Algorithms for Graph Connectivity and Cut Problems

– Connectivity Augmentation, All-Pairs Minimum Cut, and Cut-Based Clustering –

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Tanja Hartmann

aus Pforzheim

Tag der mündlichen Prüfung: 13. Juni 2014
Erster Gutachter: Prof. Dr. Dorothea Wagner
Zweiter Gutachter: Prof. Dr. Michael Kaufmann

Acknowledgements

When I signed the contract for a position as a research assistant, I did not have a doctoral degree in mind in the first place. It was more an opportunity to work with great colleagues in a comfortable atmosphere doing things I love that was offered to me by Dorothea Wagner. Before I started the position, I could already experience Dorothea's fair and benevolent leading principles in a very sincere conversation. Around four and a half years later, when I told her that Paula will be with me at the defense, she kept her word and supported me in many ways. Not least she was willing, despite her filled calendar, to review my thesis in time such that I could meet my desired deadline for the defense and finish my PhD right before my sweet little daughter was born. Thank you, Dorothea, for your endorsement, the trust you have placed in me and the great time I was allowed to spend in your group. The same holds for my second reviewer, Michael Kaufmann, who also made my desired deadline possible. Thank you, Michael, for your interest in my thesis and your willingness to be my reviewer. Beside the reviewers, I want to thank Audrey Bohlinger for her great work regarding the final organizational matters that come with handing in a thesis and preparing for a defense, as well as Henning Meyerhenke, Ralf Reussner, Peter Schmitt, and Jörg Henkel for contributing to my successful defense. Special thanks go to Roland Glantz and Sebastian Stüker for their interest in my results and the inspiring discussions on possible applications and accelerating techniques of my algorithms, which were a good preparation for the defense.

Many thanks further go to Ignaz Rutter, Martin Nöllenburg and Markus Völker for their helpful comments after proof reading. In particular my former office mate Markus has read huge parts of this thesis. Thank you, Markus, for the time you invested, the accuracy you have shown and your words of encouragement. Not least I owe it to you that I walked into the defense with self-assurance and the confidence that my results are worth a doctoral degree.

My current office mate Andreas Gemsa wrote up his thesis at almost the same time as I did. In this context we exchanged many helpful information, which saved valuable time. Thanks, Andreas, for your cooperative attitude, and thanks to all colleagues for the cooperative atmosphere in the group. Furthermore, I thank our secretaries Lilian Beckert, Elke Sauer, and Simone Meinhart, as well as our system administrators Bernd Giesinger and Ralf Kölmel, who did a great job enabling me to concentrate on my research instead of battling with bureaucratic and technical issues.

Looking further back, I consider myself very lucky that Robert Görke supervised my diploma thesis, which became the basis of my first conference publication and initiated my time as a PhD student. During this time I had the chance to work with and learn from involved coauthors in fruitful cooperation. I thank Andrea Kappes, Michael Hamann, Jonathan Rollin, Ignaz Rutter, and Christof Doll for contributing to my work. Special thanks go to Ignaz for teaching me how to write not only substantially correct but also well structured and easily readable scientific texts and for his good advice. The motivation I obtained from the cooperation with you, Ignaz, after all encouraged me to finish my PhD.

In Dagstuhl I had an inspiring discussion with Torben Hagerup on the essence of data structures. Due to this discussion the arguments for strengthening my results on dynamic Gomory-Hu trees, for which I searched for almost four years, finally came into my mind. Thus, I send my thanks to Torben Hagerup for the inspiration and to Alejandro Lopez-Ortiz, Ulrich Carsten Meyer, and Robert Sedgewick for inviting me to Dagstuhl.

Last but not least, I want to thank my family for the support and the encouragement I received. Above all, I deeply thank Toby, who not only supplied me with delicious food in the late hours I worked at this thesis, but unconditionally accompanied my way through three diplomas and this PhD, and now shares with me the most exiting experience in life: being parents of little Paula. Toby and Paula gave me the serenity that made me even enjoy the very time consuming write-up of this thesis.

Deutsche Zusammenfassung

Diese Arbeit beschäftigt sich mit der Entwicklung von Algorithmen zur Lösung graphtheoretischer Fragestellungen, die sich auf den Zusammenhang eines gegebenen Graphen beziehen. Der Zusammenhang, und damit unmittelbar verbunden, die Betrachtung von Schnitten in Graphen sind grundlegende Konzepte der Graphentheorie. Dabei unterscheidet man zwischen *Kanten-* und *Knotenzusammenhang* sowie zwischen *Kanten-* und *Knotenschnitten.* Ein Graph hat einen umso höheren Zusammenhang desto gewichtiger der minimale Schnitt ist, der nötig wird um den Graphen in zwei Teile zu trennen. Ein Schnitt ist dabei eine Kanten- oder Knotenmenge, deren Löschung zur Teilung des Graphen führt. Die Definition des *Gewichts* oder der *Kosten* eines Schnittes hängt von der betrachteten Fragestellung sowie den Eigenschaften des zugrundeliegenden Graphen ab. Während man in ungewichteten Graphen meist an der bloßen Anzahl der Kanten oder Knoten eines Schnittes interessiert ist, wird in gewichteten Graphen oftmals die Summe der Kanten- oder Knotengewichte im Schnitt betrachtet. Manche Problemstellungen betrachten darüber hinaus Schnitte, deren Gewicht erst im Verhältnis zu weiteren Kenngrößen, wie zum Beispiel der Größe der entstehenden Schnittseiten, minimal wird. Die Berechnung solch *relativ minimaler* Schnitte gilt im allgemeinen als NP-schwer. Dagegen berechnen effiziente Flussalgorithmen wie der Push-Relabel-Ansatz von Goldberg und Tarjan [57] *absolut minimale* Kantenschnitte zwischen gegebenen Knoten s und t in polynomieller Zeit. Die Verbindung von Schnitt- und Flussproblemen liefert dabei das Max-Flow/Min-Cut-Theorem, das 1956 erstmals von Ford und Fulkerson [45] formuliert wurde. Dieses Theorem beweist die Äquivalenz des Wertes eines maximalen s-t-Flusses und des Gewichts eines minimalen s-t-Kantenschnittes in einem gewichteten Graphen. Die analoge Fragestellung des Knotenzusammenhangs lässt sich ebenfalls als Flussproblem formulieren.

Bestimmte Grapheigenschaften können darüber hinaus die Nutzung speziell zugeschnittener Techniken erlauben, die zu noch schnelleren Algorithmen führen oder eine effiziente Lösung von allgemein schweren Problemstellungen ermöglichen. So sind für planare Graphen oftmals Algorithmen mit sehr schnellen, fast linearen Laufzeiten möglich. Auch für ungewichtete Graphen ist manche schnittbasierte Fragestellung einfacher zu handhaben als für gewichtete Graphen, da jede Kante denselben vorhersehbaren Beitrag zum Gewicht eines Schnittes leistet. In dieser Arbeit betrachten wir sowohl planare Graphen, deren besondere Eigenschaften wir uns zu Nutze machen um schnelle Algorithmen zu entwickeln, als auch allgemeine, gewichtete Graphen in statischen und dynamischen Szenarien, für die wir Fragestellungen beantworten, die bisher nur für ungewichtete Graphen und andere eingeschränkte Graphklassen geklärt waren.

Die Arbeit besteht aus drei Teilen, von denen jeder einen speziellen Zusammenhangsbegriff in einem bestimmten Teilgebiet der Graphentheorie betrachtet. Der erste Teil beschäftigt sich mit der Erhöhung des (absoluten) Knotenzusammenhangs durch Augmentierung eines gegebenen Graphen. Im zweiten Teil spielen (absolut) minimale Kantenschnitte zwischen allen Knotenpaaren die Hauptrolle. Der dritte Teil widmet sich schließlich besonders stark zusammenhängenden Teilgraphen, sogenannten *Clustern*, und betrachtet die *Expansion* solcher Teilgraphen, die durch relativ minimale Kantenschnitte definiert ist. Den in dieser Arbeit entwickelten Algorithmen geht außerdem eine intensive Untersuchung der jeweiligen problemspezifischen Eigenheiten voraus, die in einem tiefen Problemverständnis mündet, und so den Entwurf schneller und korrekter Algorithmen erst ermöglicht.

Teil I – Graphaugmentierung mit Zusammenhangserhöhung

Die Augmentierung eines Graphen durch zusätzliche Kanten um demselben bestimmte Eigenschaften zu verleihen, ist ein lange etabliertes Teilgebiet der Graphentheorie. Neben der Erlangung bestimmter Eigenschaften wird dabei oftmals auch die Erhaltung vorherigen Eigenschaften als Nebenbedingung gestellt. Je nach Ausgestaltung der Eigenschaften und Bedingungen sind die resultierenden Problemvarianten unterschiedlich schwer zu lösen. In dieser Arbeit betrachten wir Augmentierungsprobleme, die eine Erhöhung des Zusammenhangs zum Ziel haben. In infrastrukturellen Netzwerken geht ein starker Zusammenhang unmittelbar einher mit einer hohen Robustheit, weshalb Zusammenhangsaugmentierungen zum Beispiel in der Netzwerkplanung von großer Bedeutung sind. Auch im Gebiet des Graphenzeichnens setzen viele Konstruktionstechniken bestimmte Mindestzusammenhänge voraus. Für nicht notwendiger Weise planare Graphen wurden bereits vielfältige Problemvarianten der Zusammenhangsaugmentierung betrachtet [48, 113]. Für planare Graphen wurde die Zusammenhangsaugmentierung durch Kant und Bodlaender [88] initiiert, die Zeichenalgorithmen betrachteten, die zweifach zusammenhängende Eingabegraphen verlangen.

Teil I dieser Arbeit betrachtet ebenfalls planare Graphen, allerdings mit einer zusätzlichen Regularitätsbedingung. Genauer gesagt, es wird untersucht inwiefern planare Graphen mit Maximalgrad $d \leq k$ durch das Einfügen von Kanten zu k-regulären, planaren Graphen mit bestimmtem (absoluten) Knotenzusammenhang erweitert werden können. Diese Problemstellung wird sowohl für variable Einbettungen als auch für den Fall, dass der planare Graph zusammen mit einer festen Einbettung gegeben ist, betrachtet (siehe Fig. 1). In beiden Fällen untersuchen wir alle möglichen Werte von k (also $0 \leq k \leq 5$), sowie für jedes feste k, alle möglichen Bedingungen an den Knotenzusammenhang (von keiner Bedingung bis zur Bedingung des fünffachen Zusammenhangs). Während polynomielle Algorithmen für 1- und 2-reguläre Augmentierungsvarianten aufgrund der speziellen Struktur der initialen Graphen recht einfach zu entwerfen sind, zeigt sich, dass die Existenz einer 4- und 5-regulären Augmentierung bezüglich jeder Zusammenhangsbedingung sowohl bei variabler als auch bei fester Einbettung NP-schwer zu entscheiden ist. Interessant gestalten sich die Ergebnisse für 3-reguläre Augmentierungen. Hier sind alle Probelmvarianten NP-schwer solange eine variable Einbettung zugelassen ist. Die Einschränkung auf eine feste Einbettung führt jedoch zu polynomieller Lösbarkeit mit einer

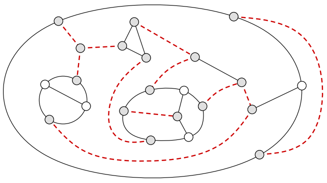


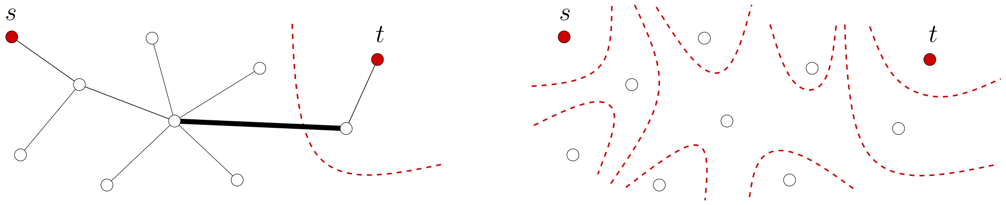
Figure 1: Planarer Graph G mit fester Einbettung (durchgezogene Kanten) und Kantenmenge einer zweifach zusammenhängenden, planaren, 3-regulären Augmentierung (gestrichelt).

Laufzeit in $O(n^{1.5})$, solange kein dreifacher Zusammenhang als Nebenbedingung an den resultierenden Graphen gestellt wird. Die Existenz einer dreifach zusammenhängenden, planaren, 3-regulären Augmentierung bleibt auch bei fester Einbettung NP-schwer zu entscheiden. Pilz [119] hat zeitgleich und unabhängig von den Ergebnissen in dieser Arbeit planare, 3-reguläre Augmentierungen bei variabler Einbettung untersucht. Der hier präsentierte NP-Schwerebeweis verstärkt Pilz' Ergebnis zur NP-Schwere dieses Problems, indem er zeigt, dass das Problem auch für zweifach zusammenhängende Eingabegraphen NP-schwer bleibt. Darüber hinaus bietet diese Arbeit für jede weitere planare, k-reguläre Augmentierungsvariante entweder einen effizienten Algorithmus oder einen NP-Schwerebeweis und beantwortet so vollständig die Frage der Komplexität planarer, regulärer Graphaugmentierungen mit zusätzlicher Erhöhung des Knotenzusammenhangs. Dies beantwortet schließlich auch die noch offenen Fragen von Pilz.

Teil II – Paarweise minimale Schnitte

Gomory und Hu [59] konnten bereits 1961 zeigen, dass $n-1$ Flussberechnungen genügen um für jedes der $\binom{n}{2}$ Knotenpaare in einem ungerichteten, gewichteten Graphen einen (absolut) minimal trennenden Kantenschnitt zu finden. Während für ungewichtete Graphen und spezielle Graphklassen, wie zum Beispiel planare Graphen, inzwischen schnellere Algorithmen existieren um für jedes Knotenpaar einen minimal trennenden Kantenschnitt zu bestimmen [17, 13], ist für allgemeine ungerichtete, gewichtete Graphen die Berechnung von $n-1$ maximalen Flüssen noch immer die schnellste Lösungsmethode.

Kernstück des Beweises von Gomory und Hu ist die Konstruktion einer Baumstruktur auf den Knoten des Graphen, deren Kanten Schnitte im zugrundeliegenden Graphen repräsentieren. Genauer gesagt, jede Baumkante $\{s,t\}$ induziert einen minimalen s-t-Kantenschnitt, dessen Gewicht an der Baumkante vermerkt ist. Aus dieser Eigenschaft ergibt sich auch, dass für Knoten s und t, die keine Kante im Baum darstellen, ein minimaler s-t-Kantenschnitt leicht am Baum ablesbar ist, nämlich in Form einer leichtesten Kante auf dem eindeutigen Pfad zwischen s und t (siehe Fig. 2). Ein Gomory-Hu-Baum ist somit eine Datenstruktur, die für jedes paar von Knoten einen minimal trennenden Kantenschnitt bereitstellt. Die Repräsentation von Schnitten in solch kompakter Form ist Grundlage vieler weiterer Algorithmen zur Lösung graphtheoretischer und kombinatorischer Optimierungsprobleme. Neben paarweisen minimalen s-t-Kantenschnitten lassen sich zum Beispiel auch global minimale Kantenschnitte in sogenannten Kakteen zusammenfassen [32]. Dagegen sind paarweise minimale Knotenschnitte nicht ohne Weiteres in einer analogen Baumstruktur darstellbar [12].



(a) Günstigste Kante (fett schwarz) auf dem Pfad zwischen s und t repräsentiert minimalen s-t-Schnitt (rot gestrichelt) im zugrundeliegenden Graphen.

(b) Menge aller Kanten im Baum repräsentiert $n-1$ kreuzungsfreie minimale s-t-Schnitte (rot gestrichelt).

FIGURE 2: Gomory-Hu-Baum auf zehn Knoten, Kanten des zugrundeliegenden Graphen sind nicht dargestellt. Baumkanten entsprechen nicht zwangsläufig Graphkanten.

Teil II dieser Arbeit stellt eine neue, effizient konstruierbare Datenstruktur vor, die die Klasse aller paarweisen minimalen Kantenschnitte repräsentiert, die eine der beiden Schnittseiten minimieren. Aufgrund ihrer Eindeutigkeit finden diese Schnitte in vielen Algorithmen Anwendung und stellen ein beliebtes Beweisinstrument dar. Auch diese Arbeit präsentiert in Teil III einen Clusteralgorithmus, der wesentlich auf der neu entwickelten Datenstruktur aufbaut.

Teil II betrachtet außerdem die bewährte Datenstruktur des Gomory-Hu-Baumes in einem dynamischen Szenario, in dem sich der zugrundeliegende Graph von Zeitschritt zu Zeitschritt durch einzelne Kanten- oder Knotenveränderungen fortentwickelt (siehe Fig. 3). Im Rahmen einer *Sensitivitätsanalyse* wurde die Gewichtsentwicklung paarweiser minimaler Kantenschnitte in sich verändernden Graphen erstmals bereits kurz nach der Vorstellung von Gomorys and Hus Baumstruktur untersucht [36, 11]. Ziel dieser Arbeit ist jedoch nicht nur die Beschreibung sich verändernder Gewichte, sondern darüber hinaus eine möglichst effiziente Bereitstellung der vollständigen Struktur eines Gomory-Hu-Baumes in jedem Zeitschritt. Für ungewichtete Graphen beschreiben Lin und Ma [100] einen ersten Ansatz eines entsprechenden Update-Algorithmus. Für gewichtete Graphen galt diese Problemstellung jedoch bislang als offen und schwer zu lösend [11].

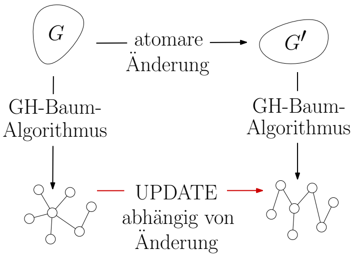


FIGURE 3: Dynamisches Szenario. Graph G unterliegt Kanten- und Knotenänderungen, die Zeitschritte induzieren.

Neben der möglichst *effizienten Konstruktion* des jeweils neuen Baumes auf der Basis des vorangegangenen Baumes, sind *glatte Übergänge* zwischen den Zeitschritten ein weiteres typisches Ziel in dynamischen Szenarien. In Bezug auf Gomory-Hu-Bäume heißt das, Bäume aufeinanderfolgender Zeitschritte sollen möglichst geringen Abweichungen unterliegen. Der in dieser Arbeit entwickelte Update-Algorithmus für Gomory-Hu-Bäume in gewichteten Graphen garantiert optimale Glattheit und beweist gleichzeitig ein großes Sparpotenzial bei der Aktualisierung vorangegangener Bäume. Selbst die Worst-Case-Laufzeit von $n-2$ Flussberechnungen kann unter der Annahme bestimmter, durchaus plausibler, Bedingungen als optimal betrachtet werden. Somit löst dieser Update-Algorithmus das offene Problem voll-dynamischer Gomory-Hu-Bäume in gewichteten Graphen.

Teil III – Schnittbasiertes Clustern

In Gegensatz zu den in Teil I betrachteten Augmentierungsproblemen, deren Ziel es ist, den Zusammenhang eines gegebenen Graphen zu erhöhen, ist das Ziel des Graphenclusterns, einen gegebenen Graphen in möglichst dichte Teilgraphen zu zerlegen, die untereinander nur schwach durch Kanten verbunden sind (siehe Fig. 4). Basierend auf der Annahme, dass reale Netzwerke eine solch inhärente Gruppenstruktur aufweisen, ist Graphenclustern ein weit verbreitetes Instrument im Bereich der Netzwerkanalyse.

Teil III dieser Arbeit beschäftigt sich mit einem Ansatz zum Clustern ungerichteter, gewichteter Graphen, der auf der Struktur eines Gomory-Hu-Baumes basiert und von Flake et al. [42] 2004 als *Cut-Clustering-Algorithmus* vorgestellt wurde. Bemerkenswert an diesem Ansatz ist, dass die resultierenden Cluster, abhängig von einem Parameter α, eine garantierte *Mindestexpansion* und damit eine Mindestqualität aufweisen, obwohl bereits die bloße Berechnung der Expansion eines (Teil-)Graphen, die durch relativ minimale Kantenschnitte bestimmt ist, NP-schwer ist [87].

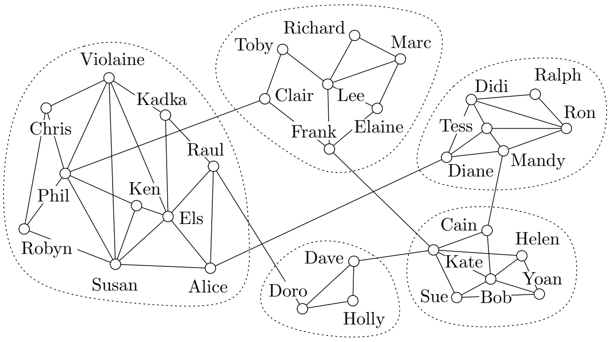


FIGURE 4: Exemplarische Clusterung eines Freundschaftsnetzwerkes. Eng befreundete Gruppen zeichnen sich durch viele Freundschaftsbeziehungen, dargestellt durch Kanten, aus.

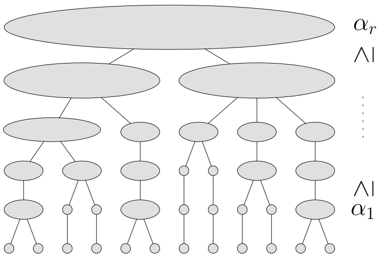


FIGURE 5: Schematisierte Clusterungshierarchie für nichtaufsteigende Parameterwerte α.

Darüber hinaus ist die Expansion der Kantenschnitte, die die einzelnen Cluster vom Rest des Graphen trennen, ebenfalls durch α nach oben beschränkt, was eine zusätzliche Qualitätsgarantie darstellt. Für nichtaufsteigende Werte von α erhält man außerdem hierarchisch geschachtelte Clusterungen, deren Grobheit zunimmt (siehe Fig. 5). Im Gegensatz zu einer einzelnen Clusterung erlaubt eine solche Hierarchie die Auswahl einer besonders geeigneten Clusterung für die jeweilige Anwendung.

Neben einer schnittbasierten Charakterisierung der Cluster und einer experimentellen Untersuchung der Aussagekraft der Qualitätsgarantien im Vergleich zu trivialen Schranken, entwickelt diese Arbeit das Verfahren von Flake et al. in zwei Richtungen weiter. Die erste Richtung betrifft die Berechnung von statischen Clusterungshierarchien. Während das von Flake et al. vorgeschlagene Berechnungsverfahren auf einer binären Suche auf dem Wertebereich für α basiert und somit nicht garantieren kann, dass alle möglichen Hierarchieebenen gefunden werden, stellt diese Arbeit ein parametrisches Suchverfahren vor, das eine garantiert vollständige Hierarchie aller möglichen Clusterungen berechnet. Außerdem wird gezeigt, dass zur Konstruktion von Clusterungshierarchien nicht nur bestimmte minimale s-t-Kantenschnitte (nämlich diejenigen, die eine Schnittseite minimieren) geeignet sind, sondern dass auch die Verwendung beliebiger minimaler s-t-Schnitte zu hierarchisch angeordneten Clusterungen führt, ohne die Qualitätsgarantien in den einzelnen Clusterungen zu verlieren. Diesen zusätzlichen Freiheitsgrad macht sich die zweite Richtung der Weiterentwicklung zu Nutze.

Die zweite Richtung der Weiterentwicklung beschreibt die Anpassung des hierarchischen Cut-Clustering-Algorithmus an dynamische Graphen. Dazu entwickelt diese Arbeit einen Update-Algorithmus, der in Teilen auf Erkenntnissen und Techniken basiert, die bereits bei der Betrachtung dynamischen Gomory-Hu-Bäume eine Rolle spielten, und der ebenfalls optimale Glattheit garantiert und ein hohes Sparpotenzial, und damit gute Laufzeiten, bei der Aktualisierung der Clusterungshierarchien in den einzelnen Zeitschritten erzielt. Die Optimalität der Glattheit wird nicht zuletzt durch den zuvor bewiesenen, zusätzlichen Freiheitsgrad bezüglich der Wahl der Kantenschnitte möglich. Dieser Update-Algorithmus stellt die erste korrekte Anpassung des Cut-Clustering-Algorithmus an dynamische Graphen dar, nachdem sich ein früherer Versuch anderer Autoren [124] als fehlerhaft erwies.

Neben der Weiterentwicklung des Cut-Clustering-Algorithmus, bildet die Einordnung der Cluster in eine Sammlung verwandter Clusterkonzepte aus dem Bereich *dominant zusammenhängender Teilgraphen*, wie sie bei der Analyse sozialer Netzwerke betrachtet werden [15], sowie die Definition eines weiteren solchen Konzeptes (sogenannte Source-Communities, oder kurz: SCs) den letzten Schwerpunkt des dritten Teils. In diesem Zusammenhang wird ein Algorithmus vorgestellt, der nach der Vorberechnung der neuen Datenstruktur aus Teil II für eine gegebene Source-Community in Linearzeit die eindeutige maximale SC-Clusterung berechnet, das heißt, die Clusterung, deren Cluster Source-Communities sind, wobei eines der Cluster der gegebenen Source-Community entspricht, und die die Eigenschaft besitzt, dass die Cluster jeder weiteren SC-Clusterung, die ebenfalls die gegebene Source-Community enthält, in den Clustern der maximalen SC-Clusterung enthalten sind. Diese Art maximaler SC-Clusterung kann zum Beispiel der Bewertung von Clusteralgorithmen wie dem Cut-Clustering-Algorithmus dienen. Findet ein solcher Algorithmus, dessen Ziel es ist, eine Clusterung aus dominant zusammenhängenden Teilgraphen zu berechnen, auf bestimmten Graphen nur wenige Cluster sinnvoller Größe und viele Singletons, so kann die Bestimmung der maximalen SC-Clusterung darüber Aufschluss geben, ob der Algorithmus überhaupt die Chance gehabt hätte, eine bessere Clusterung zu finden, oder ob es eventuell aufgrund der Graphstruktur gar keine Clusterung aus balancierteren, dominant zusammenhängenden Teilgraphen gibt.

Contents

Acknowledgements **iii**

Deutsche Zusammenfassung (German Summary) **v**

1 Introduction **1**
1.1 A Brief Outline 5
1.2 Notations and Definitions on Graphs 8
1.3 Cuts and Connectivity 11
1.4 Graph Instances 15
1.5 A Brief Glance at Complexity Theory 19

I Connectivity Augmentation **23**

2 Introduction – A Bunch of Problem Variants **25**
2.1 Contribution and Outline 26
2.2 Preliminaries 27

3 The Nature of Planar 3-Regular Augmentation **31**
3.1 NP-Completeness of 3-PRA 31
3.2 A Simple Polynomial-Time Algorithm for 3-FEPRA 34
3.3 A Systematic Study of 3-FEPRA 36

4 Algorithms for *C*-connected 3-FEPRA **57**
4.1 Construction of the Assignment Graph G_A 57
4.2 Improving the Running Times 67
4.3 Complexity of Triconnected 3-FEPRA 74

5 Complexity of Planar 4- and 5-Regular Augmentation **79**
5.1 NP-Completeness of C-connected 4-PRA/FEPRA 79
5.2 NP-Completeness of C-connected 5-PRA/FEPRA 87

Conclusion of Part I **93**

II All-Pairs Minimum Cut **95**

6 Introduction – Gomory-Hu Trees and Other Data Structures **97**
6.1 Contribution and Outline 102
6.2 Preliminaries 104

7 Gomory-Hu Trees in Static Graphs **113**
7.1 Basics for Constructing Gomory-Hu Trees 113
7.2 Unique-Cut Trees 118

8 Gomory-Hu Trees in Dynamic Graphs **129**
8.1 Finding Reusable Cuts 132
8.2 The Dynamic Gomory-Hu Tree Algorithm 134
8.3 Running Times of the Update Procedures 137

9 Optimality in Smoothness and Running Time **145**
9.1 Optimal Temporal Smoothness in Case of Edge Deletion or Cost Decrease 146
9.2 Optimal Temporal Smoothness in Case of Edge Insertion or Cost Increase 149
9.3 Optimality of Asymptotic Worst-Case Running Time 164

Conclusion of Part II **173**

III Cut-Based Clustering **175**

10 Introduction – Graph Clustering and Cohesive Subset **177**
10.1 Contribution and Outline 180
10.2 Preliminaries 182

11 Static Hierarchies of Cut Clusterings **185**
11.1 Cut Clusterings and Source Communities 185
11.2 Complete Hierarchies of Cut Clusterings 193
11.3 A Case Study on Expansion and Modularity 198

12 Maximum Source-Community Clusterings **205**
12.1 Constructing Maximum SC-Clusterings 208
12.2 Constructing Overlay Clusterings 210
12.3 An Exemplary Analysis of the Lesmis Network 211

13 The Unrestricted Cut-Clustering Algorithm **213**

14 Fully-Dynamic Hierarchies of Unrestricted Cut Clusterings **219**
14.1 The Dynamic Attempt of Saha and Mitra 220
14.2 Correct Update Procedures for Single Clusterings 222
14.3 Optimal Temporal Smoothness for Single Clusterings 227
14.4 Running Times for Single Clusterings 229
14.5 Update Procedures for Hierarchies 232

Conclusion of Part III **237**

15 Conclusion **239**

Bibliography **241**

Appendices

CHAPTER 1

Introduction

In this thesis we develop algorithms for solving graph-theoretic problems related to connectivity and minimum cuts. It consists of three parts considering connectivity augmentation problems, all-pairs minimum cut representations and cut-based clustering approaches. “Informally, an algorithm is any well-defined computational procedure [consisting of a finite number of instructions] that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a [finite] sequence of computational steps that transform the input into the output.”[27]

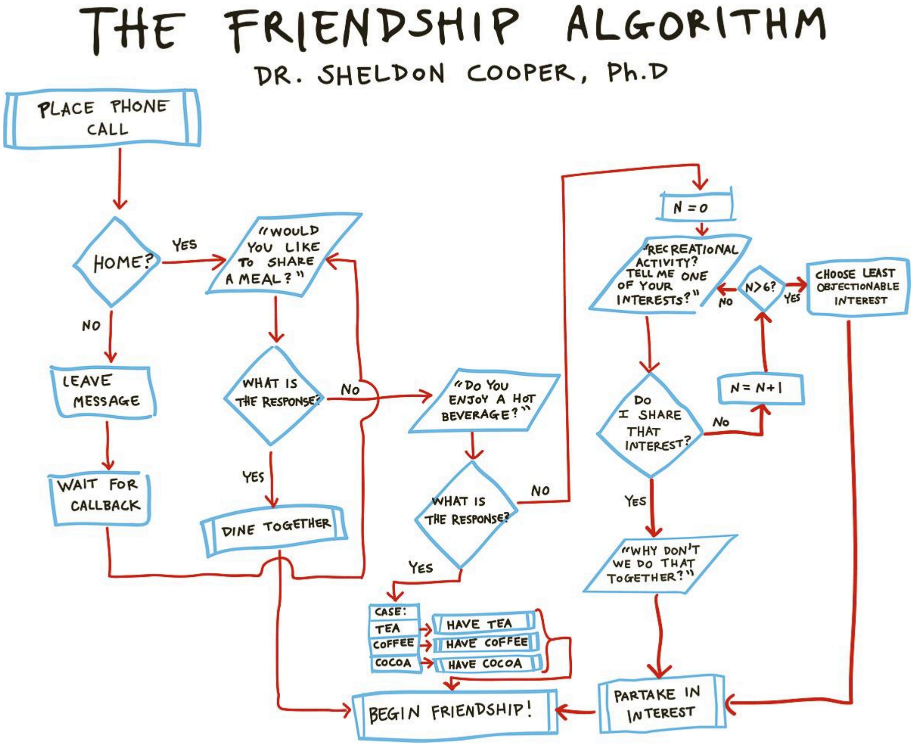


Figure 1.1: Flowchart of the Friendship Algorithm by Dr. Sheldon Cooper, flatmate (and friend?) of Leonard Hofstadter in the American sitcom *The Big Bang Theory* created by Chuck Lorre and Bill Prady. Screenshot provided by http://www.emp-online.co.uk/art_240796/, distributed by Nathan Lorah via Mathlab File Exchange (2013/03/11).

The comprehensible desire for well-defined, finite instructions to solve difficult situations also in everyday life may tempt people, and in particular nerdy guys like Sheldon (character in *The Big*

Bang Theory), to ignore the computational aspect in the notion of an algorithm, thus squeezing almost any aspect of personal relation into a flowchart. Nevertheless, Sheldon's *friendship algorithm* (Fig. 1.1) provides all characteristics usually associated with a computational algorithm. It takes a phone call as input, processes a finite number of well-defined instructions and in the end returns friendship as output. The algorithms considered in this thesis take a *graph* as input and return answers concerning graph-theoretic problems. A graph is a representation of a set of objects, called *vertices*, where some pairs of objects are connected by links or *edges*. The flow chart in Fig. 1.1, for example, can be considered as a *directed* graph, where each edge has an orientation. Graphs are well suited to model relations between objects in order to simplify a given situation for analysis. Furthermore, many fields are faced with the task of analyzing data that already entail an inherent graph structure or are directly derived from physical or virtual networks, which clearly conform to the structure of a graph. This is why the terms graph and network are often used interchangeably.

A Bit of History. Leonard Euler is said to be the first one who used a graph for abstraction in his study of the problem of the Königsberger bridges, which counts as the birth of graph theory. In his article, published in 1736, Euler considered seven bridges in the city of Königsberg (nowadays Kaliningrad, Russia) that link an island (called Kneiphof) in the River Pregel to the opposite riversides. Euler asks if it is possible to find a walkway through the city that passes each bridge exactly once (nowadays called an *Euler path*). For the sake of simplicity, he denoted the different landmasses separated by the river by capital letters and associated each bridge with the pair of landmasses it links (Fig. 1.2). This abstract description of the situation laid the foundation of graphs as mathematical objects, although the term graph was not introduced until 1878, when it was first used in the sense described above by Sylvester [134, 63]. Since each bridge in Königsberg can be passed in both directions, Euler's abstraction is an *undirected* graph, which finally enabled Euler to get to the bottom of the general existence of Euler paths. According to his famous, graph-theoretic theorem, an undirected graph admits an Euler path, that is, a walk through the graph that passes each edge exactly once, if and only if zero or exactly two vertices are part of an odd number of edges. The number of edges a vertex is part of is called the *degree* of the vertex. Due to this characterization, a simple algorithm that decides the Euler path problem consists of the following instructions. Set a vertex counter to 0. For each vertex, determine the degree. If the degree is odd, increase the vertex counter. If the vertex counter is 0 or 2 in the end, return yes. Otherwise, return no. Besides the undirected graph edges and the parity of the vertex degrees, we can observe even more fundamental graph properties at Euler's example. One property is the existence of parallel edges corresponding to bridges like a and b in Fig. 1.2 that are associated with the same pair of landmasses. A graph with parallel edges is called a *multigraph*, while a graph without parallel edges is a *simple graph*. Another property are missing edge costs. That is, in the context of the Euler path problem each bridge/edge is equally important, and thus, Euler considered an

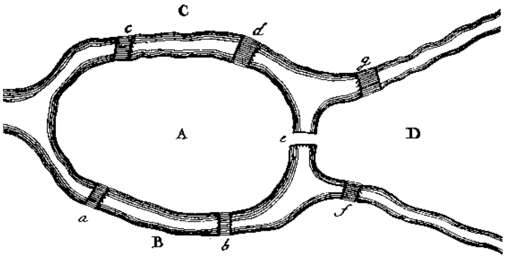


FIGURE 1.2: Euler's simplification of the landmasses (capitals) and bridges (small letters) in Königsberg. The bridges can be modeled as pairs of landmasses: $a = b = \{A, B\}$, $c = d = \{A, C\}$, $e = \{A, D\}$, $f = \{B, D\}$, $g = \{C, D\}$.

unweighted graph. Other graph-theoretic problems also consider *weighted* graphs, and seek for cheap or expensive solutions with respect to the edge costs. The third property is *connectivity*. Euler's theorem only holds if each landmass that is part of a bridge can be reached from any other landmass that is part of a bridge. Otherwise the answer to the Euler path problem is always no. In other words, Euler assumed that the given graph is connected, which equals the property that each vertex in the graph can be reached from any other vertex by passing a sequence of edges and vertices.

Connectivity plays a central role in the three parts of this thesis, which focus on graph connectivity problems from three different points of view. Part I addresses connectivity augmentation problems where the aim is to increase the connectivity of a given graph. Part II is concerned with the representation of minimum cuts in undirected, weighted graphs, and thus, besides the interesting theoretical aspects of this problem, provides data structures for solving other connectivity-related problems, as those considered in Part III. Part III deals with cut-based graph clusterings, which are, in contrast to *augmentations*, *decompositions* of graphs into well-connected subgraphs. Part III relies in large parts on Part II. In the following we provide a brief motivation for each part, where we consider the two contrary directions of connectivity augmentation and clustering decomposition first.

Connectivity and Graph Augmentation. Today the importance of connectivity becomes obvious in particular in infrastructural networks like urban accessibility networks, telecommunication networks, and electricity networks. The German extra-high voltage (EHV) network, for example, suffers from low throughput between different regions, in particular from North to South. That is, in the corresponding graph the connectivity between these regions is too low. In order to increase the connectivity and thus the performance and the robustness it is planned to augment the network by further power lines indicated by the orange lines in Fig. 1.3. The decision where to place the additional power lines is basically a graph augmentation problem with respect to special objectives and side constraints related to individual properties of the network, as for example the physical embedding. Graph augmentation problems are a classical field in graph theory with a counterpart often considered as subgraph problems. Instead of inserting edges into a graph, in a subgraph problem the question is which edges to delete in order to obtain a subgraph of special properties. In Part I of this work we concentrate on algorithms for augmentation problems on planar graphs. In this context we regard two objectives, regularity and connectivity, and the side constraint of

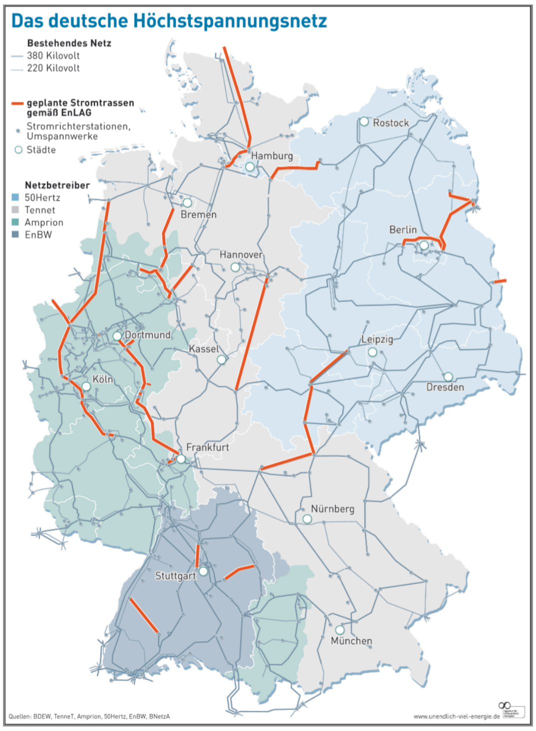

Das deutsche Höchstspannungsnetz
Bestehendes Netz
380 Kilovolt
220 Kilovolt
geplante Stromtrassen gemäß EnLAG
Stromrichterstationen, Umspannwerke
Städte
Netzbetreiber
50Hertz
Tennet
Amprion
EnBW
Rostock
Hamburg
Bremen
Berlin
Hannover
Dortmund
Kassel
Leipzig
Köln
Dresden
Frankfurt
Nürnberg
Stuttgart
München
Quellen: BDEW, TenneT, Amprion, 50Hertz, EnBW, BNetzA
www.unendlich-viel-energie.de

FIGURE 1.3: EHV network of Germany, 2013. Data provided by BDEW, TenneT, Amprion, 50Hertz, EnBW, BNetzA. Image provided by Agentur für erneuerbare Energien, www.unendlich-viel-energie.de.

preserving planarity, possibly with respect to a given embedding. A detailed outline and our contribution to connectivity augmentation problems are found in Section 2.1.

Connectivity and Graph Clustering. In contrast to connectivity augmentation problems, where the aim is to *increase* the connectivity of a given network, the *decomposition* of a network into well connected regions that are only *sparsely* connected to each other, is an opposite direction where connectivity plays a fundamental role. Such decompositions are classically addressed by graph clustering algorithms. Graph clustering, which is a branch of network analysis, has become a central tool to examine the structure of networks, with applications ranging from the field of social sciences to biology and to the growing field of complex systems. The general aim of graph clustering is often described by the clustering paradigm of *intra-cluster density and inter-cluster sparsity*, referring to the identification of well connected subgraphs, so-called *clusters*, whose connections to other subgraphs constitute bottlenecks in the network while within the subgraphs no significant bottlenecks can be found. Formally, a bottleneck in a network is a set of edges that disconnects different parts of the network when deleted. A bottleneck thus forms a *cut*. The denser the clusters and the smaller the cuts in-between the clearer is the bottleneck-property, and thus, the better is the *clustering*. Figure 1.4 exemplarily shows a clustering of a snapshot of the evolving email-communication network of the Department of Informatics at KIT. For details on this instance see Section 1.4.

Countless formalizations of the clustering paradigm exist, however, the overwhelming majority of algorithms for graph clustering relies on heuristics, for example, for some NP-hard optimization problem, and do not allow for any structural guarantee on their output. An exception, besides spectral clustering algorithms [87], is a cut-based clustering algorithm introduced by Flake et al. [42]. This approach exploits properties of minimum cuts in order to give a quality guarantee on the connectivity within the clusters and the cuts between the clusters. In Part III of this work we investigate further properties of the clusters found by this algorithm and extend the approach in different directions, based on data structures that provide useful information about the cut structure of a graph. See Section 10.1 for a detailed outline and our contribution to cut-based clustering problems.

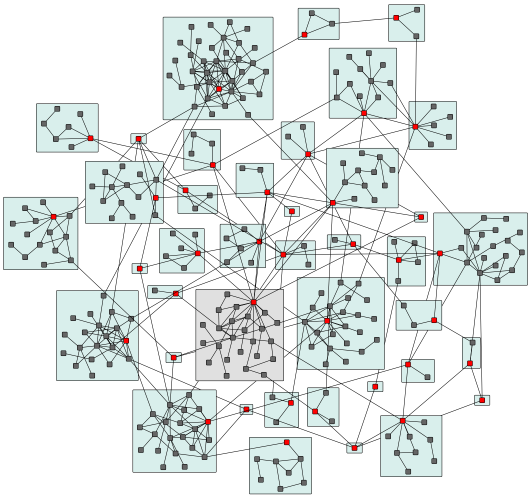


Figure 1.4: Maximum SC-clustering (Chapter 12) with respect to the gray cluster (red vertices represent sources) in the giant component of a snapshot of the email-communication network of the Department of Informatics at KIT. Clusters are indicated by filled boxes.

Connectivity and Minimum Cuts. Determining the size of a minimum cut, and thus, the connectivity in a given graph, or even finding a concrete minimum cut, are well-studied combinatorial optimization problems that can be solved efficiently. However, when we talk about algorithms for solving these kinds of problems, we need to carefully distinguish between different types of minimum cuts and connectivity as well as the many possible properties of an input graph. Some techniques that work well, for example, for directed graphs cannot be applied to undirected graphs and vice versa. In contrast, each algorithm for weighted graphs can be also used for unweighted graphs, however, for unweighted or integer weighted graphs there often exist faster approaches that exploit the special edge costs and are thus not applicable to weighted graphs. More sophisticated graph properties that support the design of algorithms faster than those for general graphs are, for example, planarity and treewidth. It further makes a difference whether we consider the *edge* or *vertex connectivity* of a graph. Intuitively, a graph is the more edge- and vertex-connected the more edges and vertices, respectively, need to be deleted in order to split the graph into two independent parts. Besides this *global* connectivity regarding the whole graph, we can further study the *local* connectivity of two designated subgraphs, like, for example, two vertices. The local connectivity describes how many edges or vertices need to be deleted in order to separate the subgraphs. A minimum set of edges that witnesses the global connectivity is a *global minimum cut*, a minimum edge set with respect to two designated vertices is called a *minimum separating cut*. There exist several elegant data structures that represent the structure of minimum cuts in a given graph. Cacti [32], for example, store all existing global minimum cuts, Gomory-Hu trees [59] provide a minimum separating cut for each vertex pair, and Picard-Queyranne DAGs [118] offer all minimum separating cuts of one designated vertex pair. In Part II of this thesis we consider Gomory-Hu trees in more detail. Besides their various applications in designing further algorithms for advanced connectivity problems [36, 11], Gomory-Hu trees are also employed in completely different contexts like, for example, robust Internet-routing [130]. In our work, Gomory-Hu trees serve as an underlying data structure for our developments on the cut-based clustering algorithm of Flake et al. [42]. Moreover, we believe they are also interesting as a fundamental object in graph theory. We give a detailed outline and point out our contribution to Gomory-Hu trees in Section 6.1.

1.1 A Brief Outline

This thesis consists of three parts, each addressing a collection of related connectivity or cut problems in simple graphs. In the following, we give a brief overview, referring to the introduction of each part for a more detailed outline and our contributions.

Part I: Connectivity Augmentation

In Part I, we study augmentation problems on undirected, unweighted, planar graphs with and without fixed embedding, where the aim is to augment the given graph to be k-regular and c-connected while preserving planarity. We consider all possible problem variants resulting from all possible choices of k and c, which are $1 \leq k \leq 5$ and $0 \leq c \leq 5$ with $c \leq k$ for variable and fixed embeddings. For each problem variant, we give either an efficient algorithm or an NP-hardness proof, therefore settling the complexity for the entire class of planar regular augmentation problems with connectivity constraints.

We show that designing algorithms for solving the variants of 1- and 2-regular connectivity augmentation is quite easy due to the simple structure of the input graphs in these cases. Note that we require the input graphs have a maximum degree of at most k. In contrast, for 4- and 5-regular connectivity augmentation, we prove the NP-hardness with respect to all connectivity constraints for variable and fixed embeddings. Interestingly, cubic connectivity augmentation turns out to be NP-hard for variable embeddings, but polynomial solvable for fixed embeddings, unless we additionally aim at 3-connectivity. Cubic augmentations are in particular interesting as the dual graph of a cubic planar graph forms a triangulation. See Section 2.1 for a more detailed outline.

Part II: All-Pairs Minimum Cut

Part II is dedicated to Gomory-Hu trees [59] on undirected, weighted graphs. A Gomory-Hu tree is an undirected, weighted tree on the vertices of a given graph such that each edge in the tree represents a minimum separating cut in the underlying graph with respect to its incident vertices. The cost of the cut is given by the cost of the tree edge. From this property, it follows directly that a minimum separating cut for two vertices that are not adjacent in the tree is given by a cheapest tree edge on the unique path between both vertices. Hence, a Gomory-Hu tree represents a minimum separating cut for each vertex pair of the graph. We first give a detailed description of the state-of-the-art algorithms for constructing Gomory-Hu trees on general graphs and explain the ideas and mechanisms these algorithms rely on. In this way we lay the basis for our new algorithmic approaches, which modify the original Gomory-Hu tree construction in order to accomplish their goals. The goal of our first algorithmic approach is the construction of a data structure, which we call *unique-cut tree*, that represents a class of special minimum separating cuts, which we call *U-cuts*. Although rarely denoted by a special name, U-cuts have many applications and appear in many proofs in the literature due to their uniqueness. In this work, the new data structure forms the basis of an algorithm presented in Part III, which returns maximum source-community clusterings with respect to given communities.

Our second algorithmic approach aims at maintaining Gomory-Hu trees also in evolving graphs. We consider a dynamic scenario where two consecutive snapshots of the underlying graph differ due to either an atomic edge change or an atomic vertex change. As usual in incremental dynamic scenarios, we are interested in updating the Gomory-Hu tree in each time step as efficiently as possible, in particular in comparison with a computation from scratch. Furthermore, we seek for smooth transitions, that is, we wish to obtain similar trees in successive snapshots. The update algorithm developed in this thesis is fast in practice, guarantees optimal temporal smoothness and is very easy to implement. See Section 6.1 for a more detailed outline.

Part III: Cut-Based Clustering

In Part III, we extend the work of Flake et al. [42] on their elegant cut-based clustering algorithm. According to the authors, we call this algorithm, which is designed for undirected, weighted graphs, *cut-clustering* algorithm. The striking feature of this approach is that, in contrast to the majority of graph clustering approaches, it provides a quality guarantee for the found clusterings in terms of expansion, which is a cut-based measure for the density of (sub)graphs. This fact is even more interesting as the computation of the expansion of a graph is already NP-hard.

In this part, we show that, beyond this quality guarantee, the clusters also provide nice cohesion properties, as already considered for a long time in social network analysis. In this context we describe *source communities* as a concept of cohesive subsets and discuss the properties of these communities in comparison with other established cohesive subsets. We then characterize the clusters that are returned by the cut-clustering algorithm in terms of source communities and show that they form a proper subclass of the class of source communities.

Flake et al. further extended their clustering algorithm to a hierarchical approach, which however is not complete, that is, in the found clustering hierarchies there might be some levels missing. Hence, we improve the approach such that the completeness of the returned hierarchies is guaranteed. Based on this complete approach, we finally conduct some experiments in order to evaluate the given quality guarantee compared to another, popular quality measure called *modularity*.

Briefly turning away from the cut-clustering algorithm of Flake et al., we also consider clusterings that consist of general source communities. At this point we draw on unique-cut trees presented in Part II, and show that this data structure allows to efficiently answer queries on inclusion-maximal source-community clusterings with respect to some given communities.

Again returning to the hierarchical algorithm of Flake et al., which gains the cohesion properties of its clusters due to the use of the special U-cuts during the construction, we finally show that the same construction technique can also be hierarchically applied with more general cuts still preserving the quality guarantee in terms of expansion. This yields a higher degree of freedom that admits choosing the most appropriate cuts regarding, for example, a special application, which compensates the loss of the cohesion properties. We adapt the in this sense unrestricted hierarchical approach to the same dynamic scenario as the Gomory-Hu trees in Part II, thereby exploiting the degree of freedom in order to achieve good temporal smoothness. See Section 10.1 for a more detailed outline.

How to Read this Work. We omit a central related work section in this introduction. Instead, related concepts and results are either presented in the introduction of each part or in the particular chapter where the relation to our work becomes clear. The introduction of each part further provides the specific notation, a detailed outline of the chapters with references to the corresponding publications, and our contributions presented in this part. If there is further background information needed in order to understand the ideas and arguments in a particular part, this information is also preliminarily provided in the introduction of the part. Each part further has its own conclusion. General notation and definitions with respect to graphs, connectivity in graphs, and minimum cuts, as well as existing approaches related to connectivity and minimum cuts are discussed in the remainder of this chapter. Furthermore, we provide a brief introduction to complexity theory and an overview on the instances we use in different experiments at the end of this chapter. The sources of the used instances is only references once, namely in this preliminary section on graph instances. In the description of the experiments we simply refer to this section. The same holds for the template library we used in our experiments.

Part I and Part II of this work can be read independently from Part III and from each other. Part III, however, in large parts relies on Part II. To be exact, the dependencies are as follows. Apart from the case study in Section 11.3, the explanations on static hierarchies of cut clusterings in Chapter 11 extensively use U-cuts and M-sets, which are presented in Section 6.2.2. The maximum source-community clusterings in Chapter 12 are based on unique-cut trees developed in Section 7.2. To understand the unrestricted cut-clustering approach in Chapter 13 it is necessary to know, besides the static clustering algorithm in Section 11.1.1, the basics of the Gomory-Hu tree construction provided in Section 7.1. Finally, Chapter 14 cannot be read without Chapter 13, the knowledge about U-cuts and M-sets, and the results on dynamic Gomory-Hu trees in Chapter 8.

We further point out the main tools that are used to obtain the results in Part II and Part III, and thus, are referred in many proofs. These tools are given by four lemmas, which have a very similar structure. The first lemma, we call it the Non-Crossing Lemma (7.2), was already proven by Gomory and Hu [59], and Gusfield [66], and is presented in Section 7.1.1. It ensures the correctness of the original Gomory-Hu tree construction as it admits to bend minimum separating cuts along other minimum separating cuts without changing costs, thus preventing crossings. Since non-crossing cuts are either nested or disjoint, this lemma tells us something about the nesting behavior of minimum separating cuts in a given graph. The second lemma,

Lemma 8.9, adapts the statement of the Non-Crossing Lemma to a dynamic scenario, more precisely, to the case where an edge in the underlying graph is deleted or the cost of an edge decreases. In this situation we are busy with cuts in two different graphs, namely the current graph G and the graph $G^{\ominus}$ resulting from G due to the edge change. Lemma 8.9, which is proven in Section 8.2, then describes how to bend arbitrary cuts in $G^{\ominus}$ along minimum separating cuts in G without changing costs. The third lemma, Lemma 9.4 in Section 9.2.1, is the analog lemma for the situation where an edge is inserted into G or the cost of an edge increases, resulting in $G^{\oplus}$. The adaption of the original Gomory-Hu tree construction to evolving graphs in Chapter 8 as well as the update algorithm for single clusterings developed in Section 14.2 strongly rely on these lemmas. The fourth lemma, Lemma 13.2 in Chapter 13, finally characterizes the nesting behavior of minimum separating cuts on different levels of a clustering hierarchy, which admits our results regarding static and dynamic clustering hierarchies. The proofs of these lemmas are very technical and follow all the same structure. Finally, we point out Lemma 7.6 in Section 7.2.1, which can be seen as an equivalent of the Non-Crossing Lemma (7.2), since it characterizes the nesting behavior of M-sets. Hence, it is referred in almost every proof related to U-cuts or M-sets.

Writing Style. We generally emphasize terms that are going to be defined properly in this work, that are borrowed from other authors or that are somehow crucial. Such terms are emphasized at their first occurrence, also if the definition does not directly follow. We use small caps for algorithms and procedures that are given in pseudo code, expressions that describe graph-theoretic problems like, for example, the all-pairs minimum cut problem are not written in a special font. For problems that are mentioned very often or occur in different variants, we introduce abbreviations, which are then written in small caps. For the names of graph instances that appear outside of tables in the text, we use a font without serifs. We mostly avoid using citations as syntactical elements, however, citations may be denoted at the end of statements without any mention of author names or publication type. Usually we only cite the latest reference of a result, however, in few cases, we refer to claimed results that have been proven to be incorrect by a later result. For these results we tried to cite all occurrences in order to caution against all sources of erroneous information. Finally, we remark that whenever we talk about connectivity without further concretization, we mean vertex connectivity in Part I and edge connectivity in Part II and III. Moreover, we sometimes do not follow the usual notation of minimum s-t-cuts and maximum s-t-flows, but use vertices u and v or x and y, instead, in order to stress the fact that the actual direction of the cut or flow is not important.

1.2 Notations and Definitions on Graphs

In this section we present the notation used throughout this thesis and give the definitions of most terms that occur. Some concepts that are, however, only incidentally mentioned, are not described in detail. Special terms that are explicitly used in only one part of this work are defined in the introduction of the particular part or directly where they occur.

Undirected Graphs. Apart from maximum flows, which are usually considered in *directed* graphs, and special directed trees, we consider only *undirected* graphs in this work. The graphs in Part II and III are further *weighted*, while in Part I we assume *unweighted* graphs. An undirected, weighted graph is a graph $G = (V, E, c)$ with vertices V, edges E and a positive edge cost function $c : E \to \mathbb{R}_0^+$, writing $c(u, v)$ as a shorthand for $c(\{u, v\})$ with $\{u, v\} \in E$. The notations V, E, c, as well as $n := |V|$ and $m := |E|$ may be used without explicitly denoting a graph, as long as the underlying graph becomes clear from the context. If not of current interest, we omit the cost function c in the notation. The *degree* $\deg(v)$ of a vertex v describes the sum of the costs of the edges incident to v. Since unweighted graphs can be considered as graphs with uniform edge costs, usually 1, this definition also applies in the unweighted case. A vertex adjacent to a vertex v is called a *neighbor* of v. The set of all neighbors of v is denoted by $N(v)$. If necessary, we denote the underlying graph as an index at the degree or the set of neighbors.

In the literature, the terms graph and network are sometimes used interchangeably, and the edge cost is also called *edge weight* or *edge capacity*. Analogously, (edge) weighted graphs are also named *(edge) capacitated* graphs or networks. We call our graphs weighted, although we talk about edge costs, instead of weights, and denote the cost function by c, instead of w. We do not consider costs assigned to vertices in this work.

The graphs considered in this work are further *simple*, that is, they do neither contain parallel edges between the same pair of vertices nor *self-loops*, that is, edges $\{v, v\}$ that start and end at the same vertex v. In the literature, integer edge costs are sometimes replaced by parallel unweighted edges, which results in different values for m. Hence, comparing running times of algorithms that depend on m must be done carefully.

A *subdivision* of a graph is obtained by iteratively subdividing edges in the graph. Subdividing an edge $\{u, v\}$ means adding a new vertex w and replacing $\{u, v\}$ by two edges $\{u, w\}$ and $\{w, v\}$. A *(vertex-induced) subgraph* of a graph $G = (V, E, c)$ is a graph on a subset $V' \subseteq V$ that contains an edge $\{u, v\}$ with cost $c(u, v)$ if and only if $\{u, v\} \subseteq V'$ and $\{u, v\} \in E$. We call a set $V' \subsetneq V$ a *proper subset* of V, and say that two sets are *nested* if one set is a (not necessarily proper) subset of the other set. Two sets that are not disjoint are *overlapping*. We call a collection of sets *hierarchically nested* if they are pairwise nested or disjoint. Furthermore, we define the *degree of a (sub)set* analogously to the degree of a vertex. The terms proper, nested and hierarchically nested, as well as the notion of the degree are adapted analogously to subgraphs and subclasses.

We further reserve the term *node* for compound vertices of abstracted graphs, which may contain several basic vertices of a concrete graph; however, we identify singleton nodes with the contained vertex without further notice. *Contracting* a set $N \subseteq V$ in G means replacing N by a single node, and leaving this node adjacent to all former adjacencies u of vertices of N, with an edge cost equal to the sum of all former edges between N and u. Analogously, we contract a set $M \subseteq E$ or a subgraph of G by contracting the corresponding vertices.

Special Graphs. We define a *simple path* of n vertices as a sequence $v_1, \dots, v_n$ of vertices such that $v_i \neq v_j$ for $i \neq j$, $\{i, j\} \subseteq \{1, \dots, n\}$, and $\{v_i, v_{i+1}\}$ forms an edge for $i = 1, \dots, n-1$. A simple path where also $\{v_1, v_n\}$ forms an edge is called a *simple cycle*. If we further allow the vertices to appear more than once in the sequence $v_1, \dots, v_n$, we get a (general) path or (general) cycle, respectively, that is not necessarily simple anymore. It might happen that we omit the commas when denoting a path, simply writing xyz for, for example, a path of two edges or a cycle of three edges, or uv for an edge. The length of a path or a cycle is either measured by the number of edges or the number of vertices, as convenient.

A *wheel graph* of n vertices is a graph formed by connecting a designated vertex (the so-called center) to each vertex of a simple cycle formed by the $n - 1$ remaining vertices. Deleting in a wheel graph of n vertices the edges of the simple cycle formed by the $n - 1$ vertices apart from the center yields a *star* of n vertices. A *tree* is a graph that has no (general) cycle as subgraph. Note that a star is a special tree. In Part II and III we deal a lot with trees, also with directed trees. The path between two vertices in a tree is unique, and we write $\pi(u,v)$ for the path between u and v. In directed trees, we use the same notation and stress the direction of the path by words. The special graphs K_5 and $K_{3,3}$ play a fundamental role in *planarity testing* (see Part I). The graph K_5 consists of five vertices that are pairwise connected by an edge. The graph $K_{3,3}$ consists of three red and three green vertices, such that each green vertex is connected to each red vertex, while vertices of the same color are not connected. Both graphs are not planar. The complement graph $G^c = (V, E^c)$ of an (unweighted) graph $G = (V, E)$ is obtained from G by replacing the edge set E by the set of complement edges $E^c := \binom{n}{2} \setminus E$, that is, the set of all edges that are not contained in G.

Directed Graphs. Regarding the few situations where we also consider directed graphs (or digraphs), we briefly introduce some basic notations for digraphs. We distinguish directed arcs or edges from undirected edges by surrounding them by round brackets, while undirected edges are written with curly brackets. The first vertex u of a directed edge (u,v) is called the *tail*, the second vertex v is called the *head* of (u,v). The edge (u,v) is thus oriented from u to v, pointing at v. The degree of a vertex v in a digraph usually distinguishes between *in-going* and *out-going* edges, that is, edges for which v is the head or the tail. So there is an *in-degree* and an *out-degree*. The neighbors of v are split analogously into neighbors with respect to in-going and out-going edges. *Vertex-induced subgraphs of digraphs*, *directed paths* and *directed cycles* are defined analogously to vertex-induced subgraphs of undirected graphs, (general) paths and (general) cycles. The edges in a path are oriented from v_i to v_{i+1}, and the additional edge in a cycle is oriented from v_n to v_1. A digraph is called *acyclic* if it has no cycle as subgraph. A vertex that can be reached from a vertex v by a directed path is a *successor* of v, while a vertex from which v can be reached by a directed path is a *predecessor* of v. We will use these terms intensively in the context of directed trees in Part II and III.

Dynamically Changing Graphs. Dynamic or *evolving* graphs appear in Part II and III of this work. We call a graph *dynamic* if its edges or vertices vary over time. The current status of the graph at a certain time step is also called graph or *snapshot* of the dynamic graph. In this work, we consider only atomic changes in dynamic graphs, that is, the deletion of an edge or a degree-0 vertex, the insertion of an edge or a degree-0 vertex, and the decrease or increase of the cost of an edge. Note that in Part II we will distinguish between edge insertion and increasing cost as well as between edge deletion and decreasing cost, while in Part III edge insertion and deletion are considered as special cases of increasing and decreasing costs.

A change in a graph G either involves an edge $\{b,d\}$ or a vertex b. If the cost of $\{b,d\}$ in G decreases by $\Delta > 0$ or $\{b,d\}$ with $c(b,d) = \Delta > 0$ is deleted, the change yields $G^{\ominus}$. Analogously, inserting $\{b,d\}$ or increasing the cost yields $G^{\oplus}$. We denote the cost function after a change by $c^{\ominus}$ and $c^{\oplus}$, respectively. Since we assume that only degree-0 vertices are deleted from or inserted into G, vertex changes do not affect the edge set or the cost function. Hence, we simply denote the graph after such an change by $G - b$ or $G + b$, respectively. If we consider the graph after an arbitrary change, without any further information about the type of the change, we denote it by $G^{\textcircled{\scriptsize U}}$.

Depth-First-Search and Breadth-First Search. Depth-first search (DFS) and breadth-first search (BFS) are standard methods for traversing a given graph in time $O(n+m)$, visiting each vertex. The idea is to start at an arbitrary vertex and explore the graph along a path (DFS) or along the neighbors (BFS) as far as possible. If the DFS reaches a vertex that has been already visited, it tracks the path back until the first possibility to branch. The BFS continues visiting the neighbors of the neighbors until each vertex has been visited at least once. We omit a more detailed description here and refer to the textbook of Cormen et al. [27], instead.

1.3 Cuts and Connectivity

Although, at two points in this thesis, there appear directed graphs, we consider cuts and connectivity aspects only in undirected graphs. Thus, we concentrate on definitions and algorithms for undirected graphs. Furthermore, we consider unweighted graphs as graphs with cost 1 assigned to each edge. Hence, all definitions and algorithmic approaches considered in the following for weighted graphs also apply to unweighted graphs.

Cuts in Weighted Graphs. A *cut* in an undirected, weighted graph $G = (V, E, c)$ is a partition of V into two non-empty *cut sides* S and $V \setminus S$. The cost $c(S, V \setminus S)$ of a cut in G is the sum of the costs of all edges *crossing* the cut, that is, edges $\{u, v\}$ with $u \in S$, $v \in V \setminus S$. Since a cut is clearly defined by already one of its cut sides, say S, the cost can be also denoted by $c(S)$. Note that the cost $c(\{v\})$ with $v \in V$ equals the degree of v. For two disjoint sets $A, B \subseteq V$, which not necessarily form a cut, we define the cost $c(A, B)$ analogously. Whenever we are not interested in the exact cut sides of a cut, we use θ, possibly with some indices, to denote a cut. Another possibility to define a cut is to consider the *cut set*, that is, the set of edges crossing the cut. In connected graphs, that is, in graphs that contain at least one path between any vertex pair (see below), both definitions are equivalent. In disconnected graphs, however, the cuts sides of two cuts with the same cut set may differ in those connected components that are not split by the cuts. In the context of cut-based clustering in Part III of this thesis, the cut sides play an important role. Hence, we stick with the definition of cuts based on cut sides, and consider cuts with the same cut sets as equivalent (see also the definitions in Section 6.2).

We say two cuts are *nested* if their cut sides are nested. Two cuts are *non-crossing* if their cut sides are nested or at least two cut sides are disjoint. Otherwise we say that two cuts *cross*. We call a set of cuts *non-crossing* or *hierarchically nested* if all cuts are pairwise non-crossing. Two vertices $u, v \in V$ are *separated* by a cut if they lie on different cut sides. A *minimum u-v-cut* is a cut that separates u and v and is the cheapest cut among all cuts separating these vertices. The cost of a minimum u-v-cut defines the *(local) edge connectivity* of u and v, which is denoted by $\lambda(u, v)$, possibly with an index indicating the underlying graph. We call a cut a *minimum separating cut* if there exists an arbitrary vertex pair $\{u, v\}$ for which it is a minimum u-v-cut; $\{u, v\}$ is called a *cut pair* of the minimum separating cut. Analogously, we can also consider minimum separating cuts with respect to vertex sets, instead of vertices, that is, minimum S-T-cuts with $S, T \subsetneqq V$, $S \cap T = \emptyset$.

Due to the equivalence of $\lambda(u, v)$ and the *value* of a *maximum u-v-flow* in an undirected, weighted graph, as stated by Ford and Fulkerson [45] in their popular *max-flow min-cut theorem*[1], and due to the duality of finding a minimum u-v-cut and a maximum u-v-flow in linear programming [28], computing a minimum u-v-cut in an undirected, weighted graph basically means computing a maximum u-v-flow in this graph. We introduce the maximum-flow problem below and give a brief idea how fast it can be solved.

---

[1] This first version of the theorem, which is closely related to Menger's theorem, was on undirected graphs. Later, the authors generalized it to the directed case. Today there exist various variants of this statement.

Global Minimum Cuts and Global Edge Connectivity. A cheapest cut among all minimum separating cuts (that is, with respect to all vertex pairs) in an undirected, weighted graph is a *global minimum cut.* Interestingly, a global minimum cut can be computed at least as fast as a maximum flow, that is, a minimum separating cut for a single vertex pair. Hao and Orlin [74] use $2n - 2$ specially reduced maximum-flow computations for constructing a global minimum cut in the same asymptotic time, namely $O(nm\log(n^2/m))$, as a maximum flow. Nagamochi and Ibaraki [112] showed that computing a maximum flow is not necessary at all. Instead they introduced maximum adjacency orderings (MA orderings), which turned out to be a powerful tool to reduce the running time for many edge-connectivity problems in undirected graphs. Their algorithm presented for determining the edge connectivity of weighted graphs runs in time $O(nm + n^2\log n)$, and is still one of the fastest approaches to solve this problem. The algorithm of Stoer and Wagner [133] even refines the approach of Nagamochi and Ibaraki resulting in the same asymptotic running time but a simpler method. This algorithm can be further extended such that it also returns $n - 1$ flows that together separate each vertex pair, and thus, witness the found global minimum cut [6]. Constructing this certificate takes $O(nm)$ additional time. A randomized approach in $O(m\log^3 n)$ is given by Karger [90]. Due to the correspondence of cuts in the original and cycles in the dual graph, planar graphs admit even faster running times for computing a global minimum (weighted) cut. Italiano et al. [85] achieved a running time in $O(n\log n\log\log n)$, which recently could be even improved by a $\log n$ factor by Laçki and Sankowski [98].

Ignoring the edge costs, global minimum cuts establish the concept of *global (unweighted) edge connectivity* of undirected graphs. Note that, in this work, local edge connectivity refers to weighted graphs, while global edge connectivity implies an unweighted graph. In weighted graphs, we consider the cost of a global minimum cut whenever we refer to the global *weighted* edge connectivity. The global (unweighted) edge connectivity of a graph G describes how many edges cross a minimum separating cut, or in other words, how many edges need to be at least deleted in order to split the graph into two parts. According to Menger's Theorem [107], this is further equivalent to the question how many edge-disjoint paths exist at least between each vertex pair in G. According to the latter, a graph is *c-edge-connected* if it contains at least c edge-disjoint paths between each vertex pair. A 1-edge-connected graph is simply called *connected.* A graph that is not connected is called *disconnected, unconnected* or *non-connected.* A disconnected graph consists of at least two connected subgraphs that are pairwise separated by a global minimum cut of cost 0. These *connected components* can be computed in $O(n + m)$ time by a DFS or a BFS. Another definition says, a graph G is *c-edge-connected* if at least c edges need to be removed in order to make G disconnected, or the other way round, if G is connected and removing any set of at most $c - 1$ edges leaves G connected. Note that the latter excludes $c = 0$.

The global edge connectivity ignoring edge costs can be computed even faster than a global minimum (weighted) cut. Nagamochi and Ibaraki [112], presented an algorithm that runs in $O(nm)$ time, while Gabow's algorithm [51] provides a running time in $O(m + \lambda^2 n\log(n/\lambda))$, with λ the found edge connectivity. We remark that for planar graphs, as considered in Part I of this work, the edge connectivity is bounded by 5, since each planar graph has at least one vertex with at most five neighbors. Hence, for planar graphs, Gabow's algorithm already has a near linear running time. Furthermore, there exists an $O(n)$ time algorithm of Yuster and Zwick [4] for finding shortest cycles of length ≤ 5, which implies a linear time algorithm also for minimum cuts, due to the duality of cuts and cycles.

Vertex Connectivity. The *vertex connectivity* of a graph G is the analog to the edge connectivity of G. A graph G is c-vertex-connected if it is connected and removing any set of at most $c-1$ vertices leaves G connected. This equals again the existence of at least c vertex-disjoint path between any pair of vertices in G. A set of vertices that decomposes G into connected components when it is deleted, is called a *vertex cut* or a *vertex separator*. We note, that in this work, we do not consider minimum separating vertex cuts with respect to two vertices in a graph.

Henzinger et al. [82] showed that the vertex connectivity of an undirected graph can be determined in $O(\kappa^2 n^2)$ time, with κ the found connectivity value, Gabow [53] presented an algorithm with a running time in $O(n+\min\{\kappa^{5/2}, \kappa^{3/4}\}\kappa n)$, which is one of the fastest known. As edge connectivity, also vertex connectivity is bounded by 5 in planar graphs, since the neighbors of a vertex with minimum (unweighted) degree always form a vertex separator of size at most 5. Furthermore, a graph is 1-vertex-connected if and only if it is 1-edge-connected, which can be tested by a DFS/BFS. Testing 2-vertex connectivity can be also done in linear time [19, 52], and we note that these algorithms do not require planarity. Whether a planar graph is 3-vertex-connected is indicated by SPQR-trees, a data structure that, similar to Gomory-Hu trees, represents all vertex separators of size 2 in a 2-vertex-connected planar graph. An SPQR-tree can be again constructed in linear time, as claimed by Hopcroft and Tarjan [83]. However, Gutwenger and Mutzel [70] show the incorrectness of the linear time algorithm proposed by Hopcroft and Tarjan, and give a correct linear time implementation. Other techniques related to vertex connectivity are, for example, *ear decompositions* [102], and *s-t-numberings* and *bipolar orientations* [19].

1.3.1 Maximum Flows

We give just a brief impression on the classical maximum-flow problem, since, although we use a maximum-flow algorithm in order to compute minimum separating cuts, the results in this work do not depend on a concrete maximum-flow algorithm. Instead, we evaluate our algorithmic approaches (in Part II and Part III, Part I does not employ minimum separating cuts) by counting necessary cut computations. That is, we use the maximum-flow problem as a black box and do not care about the actual running time of the implemented algorithm for solving this problem. The only point where we consider maximum flows in slightly more detail is in Section 6.2.3. There we need the notion of maximum flows in undirected, weighted graphs in order to illustrate the nature of Picard-Queyranne DAGs and to substantiate our *recomputation conjecture* in the context of dynamic Gomory-Hu trees. Based on this conjecture, we discuss the worst-case running time of our update procedures for Gomory-Hu trees in Section 9.3.

The classical maximum-flow problem considers a directed, weighted network $G = (V, E, c)$ with edge costs in $\mathbb{R}_0^+$. The edge costs are usually called *capacities* in the context of flows. In Section 6.2.3 we describe how to interpret an undirected, weighted graph as such a directed *flow network*, such that applying any maximum-flow algorithm becomes possible.

For a fixed vertex pair $\{s,t\} \subseteq V$, a flow from s to t (or an s-t-flow for short) in G is a function $f : E \rightarrow \mathbb{R}_0^+$ that satisfies the *flow conservation property* and the *capacity constraint*. The flow conservation property is satisfied if the total incoming flow equals the total outgoing flow at each vertex in $V \setminus \{s,t\}$, while at the source s the total outgoing flow dominates the total incoming flow, and vice versa at the target t. More formally, this is $\Delta(v) := \sum_{x\in N_i(v)} f(x,v) - \sum_{y\in N_o(v)} f(v,y) = 0$ for all $v \in V \setminus \{s,t\}$, while $\Delta(s) \leq 0$ and $\Delta(t) \geq 0$, with $N_i(v)$ the vertices connected to v by an incoming edge (with respect to v) and $N_o(v)$ the neighbors of v

reachable by an outgoing edge of v. The capacity constraint is satisfied if $f(e) \leq c(e)$ for all $e \in E$. The *value* $\mathfrak{v}(f)$ of f is the total amount of flow that passes from s to t in G, that is, $\mathfrak{v}(f) := |\Delta(s)| = \Delta(t)$. The flow f is *maximum* if there is no other flow f' in G with $\mathfrak{v}(f') > \mathfrak{v}(f)$.

Flow Algorithms. There exists a vast number of algorithms to compute a maximum s-t-flow in a given flow network, and several techniques these algorithms rely on. The two most popular techniques are probably the *push-relabel* technique and the idea of finding *augmenting paths*. While the first method, roughly speaking, pushes as much flow as possible into the network, starting from the source, and then routes back the flow that cannot get through the first bottleneck that is reached, the second technique iteratively searches for a path from s to t where the capacities of the edges are not yet reached, such that some flow can still pass through. These techniques are used in different modifications and combined with further ideas resulting in various algorithms of different worst-case running times, which in parts are difficult to compare. This is because some are given with respect to n and m, which makes a difference for dense and sparse graphs, others only focus on n, and some depend on further parameters as, for example, the maximum edge capacity or the final maximum-flow value in G. Furthermore, some authors provide an amortized running time analysis for their approaches or claim expected running times.

Hence, we exemplarily mention only one algorithm, namely the push-relabel algorithm of Goldberg and Tarjan [57], which is one of the fastest and most popular maximum-flow algorithms with a running time in $O(nm \log(n^2/m))$. It is part of many implementation libraries like, for example, the C++ template library LEMON (see Section 1.4). Furthermore, a highly efficient implementation of this algorithm is distributed freely (at least for non-commercial use) by IG Systems (Inc.) under the name HI_PR. Further advantages of this method are the possibility to deduce a minimum separating cut already from a *preflow*, which is a preliminary state of a flow during the computation, and in some versions, the indication of the vertices on the source side of a minimum separating cut by the labeling. Depending on the input graph, computing a cut thus might be significantly faster than a full flow computation, and deciding the cut side of a given vertex becomes possible in constant time. A more detailed description on the most important maximum-flow methods, their relations to each other, some interesting historical notes and an overview on the different running times achieved so far the maximum-flow problem is given in Volume A of Schrijver's book on combinatorial optimization [129]. We further note that special graph properties admit different, often faster, algorithms. For unweighted graphs, for example, the maximum-flow problem corresponds to the problem of finding a maximum number k of edge-disjoint paths from s to t, and can thus be solved in $O(k^2 n)$ time in directed graphs [111], and in $O(m+nk^{3/2})$ time in undirected graphs [91]. Weighted, planar graphs even admit running times in $O(n \log n)$ for the directed case [37, 16], and $O(n \log \log n)$ for the undirected case [85]. For (undirected and directed) weighted graphs with bounded treewidth, Hagerup et al. [71] could even give a (sequential) linear time algorithm, resulting from a parallel algorithm with a running time in $O(\log n)$. Note that for general networks the maximum-flow problem is P-complete and therefore unlikely to be solvable by an efficient parallel algorithm.

Further Flow Problems. Besides the classical maximum-flow problem there exists a variety of further flow problem as, for example, the minimum-cost flow problem [56] where each edge is associated with an additional cost for sending flow through it. The task is then to find a cheapest flow of value at least d. Another example is the multi-commodity flow problem, where several pairs of sources and sinks are given and between each pair the flow is demanded to transport a designated amount of commodities. Finding an integer multi-commodity flow is NP-complete [39], however, for fractional flows the problem is solvable by fast approximation schemes [89]. This problem can be also considered as a minimum-cost problem or as a maximization problem. A further variant is the maximum concurrent flow problem, where the task is to maximize the minimal fraction of the flow of each commodity to its demand. Besides edge capacities and edge costs also vertex capacities and vertex costs are considered in the context of flow problems. Some of these problem variants can be again reduced to flow problems of only edge constraints. Flow problems with also a lower bound on the flow of each edge arise for example from transportation systems [81]. See also [2] for a survey on Network Flows. Finally we also point out the multiterminal network flow problem, which sounds like another variant of the classical maximum-flow problem, however, this problem, originally proposed by Mayeda [106], is not about finding a function $f : E \to \mathbb{R}_0^+$ on edges. Instead, it asks for the local connectivity $\lambda(u, v)$ of each vertex pair $\{u, v\}$ in an undirected, weighted graph G, and is thus closely related to Gomory-Hu trees as we will see in Section 6.

1.4 Graph Instances

This work presents several small and two more detailed experimental studies on dynamic Gomory-Hu trees and hierarchical cut clusterings. Each of these studies uses a collection of the graph instances introduced in following. The exact references to the sources of the data is only given in this section, and we refer to this section in the description of the experiments. We distinguish three categories of instances depending on their origin. All instances are considered as undirected graphs even if the underlying data provides information about edge directions. The first category describes a bunch of various static instances collected from different sources in the context of the 10th DIMACS Implementation Challenge on Graph Partitioning and Graph Clustering [9]. We have chosen these instances, since they can be considered as benchmark instances for graph clustering, and thus, fit our purposes very well. The second category subsumes instances obtained from email data provided by the Department of Informatics at KIT[2], where each entry represents an email between two members of the department, provided with an exact time stamp. From these data we construct static and dynamic graph instances as described below. The third category finally considers two further instances, one we obtained from a further source and one we gathered ourselves.

For our implementations, we used the template library LEMON (version 1.2.1), which is an open source graph library written in the C++ providing implementations of common data structures and algorithms with focus on combinatorial optimization tasks connected mainly with graphs and networks. The library is part of the COIN-OR project (http://www.coin-or.org). More details can be found on the webpage http://lemon.cs.elte.hu/trac/lemon/.

1.4.1 Instances of the 10th DIMACS Implementation Challenge

We use only a small extract of the graph instances provided for the implementation challenge. The complete collection can be obtained from the download section of the challenge webpage (http://www.cc.gatech.edu/dimacs10/index.shtml). The DIMACS instances are again organized

---

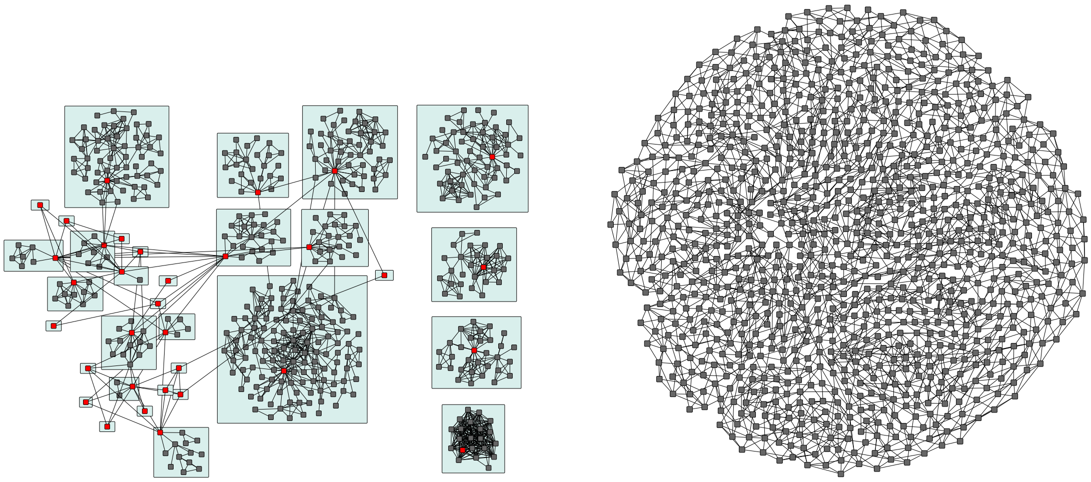
[2] Dynamic network of email communication at the Department of Informatics at Karlsruhe Institute of Technology (KIT). Data collected, compiled and provided by Robert Görke and Martin Holzer of ITI Wagner and by Olaf Hopp, Johannes Theuerkorn and Klaus Scheibenberger of ATIS, all at KIT. 2011. i11www.iti.kit.edu/projects/spp1307/emaildata

in several categories. We list the used instances in tables that show the graph sizes and the numbers of connected components. Instances marked by (*) provide edge costs different from 1, instances marked by (°) are generated instances. The graphs used from the category *Clustering Instances*, which provides real-world instances and some generated graphs that are often used as benchmarks in the graph-clustering communities, are listed in Table 1.1. The following list further gives brief explanations about the data background, as provided at the challenge webpage. The challenge webpage further specifies the original sources of the data. We point out the instances netscience and delaunay_n10, delaunay_n11, and delaunay_n12, which show a very contrary behavior in our experiments due to their different structures. While netscience shows clearly indicated clusters, the Delaunay triangulations have a very regular structure. Figure 1.5 shows netscience and delaunay_n10 as an example of the Delaunay graphs. The instance lesmis is further depicted by Fig. 12.6 and the network karate is given by Fig. 11.4.

Table 1.1: Real-world instances and randomly generated graphs from the DIMACS category *Clustering Instances.*

graph	n	m	comp	graph	n	m	comp
karate	34	78	1	dolphins	62	159	1
lesmis*	77	254	1	polbooks	105	441	1
adjnoun	112	425	1	football	115	613	1
jazz	198	2742	1	celegansneural*	297	2148	1
celegans_metabolic	453	2025	1	email	1133	5451	1
polblogs	1490	16715	268	netscience	1589	2742	396
data	2851	15093	1	power	4941	6594	1
hep-th	8361	15751	1332	PGPgiantcompo	10680	24316	1
astro-ph	16706	121251	1029	cond-mat	16726	47594	1188
as-22july06	22963	48436	1	cond-mat-2003	31163	120029	1599
cond-mat-2005	40421	175691	1798	G_n_pin_pout°	100000	501198	6

- jazz – jazz musicians network
- celegans_metabolic – metabolic network of Caenorhabditis elegans (roundworm)
- email – network of email interchanges between members of the Univeristy Rovira i Virgili (Tarragona)
- PGPgiantcompo – giant component of the network of users of the Pretty-Good-Privacy algorithm for secure information interchange
- adjnoun – adjacency network of common adjectives and nouns in the novel David Copperfield by Charles Dickens
- as-22july06 – a symmetrized snapshot of the structure of the Internet at the level of autonomous systems, reconstructed from BGP tables posted by the University of Oregon Route Views Project
- astro-ph – network of coauthorships between scientists posting preprints on the Astrophysics E-Print Archive between January 1, 1995 and December 31, 1999
- celegansneural – weighted network representing the neural network of Caenorhabditis elegans (roundworm)
- cond-mat – network of coauthorships between scientists posting preprints on the Condensed Matter E-Print Archive between January 1, 1995 and December 31, 1999
- cond-mat-2003 – updated network of coauthorships between scientists posting preprints on the Condensed Matter E-Print Archive. This version includes all preprints posted between



(a) Large components of netscience. (b) Instance delaunay_n10.

FIGURE 1.5: Cut clustering with best modularity value among all cut clusterings in the complete hierarchy (see Section 11.2) of the network netscience (red vertices correspond to representatives), and the network delaunay_n10. The complete cut-clustering hierarchy for the latter consists of only the two trivial clusterings.

January 1, 1995 and June 30, 2003. The largest component of this network, which contains 27519 scientists, has been used by several authors as a test-bed for community-finding algorithms for large networks

- cond-mat-2005 – updated network of coauthorships between scientists posting preprints on the Condensed Matter E-Print Archive. This version includes all preprints posted between January 1, 1995 and March 31, 2005
- dolphins – social network of frequent associations between 62 dolphins in a community living off Doubtful Sound, New Zealand
- football – network of American football games between Division IA colleges during regular season Fall 2000
- hep-th – network of coauthorships between scientists posting preprints on the High-Energy Theory E-Print Archive between January 1, 1995 and December 31, 1999
- karate – social network of friendships between 34 members of a karate club at a US university in the 1970s
- lesmis – coappearance network of characters in the novel Les Miserables
- netscience – coauthorship network of scientists working on network theory and experiment
- polblogs – network of hyperlinks between weblogs on US politics, recorded in 2005
- polbooks – network of books about US politics published around the time of the 2004 presidential election and sold by the online bookseller Amazon.com, edges between books represent frequent copurchasing of books by the same buyers.
- power – network representing the topology of the Western States Power Grid of the US
- G_n_pin_pout – graph generated using a two-level Gnp random-graph generator

A few graphs are taken from further categories. The *Delaunay graphs* delaunay_n10, delaunay_n11 and delaunay_n12 have been generated as Delaunay triangulations of random points in the unit square. The graph rgg_n_2_15_s0 is a *random geometric graph* with 2^{15} vertices. Each vertex is a random point in the unit square and edges connect vertices whose Euclidean distance is below $0.55 \ln n/n$. This threshold was chosen in order to ensure that the graph is almost connected.

Chris *Walshaw's graph partitioning archive* contains 34 graphs that have been very popular as benchmarks for graph partitioning algorithms. We have taken the instance data from there. More details are provided in Table 1.2.

TABLE 1.2: Real-world and randomly generated instances from other DIMACS categories.

graph	n	m	comp	graph	n	m	comp
delaunay_n10°	1024	3056	1	delaunay_n11°	2048	6127	1
data	2851	15093	1	delaunay_n12°	4096	12264	1
rgg_n_2_15_s0°	32768	160240	6				

1.4.2 Instances Obtained from Email Data

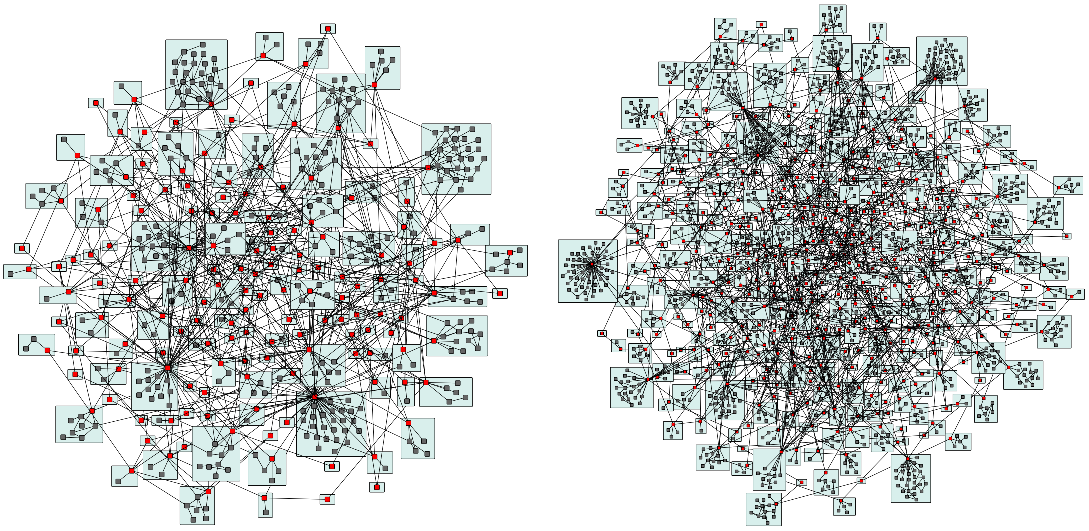
For our experiments on dynamic graph algorithms, we constructed evolving instances from these data as follows. The members of the department correspond to vertices and the edges result from email contacts between the individuals. Starting with the first recorded email, each new email and each email that becomes older than 72 hours then indicates a new time step. The edges in a snapshot at a particular time step are weighted by the number of emails sent between two corresponding individuals during the last 72 hours. Singletons are deleted as soon as the last incident email contact to a neighbor times out. This also results in another time step. Analogously, if an email is sent between from or to an individual that is not yet represented by a vertex, a new vertex is inserted, again resulting in a new time step. In this way we receive a series of snapshots, one for each time step, where consecutive snapshots can be obtained from each other by exactly one atomic change, namely the deletion or insertion of a vertex or the decrease or increase (including the deletion and insertion) of the cost of an edge. An exemplary snapshot of a dynamic graph obtained in this way is shown in Fig. 1.4. In Section 11.2.2 we consider two further snapshots, emailgraph550K_19 and emailgraph550K_26, also as static graphs. Figure 1.6(a) exemplarily shows the instance emailgraph550K_19 clustered with the complete hierarchical cut-clustering approach introduced in Section 11.2.

1.4.3 Further Instances

Additionally to the instances listed above, we also use the protein interaction network bo_cluster published by Jeong et al. [86] and a snapshot of the linked wiki pages at dokuwiki_org gathered ourselves. The sizes and numbers of components are given in Table 1.3. Figure 1.6(b) exemplarily shows the instance bo_cluster clustered with the complete hierarchical cut-clustering approach introduced in Section 11.2.

TABLE 1.3: Testbed of further real-world networks.

graph	n	m	comp	graph	n	m	comp
bo_cluster	2114	2203	417	dokuwiki_org	4416	12914	110



(a) Giant component of emailgraph550K_19.

(b) Giant component of bo_cluster.

FIGURE 1.6: Cut clusterings with best modularity values among all cut clusterings in the complete hierarchies (see Section 11.2) of the networks emailgraph550K_19 and bo_cluster.

1.5 A Brief Glance at Complexity Theory

This section is meant to give a short review on the terms and concepts related to computational complexity theory that are used in this thesis. It is neither comprehensive nor suitable to learn about complexity theory from scratch. For further explanations we recommend the textbook of Sanjeev Arora and Boaz Barak [7]. Since we exclusively consider graph-theoretic problems in this work, we restrict the following descriptions and explanations to graph instances.

Graph-Theoretic Problems. A graph-theoretic decision problem considers a graph instance and usually asks for the existence of an object somehow related to the given graph. Such an object can be for example a cut of certain cost, a special subgraph, or a clustering with special properties. Some problems also asks for a graph property, instead of the existence of an object. However, in most cases the property is characterized by the existence or nonexistence of an object, such that the problem can be reformulated accordingly. Whether a graph is, for example, planar, is a matter of the existence of a planar embedding or the nonexistence of a subgraph that is a subdivision of K_5 or $K_{3,3}$. Whether an undirected, unweighted graph is c-edge-connected is determined by the existence or nonexistence of a cut with cost at least c. The input size of a graph instance is usually measured by the number of nodes or the number of edges or both, where the number of edges is at most the squared number of nodes. If not denoted otherwise, the running time of an algorithm is described by the number of instructions that are performed to solve a problem. However, in most cases, one is not interested in the exact running time, that is, the number of instructions needed to solve a problem for a concrete instance, but in the behavior of the running time with respect to increasing input sizes of the instances. The latter yields a function in the input size which can be simplified by the help of the *big O notation.* The big O is the only *Landau symbol* we will use in this thesis. A function f is in the class $O(g)$ if, asymptotically, it does not grow faster than g multiplied by a constant factor. That is, we can use a simple function for g that ignores constant factors and addends in f that are dominated by faster growing components, and thus, describe the asymptotical behavior of f by the simple notation of $O(g)$.

Polynomial-Time Solvability of Graph-Theoretic Problems. A problem is *solvable in polynomial time* if there exists a *polynomial-time algorithm* that solves it. An polynomial-time algorithm is an algorithm with an asymptotic running time bounded by a polynomial g in the input size, that is, the running time is in $O(g)$. The class of all polynomial-time solvable decision problems (including also the problems that are not graph-theoretic) is denoted by P. Most problems considered in this work are in the class P. For some problems considered in Part I, however, we cannot give a polynomial-time algorithm, but prove that they are in the class NP, which contains the class P.

The Class NP. Roughly speaking, the class NP contains all decision problems for which the *yes-instances* (that is, the instance with positive answer) have a polynomial-time checkable *certificate* for the correctness of a given answer. In the context of graph-theoretic problems, such a certificate is usually already given by the object the problem relies on, that is, an object that witnesses by its existence that a given instance is a yes-instance is a certificate. The certificate is *polynomial-time checkable* if there exists a polynomial-time algorithm that decides whether or not a given object of the structure of the certificate witnesses that the given instance is a yes-instance, where the polynomial running time of the algorithm is supposed to be polynomial in the input size of the instance (not the certificate). This implies that also the size of the certificate is polynomial in the size of the instance. The abbreviation NP stands for *non-deterministically polynomial-time.* The idea is that an object of the structure of a certificate can be chosen randomly, and thus, guessing well yields a polynomial-time decision for yes-instances, and implicitly, since guessing well is a priori not possible for no-instances, also to a polynomial-time decision for no-instances, which in total leads to a polynomial-time algorithm solving the problem.

To show that a graph-theoretic decision problem, where the yes-instances are characterized by the existence of an object, is in NP, it thus suffices to argue that deciding whether or not a randomly chosen potential certificate witnesses a yes-instance is possible in polynomial time.

Polynomial-Time Reduction. The idea of a polynomial-time reduction of one decision problem to another is the following. If one problem is polynomial-time solvable then the other problem can be also solved in polynomial time by first reducing it (in polynomial time) to the former problem and then solving this problem. In practice this means, problem A is *reducible* to problem B if each instance of problem A can be transformed, in polynomial time, into an instance of B such that the instance of A is a yes-instance if and only if the instance of B is a yes-instance.

Interestingly, the *satisfiability problem* (which is no graph-theoretic problem) is a problem in NP for which Cook [26] had proven that each further problem in NP can be reduced to it in polynomial time. That is, if we could find a polynomial-time algorithm to solve the satisfiability problem, we would have proven that the classes P and NP are identical. The question whether P=NP is one of the most popular open problems in the field of theoretical informatics. A problem to which all problems in NP can be reduced in polynomial time is called *NP-hard.* If the problem is also in NP, it is called *NP-complete.* Meanwhile, there are many NP-hard and NP-complete problems known, besides the satisfiability problem. Due to the nice concept of reduction, proving the NP-hardness of a problem can be done by simply proving that the satisfiability problem or any other NP-hard problem can be reduced to the given problem. In Part I of this thesis, we conduct some NP-hardness proofs. In Part III we further mention some NP-hard optimization problems in the context of graph clustering. This actually means that the corresponding decision problem is NP-hard. An optimization problem can be transformed into a decision problem by the use of a lower bound for the objective function. So the problem does no longer ask for an optimal solution, but for the existence of a solution at least as good as the given bound.

We finally remark that, although the minimization of an edge cut with respect to its cost given by the sum of all crossing edges is in P, there also exist many NP-hard cut-based minimization problems as, for example, the *sparsest cut problem* (which will come across in Part III as the problem of determining the *expansion* of a graph) or the *balanced separator problem*. Also the *maximum-cut problem* is NP-hard [93]. Since up to now there is no polynomial-time algorithm known for any NP-hard problem, people apply different concepts like *fixed-parameter tractability* or *approximation algorithms*. Madry [105] recently developed a general framework that admits poly-logarithmic approximation algorithms for NP-hard cut-based minimization problems.

Part I

Connectivity Augmentation

CHAPTER 2

Introduction – A Bunch of Problem Variants

An *augmentation* of an (undirected) graph $G = (V, E)$ is a set $W \subseteq E^c$ of edges of the complement graph. The *augmented graph* $G' = (V, E \cup W)$ is denoted by $G + W$. We study several problems where the task is to augment a given planar graph to be k-regular (for $k = 1, \dots, 5$), while preserving planarity. The problem of augmenting a graph with the goal that the resulting graph has some additional properties is a well-studied problem and has several applications, for example, in network planning [38], where one seeks to increase the robustness at small cost. Hence, a typical goal is to increase the connectivity of the graph while adding few edges. Frederickson and Ja'Ja' [49] study the problem of making a graph biconnected by adding few edges. They show that the problem of biconnecting a graph at minimum cost is NP-hard, even if all edge costs are in $\{1, 2\}$. Watanabe and Nakamura [138] give an $O(c \min\{c, n\} n^4 (cn + m))$ algorithm for minimizing the number of edges to make a given graph with n vertices and m edges c-edge-connected. Frank [48] and Nagamochi and Ibaraki [113] provide surveys on connectivity augmentation of undirected and directed, unweighted and weighted graphs with respect to edge and vertex connectivity. Motivated by graph drawing algorithms that require biconnected input graphs, Kant and Bodlaender [88] initiated the study of augmenting planar graphs to increase connectivity while preserving planarity. They show that minimizing the number of edges for the biconnected case is NP-hard and give efficient 2-approximation algorithms for both, 2-edge and 2-vertex connectivity augmentation. Rutter and Wolff [122] give a corresponding NP-hardness result for planar 2-edge connectivity. Moreover they study the complexity of geometric augmentation problems, where the input graph is a plane geometric graph and additional edges have to be drawn as straight-line segments. Abellanas et al. [1], Tóth [135] and Al-Jubeh et al. [3] give upper bounds on the number of edges required to make a plane straight-line graph c-connected for $c = 2, 3$. For a survey on plane geometric graph augmentation see [84].

We study the problem of augmenting a graph to be k-regular for all possible k while preserving planarity. In doing so, we additionally seek to raise the connectivity as much as possible. Since the minimum degree of a planar graph does not exceed 5 and the connectivity does not exceed the minimum degree in a graph, it is $0 \leq c \leq k \leq 5$. Specifically, we study the following problems.

Problem: Planar k-Regular Augmentation (k-PRA)
Instance: Planar graph $G = (V, E)$
Task: Find an augmentation W such that $G + W$ is k-regular and planar.

Table 2.1: Overview of problem variants and results.

$k=1$	k = 2	k = 3		k = 4	k = 5	
Pra/Fepra	Pra/Fepra	Pra	Fepra	Pra/Fepra	Pra/Fepra	c
$O(n)$ Thm. 2.1	$O(n)$ Thm. 2.2	NPC Thm. 3.1	$O(n^{1.5})$ Thms. 4.16, 4.17,4.18	NPC Thm. 5.1	NPC Thm. 5.3	0
						1
						2
			NPC Thm. 4.19			3
						4
						5

Problem: Fixed-Embedding Planar k-Regular Augmentation (k-Fepra)
Instance: Planar graph $G = (V, E)$ with a fixed (topological) planar embedding
Task: Find an augmentation W such that $G + W$ is k-regular, planar, and W can be added in a planar way to the fixed embedding of G.

Moreover, we study *c-connected* k-Pra and *c-connected* k-Fepra, for $c = 1, \dots, 5$, where the goal is to find a solution to k-Pra and k-Fepra such that the augmented graph is c-connected. For simplicity, the basic problems k-Pra and k-Fepra (without further connectivity constraints) are considered as c-connected problem variants with $c = 0$. Table 2.1 gives an overview of the problem family considered in this work and our results. In particular, we give for each variant either an efficient algorithm or an NP-hardness proof.

While it is not difficult to see that c-connected k-Pra/k-Fepra can be solved in polynomial time for $k = 1, 2$, we show that for $k = 4, 5$ all problem variants are NP-complete. For $k = 3$, however, considering a variable embedding or a fixed embedding as well as requiring different connectivity properties makes a difference. Using a modified version of an NP-hardness reduction by Rutter and Wolff [122], we show that c-connected 3-Pra is NP-complete for $c = 0, \dots, 3$, even if the input graph is biconnected. Note that a triconnected graph of maximum degree 3 must be 3-regular. We further prove that 3-connected 3-Fepra is again NP-hard. Our main results are efficient algorithms for c-connected 3-Fepra for $c = 0, 1, 2$.

We note that Pilz [119] has simultaneously and independently studied the planar 3-regular augmentation problem. He showed that it is NP-hard and posed the question on the complexity if the embedding is fixed. Our hardness proof strengthens his result (to biconnected input graphs) and our algorithmic results answer his open question.

2.1 Contribution and Outline

Our results on planar 3-regular augmentation are jointly published with Jonathan Rollin and Ignaz Rutter in [77]. The comprehensive results presented in this thesis have been recently accepted under minor revision at Algorithmica [78].

Chapter 2: Introduction – A Bunch of Problem Variants

We introduce basic notions used throughout this part and briefly cover the results on c-connected 1- and 2-Pra/Fepra at the end of this chapter.

Chapter 3: The Nature of Planar 3-Regular Augmentation

We prove the NP-completeness of c-connected 3-Pra in Section 3.1 and give a simple $O(n^{2.5})$-time decision algorithm for 3-Fepra in Section 3.2. Unfortunately, this simple algorithm is not suitable for incorporating additional constraints, such as increasing the connectivity. Therefore, we study 3-Fepra more systematically in Section 3.3. The problem 3-Fepra is equivalent to finding a *node assignment* that assigns the vertices of degree less than 3 to the faces of the graph such that, for each face f, an augmentation exists that can be embedded inside f in a planar way and raises the degrees of all its assigned vertices to 3. We completely characterize these assignments and show that their existence can be tested efficiently.

As an intermediate result in Section 3.3, we further show that the non-planar 3-regular augmentation problem can be decided in $O(n)$ time. This problem is equivalent to finding a degree-constrained subgraph in the complement of a graph where each vertex v has degree $3 - \deg(v)$ in the subgraph, with $\deg(v)$ the degree of v in the input graph.

Chapter 4: Algorithms for C-connected Planar 3-Regular Augmentation with Fixed Embedding

In Section 4.1 we strengthen our characterizations of the node assignments to the case where the graph should become c-connected for $c = 1, 2$ and show that our algorithm can be extended to incorporate these constraints. Up to this point all algorithms have a running time of $O(n^{2.5})$. Based on the characterizations given in Section 3.3 and Section 4.1, we improve the running time for c-connected 3-Fepra ($c = 0, 1, 2$) to $O(n^{1.5})$ in Section 4.2. In Section 4.3 we provide the hardness proof for 3-connected 3-Pra.

Chapter 5: Complexity of Planar 4- and 5-Regular Augmentation

The NP-hardness of c-connected k-Pra/k-Fepra for $k = 4, 5$ is finally proven in Chapter 5. We conclude and pose open questions at the end.

2.2 Preliminaries

In Part I of this work, we consider *planar k-regular* graphs. A graph is planar if it admits a *planar embedding* into the Euclidean plane, where each vertex (edge) is mapped to a distinct point (Jordan curve between its endpoints) such that curves representing distinct edges do not cross. It is further k-regular if each vertex is adjacent to exactly k neighbors. Since planarity and k-regularity do not care about multiple edges, loops, directed or weighted edges, we may assume simple, unweighted, undirected graphs in this part. We call an augmentation W *k-regular* if $G + W$ is k-regular, and W is called *planar* if $G + W$ is planar. A planar embedding of a graph subdivides the Euclidean plane into *faces*. We denote the set of all faces of a planar graph by $\mathcal{F}$. When we seek a planar augmentation preserving a fixed embedding, we require that the additional edges can be embedded into these faces in a planar way. Whether a given graph is planar can be tested in linear time using, for example, an algorithm by Boyer and Myrvold [18], which either returns a *Kuratowski subdivision* of either K_5 or $K_{3,3}$ disproving the planarity of the input graph, or a planar embedding confirming planarity.

We further introduce *generalized perfect matchings*, which will be our main algorithmic tool in this part. Given a graph $H = (V, E)$ and a mapping $d \colon V \to \mathbb{N}$ specifying a demand for each vertex, a *generalized perfect matching* is a graph $H' = (V, E')$ with $E' \subseteq E$ such that $\deg_{H'}(v) = d(v)$ for all $v \in V$, with $\deg_{H'}(v)$ the degree of vertex v in H'. By a result of Gabow [50], where this is referred to as a degree-constrained subgraph, the existence of a generalized perfect matching can be decided in time $O(\sqrt{\sum_{v \in V} d(v)}|E|)$.

For a vertex set V, we denote by $V^{\textcircled{i}} \subseteq V$ the set of vertices of degree i. In particular, for the 3-regular augmentation problems, we will frequently refer to the set of vertices with degree less than 3. For ease of use, we denote this set by $V^{\textcircled{\star}} = V^{\textcircled{0}} \cup V^{\textcircled{1}} \cup V^{\textcircled{2}}$. Clearly, an k-regular augmentation W must contain $k-i$ edges incident to a vertex in $V^{\textcircled{i}}$. We say that a vertex $v \in V^{\textcircled{i}}$ has $k-i$ *(free) valencies* and that an edge of an augmentation incident to v *satisfies* a valency of v. Two valencies are adjacent if their vertices are adjacent.

An augmentation W with respect to a fixed embedding of G is *valid*, if the endpoints of each edge in W share a common face in G. We assume that a valid augmentation is associated with a (not necessarily planar) embedding of its edges into the faces of G such that each edge is embedded into a face shared by its endpoint. Without loss of generality, we further assume that this embedding is such that any two edges cross at most once and edges that share a common endpoint do not cross. A valid augmentation is *planar* if the edges can be further embedded in a planar way into the faces of G. A valid k-regular augmentation can be efficiently found by computing a generalized matching in the subgraph of G^c that contains edges only between vertices that share a common face of G. In order to construct a valid k-regular augmentation for a planar graph G with a fixed planar embedding, we choose $k-i$ for the demand of a vertex in $V^{\textcircled{i}}$, for $i = 0, ..., k$. Using Gabow's algorithm such a matching can then be computed in $O(\sqrt{|V|}|E|) = O(n^{2.5})$ time.

Recall that a graph G is *connected* if it contains a path between any pair of vertices, and it is *c-(edge)-connected* if it is connected and removing any set of at most $c-1$ vertices (edges) leaves G connected. A 2-connected graph is also called *biconnected* and a 3-connected graph is also called *triconnected*. We note that, generally, c-connectivity implies c-edge connectivity, and the notions of c-connectivity and c-edge connectivity coincide on graphs of maximum degree 3. In particular, a graph of maximum degree 3 is biconnected if and only if it is connected and does not contain a *bridge*, that is, an edge whose removal disconnects the graph.

Solutions for C-connected Planar 1- and 2-Regular Augmentation. We briefly cover the results on c-connected k-PRA/FEPRA for $k = 1, 2$. Observe that a given input graph (of maximum degree 1) can be augmented to be 1-regular, that is, the graph becomes a perfect matching, if and only if it has an even number of vertices. A connected 1-regular augmentation exists if and only if the graph has exactly two vertices. Note that, since the graph cannot contain cycles, it cannot become non-planar. That is, it makes no difference whether or not we assume a fixed embedding, and we obtain the following theorem.

Theorem 2.1. *For $c = 0, 1$ c-connected* 1-PRA *and* 1-FEPRA *can be solved in $O(n)$ time.*

For the 2-regular case, we state an analogous theorem.

Theorem 2.2. *For $c = 0, 1, 2$ c-connected* 2-PRA *and* 2-FEPRA *can be solved in $O(n)$ time.*

To see the statement for the 2-regular case, first consider the variant without a fixed embedding. We want to decide whether there is an augmentation that makes the input graph (of maximum degree 2) consist of a set of simple cycles. We disregard the already existing cycles and remove them from the graph as they are not relevant. The remaining degree-0 and degree-1 vertices, if any, can be augmented to form a (single) cycle if and only if there are at least three of them. The augmentation becomes connected (and as a cycle also 2-connected) if the input graph is either already a single cycle or if it does not contain a cycle.

Now assume that the input graph additionally has a fixed embedding. Note that this makes a difference only if the input graph contains at least one cycle. But then a connected/biconnected augmentation is impossible unless the graph consists of a single cycle. For the case where we do not care about connectivity, it is necessary and sufficient that, for each face f of G, the graph induced by the degree-0 and degree-1 vertices incident to f can be augmented to form a (single) cycle (see the variant without a fixed embedding). All cases can easily be decided in $O(n)$ time.

CHAPTER 3

The Nature of Planar 3-Regular Augmentation

In this chapter, we give a first assessment of the complexity of planar 3-regular augmentation problems. More specifically, we show that 3-Pra is NP-complete, even if the input graph is already biconnected. On the other hand, we give a first simple polynomial-time algorithm for 3-Fepra. Afterwards, we study the nature of 3-Fepra in more detail. The insights we gain from this study will help to design efficient algorithms solving 3-Fepra even with additional connectivity constraints in Chapter 4. We start with the variable embedding case.

3.1 NP-Completeness of 3-Pra

We prove the following theorem.

Theorem 3.1. *The problem c-connected* 3-Pra *is NP-complete for* $c = 0, 1, 2, 3$, *even if the input graph is biconnected.*

Proof. Obviously, 3-Pra is in NP for any value of c since, given a planar graph G, we can guess a set $W \subseteq \binom{V}{2}$ of non-edges of G and then test efficiently whether $G + W$ is 3-regular, planar, and has the required connectivity. We prove NP-hardness by reducing from the planar 3-satisfiability problem (Planar3Sat), which is known to be NP-hard [99]. The reduction is inspired by and indeed very similar to a reduction of Rutter and Wolff [122], showing that it is NP-hard to find a smallest edge set that augments a given graph to be 2-edge connected and planar. An instance of Planar3Sat is a 3Sat formula φ whose variable–clause graph is planar. Such a graph can be laid out (in polynomial time) such that the variables correspond to pairwise axis-aligned rectangles on the x-axis and clauses correspond to non-crossing three-legged "combs" above or below the x-axis [95]; see Fig. 3.1.

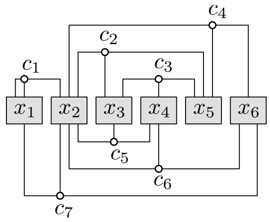


Figure 3.1: Layout of a planar 3Sat formula.

We now construct a biconnected planar graph G_φ that admits a planar 3-regular augmentation if and only if φ has a satisfying truth assignment. We further show that any 3-regular planar augmentation of G_φ indeed makes G_φ 3-connected. Therefore, the following construction proves NP-hardness for all possible connectivity constraints.

The graph G_φ consists of *gadgets*, which are subgraphs that represent the variables, literals, and clauses of φ; see Fig. 3.2. For each gadget, we will argue that there are only a few ways to embed and augment it to be 3-regular and planar. In the figure, the gadgets consist of the (black

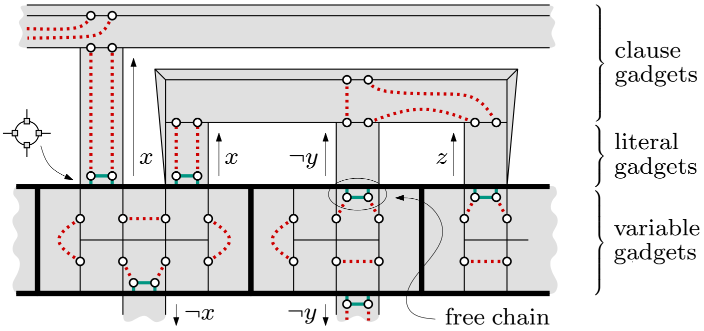


FIGURE 3.2: Part of the graph G_φ for a SAT formula φ that contains the clause $(x \vee \neg y \vee z)$. The augmentation (dotted edges) corresponds to the assignment $x = y =$ `false` and $z =$ `true`.

and green) solid line segments; the (red) dotted line segments represent non-edges of G_φ that are candidates for an augmentation of G_φ. All bends and junctions of line segments represent vertices of degree at least 3. Vertices of degree greater than 3 are actually modeled by small cycles of vertices of degree 3, as indicated in the left of Fig. 3.2. Vertices with degree less than 3 are highlighted by empty disks. The set of black solid edges forms a subgraph of G_φ that we call the *frame*. The green solid edges form *free chains*, which connect two degree-2 vertices to the frame. The thick black edges of the frame bound the variable gadgets. Variable gadgets that correspond to neighboring variables in the layout of the variable–clause graph of φ share a common boundary. Hence G_φ is always connected. Additionally, we identify the left boundary of the leftmost variable gadget with the right boundary of the rightmost variable gadget.

Consider the graph G'_φ that is obtained from the frame by contracting all vertices of degree 2 and all cycles that are used to model vertices of degree greater than 3. The graph G'_φ coincides with a graph for which Rutter and Wolff show in their reduction that it is 3-connected [122], and thus, that it has a unique planar embedding [140]. Conversely, the frame of G_φ can be obtained from G'_φ by subdividing edges and replacing vertices of degree at least 4 by cycles, which obviously preserves the uniqueness of the planar embedding. In other words, the embedding of G_φ is fixed up to the embedding of the free chains, which may be embedded in two distinct faces, each. Furthermore, the frame is biconnected, since subdividing edges and replacing vertices of degree at least 4 by cycles in the 3-connected graph G'_φ preserves at least biconnectivity. Since G_φ is obtained from the frame by adding free chains, which are paths between existing endpoints, G_φ is also biconnected.

A planar 3-regular augmentation W of G_φ finally fixes the embedding of the free chains, and thus, the embedding of the whole graph G_φ. Furthermore, W induces an assignment of the degree-2 vertices of G_φ to incident faces by considering a vertex v as assigned to a face f if, in the planar embedding of $G_\varphi + W$, the edge of W incident to v is embedded in the (former) face f. The assignment induced by W has the following properties.

(P1) Each face is assigned an even number of vertices.

(P2) Each face that is assigned two adjacent vertices is assigned at least four vertices.

We call an assignment with these properties *valid*. Conversely, it is readily seen that given a valid assignment for G_φ, a planar 3-regular augmentation can always be constructed. We thus need to show that G_φ admits an embedding with a valid assignment, if and only if φ is satisfiable.

A variable gadget consists of two rows of square faces where the horizontal edge between the two leftmost faces and the horizontal edge between the two rightmost faces is missing. Effectively, the inner faces of a variable box form a cycle. Starting from the leftmost (rectangular) face, we call the faces *odd* and *even*. The number of even and odd faces per variable gadget depends on the number of clauses that contain the corresponding literals. Each interior vertical edge is subdivided by a degree-2 vertex. Due to property (P1), a valid assignment assigns these subdivision vertices either all to the odd faces or all to the even faces of the variable. Regarding the satisfiability of φ, the interpretation will be such that a variable is `true` if all its subdivision vertices are assigned to even faces and `false` otherwise.

A literal gadget consists of a square face that lies immediately above or below the variable gadget. A positive literal (such as the one labeled with x in Fig. 3.2) is attached to an even face, a negated literal (such as the one labeled with $\neg y$ in Fig. 3.2) is attached to an odd face. Each literal gadget contains two adjacent subdivision vertices at the edge it shares with the clause gadget, and one free chain. The latter is attached to the boundary shared by the literal gadget with the variable gadget. Due to property (P2), a valid assignment either fixes the embedding of the free chain to the face of the literal gadget and assigns the subdivision vertices to the same face, or it embeds the chain in the face of the attached variable gadget and assigns the subdivision vertices to the face of the attached clause gadget.

Finally, each clause gadget consists of a single rectangular face that contains two adjacent subdivision vertices and a valid assignment assigns at least two other degree-2 vertices (besides the subdivision vertices of the clause gadget) to the face of the clause gadget, due to property (P2).

If the subdivision vertices of a variable are assigned to the even faces, that is, the variable is `true`, none of the free chains of the negated literal gadgets, which are attached to the odd faces, can be embedded into the faces of the variable gadget. Hence, they must be all embedded into the faces of the literal gadgets, which implies that the negated literal gadgets all behave the same. The case where the variable is `false`, that is, where the subdivision vertices of the variable gadget are assigned to the odd faces, is symmetric. Consequently, the subdivision vertices of the literal gadgets whose free chains are all embedded into the faces of the literal gadgets, are not assigned to the face of the attached clause gadget, and thus, do not provide any of the additional valencies that are required at the face of the clause gadget by a valid assignment.

However, if G_φ admits an embedding with a valid assignment, then, for each clause gadget, the subdivision vertices of at least one literal gadget are assigned to the face of the clause gadget and the free chain of this literal gadget is embedded into the face of the attached variable gadget. Hence, the variable is `true` if the literal gadget is positive and `false` if the literal gadget is negated. In other words, if G_φ admits an embedding with a valid assignment, each clause gadget is attached to at least one positive literal gadget whose free chain is embedded into a face of a variable gadget that is assigned `true` or to at least one negated literal gadget whose free chain is embedded into a face of a variable gadget that is assigned `false`. This eventually induces a satisfying truth assignment for φ.

Conversely, it is easy to see that, if φ has a satisfying truth assignment, a corresponding assignment can be found. Furthermore, our reduction—including the computation of the embedding of the variable–clause graph—is polynomial, since we use only a constant number of vertices and edges for each literal and clause gadget, and the number of vertices used for the variable gadgets only depends on the number of clauses that contain the variable. For each such clause a constant number of vertices and edges is used in the variable gadget.

It remains to show that any planar 3-regular augmentation makes G_φ 3-connected. To this end, we construct the augmented graph from the 3-connected graph G'_φ with steps that preserve

3-connectivity. First, replacing each vertex of degree at least 4 by a cycle of degree-3 vertices obviously preserves 3-connectivity. The remaining edges and vertices can all be inserted by subdividing two edges that share a face and connecting the new vertices by an edge. This is one of the well-known Barnette-Grünbaum operations [10], which preserve 3-connectivity. ☐

3.2 A Simple Polynomial-Time Algorithm for 3-FEPRA

Next, we give a simple, polynomial-time algorithm for 3-FEPRA. In a first step, we seek a valid 3-regular augmentation W of G, that is, an augmentation such that, for each edge, there exists a face of G that is incident to both endpoints of the edge. Since G has a fixed planar embedding, we assume that each edge in W is embedded within a face of G, possibly crossing some other edges in W that are embedded in the same face. Such an augmentation can be found in $O(n^{2.5})$ time using Gabow's algorithm [50]. In a second step, we show that W can always be transformed into a planar 3-regular augmentation. The main idea is to show that, as long as the augmentation is not planar, we can rewire edges such that we get a new augmentation with fewer crossings.

Let $G = (V, E)$ be a planar graph of maximum degree 3 with a fixed planar embedding and let W be a valid 3-regular augmentation with at most one crossing between any pair of edges (resulting from a generalized perfect matching as described above).

Proposition 3.2. *Let $G = (V, E)$ be a planar graph of maximum degree 3 with a fixed planar embedding. Let W be a valid 3-regular augmentation of G. Then G admits a planar 3-regular augmentation.*

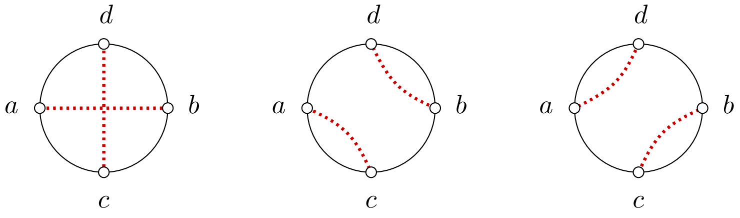
Proof. We assume without loss of generality that $G+W$ is connected. Otherwise, the connected components can be treated independently. Next we show that we can even assume that G is connected.

Claim: There is a set $P \subseteq W$ such that $G+P$ is connected and the edges of W can be embedded into the faces of G in such a way that the edges of P are crossing-free.

We choose $P \subseteq W$ as a set of edges that spans all connected components of G. More precisely, P is the edge set of a spanning tree of the graph that is obtained from $G + W$ by contracting each connected component of G to a vertex. Such a set exists since $G + W$ is connected. Clearly, $G + P$ is planar, and, since no edge of P subdivides a face of G, the remaining edges of W can be embedded inside the faces of $G + P$. This proves the claim.

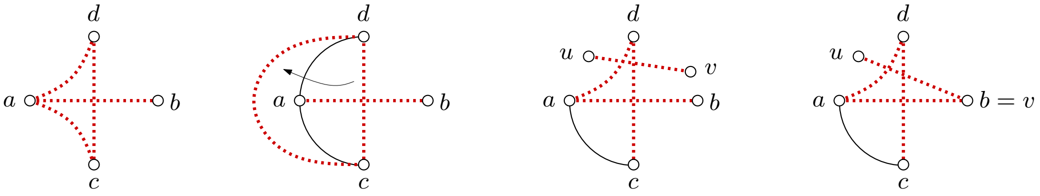
We now choose P as in the claim, change the embedding of the edges of $W \setminus P$ such that the edges in P are free of crossings and then consider $W \setminus P$ as an augmentation of the connected graph $G + P$.

In the following we can thus assume G is connected. In particular, each face f of G has now a single facial cycle C_f. We note that each vertex that forms a vertex separator occurs multiple times along C_f. When considering an edge $\{u, v\}$ embedded inside f this naturally defines a chord of C_f by taking for each endpoint its occurrence on C_f that corresponds to its embedding inside f. Assuming that any pair of edges in W crosses at most once and adjacent edges do not cross, we thus find that two edges cross each other if and only if their endpoints along C_f alternate. Our strategy for the proof is now to show that in this case we can modify W or its embedding such that the number of crossings decreases. Since the number of crossings is at most n^2 (as G is connected, and thus, there are at most $n/2$ edges in W), this yields a planar augmentation after at most n^2 modification steps.



(a) Crossing edges. (b) Exchange. (c) Exchange.

FIGURE 3.3: Illustration of the replacement of two crossing edges $\{a, b\}$ and $\{c, d\}$ by either $\{a, c\}$ and $\{b, d\}$ or by $\{a, d\}$ and $\{b, c\}$.



(a) $\{a, c\}, \{a, d\} \in W$. (b) $\{a, c\}, \{a, d\} \in E$. (c) $\{a, c\} \in E$, $\{a, d\} \in W$. (d) Case (c) with $b = v$.

FIGURE 3.4: The different cases for reducing crossings or the potential. Edges of G are drawn as solid line segments, edges of W as dashed line segments.

Let $e_1 = \{a, b\}$ and $e_2 = \{c, d\}$ be two edges that cross inside a face f, that is, their endpoints along C_f alternate. Without loss of generality, their circular order along C_f is $acbd$; see Fig. 3.3(a). If there is at least one pair of edges $\{a, c\}$ and $\{b, d\}$ or $\{a, d\}$ and $\{b, c\}$ such that $G + W$ does not yet contain an edge of this pair, we can exchange $\{a, b\}$ and $\{c, d\}$ with this pair, that is, with either $\{a, c\}$ and $\{b, d\}$ or $\{a, d\}$ and $\{b, c\}$; see Fig. 3.3(b) and Fig. 3.3(c) for an illustration. Note that any edge of $W \setminus \{e_1, e_2\}$ that crosses one of the replacement edges also crosses e_1 or e_2, and if an edge crosses both replacement edges, then it crosses also both e_1 and e_2. Since the two replacement edges do not cross each other, such a replacement decreases the number of crossings.

Otherwise, if such a replacement is not possible, assume without loss of generality that $G+W$ contains $\{a, c\}$ from the first and $\{a, d\}$ from the second pair. We distinguish cases based on whether these edges belong to G or to W.

First, assume that $\{a, c\}$ and $\{a, d\}$ both belong to W; see Fig. 3.4(a). But W also contains the edge $\{a, b\}$. This implies that a has degree 0 in G, contradicting the assumption that G is connected.

Second, assume that $\{a, c\}$ and $\{a, d\}$ both belong to G; see Fig. 3.4(b). They hence do not cross any edges. Since W contains $\{a, b\}$, the vertex a has degree 2 in G, which implies that $\{a, c\}$ and $\{a, d\}$ form a path. Thus, we simply redraw the edge $\{c, d\}$ along this path on the other side than $\{a, b\}$. Afterwards $\{c, d\}$ is free of crossings.

Third, assume that one of $\{a, c\}$ and $\{a, d\}$ belongs to G and one is in W. Without loss of generality, assume $\{a, c\} \in E$ and $\{a, d\} \in W$; see Fig. 3.4(c). If $\{a, d\}$ is not crossed by any edge, we can redraw $\{c, d\}$ as in the previous case, decreasing the number of crossings. Hence assume that there is an edge $\{u, v\} \in W$ that crosses $\{a, d\}$. We will show that it is possible to exchange $\{u, v\}$ and $\{a, d\}$ with either the pair of edges $\{a, u\}$ and $\{d, v\}$ or with $\{a, v\}$ and $\{d, u\}$, thus decreasing the number of crossings. First, observe that c is distinct from u and v. Otherwise, this would imply that both a and c have degree 1 in G, contradicting the connectivity of G. We now distinguish two subcases, based on whether b is distinct from u and v or coincides with one of them.

If b is distinct from both u and v (Fig. 3.4(c)), then a is fine with both possible exchanges. An exchange is then possible, unless d is adjacent to both u and v. However, d is already adjacent to a and c, and thus to at most one of u and v. Hence an exchange can be used to find a solution with fewer crossings.

Now assume that b coincides with one of u or v, without loss of generality $b = v$; see Fig. 3.4(d). Then both b and d are incident to two edges of W, and are hence leaves of G. Since G is connected, this in turn implies that they are not adjacent, and hence v (which is equal to b) and d are not adjacent in $G + W$. We thus have that exchanging $\{a, d\}$ and $\{u, v\}$ with $\{a, u\}$ and $\{v, d\}$ is possible, and hence decreases the number of crossings. □

Together with the above discussion showing that an augmentation W as assumed in Proposition 3.2 can be found in $O(n^{2.5})$ time, we immediately obtain the following.

Theorem 3.3. *The problem* 3-Fepra *(without further connectivity constraints) can be decided in $O(n^{2.5})$ time.*

While this easily establishes the polynomial-time solvability of 3-Fepra, this approach has a number of shortcomings. First, although the steps described in the proof of Proposition 3.2 can be used to find a planar augmentation in $O(n^3)$ time ($O(n^2)$ rewiring steps, each costing $O(n)$ time), the conversion step from a valid augmentation to a planar one is a major bottleneck. Second, the algorithm offers little structural insight into the problem. In particular, it is very unclear how the running time could be improved and how other requirements, such as additionally raising the connectivity, can be included into such an approach. In the next section we will study 3-Fepra more systematically and address all the mentioned issues. Among others, we will reduce the running time for the transformation of a valid 3-regular augmentation to a planar 3-regular augmentation to linear.

3.3 A Systematic Study of 3-Fepra

We want to decide for a graph $G = (V, E)$ with a fixed planar embedding, whether there exists an augmentation W such that $G + W$ is 3-regular and the edges in W can be embedded into the faces of G in a planar way. We will see that this problem is equivalent to finding a node assignment that assigns the vertices with degree less than 3 to the faces of the graph such that, for each face f, an augmentation exists that can be embedded inside f in a planar way and raises the degrees of all its assigned vertices to 3. In this section we characterize those node assignments and exploit the found properties for the design of efficient algorithms solving 3-Fepra even with additional connectivity constraints.

Recall that $\mathcal{F}$ denotes the set of faces of G and that $V^{\circledast}$ is the set of vertices with free valencies. A *node assignment* is a mapping $A\colon V^{\circledast} \to \mathcal{F}$ such that each $v \in V^{\circledast}$ is incident to $A(v)$. Each valid 3-regular augmentation W induces a node assignment by assigning each vertex v to the face where its incident edges in W are embedded: this is well-defined, since vertices in $V^{\textcircled{0}} \cup V^{\textcircled{1}}$ are incident to a single face. A node assignment is *realizable* if there exists a valid 3-regular augmentation that induces it. It is *realizable in a planar way* if it is induced by some planar 3-regular augmentation. We also call the corresponding augmentation a *realization.* Recall that the existence of a valid 3-regular augmentation, and thus, of a realizable node assignment, can be efficiently decided by computing a generalized matching in the subgraph of G^c that contains edges only between vertices that share a common face. According to Proposition 3.2, G thus admits a planar 3-regular augmentation if and only if it admits a realizable node assignment.

Both, valid augmentations and node assignments, are local by nature, and can be considered independently for distinct faces. Let A be a node assignment and let f be a face. We denote by V_f the vertices that are assigned to f. We say that A is *realizable for* f if there exists an augmentation $W_f \subseteq \binom{V_f}{2}$ such that in $G + W_f$ all vertices of V_f have degree 3. It is *realizable for* f *in a planar way* if additionally W_f can be embedded into f without crossings. We call the corresponding augmentations *(planar) realizations for* f. The following lemma is obtained by gluing (planar) realizations for all faces.

Lemma 3.4. *A node assignment is realizable (in a planar way) for a graph* G *if and only if it is realizable (in a planar way) for each face* f *of* G*.*

Proof. Consider a node assignment A. If A is realizable (in a planar way), there exists a corresponding valid (planar) augmentation W. Then, for each face f, the set $W_f \subseteq W$ of edges embedded inside f forms a (planar) realization for f. Conversely, assume that A is realizable (in a planar way) for each face f. Then, for each face f, there is a corresponding (planar) realization W_f of A for f. Hence $W := \bigcup_{f \in \mathcal{F}} W_f$ is a valid (planar) augmentation that realizes A. □

Note that a node assignment induces a unique corresponding assignment of free valencies, and we also refer to the node assignment as assigning free valencies to faces. In the spirit of the notation $G + W$, we use $f + W_f$ to denote the graph $G + W_f$, where the edges in W_f are embedded into the face f. If W_f consists of a single edge e, we write $f + e$, and sometimes we use f to also denote the subgraph of G that is induced by the vertices incident to the face f. For a fixed node assignment A we sometimes consider a *partial augmentation* W_f such that some vertices assigned to f have still a degree less than 3 in $f + W_f$. Then, A is *realizable for* $f + W_f$ if there exists an extension W'_f such that $W_f \cup W'_f$ forms a complete realization of A for f. We interpret A as a node assignment for $f + W_f$ that assigns to f all vertices that were originally assigned to f by A and do not yet have degree 3 in $f + W_f$. Observe that in doing so, we still assign to the face f (although the edges in W_f may split f into several new faces) but when considering free valencies and adjacencies, we consider $f + W_f$.

3.3.1 (Planarly) Realizable Assignments for a Face

Throughout this section we consider an embedded graph G together with a fixed node assignment A and a fixed face f of G. The goal of this section is to characterize when A is realizable (in a planar way) for f. We first collect some necessary conditions for a realizable assignment.

Condition 3.5 (parity)**.** *The number of free valencies assigned to* f *is even.*

Furthermore, we list eight *indicator sets* of vertices that are assigned to f and that demand additional valencies outside the set to which they can be matched, as otherwise an augmentation is impossible. At the same time, each vertex in an indicator set also provides some free valencies. Figure 3.5 illustrates the indicator sets.

(1) **Joker:** A vertex in $V^{②}$ whose neighbors are not assigned to f demands *one* valency.

(2) **Pair:** Two adjacent vertices u and v in $V^{②}$ demand *two* valencies, one not adjacent to u and one not adjacent to v.

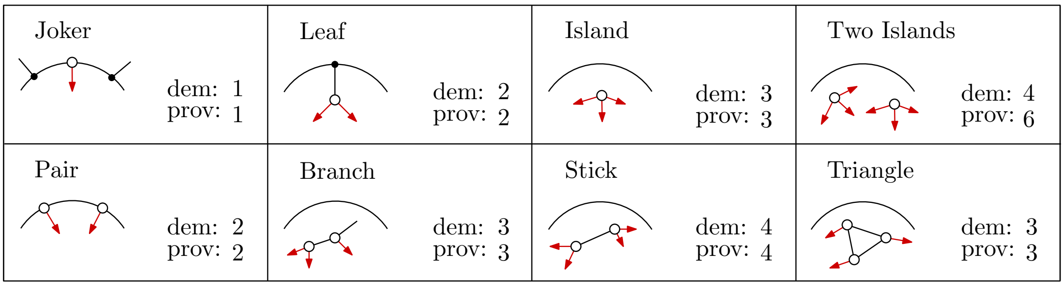


FIGURE 3.5: Indicator sets and the amount of valencies they demand and provide.

(3) **Leaf:** A vertex in $V^{\textcircled{1}}$ whose neighbor has degree 3 demands *two* valencies from two *distinct vertices.*

(4) **Branch:** A vertex in $V^{\textcircled{1}}$ and an adjacent vertex in $V^{\textcircled{2}}$ demand *three* valencies from at least *two distinct vertices* with at most one valency adjacent to the vertex in $V^{\textcircled{2}}$.

(5) **Island:** A vertex in $V^{\textcircled{0}}$ demands *three* valencies from *distinct vertices.*

(6) **Stick:** Two adjacent vertices of degree 1 demand *four* valencies; *no three* from the *same vertex.*

(7) **Two Islands:** Two vertices in $V^{\textcircled{0}}$ demand *four* valencies; *no three* from the *same vertex.*

(8) **Triangle:** A cycle of three vertices in $V^{\textcircled{2}}$ demands *three* valencies.

We note that a branch may properly overlap with other indicator sets. In particular, a branch $\{u, v\}$ may share its degree-2 vertex v with another branch $\{v, w\}$ with $u \neq w$. In this case it is crucial that only one valency of w may be used to satisfy the demand of the branch $\{u, v\}$.

Condition 3.6 (matching)**.** *The demands of all indicator sets formed by vertices assigned to f are satisfied.*

Each indicator set contains at most three vertices and provides at least the number of valencies it demands; only two islands provide more. The demand of a joker is implicitly satisfied by the parity condition. We call an indicator set with maximum demand *maximum indicator set*, and we denote its demand by $k_{\max}$. Note that $k_{\max} \leq 4$. We observe that inserting edges does not increase $k_{\max}$.

Observation 3.7. *Inserting an edge uv into f does not increase $k_{\max}$.*

Proof. Let k and k' denote $k_{\max}$ before and after the insertion of uv, respectively. We show $k' \leq k$. If $k' = 4$, then after the insertion there is a stick or two islands. Since a stick can only be obtained from two islands, we have $k = 4$. If $k' = 3$, then after the insertion there is a branch, an island, or a triangle. Since a branch can only be obtained from an island or a stick and a triangle can only be obtained from a branch, we have $k \geq 3$. If $k' = 2$, then, after the insertion, there is a pair or a leaf. Since a pair can only be obtained from two leaves, we have $k \geq 2$. ☐

The following lemma reveals the special role of maximum indicator sets.

Lemma 3.8. *Let S be a maximum indicator set in f and A a node assignment that satisfies the parity condition for f. Then A satisfies the matching condition for f if and only if the demand of S is satisfied.*

Proof. Clearly, if A satisfies the matching condition, then in particular the demand of S is satisfied. Hence, assume that the demand of S is satisfied. We prove that for any indicator set U of vertices assigned to f the demands are satisfied. Since the demand of a joker is always satisfied by the parity condition, we omit considering jokers in the following. Observe further that the demand of an indicator set that is contained in S is trivially satisfied, we may thus assume that U contains vertices outside S. We distinguish cases based on the demand $k_{\max}$ of S.

Case I: $k_{\max} = 4$. Then S consists either of a stick or of two islands. Let U be any indicator set distinct from S. Assume that U demands four vertices. If U is disjoint from S, then S provides the demanded valencies. Otherwise, both S and U consist of a pair of isolated vertices, and they share a common vertex. Since the demand of S is satisfied, there are at least two more assigned valencies provided by vertices outside of $S \cup U$. Together with $S \setminus U$, they provide the demanded valencies for U. The same argument applies if U consists of an island disjoint from S, and hence demands three valencies.

If U demands three or fewer valencies and it is not an island disjoint from S, then it is either contained in S or disjoint from it and no island. In the former case its demand is satisfied, in the latter case the demand is satisfied by the two vertices in S since the only indicator set demanding valencies from three different vertices is an island that is not contained in U.

Case II: $k_{\max} = 3$. Then S consists either of a triangle, an island, or a branch. If S is a triangle, then any other indicator set is either completely contained in S or disjoint from it, and it hence provides the necessary valencies (even for an isolated vertex).

If S consists of an island s, observe that $k_{\max} = 3$ implies that there is no other island assigned to f. The island s provides the necessary valencies for all indicator sets, except for a branch or a leaf. Assume that U is a branch. Since s demands three valencies from distinct vertices, there is a vertex $v \notin U \cup \{s\}$ assigned to f. Together s and v provide the valencies for U. The case that U is a leaf can be treated analogously.

Finally, consider the case that S consists of a branch. If U consists of an island u, then there must be a vertex $v \notin S \cup \{u\}$ providing a valency for S. Then $S \cup \{v\}$ provide the demanded valencies for u. If U is not an island, it demands at most three valencies from at most two different vertices. Hence, if U is disjoint from S, then S provides the demanded valencies for U. It remains to deal with the case that U is a pair or a branch sharing a degree-2 vertex with S. If U is a branch the situation for U and S is completely symmetric, and the demands for U are satisfied. If U is a pair, there exists again a vertex $v \notin S \cup U$ providing a valency for S, and hence, $S \cup \{v\}$ provide the demanded valencies for U.

Case III: $k_{\max} = 2$. Since the demands of jokers are always satisfied due to the parity condition, in this case all critical indicator sets consist either of pairs or of leaves. If S and U are both leaves, their situation is again completely symmetric. If S and U are a leaf and a pair, respectively, they mutually satisfy their demands. It remains to deal with the case that S and U both consist of pairs. If S and U are disjoint, they mutually satisfy their demands. If they share a vertex, then S and U are again completely symmetric. □

The necessity of the parity and the matching condition is obvious; we give the following characterization proving that the parity and the matching condition are also sufficient for a node assignment to be realizable for f.

Proposition 3.9. *A node assignment A is realizable for a face f if and only if A satisfies the parity and matching condition for f.*

Proof. Clearly, the parity condition and the matching condition for f are necessary for A to be realizable for f. Conversely, we show that, if A satisfies both conditions, then a corresponding realization for f exists. We prove the existence of a realization by construction. We defer the case that A assigns less than five vertices to f to the end. In the following we assume that A assigns at least five vertices to f and satisfies the parity and matching condition for f.

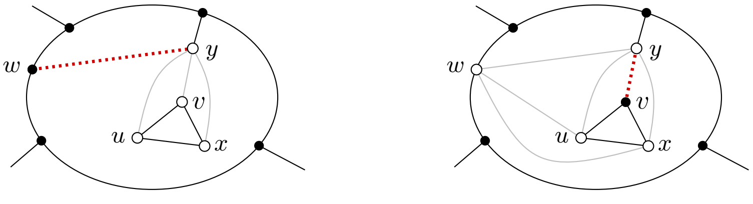
Suppose there exists a partial augmentation W_1 of f such that A still assigns $k \geq 4$ vertices to $f + W_1$, each assigned vertex has degree 2, and if A assigns exactly four degree-2 vertices in $f + W_1$, then no three of them form a cycle. We define the graph H^c that consists of the vertices assigned in $f + W_1$ and contains an edge between two vertices if and only if they are not adjacent in $f + W_1$. Our first goal is to show that H^c admits a perfect matching W_2. Then $W_1 \cup W_2$ is a 3-regular augmentation, and thus, a realization for f. Our second goal is to construct the partial augmentation W_1 for all possible assignments for f. To this end, we first construct a partial augmentation W_1' such that there are at least four vertices assigned in $f + W_1'$, each of which has degree 2. In a second step we transform W_1' to a partial augmentation W_1 that additionally avoids that $f + W_1$ contains a 3-cycle if there are only four vertices assigned in $f + W_1$. After that, it remains to deal with the case that A assigns less than five vertices to f.

Claim: H^c admits a perfect matching. Since each assigned vertex in $f + W_1$ has degree 2, it has at most two adjacencies in W_1, and thus, at least $(k-1) - 2 = k - 3$ adjacencies in H^c. For $k \geq 6$, this implies a minimum degree of $k - 3 \geq k/2$. In this case, by a theorem of Dirac [33], H^c contains a Hamiltonian cycle, which in turn contains the claimed perfect matching. For $k \leq 5$, observe that k must be even, and thus $k = 4$. The graph H^c thus has minimum degree 1. However, the only graph with minimum degree 1 on four vertices that does not admit a perfect matching is the star on four vertices. But then the three leaves of the star form a cycle of assigned degree-2 vertices in $f + W_1$, contradicting the assumption; see Fig. 3.6(a). Thus W_2 always exists.

In order to construct now the partial augmentation W_1 for f, which will result from a preliminary partial augmentation W_1', we begin with the following observation. Let S denote an island or a stick assigned to f and let e denote an edge between two valencies in f. Splitting e and connecting the resulting half-edges to the vertex, respectively the vertices, in S yields a partial augmentation $\{e_1, e_2\}$ such that the vertices in S have degree 2 in $f + \{e_1, e_2\}$. We refer to this procedure as *clipping in* the set S.

Now we construct a partial augmentation W_1' for all possible assignments for f such that there are at least four degree-2 vertices assigned in $f + W_1'$. In order to identify pairs of islands with sticks, in a first step we arbitrarily choose pairs of islands and connect them by an edge. Note that this in particular means that there remains at most one island assigned to f. We denote the set of all assigned sticks (and possibly the assigned island) by X. In a second step, we connect the assigned degree-1 vertices that are not in X pairwise, thus obtaining assigned degree-2 vertices. Connecting degree-1 vertices that are not in X, that is, that do not form a stick, is feasible since they are mutually non-adjacent. After this step, there remains at most one assigned degree-1 vertex besides the degree-1 vertices in X. This degree-1 vertex is either a leaf or part of a branch. We distinguish the possible assignments on whether there is one or no such degree-1 vertex assigned to f and whether X contains a stick, only an island, or is empty.

Assume there is exactly one degree-1 vertex v assigned to f that forms a leaf or is part of a branch. If X contains a stick we connect the vertices of this stick to the two valencies provided by v and clip in the possibly remaining elements in X. In doing so, v becomes a degree-3 vertices.



(a) $f + W_1'$ with 3-cycle uvx. (b) $f + W_1$ without 3-cycle.

FIGURE 3.6: Face f (solid black) and partial augmentations W_1' and W_1 (dotted). Assigned vertices are depicted as empty disks. Complement graph H^c (solid gray) admits perfect matching in (b) but not in (a).

If X consists of a single island, there exists a further valency at a degree-2 vertex $u \neq v$ due to the demand of this island. Hence, we connect the island to v and u yielding degree-2 vertices at the former island and v and a degree-3 vertex at u. If X is empty, there exists again a further valency at a degree-2 vertex $u \neq v$ due to the demand of v. Hence, we connect v to u yielding a degree-2 vertex at v and a degree-3 vertex at u. In all cases, there still remain four assigned degree-2 vertices in $f + W_1'$, since A initially assigned at least five vertices to f and we obtain only one degree-3 vertex per case.

Now assume there is no degree-1 vertex assigned to f apart from those in X. If $|X| \geq 2$, we connect all elements in X such that we obtain a cycle of degree-2 vertices. This is possible, since X contains at most one island. If X consists of a single stick, there are at least four further degree-2 vertices assigned due to the demand of this stick. Hence we connect each vertex of the stick to one of the degree-2 vertices. This yields at least four assigned degree-2 vertices in total for $f + W_1'$. If X consists of a single island, there are again at least four further degree-2 vertices assigned, since A assigns at least five vertices in total. Furthermore, there is an additional degree-2 vertex assigned due to the parity condition. Hence, connecting the island to two of the degree-2 vertices yields again at least four assigned degree-2 vertices in total for $f + W_1'$.

Claim: W_1' can be transformed into the partial augmentation W_1. By construction, there are at least four vertices assigned to $f + W_1'$, each of which has degree 2. Now assume there are exactly four (degree-2) vertices u, v, x, y assigned in $f + W_1'$ but uvx form a 3-cycle in $f + W_1'$. We claim that W_1' contains an edge $\{w, y\}$ incident to y. Then $W_1 = (W_1' \setminus \{\{w, y\}\}) \cup \{\{y, v\}\}$ is the required partial augmentation for f; see Fig. 3.6.

Assume W_1' does not contain an edge $\{w, y\}$ incident to y. Then, there is no additional vertex assigned to f, since, by construction of W_1', a vertex assigned to f, which has degree at most 2, becomes a degree-3 vertex, and thus is longer assigned in $f + W_1'$, if and only if it is connected by an edge in W_1' (to a degree-2 vertex in $f + W_1'$). That is, u, v, x, y are the only vertices providing valencies in f and y provides only one valency in f since it already has degree 2. On the other hand, depending on which edges of the 3-cycle are already in E, the vertices u, v, x induce a 3-cycle, a branch or two islands in f. We see that, in each case, the demand of the corresponding indicator set is not satisfied. Thus, this situation cannot occur since A satisfies the matching condition.

It remains to deal with the case that there are less than five vertices assigned to f.

Claim: If A assigns less than five vertices to f it is realizable. Let V_f ($|V_f| \leq 4$) denote the vertices assigned to f. We distinguish cases based on the number of assigned degree-0 vertices. For simplicity we skip the index indicating f and denote the degree-0 vertices in f by $V^{⓪}$.

Case 1: $|V^{\textcircled{0}}| = 4$. In this case, all assigned vertices have degree 0, and $\binom{V_f}{2}$ is the claimed realization.

Case 2: $|V^{\textcircled{0}}| = 3$. The parity condition implies that a fourth vertex with an odd number of valencies must be assigned to f. Since it may not have degree 0, it must have degree 2. But then the demand of two islands formed by two of the degree-0 vertices is not satisfied, contradicting the assumption that the matching condition holds for A. Thus this case cannot occur.

Case 3: $|V^{\textcircled{0}}| = 2$. Let $V^{\textcircled{0}} = \{u, v\}$. The matching condition implies that there are at least four valencies provided by vertices in $V_f \setminus V^{\textcircled{0}}$. However, $|V_f \setminus V^{\textcircled{0}}|$ contains at most two vertices of degree at least 1. Hence, the four additional valencies are provided by two degree-1 vertices x and y. The set $\{\{u, v\}, \{x, u\}, \{x, v\}, \{y, u\}, \{y, v\}\}$ forms the claimed realization.

Case 4: $|V^{\textcircled{0}}| = 1$. Let $V^{\textcircled{0}} = \{u\}$. The matching condition implies that there are at least three valencies provided by vertices in $V_f \setminus V^{\textcircled{0}}$. If $V_f \setminus V^{\textcircled{0}}$ consists of three degree-2 vertices x, y, z, then the set $\{\{u, x\}, \{u, y\}, \{u, z\}\}$ is the claimed realization. The only further possibility for $V_f \setminus V^{\textcircled{0}}$, due to the parity and the matching condition, is a degree-2 vertex x and two degree-1 vertices y and z. If the two degree-1 vertices form a stick, then the matching condition for this stick is not satisfied. Thus, y and z are not adjacent, and adding $\{y, z\}$ brings us back to the previous subcase, where x, y and z all have degree-2. This finishes the cases where at least one degree-0 vertex is assigned to f.

If there are no degree-0 vertices assigned to f, we distinguish cases based on the number of assigned degree-1 vertices. Analogous to the notion of the degree-0 vertices above, we now denote the degree-1 vertices in f by $V^{\textcircled{1}}$.

Case 5: $|V^{\textcircled{1}}| = 4$. Let $V^{\textcircled{1}} = \{u, v, x, y\}$. Since each degree-1 vertex has at most one adjacency in $V^{\textcircled{1}}$, without loss of generality u and v are not adjacent to x and y, and the set $\{\{u, x\}, \{u, y\}, \{v, x\}, \{v, y\}\}$ forms the claimed realization.

Case 6: $|V^{\textcircled{1}}| = 3$. Assume there is a fourth vertex x assigned to f. Due to the parity condition, x provides an even number of valencies, which implies that x has degree-1 contradicting $|V^{\textcircled{1}}| = 3$. Hence, it is $V_f = V^{\textcircled{1}}$. Let $V^{\textcircled{1}} = \{x, y, z\}$. The matching condition for a stick is not satisfied. Thus, the vertices in $V^{\textcircled{1}}$ are not adjacent and the set $\{\{x, y\}, \{y, z\}, \{x, z\}\}$ forms the claimed realization.

Case 7: $|V^{\textcircled{1}}| = 2$. Let $V^{\textcircled{1}} = \{u, v\}$. Again, the matching condition for a stick is not satisfied. Hence, u and v are not adjacent. The demand of u implies that there is a degree-2 vertex x in $V_f \setminus V^{\textcircled{1}}$, and the parity condition implies the existence of yet another degree-2 vertex y in $V_f \setminus V^{\textcircled{1}}$. If both u and v were adjacent to one of the degree-2 vertices, say x, then $\{u, x\}$ would form a branch whose demand is not satisfied. Hence, without loss of generality, v is not adjacent to x and u is not adjacent to y, and the set $\{\{u, v\}, \{x, v\}, \{u, y\}\}$ forms the claimed realization.

Case 8: $|V^{\textcircled{1}}| = 1$. Let $V^{\textcircled{1}} = \{u\}$. It follows from the parity and the matching condition that there are exactly two degree-2 vertices x and y in $V_f \setminus V^{\textcircled{1}}$. If one of these degree-2 vertices were adjacent to u, the corresponding set would form a branch whose demand is not satisfied. Thus, they are not adjacent to u, and $\{\{u, x\}, \{u, y\}\}$ forms the claimed realization.

Case 9: $|V^{\textcircled{1}}| = 0$. Now all vertices assigned to f have degree 2. If there are only two such vertices x and y, the matching condition implies that they are not adjacent, and the edge $\{x, y\}$ is the claimed realization. Otherwise, there are exactly four degree-2 vertices assigned to f. If it is not possible to match them up to a realization, the graph H^c that consists of degree-2 vertices assigned to f and contains an edge between two vertices if and only if they are not adjacent in f, does not admit a perfect matching. Observe that H^c has minimum degree 1, and thus, H^c is again the star on four vertices. But then the three leaves of the star form a 3-cycle in f, whose demand is not satisfied. This contradicts the assumption that the matching condition holds. □

We note that the notion of valencies also applies for general (possibly non-planar) graphs of maximum degree 3, and the indicator sets are also well defined with respect to all valencies in a non-planar graph. Hence, the parity and the matching condition are not restricted to planar graphs (and in particular not to a fixed embedding). Furthermore, the construction of the realization in the proof of Proposition 3.9 only exploits the non-adjacencies in the input graph but neither the planarity nor the embedding. Hence, in actual fact, Proposition 3.9 characterizes when the complement graph of a graph $G = (V, E)$ with maximum degree 3 admits a generalized perfect matching with demand $3 - i$ for each vertex $v \in V^{\textcircled{1}}$. This is equivalent to the non-planar 3-regular augmentation problem and directly implies that Proposition 3.9 also holds if the face f already contains a partial (possibly non-planar) augmentation W_f.

Corollary 3.10. *A node assignment A is realizable for a face $f + W_f$, with W_f a partial (possibly non-planar) augmentation of f, if and only if A satisfies the parity and matching condition for $f + W_f$.*

From an algorithmic point of view, the proof of Proposition 3.9 is not very useful in order to efficiently construct a realization. In the following, we thus develop another approach. We give a rule that allows to iteratively insert edges into f such that the inserted edges finally form a (not necessarily planar) realization. Given a node assignment A that satisfies the parity condition and the matching condition for a face f and possibly a partial augmentation W_f, the following rule picks an edge e that is not yet in $f + W_f$ such that, after the insertion of e into f, the remaining assignment still satisfies the parity and the matching condition. Corollary 3.10 then guarantees that iteratively applying this rule yields the claimed realization.

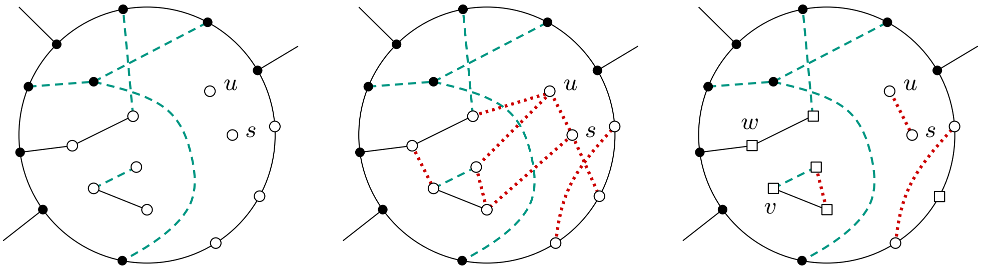
Rule 3.11.

1. *If $k_{\max} \geq 3$, let S denote a maximum indicator set. Choose a vertex u of minimum degree in S and connect u to an arbitrary assigned vertex $v \notin S$.*
2. *If $k_{\max} = 2$ and there is a leaf u, choose $S = \{u\}$, and connect u to an assigned vertex v.*
3. *If $k_{\max} = 2$ and there is no leaf but a path xuy of assigned vertices in $V^{\textcircled{2}}$, choose $S = \{x, u, y\}$. Connect u to an arbitrary assigned vertex $v \notin S$.*
4. *If $k_{\max} = 2$ and there is neither a leaf nor a path of three assigned vertices in $V^{\textcircled{2}}$, let S denote a pair $\{u, w\}$. Connect u to an arbitrary assigned vertex $v \notin S$.*
5. *If $k_{\max} = 1$, choose $S = \{u\}$, where u is a joker, and connect u to another joker v.*

Lemma 3.12. *Assume A satisfies the parity and matching condition for $f + W_f$, where W_f is a partial augmentation for f, and let e be an edge chosen according to Rule 3.11. Then A satisfies the same conditions for $f + W_f + e$.*

Proof. First observe that, according to Corollary 3.10, the partial augmentation W_f can be extended to a 3-regular augmentation $W_f + W'_f$ for f. Furthermore, if e is contained in W'_f, then A immediately satisfies the parity and the matching condition for $f + W_f + e$. Hence, we assume $e \notin W'_f$ and prove the assertion of Lemma 3.12 by locally reworking the extension W'_f such that afterwards it contains e. For simplicity, we skip the index for the face f in the notation and we write $\bar{f}$ instead of $f + W_f$.

More precisely, we do the following. We delete a set $W^{\ominus} \subseteq W'$ that depends on the set S and the vertex v chosen in Rule 3.11. After this deletion, A still satisfies the parity condition



(a) $f + W_f$. (b) $f + W_f + W_f'$. (c) $f + W_f + (W_f' - W^{\ominus})$.

FIGURE 3.7: The different layers of partial augmentations in a face f. (a) Dashed (green) edges represent the partial augmentation W_f. Vertices of degree 3 are filled, vertices providing valencies are empty. Vertex pair $\{u, s\} =: S$ induces two islands. (b) Dotted (red) edges represent the extension W_f', u and s are matched in W_f'. (c) Dotted (red) edges represent the set $W_f' - W^{\ominus}$ with respect to S and v, vertices in X are depicted as empty boxes.

and the matching condition since reinserting $W^{\ominus}$ would yield again a 3-regular augmentation for f. It remains to show that A also satisfies the parity condition and the matching condition if we just insert the edge $e = \{u, v\}$. We prove this by either reinserting the whole set $W^{\ominus}$ and then locally rewiring the edges in $W^{\ominus}$ such that $\{u, v\}$ becomes an edge in $W^{\ominus}$ or by just inserting e, which preserves the parity condition, and then arguing that the demand of any remaining maximum indicator set is still satisfied. The latter guarantees the matching condition according to Lemma 3.8.

The edges in $W^{\ominus}$ are chosen as follows. We delete all edges in W' that are incident to vertices in S and, if v has still degree 3, we delete an additional edge of W' that is incident to v. Furthermore, if $S = \{u, s\}$ represents two islands and u and s are adjacent in W', we reinsert the edge $\{u, s\}$, therefore excluding this edge from $W^{\ominus}$. In this way, we identify two islands with a stick. The free valencies resulting from the deletion of $W^{\ominus}$ are provided by the vertices in S and possibly by further vertices, which we denote by the set X. The sets S and X are disjoint and we get $u \in S$ and $v \in X$. Figure 3.7 shows an example.

In order to avoid parallel edges when rewiring edges in $W^{\ominus}$, we need to carefully study the adjacencies in $\bar{f} + (W' - W^{\ominus})$ and $W^{\ominus}$. For purposes of clarity, we will say that two vertices u and v are *matched* if they are adjacent in $W^{\ominus}$, and, unless specified otherwise, we reserve the word *adjacent* for adjacencies in the graphs $\bar{f}$ and $\bar{f} + (W' - W^{\ominus})$. We state the following properties regarding the set $W^{\ominus}$.

1) If $W^{\ominus}$ matches two vertices in S, then S is a path xuy according to subrule 3.
2) In this case, among the vertices of S, exactly u is matched to a vertex in X.
3) The vertex v is either matched to only vertices in S or to exactly one vertex in X.
4) In the former case all vertices of S are only matched to vertices in X.

For property 1) observe that, except in subrule 3, the set S (chosen in $\bar{f}$) is an indicator set. However, except for two islands, the vertices of each indicator set are already adjacent (in $\bar{f}$), and thus cannot be matched (in $W^{\ominus}$). For the case that S consists of two islands our special treatment (that is, excluding edges between islands from $W^{\ominus}$) ensures property 1. Property 2) is clear, since x, y and u already have degree 2 in $\bar{f}$ and $W^{\ominus}$ contains $\{x, y\}$, property 3) follows immediately from the construction of $W^{\ominus}$. For property 4) observe that, if $W^{\ominus}$ contains an edge between two vertices in S, then, by property 2), it follows that S is a path xuy. Since v must be matched to a vertex in S but not to u (since $e \notin W'$), this contradicts property 2).

We use property 3) to distinguish two different scenarios. In the first scenario, v is matched only to vertices in S. Then property 4) implies that vertices in S are only matched to vertices in X, and we use this to show that we can rewire the edges of $W^{\ominus}$ incident to u and v such that $W^{\ominus}$ contains the edge $\{u, v\}$. In the second scenario, we have that v is matched to exactly one vertex in X, and hence v provides exactly one valency after the removal of $W^{\ominus}$. Then inserting the edge $e = uv$, thus forming $\bar{f} + (W' - W^{\ominus}) + e$, increases the degree of v to 3. In this case we prove that the demand of a maximum indicator set given by the remaining valencies is satisfied.

According to the discussion above, we now consider the graph $\bar{f} + (W' - W^{\ominus})$ and show for each possible type of S that A satisfies the parity condition and the matching condition after the insertion of e, at that distinguishing whether v is matched to S or X. If v is matched to S, we reinsert $W^{\ominus}$ and locally rewire edges in $W^{\ominus}$ in order to construct the edge $\{u, v\}$. This preserves both, the parity and the matching condition. If v is matched to X, we insert e, which preserves the parity condition, and prove, by checking the demands of maximum indicator sets, that also the matching condition is satisfied. This will conclude the proof. Following its appearance in the subrules, the set S can be a pair of two islands, a stick, a triangle, an island, a branch, a leaf, a path of three vertices, a pair bounded by non-assigned vertices, or a joker.

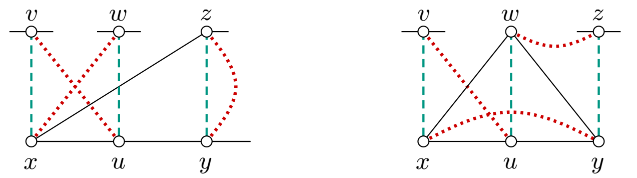
Scenario I: *The vertex v is only matched to vertices in S (in $W^{\ominus}$).* Recall from property 4) that, in this scenario, the vertices in S are exclusively matched to vertices in X. Since, by assumption, v is not matched to u, there are at least two vertices in S. Note that $|S|$ is bounded by 3. Hence, we distinguish two cases according to the size of S. For both cases we will locally rework the 3-regular augmentation W' by rewiring edges in $W^{\ominus}$ such that $e = \{u, v\}$ becomes an augmentation edge. The main issue in this context is to make sure that the rewiring does not cause parallel edges. When we wish to add an edge $\{p, q\}$, we thus have to make sure that p and q are neither matched nor adjacent. We note that, by construction of $W^{\ominus}$, an edge $\{p, q\}$ with $p \in S$ and $q \in X$ exists in $\bar{f} + W'$ if and only if it is in $\bar{f}$ or in $W^{\ominus}$. The edge $e = \{u, v\}$, however, can be always inserted without causing any problems, since by assumption $e \notin \bar{f} + W'$.

Case I: $|S| = 2$. Assume $S = \{u, s\}$. Then S is a pair of two islands, a stick, a branch or a pair bounded by vertices that are not assigned to f. The vertex v is only matched to s, and u is chosen in Rule 3.11 such that, in $\bar{f}$, its demand is at least as large as the demand of s. Since $\{v, s\} \in W^{\ominus}$ but $\{v, u\} \notin W^{\ominus}$, it follows that $W^{\ominus}$ contains an edge between u and a vertex $w \in X$ that is not matched to s. We show that we can choose w such that it is also not adjacent to s. Once this is done, we can replace $\{u, w\}$ and $\{v, s\}$ by $\{u, v\}$ and $\{w, s\}$ in $W^{\ominus}$, yielding an augmentation containing e.

If S it not a branch, then, after removing from $\bar{f}$ all non-assigned vertices, there are no edges from S to nodes outside S; this implies that s and w (which are both assigned to f) are not adjacent. If S is a branch, then the demand of u is strictly greater than the demand of s in $\bar{f}$, and hence there are at least two candidates for w, of which at most one can be adjacent to s.

Case II: $|S| = 3$. Then $S = xuy$ is a triangle or a path of degree-2 vertices according to subrule 3. We consider the adjacencies of S in $W^{\ominus}$. If S is a triangle, assume without loss of generality that v is at least matched to x and let w denote the vertex in X that is matched to u. Then x is neither matched nor adjacent to w (since x is adjacent to u and y and matched to v). Hence, rewiring the edges $\{u, w\}$ and $\{x, v\}$ in order to get $\{u, v\}$ and $\{x, w\}$ is feasible.

If S is a path of degree-2 vertices, assume again that v is matched to x. In particular, v is exclusively matched to x, since subrule 3 already excludes the existence of a leaf in $\bar{f}$. For



(a) Disjoint non-neighbors.

(b) Only a common non-neighbor.

FIGURE 3.8: Strategies for replacing $W^{\ominus}$ in case $S = xuy$ is a path of degree-2 vertices. Solid (black) edges belong to $\bar{f}$, dashed (green) edges represent $W^{\ominus}$, dotted (red) edges represent the claimed augmentation. (a) w is a non-neighbor of x and y, while z is (always) a non-neighbor of y. (b) z is the only non-neighbor of x and y.

the same reason, u and y are matched to distinct vertices w and z in X, and thus $W^{\ominus} = \{\{v,x\},\{u,w\},\{y,z\}\}$. Our goal is to rewire $\{v,x\}$, $\{u,w\}$ and $\{y,z\}$ in $W^{\ominus}$ in order to obtain an augmentation containing e.

We observe that x can be adjacent to at most one vertex in $\{w,z\}$ and y can be adjacent only to w among $\{w,z\}$ (since it is already matched to z). Hence, both x and y have at least one non-neighbor in $\{w,z\}$. If x and y have distinct non-neighbors in $\{w,z\}$ (see Fig. 3.8(a)), we change the augmentation as follows. We connect each of x and y to its non-neighbor and u to v. This results in an augmentation containing e. If x and y have no distinct non-neighbors but a common non-neighbor in $\{w,z\}$ (see Fig. 3.8(b)), it follows that they are both adjacent to w. In this case, after removing $W^{\ominus}$, $xuyw$ forms a cycle of length 4. Thus, replacing $W^{\ominus}$ by $\{\{x,y\},\{u,v\},\{w,z\}\}$ yields the claimed augmentation.

Scenario II: *The vertex v is matched to exactly one vertex in X (in $W^{\ominus}$)*, which we denote by w. Recall that, in this scenario, $W^{\ominus}$ does not match any two vertices in S, unless S is a path xuy given by subrule 3. Hence, all edges in $W^{\ominus} \setminus \{\{v,w\}\}$ run between the two disjoint vertex sets S and X. Consequently, the number of valencies after the deletion of $W^{\ominus}$ is twice the demand of S plus two valencies at v and w, for all types of S apart from the path xuy. We will now distinguish cases according to the value of $k_{\max}$ in $\bar{f}$ and show that, after the deletion of $W^{\ominus}$ and the insertion of e, there are still enough valencies to satisfy the demand of any maximum indicator set in $\bar{f} + (W' - W^{\ominus}) + e$. We remark that, according to Observation 3.7, $k_{\max}$ does not increase in $\bar{f} + (W' - W^{\ominus}) + e$.

Case I: $k_{\max} = 4$. In this case, $S = \{u,s\}$ is a pair of two islands or a stick, and thus, consists of connected components in $\bar{f}$ as well as in $\bar{f} + (W' - W^{\ominus})$. After the insertion of e, S is connected to X via v, however v then is a degree-3 vertex due to the scenario. Since none of the remaining valencies in S is adjacent to a valency in X in $\bar{f} + (W' - W^{\ominus}) + e$, after the insertion of e, a connected indicator set is either a subset of S or a subset of X. We observe further that the vertex $w \in X$ (which was matched to v) is the only vertex in X that possibly induces an island in X, since S contains only two vertices to which the vertices in X could have been matched. Hence, two islands do not occur in X. However, if w is an island and S is a pair of two islands in $\bar{f}$, after the insertion of e, the vertices w and s form a pair of two islands.

Now suppose S consists of two islands (in $\bar{f}$ and $\bar{f} + (W' - W^{\ominus})$). It then follows from the construction of $W^{\ominus}$ that u and s were not matched (in W'). After the deletion of $W^{\ominus}$ and the insertion of e, S then provides $6 - 1 = 5$ valencies at two vertices, and the maximum indicator set in S is the island induced by s. The set X provides $6 + 2 - 1 = 7$ valencies at at least three vertices, due to the demand of S. This already satisfies the demand of the island $s \in S$, and thus, the demand of any indicator set, unless there is an indicator set that demands more than three valencies. However, the only possible indicator sets in $\bar{f} + (W' - W^{\ominus}) + e$ that demand more than three valencies is a stick in X, whose demand is already satisfied by the valencies in S, and, if w is an island, the pair of islands w and s. In the latter case, the demand is satisfied by the four valencies at the at least two remaining vertices in X. Recall that w is the only island in X, and hence, no three of these four valencies are provided by a the same vertex.

Suppose S is a stick. Then, all indicator sets in $\bar{f} + (W' - W^{\ominus}) + e$ are connected and either contained in S or X. After the deletion of $W^{\ominus}$ and the insertion of e, S provides $4 - 1 = 3$ valencies at two vertices, and the maximum indicator set in S is a branch. The set X provides $4 + 2 - 1 = 5$ valencies at at least two vertices. This already satisfies the demand of the branch in S, since we have seen before that the valencies in X and S are not adjacent. Thus, the demand of any maximum indicator set is satisfied, unless there is an indicator set in X that demands more than three valencies. However, the only such indicator set that is possible in X is a stick. In this case, note that this stick only provides four of the five valencies provided by X. Hence, there must be a third vertex in X, which, together with S satisfies the demand of the stick.

Case II: $k_{\max} = 3$. In this case S is a triangle, an island or a branch. If S is a triangle or an island, it forms again a connected component in $\bar{f}$ and $\bar{f} + (W' - W^{\ominus})$, and after the insertion of e, S does neither contain an island nor is one of the remaining valencies in S adjacent to a valency in X. Hence, each indicator set in $\bar{f} + (W' - W^{\ominus}) + e$ is either a subset of S or a subset of X, and the number of valencies in X is just the demand of S plus 2. So we consider the cases where S is a triangle or an island first.

Suppose S is a triangle. After the deletion of $W^{\ominus}$ and the insertion of e, S then provides $3 - 1 = 2$ valencies at two vertices, and the maximum indicator set in S is a pair. The set X provides $3 + 2 - 1 = 4$ valencies at at least two vertices, since one vertex can provide at most three valencies. This already satisfies the demand of the pair in S and we are done, unless there is an indicator set in X that demands three valencies. Recall, that X cannot contain an indicator set with demand 4, since this contradicts $k_{\max} = 3$ in $\bar{f}$. If X contains an island, a triangle or a branch, we see the following. In the case of an island the demand is satisfied by S together with one of the remaining vertices in X. In the case of a triangle and a branch, the vertices in the indicator set provide less than four valencies. Hence, there must be a further vertex in X, which together with S satisfies the demand.

Now suppose S is an island. Then S provides again $3 - 1 = 2$ valencies but now at only one vertex, which trivially represents the maximum indicator set in S. The set X provides again $3 + 2 - 1 = 4$ valencies at at least three vertices, due to the demand of S. This satisfies the demand of the island S and we are done if X does not contain a triangle, a branch or an island. Otherwise, if X contains a triangle, a branch or an island, observe that in each case there are enough vertices and valencies left in X which, together with the vertex in S, satisfy the demands.

Now we consider the special case where $S = \{u, s\}$ is a branch. In this case it may happen that, after the deletion of $W^{\ominus}$ and the insertion of e, a vertex $z \in X$ is adjacent to the degree-2 vertex s of the branch S. In this case, u must have been matched to z in $W^{\ominus}$, and thus, $z \neq v$. Then, $s \in S$ and $z \in X$ can together form an indicator set. If this indicator set has a demand of at most 2, we can ignore it, since u and s already induce a pair. Otherwise, z must have been a degree-1 vertex after the deletion of $W^{\ominus}$. Hence the only indicator set of demand 3 that can contain z and s is a branch. Thus, in the following, we check the demand of the pair in S, which is the maximum indicator set in S after the insertion of e, any possible indicator set in X that has a demand of 3 and the branch $\{z, s\}$, which still might be a maximum indicator set even if the remaining vertices in X only induce indicator sets with a demand of at most 2.

After the deletion of $W^{\ominus}$ and the insertion of e, S provides $3-1=2$ valencies at two vertices, one of them (namely the valency at u) not adjacent to any vertex in X. The set X provides $3+2-1=4$ valencies at at least two vertices, and thus, clearly satisfies the demand of the pair in S. We observe further that the demand of a triangle, an island or a branch in X can be satisfied by the same arguments as used when S was supposed to be a triangle. Finally, if $s \in S$ and $z \in X$ form a branch (with z the degree-1 vertex), there remain two valencies in X and the valency at $u \in S$, which together satisfy the demand, since u is not adjacent to z.

Case III: $k_{\max} = 2$. In this case S is a leaf, a path xuy of three degree-2 vertices or a pair bounded by non-assigned vertices. Recall that the path xuy plays a special role, since here it may happen that $W^{\ominus}$ contains the edge $\{x, y\}$. Thus, we consider the remaining types of S first.

If S is a leaf or a pair bounded by non-assigned vertices, it holds that, after the deletion of $W^{\ominus}$ and the insertion of e, S only consists of a joker, while X provides $2+2-1=3$ valencies at at least two vertices, since an island in X contradicts $k_{\max} = 2$. Hence, X obviously satisfies the demand of the joker S. Vice versa, a maximum indicator set in X larger than a joker is a leaf or a pair. The demand of a leaf is satisfied by the joker in S and one of the remaining vertices in X. A pair provides only two of the three valencies in X, thus there is a third vertex in X, which together with S satisfies the demand.

Now suppose $S = xuy$ is a path of degree-2 vertices according to subrule 3. Then S provides $3-1=2$ valencies at the two vertices x and y after the deletion of $W^{\ominus}$ and the insertion of e. The number of the valencies provided by X, however, depends on whether the edge $\{x, y\}$ is in $W^{\ominus}$. Note that, in any case, each vertex in X provides only one valency, as otherwise there would have been a leaf in $\bar{f}$ before, which is excluded by subrule 3. Hence we get the following. If $\{x, y\} \in W^{\ominus}$, then X provides $1+2-1=2$ valencies at two vertices, otherwise it provides $3+2-1=4$ valencies at four vertices. Altogether, we get in both cases two valencies from S and at least two valencies from X, which yields at least four valencies at four distinct vertices. On the other hand, a maximum indicator set in $S \cup X$ is either a joker or a pair (a leaf is again excluded by subrule 3), and thus, the demand is obviously satisfied by the remaining valencies.

Case IV: $k_{\max} = 1$. In this case, S is a joker, which provides no valencies and induces no indicator set after the deletion of $W^{\ominus}$ and the insertion of e, while X only consists of jokers. At the same time X provides $1+2-1=2$ valencies. That is, X consists of exactly two jokers, whose demands are obviously satisfied. ☐

The construction of a realization with the help of Rule 3.11 inserts at most three edges per vertex assigned to f and inserting an edge can be done in $O(1)$ time. To this end we count the islands and then determine, in constant time, for each vertex the maximum indicator set that contains the vertex and group the vertices according to the demands of this sets. Whenever an edge is added to the current augmentation we need to update this information for only a constant number of vertices. For a single vertex this update can be again done in constant time. Hence, a non-planar realization, and thus, a degree-constrained subgraph in the complement of a graph of maximum degree-3 that solves the non-planar 3-regular augmentation problem, can be constructed in $O(n)$ time.

Our next goal is to extend this characterization and the construction of the realization to the planar case. Consider a path of degree-2 vertices that are incident to two distinct faces f and f' but are all assigned to f. Observe that, if only one connected component assigns valencies to f, then a *planar* realization for f cannot connect two vertices of such a path. Otherwise, we obtain a face to which only a path of degree-2 vertices is assigned. Iteratively connecting further degree-2 vertices in this face yields a face which is assigned exactly one path of degree-2 vertices of length 1 or 2. However, such faces do not admit any realization.

Thus, a planar realization for f may connect two vertices of a path of assigned degree-2 vertices that are incident to two distinct faces only if some vertex of the path can be connected to a valency from a different connected component, which then splits the path into two pieces. For cycles, the situation is similar, except that adjacencies from two different components are needed to split a cycle. Hence the following sets of vertices demand additional valencies, which gives a new condition, which is necessary for the existence of a planar realization.

(1) A path π of $k > 2$ assigned degree-2 vertices that are incident to two distinct faces (and whose end vertices are not adjacent) demands either k further valencies or at least one valency from a different connected component.

(2) A cycle π of $k > 3$ assigned degree-2 vertices (which are incident to two distinct faces) demands either k further valencies or at least two valencies from two distinct connected components different from π.

Condition 3.13 (planarity)**.** *The demand of each path of* $k > 2$ *and each cycle of* $k > 3$ *degree-2 vertices that are incident to two faces and that are assigned to* f *is satisfied.*

Obviously, the planarity condition is satisfied if and only if the demand of a longest such path or cycle assigned to f is satisfied. Furthermore, the parity, matching, and planarity condition together are necessary for the existence of a planar realization for the face f. In the following, we prove, by construction, for a node assignment A and a face f, that these conditions are also sufficient. To construct a planar realization for f, we give a refined selection rule that iteratively chooses edges that can be embedded in f, such that the resulting augmentation is a planar realization of A for f if A satisfies the parity, matching, and planarity condition for f. The new rule considers the demands of both maximum paths and cycles and maximum indicator sets, and at each moment picks a set with highest demand. If an indicator set is chosen, essentially Rule 3.11 is applied. However, we exploit the freedom to choose the endpoint v of $e = uv$ arbitrarily, and choose v either from a different connected component incident to f (if possible) or by a right-first (or left-first) search along the boundary of f. This guarantees that, even if inserting the edge e splits f into two faces f_1 and f_2, one of them is incident to all vertices that are still assigned to f. Slightly overloading notation, we denote this face by $f + e$ and consider all remaining valencies assigned to it. We show in Lemma 3.15 that A then again satisfies all three conditions for $f + e$. It then follows immediately that iteratively applying the modified rule yields a planar realization for f.

Rule 3.14. *Phase 1: Different connected components assign valencies to* f*. Consider all paths and cycles of assigned degree-2 vertices, regardless whether the degree-2 vertices are assigned to two distinct faces or only to* f*:*

1. *If there exists any path or cycle of more than* $k_{\max}$ *assigned degree-2 vertices, let* u *denote the middle vertex* $v_{\lceil k/2 \rceil}$ *of the longest such path or cycle* $\pi = v_1, \dots v_k$*. Connect* u *to an arbitrary assigned vertex* v *in another component.*
2. *If all paths or cycles of assigned degree-2 vertices have length at most* $k_{\max}$*, apply Rule 3.11, choosing the vertex* v *in another component.*

Phase 2: All assigned valencies are on the same connected component. Consider only paths of assigned degree-2 vertices that are incident to two distinct faces:

1. *If there exists a path that is longer than* $k_{\max}$*, let* u *denote the right endvertex* v_k *of the longest path* $\pi = v_1, \dots v_k$*. Choose* v *as the first assigned vertex found by a right-first search along the boundary of* f*, starting from* u*.*
2. *If all paths have length at most* $k_{\max}$*, apply Rule 3.11, choosing* v *as follows:*
   Let v_1, v_2 *denote the first assigned vertices not adjacent to* u *found by a left- and right-first search along the boundary of* f *starting from* u*, respectively. If* S *is a branch and one of* v_1, v_2 *has degree 2, choose it as* v*. In all other cases choose* $v = v_1$*.*

Lemma 3.15. *Assume* A *satisfies the parity, matching, and planarity condition for* f *and let* e *be an edge chosen according to Rule 3.14. Then* A *satisfies the same conditions for* $f + e$*.*

Proof. We first observe that inserting an edge e preserves the parity condition. Hence it suffices to show that A satisfies the matching condition and the planarity condition after the insertion of e. We prove the planarity condition first. To this end, we consider a path or cycle according to the planarity condition, that is, a path or cycle of at least three and four degree-2 vertices, respectively, that is incident to $f + e$ and a second face, and that is a longest such path or cycle incident to $f + e$. We denote this path or cycle by ρ. Then the planarity condition is satisfied for $f + e$ if and only if the demand of ρ is satisfied. In a second step we then prove that A also satisfies the matching condition for $f + e$.

Consider ρ for $f + e$ as described above. If ρ is a cycle, we are still in the first phase of Rule 3.14, and hence ρ did not arise by inserting e, since the first phase only inserts edges between different connected components. Thus, ρ was a cycle of the same length before the insertion of e. If ρ is a path, we distinguish three cases regarding the shape of ρ before the insertion of e. First, ρ was already a path of the same length incident to two faces (namely f and a second face). Second, ρ was part of a longer path incident to two faces. Third, u and v were both degree-1 vertices incident to f and ρ arose by connecting them. Since ρ is supposed to be incident to two faces, u and v belonged to the same connected component. In the following, we argue for each case that the demand of ρ is satisfied after the insertion of $e = uv$. The case where ρ is supposed to be a cycle is analogous to the first case of ρ being a path. Hence, we consider these two cases together.

Case I: If, for f, ρ was already a path or cycle (incident to two distinct faces) of the same length as in $f+e$, there must have been another path or cycle ρ' (distinct from ρ) for f that was at least as long as ρ, from which Rule 3.14 has chosen u. That is, after the insertion of e, the valencies at $\rho' \setminus \{u\}$ either completely satisfy the demand of ρ (if $|\rho'| > |\rho|$ for f) or they satisfy the demand of all but one vertex in ρ (if $|\rho'| = |\rho|$ for f). Recall that after the insertion of e all remaining valencies are incident to $f + e$. In the latter case, the total number of valencies at $\rho' \setminus \{u\}$ and ρ is odd for $f + e$. Hence, the parity condition guarantees the existence of the missing valency to completely satisfy the demand of ρ after the insertion of e.

Case II: If, for f, the path ρ was part of a longer path or cycle (incident to two faces), let ρ' denote this longer path or cycle for f. Since A satisfied the planarity condition for f, the demand of ρ' was satisfied by valencies outside ρ'. That is, there were either at least as many valencies outside ρ' as provided by ρ', or sufficiently many other connected components had assigned valencies. In the first case, since Rule 3.14 does never connect vertices of the same path or cycle, after the insertion of e there are still sufficiently many valencies outside ρ' to satisfy the demand of the remaining part of ρ', which is ρ. In the second case the demand of ρ' was satisfied by valencies from different connected components. Hence, we were in the first phase of Rule 3.14 and ρ' was split in the middle by the insertion of e if it was a path and cut open if it was a cycle. In the latter case, the planarity condition for f guarantees two valencies assigned from two different connected components, only one of which used up by e. The other valency from a different connected component then satisfies the demand of the path ρ. Finally, if ρ' was a path, it is split into two disjoint paths, one of which is ρ. These two paths mutually satisfy their demands, except for at most one valency in case ρ' had even length. But then the parity condition for $f + e$ implies the existence of an additional valency outside ρ.

Case III: If, for f, u and v were both degree-1 vertices at a common connected component and ρ arose by connecting them, u must have been chosen in the second phase by the subrule that calls Rule 3.11. In this phase it is $k_{\max} \leq 3$ and there is only one connected component that assigns valencies to f. We observe further that, in fact, it must have been $k_{\max} = 3$, since the applied subrule requires that there exists no path for f that is longer than $k_{\max}$. If $k_{\max}$ was less than 3 for f, the length of ρ could also not exceed 2 (after the insertion of e) and thus, ρ would be no path according to the planarity condition. Consequently, u must have been chosen from a branch $S := \{u, s\}$, since a branch is the only maximum indicator set for $k_{\max} = 3$ that contains a degree-1 vertex. Figure 3.9 exemplarily shows this situation. Since v was also a degree-1 vertex, it further follows from the applied subrule that the search in both directions to the left and to the right starting from u only passed degree-3 vertices before it terminated at the first degree-1 vertex v_1 or v_2 on either side, and thus $v = v_1$. Hence, the degree-2 vertex s of the branch $S = \{u, s\}$ must have been adjacent to a degree-3 vertex, and the same holds for v. That is, after connecting u and v, ρ consists of exactly three vertices, namely s, u and v, which provide an odd number of valencies. Due to the parity condition for $f+e$ it is thus $v_1 \neq v_2$. However, v_2 is also a degree-1 vertex. Hence, the number of valencies provided by ρ together with v_2 is still odd and the parity condition further guarantees another valency outside ρ, which together with v_2 satisfies the demand of $\rho = suv$. Thus, the planarity condition is preserved in all cases.

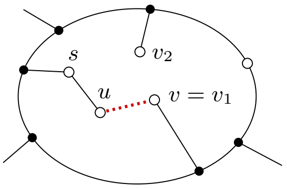


FIGURE 3.9: Situation of Case III in the proof of Lemma 3.15, where u and v were degree-1 vertices and ρ arose by connecting u and v. The dotted (red) edge was chosen by Rule 3.14.

To finish the proof, it remains to show that the matching condition is satisfied for $f + e$. The subrules calling Rule 3.11 obviously inherit this property from Rule 3.11. Thus, it suffices to prove the matching condition for the subrules where u is chosen from a path or cycle π that is longer than $k_{\max}$. To this end, we consider a maximum indicator set S after the insertion of e and show that the demand of S is satisfied for each possible length of the previously considered path or cycle π. This proves the matching condition according to Lemma 3.8. In the following, we distinguish whether S contains some of the remaining vertices of π after the insertion of e or not, that is, whether $S \cap (\pi \setminus \{u\}) = \emptyset$ or $S \cap (\pi \setminus \{u\}) \neq \emptyset$.

Case I: $S \cap (\pi \setminus \{u\}) = \emptyset$. In this case, the valencies provided by the set $\pi \setminus \{u\}$ after the insertion of e are all outside S and no two valencies are provided by the same vertex. Furthermore, the length of π was at least $k_{\max}+1$ before the insertion of e and $k_{\max}$ did not increase due to the insertion of e, according to Observation 3.7. Hence, the remaining valencies provided by $\pi \setminus \{u\}$ clearly satisfy the demand $k_{\max}$ of the maximum indicator set S after the insertion of e, even if S is a branch that is adjacent to one of the vertices in $\pi \setminus \{u\}$.

Case II: $S \cap (\pi \setminus \{u\}) \neq \emptyset$. In this case, S is either a branch or a pair. Any other indicator set does either not contain a degree-2 vertex at all or is a triangle, which obviously cannot intersect with the path $\pi \setminus \{u\}$. Note that $\pi \setminus \{u\}$ is always a path even if π was a cycle before. If S is a branch, it shares exactly one vertex with $\pi \setminus \{u\}$, if S is a pair, it is $S \subseteq \pi \setminus \{u\}$, since π was supposed to be the longest path or cycle for f. Hence, if S is a pair, $\pi \setminus (\{u\} \cup S)$ still provides $k-3$ valencies, with k the length of π for f, and $k-2$ valencies if S is a branch. On the other hand, a branch demands one more valency than a pair. That is, after the insertion of e, the number of valencies that are missing besides the valencies in $\pi \setminus (\{u\} \cup S)$ in order to satisfy the demand of S is the same for S being a branch or a pair and depends on the size $k-1$ of the set $\pi \setminus \{u\}$ after the insertion of e. Hence we distinguish the possible values for $k-1$.

If $k-1=4$ there remain three valencies at three distinct vertices in $\pi \setminus (\{u\} \cup S)$ if S is a branch, and two valencies if S is a pair. This clearly satisfies the demand of S in both cases.

If $k-1=3$, then S together with the remaining vertices in $\pi \setminus (\{u\} \cup S)$ provides an odd number of valencies, namely five valencies if S is a branch and three valencies if S is a pair. Hence, for both cases, the parity condition for $f+e$ guarantees a further valency outside S, which together with the valencies in $\pi \setminus (\{u\} \cup S)$ satisfies the demand of S.

Finally, we consider the case where $k-1=2$. That is, π consisted of only three vertices before the insertion of e. Since π is supposed to be longer than $k_{\max}$ before the insertion of e and in particular of S after the insertion of e, S can be only a pair and $k_{\max}=2$ before the insertion of e. Note that, for the same reason, the case $k-1<2$ does not occur. The latter implies that π is not a 3-cycle but a path of three degree-2 vertices. However, if S is a pair for $f+e$, the vertex u chosen by Rule 3.14 then must have been an endvertex of π since it is $S \subseteq \pi \setminus \{u\}$. This implies that u was chosen by the first subrule of the second phase of Rule 3.14. Consequently, f was incident to only one connected component and there were three valencies outside π due to the planarity condition for f. After the insertion of e at least two of these valencies are still assigned to $f+e$, and thus, the demand of S is satisfied. □

Given a node assignment A and a face f satisfying the parity, matching, and planarity condition, iteratively picking edges according to Rule 3.14 hence yields a planar realization of A for f. Applying this to every face yields the following characterization.

Proposition 3.16. *There exists a planar realization W of A if and only if A satisfies for each face the parity, matching, and planarity condition; W can be computed in $O(n)$ time.*

Proof. We construct the planar realization for each face individually. To construct a local realization for a face with a positive number of assigned vertices, we repeatedly apply Rule 3.14 to select an edge. By Lemma 3.15 this yields a planar local realization. It is not hard to see that repeatedly applying Rule 3.14 for a face f can be done in time proportional to the number of vertices incident to f. To allow fast left-first and right-first searches, we maintain a circular list containing the vertices incident to f with degree less than 3, and remove vertices reaching degree 3 from this list. Thus, also in phase 2 of Rule 3.14 the second vertex can be found in $O(1)$ time. □

3.3.2 Globally Realizable Node Assignments and Planarity

Up to this point we have seen a characterization of the node assignments that admit a planar realization and we have seen a simple algorithm, namely iteratively applying Rule 3.14, to construct a planar realization for a given node assignment. In this section we show how to decide if a node assignment exists that admits a planar realization and how to compute such a node assignment. By Proposition 3.16, this is equivalent to finding a node assignment satisfying for each face the parity, matching, and planarity condition. In a first step, we show that the planarity condition can be neglected as an assignment satisfying the other two conditions can always be modified to additionally satisfy the planarity condition. Note that this implicitly follows from Proposition 3.2, but we give a much more precise description in the following.

Lemma 3.17. *Given a node assignment A that satisfies the parity and matching condition for all faces, a node assignment A' that additionally satisfies the planarity condition for all faces can be computed in $O(n)$ time.*

Proof. Assume that f is a face for which the planarity condition is not satisfied, and let $\pi = v_1, \dots, v_k$ denote a longest path (or cycle) of degree-2 vertices, all assigned to f, that violates the planarity condition. Let f' denote the other face (distinct from f) incident to π. Let $u = v_1$. Choose $v = v_3$ if $k = 3$, and $v = v_{\lceil (k+1)/2 \rceil}$ otherwise. We modify A by reassigning u and v to f'. We claim that the resulting assignment has two properties, namely 1) for f' least the same conditions are satisfied as before the reassignment, and 2) for f the parity condition, the matching condition and the planarity condition are satisfied. Hence, applying this reassignment for all faces for which the planarity condition is not satisfied, yields the claimed assignment A'.

Note that, since π is either a path of length more than 2 or a cycle of length more than 3, the two vertices u and v are distinct and non-adjacent. To see property 1) consider the face f'. Obviously, the reassignment preserves the parity condition for f'. For the matching condition assume that M is an augmentation of f' with respect to A. Then $M \cup \{uv\}$ is an augmentation of f' with respect to A', thus the matching condition is preserved. Moreover, if M is a planar augmentation, then uv can be added in a planar way, showing that also the planarity condition is preserved for f'.

Concerning property 2), the reassignment obviously preserves the parity condition for f. For the matching and the planarity condition assume that, after the reassignment, there exists a set T of vertices assigned to f that demands k' additional free valencies by either the matching condition or the planarity condition. First, observe that $k' \leq k-1$, as π would not have violated the planarity condition, otherwise. If $k' = k-1$, then T is disjoint from π, which provides $k-2$ free valencies (recall that π was a longest path before the reassignment and u and v have been reassigned), and the parity condition implies the existence of an additional free valency assigned to f, thus ensuring that the demand of T is satisfied. The same argument works for all cases where T is disjoint from π. Thus assume that T and π are not disjoint. Since π was chosen as a maximal path or cycle incident to two faces, it follows that T is a subset of π. Note that the reassignment splits π into two disjoint subpaths π_1 and π_2 consisting of $\lceil (k-2)/2 \rceil$ and $\lfloor (k-2)/2 \rfloor$ vertices, respectively. Then, π_2, possibly together with an additional free valency provided by the parity condition (if k is odd), provides the necessary valencies for π_1 and vice versa. Thus, also the demand of T is satisfied. For f, the new assignment hence satisfies the matching condition as well as the planarity condition, and property 2) holds.

Observe that once a longest path or cycle violating the planarity condition has been found, the reassignment for a face f takes only $O(1)$ time. Moreover, since we only need to consider maximal sequences of assigned degree-2 vertices, such a path or cycle can be found in time proportional to the size of f. The test whether the planarity condition for this path is satisfied can be performed in the same running time. Thus A' can be computed from A by simply traversing all faces, spending time proportional to the face size in each face. Hence, computing A' from A takes $O(n)$ time. □

Proposition 3.16, which guarantees a planar realization for each node assignment that satisfies the parity, matching and planarity condition, and Lemma 3.17, saying that in fact the planarity condition can be neglected, together imply the following characterization.

Proposition 3.18. *A planar graph of maximum degree* 3 *with a fixed planar embedding admits a planar 3-regular augmentation, if and only if it admits a node assignment that satisfies for all faces the parity and matching condition.*

To find a node assignment satisfying the parity and matching condition for a given planar graph $G = (V, E)$ of maximum degree 3 and a fixed planar embedding of G, we compute a (generalized) perfect matching in the following (multi-)graph $G_A = (V^{\circledast}, E')$, called *assignment graph.* It is defined on $V^{\circledast} = V^{\textcircled{0}} \cup V^{\textcircled{1}} \cup V^{\textcircled{2}}$, and the demand of a vertex in $V^{\textcircled{i}}$ is $3-i$ for $i = 0, 1, 2$. For a face f let $V_f^{\circledast} \subseteq V^{\circledast}$ denote the vertices incident to f. For each face f of G, G_A contains the edge set $E'_f = \binom{V_f^{\circledast}}{2} \setminus E$, connecting non-adjacent vertices in $V^{\circledast}$ that share the face f. We seek a perfect (generalized) matching M of G_A satisfying exactly the demands of all vertices. The interpretation is that we assign a vertex v to a face f if and only if M contains an edge incident to v that belongs to E'_f. The following lemma states the correspondence between the perfect matchings of G_A, as described above, and the node assignments for G that admit a planar realization.

Lemma 3.19. *A perfect matching of G_A corresponds to a node assignment that satisfies the parity and matching condition for all faces, and vice versa.*

Proof. First assume that M is a perfect matching of G_A, and let A be the corresponding assignment. Observe that for each face f, the edge set $E'_f \cap M$ is exactly a realization of A for f, and hence, by Proposition 3.9, A satisfies the parity condition and the matching condition for f. Conversely, again by Proposition 3.9, for a node assignment A that satisfies the parity condition and the matching condition for each face f, we find a realization W_f for each face. Note that by definition of E'_f we have $W_f \subseteq E'_f$, and thus $\bigcup_{f \in \mathcal{F}} W_f$ yields a perfect matching of G_A inducing A. □

This finally implies the following algorithm, which runs in $O(n^{2.5})$ time, as proven by the next theorem. For a given planar input graph G with n vertices, maximum degree 3, and a fixed planar embedding, we perform the following steps.

(1) We construct the assignment graph G_A.
(2) We check whether G_A admits a perfect matching.
(3) If G_A admits a perfect matching, from the node assignment A, which is induced by the matching in G_A, we construct a new node assignment A' that additionally satisfies the planarity condition (if G_A admits no perfect matching, G does not admit a planar 3-regular augmentation).
(4) We construct a corresponding planar realization of A'.

Theorem 3.20. *The problem* 3-Fepra *can be solved in $O(n^{2.5})$ time.*

Proof. We can construct the assignment graph $G_A = (V^{\circledast}, E')$ in $O(n^2)$ time, with n the number of vertices in the input graph G. Checking whether G_A admits a perfect matching can be done in $O(\sqrt{|V^{\circledast}|}|E'|) = O(n^{2.5})$ time, using the algorithm due to Gabow [50], as introduced at the beginning. According to the characterization in Proposition 3.18 and the correspondence between perfect matchings in G_A and node assignments satisfying the characterization in Lemma 3.19, this decides whether G admits a planar 3-regular augmentation. By the same lemma, we obtain a node assignment A (induced by the matching in G_A) that satisfies the parity condition and the matching condition for each face. Using Lemma 3.17, we obtain in $O(n)$ time a node assignment A' that additionally satisfies the planarity condition for each face. A corresponding planar realization of A' can then be obtained in $O(n)$ time by Proposition 3.16. $\square$

CHAPTER 4

Algorithms for C-connected Planar 3-Regular Augmentation with Fixed Embedding

In this chapter, we generalize the results obtained for arbitrary 3-Fepra to efficiently solve c-connected 3-Fepra for $c = 1, 2$. The triconnected case is shown to be NP-hard in Section 4.3. For both problem variants, we give again a characterization of the node assignments that admit the required planar realizations. Moreover we describe how to construct an assignment graph G_A such that any perfect matching in G_A induces a node assignment according to the given characterization, and vice versa. We will see that the condition on (bi)connectivity, which is now additionally necessary for the characterization (besides the parity and the matching condition), is again defined locally for each face. Hence, we again construct the assignment graph G_A by constructing an edge set E'_f for each face f, and say that a perfect matching M of G_A induces a node assignment for G that assigns to each face f the vertices in $V^{\circledstar}$ that are incident to edges in $M \cap E'_f$. This finally implies the same algorithmic steps as for solving arbitrary 3-Fepra, however some details in the construction of G_A and the final augmentation differ.

4.1 Construction of the Assignment Graph G_A

The construction of G_A distinguishes two types of faces. The first type are the faces for which the subgraph H_f of $G = (V, E)$ consisting of the vertices and edges of G incident to f is already (bi)connected. For these faces the condition on (bi)connectivity is trivially satisfied by any node assignment. Hence, we call these faces *(bi)connected* and, for the assignment graph, we construct the edge set $E'_f = \binom{V_f^{\circledstar}}{2} \setminus E$ for each such face as in Section 3.3.2. For the remaining faces, which we call *non-(bi)connected*, we need to explicitly enforce the condition on (bi)connectivity when constructing E'_f. To this end, we construct additional dummy vertices, which also demand adjacencies in a perfect matching of G_A. So the vertex set of G_A is now $V^{\circledstar} \cup V_D$, where V_D denotes the set of dummy vertices. The edge set of G_A is again $E' = \bigcup_{f \in \mathcal{F}} E'_f$, with E'_f for (bi)connected faces as defined above and for non-(bi)connected faces as constructed in the following. We start discussing some preliminary insights, which we will exploit when constructing the edge set E'_f for non-(bi)connected faces.

Preassigned Vertices. An *empty 3-cycle* is a cycle of three vertices in $V^{\textcircled{2}}$ that is incident to a face that does not contain any further vertices. Although, technically, empty 3-cycles are incident to two faces, no augmentation edges can be embedded on the empty side of the 3-cycle, hence the assignment of the vertices of empty 3-cycles is already fixed. The same holds for the vertices in $V^{⓪} \cup V^{①}$ and the vertices in $V^{②}$ that are incident to bridges. They are all incident to only a single face. We denote these *preassigned* vertices by $\overline{V}$ and write $\overline{V}_f$ if we only consider the vertices in $\overline{V}$ that are incident to the face f. The vertices in $\overline{V}$ induce a preliminary node assignment $\overline{A}\colon \overline{V} \to \mathcal{F}$ that is defined piecewise on the faces of G by $\overline{A}(\overline{V}_f) := \{f\}$ for all faces. We denote the restriction of $\overline{A}$ to a face f by $\overline{A}_f := \overline{A}|_{\overline{V}_f}$.

Therefore, it suffices for a perfect matching in G_A to consider the remaining vertices in $V^{②}$ that are not yet preassigned. The set of remaining *non-preassigned* vertices in $V^{\circledast}$ is denoted by $\ddot{V}$. Analogously, we denote the vertices in $\ddot{V}$ that are incident to a face f by $\ddot{V}_f$. Note that the vertices in $\ddot{V}$ are incident to two distinct faces f and f'. Each vertex in $\ddot{V}$ is thus in two sets $\ddot{V}_f$ and $\ddot{V}_{f'}$.

Ensuring the Matching Condition. The vertices of an indicator set assigned to f are either in $\overline{V}_f$ or in $\ddot{V}_f$. Hence, for non-(bi)connected faces, we ensure the matching condition for each vertex set $\overline{V}_f$ and $\ddot{V}_f$ individually. For a maximum indicator set in $\overline{V}_f$, we will ensure the matching condition with the help of a dummy vertex. Similarly, dummy vertices will be also used to ensure the parity condition and the condition on (bi)connectivity. For a maximum indicator set in $\ddot{V}_f$, we will see that if the parity condition and the condition on (bi)connectivity are satisfied, there are always enough vertices preassigned or required by dummy vertices to already satisfy its demand. Hence, there is no need to explicitly ensure the matching condition for non-preassigned vertices. This in particular allows to also assign adjacent vertices of $\ddot{V}_f$ to f.

4.1.1 Connected 3-FEPRA

Before we describe the construction of the assignment graph for connected 3-FEPRA, we characterize the node assignments that admit a connected planar 3-regular augmentation. Let f be a face of G, and let z denote the number of connected components incident to f. Obviously, an augmentation connecting all these components must contain at least a spanning tree on these components, which consists of $z-1$ edges and thus needs $2(z-1)$ valencies assigned to f. Hence, the following *connectivity condition* is necessary for a node assignment A to admit a connected realization for f.

Condition 4.1 (connectivity)**.** *(1) If $z > 1$, each connected component incident to f must have at least one valency assigned to f.*
(2) The number of valencies assigned to f must be at least $2z-2$.

By iteratively applying Rule 3.14, we show that this condition is also sufficient; besides the parity, the matching and the planarity condition. This yields the following characterization.

Proposition 4.2. *There exists a connected planar realization W of A if and only if A satisfies for each face the parity, matching, connectivity, and planarity condition; W can be computed in $O(n)$ time.*

Proof. Clearly the conditions are necessary. We prove that they are also sufficient by iteratively applying Rule 3.14. Let A be a node assignment that satisfies for each face the parity, the matching, the connectivity and the planarity condition. We construct the connected planar realization W for each face individually. To this end we observe that, for a face f with $z > 1$, we can always choose an edge $e = uv$ in the first phase of Rule 3.14 such that A satisfies the connectivity condition also for $f + e$, and the first phase does not end until f is incident to only one connected component. At this point the connectivity condition is trivially satisfied for f and by Lemma 3.15 also the remaining conditions are still satisfied for f. Hence, choosing further edges according to the second phase of Rule 3.14 finally yields a connected planar realization for f. The running time can be argued as in the proof of Proposition 3.16. □

The following lemma follows from Lemma 3.17 by showing that the reassignment which establishes the planarity condition preserves the connectivity condition.

Lemma 4.3. *Given a node assignment A that satisfies the parity, matching, and connectivity condition for all faces, a node assignment A' that additionally satisfies the planarity condition can be computed in $O(n)$ time.*

Proof. To see this, recall that Lemma 3.17 reassigns from each face at most two vertices to a distinct face if the planarity condition is not satisfied. Clearly, assigning more vertices to a face does not invalidate the connectivity condition. Thus, an invalidation of the connectivity condition for a face f may only happen when two vertices assigned to f are reassigned to a different face. Note that if $z > 2$ for f, the planarity condition is implied by connectivity condition (1). Thus a reassignment only happens for faces with $z = 1, 2$. If $z = 1$, the connectivity condition holds trivially. If $z = 2$, observe that connectivity condition (2) is implied by condition (1), and since the reassignment does not reassign the last valency of a connected component, connectivity condition (1) is preserved. □

Proposition 4.2, which guarantees a connected planar realization for each node assignment that satisfies the parity, matching, connectivity and planarity condition, and Lemma 4.3, saying that in fact the planarity condition can be neglected, imply the following characterization.

Proposition 4.4. *A planar graph of maximum degree* 3 *with a fixed planar embedding admits a connected planar 3-regular augmentation if and only if it admits a node assignment that satisfies for all faces the parity, matching, and connectivity condition.*

Assignment Graph Construction for Connected 3-FEPRA. We describe an assignment graph $G_A = (V^{\circledast} \cup V_D, E')$, whose construction is such that there is a correspondence between the perfect matchings of G_A and node assignments satisfying the parity, matching, and connectivity condition. We construct an individual edge set $E'_f \subseteq E'$ for each face f, distinguishing connected faces (for which the connectivity condition is trivially satisfied) and non-connected faces, also called *disconnected*, as introduced at the beginning of Chapter 4. In the context of connected 3-FEPRA, a face is connected if and only if it is $z = 1$ for this face. For connected faces, we use the same set $E'_f = \binom{V_f^{\circledast}}{2} \setminus E$ as in Section 3.3.2.

In the following we consider disconnected faces, which are the faces with $z > 1$. Let f be a face that is incident to at least two connected components. We first determine the sets $\overline{V}_f$ and $\ddot{V}_f$ of preassigned and non-preassigned vertices and the preliminary node assignment $\overline{A}_f$. For each component C incident to f that does not provide a preassigned vertex, we then add a dummy vertex d_C with demand 1 and connect it to all vertices in $\ddot{V}_f \cap C$; this ensures connectivity condition (1). Let $\ddot{c}$ denote the number of these dummy vertices, and note that there are exactly $\ddot{c}$ valencies assigned to f due to these dummy vertices. Let $\overline{a}$ denote the number of valencies assigned to f by $\overline{A}_f$, and let δ denote the maximum remaining demand of an indicator set in $\overline{V}_f$, that is, the smallest number of valencies that must be additionally provided by vertices in $\ddot{V}_f$ such

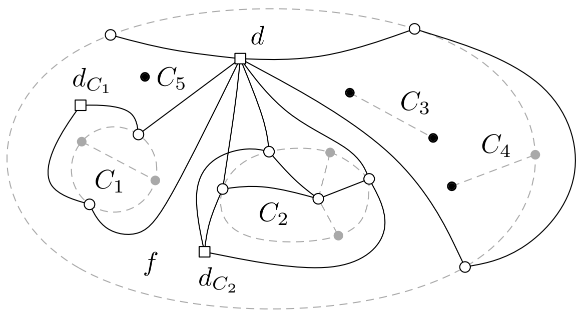


FIGURE 4.1: Graph (dashed lines) with one disconnected face f and five connected faces. Degree-3 vertices are filled gray, preassigned vertices are filled black, remaining vertices with degree less than 3 are empty. The solid lines depict the corresponding assignment graph (dummy vertices as empty boxes).

that the demand of any maximum indicator set in $\overline{V}_f$ is satisfied (by $\overline{A}_f$, the $\ddot{c}$ valencies due to the dummy vertices and the additional valencies). To ensure the matching condition for f, at least δ further valencies need to be assigned to f. For connectivity condition (2) we need at least $2z-2$ valencies assigned to f, while we already know that $\overline{a}+\ddot{c}$ valencies are definitely assigned to f. We thus create a dummy vertex d whose demand is set to $s = \max\{2z-2-(\overline{a}+\ddot{c}), \delta, 0\}$ and connect d to all vertices in $\ddot{V}_f$. The vertices in $\ddot{V}_f$ have demand 1. If $s+\overline{a}+\ddot{c}$ is odd, we increase s by one to also guarantee the parity condition.

We will see in the proof of the following lemma that the construction so far already assigns enough valencies to f to satisfy the demand of any maximum indicator set possibly arising from assigning vertices of $\ddot{V}_f$ to f. Hence, we finally allow an additional arbitrary even number of vertices in $\ddot{V}_f$ to be assigned to f. To achieve this we just add all pairs of vertices in $\ddot{V}_f$ as edges to E'_f. Note that this is different from the connected faces, where we only allow pairs of non-adjacent vertices of $\ddot{V}_f$.

Observe that, for disconnected faces, the edges in E'_f are only incident to dummy vertices and vertices in $\ddot{V}_f$, since preassigned vertices do not need to be considered. Figure 4.1 shows an example of an assignment graph; for clarity edges connecting vertices in $\ddot{V}_f$ are omitted for disconnected faces.

Lemma 4.5. *A perfect matching of G_A (together with $\overline{A}$) corresponds to a node assignment that satisfies the parity, matching, and connectivity condition for all faces, and vice versa.*

Proof. Let M be a perfect matching of G_A and let A denote the corresponding node assignment together with $\overline{A}$. We show that A satisfies the parity, matching, and connectivity condition for all faces. For connected faces, A satisfies the connectivity condition by definition. The matching condition and the parity condition are satisfied since the set $E'_f \cap M$ is a perfect matching with respect to all vertices in $V_f^{\circledast}$, as we did not distinguish preassigned and non-preassigned vertices for the construction of E'_f.

Now assume f is a disconnected face, for which it is $z > 1$. Using the definitions from above, there are $\overline{a}$ valencies assigned to f by $\overline{A}_f$, $\ddot{c}$ valencies from vertices adjacent to dummy vertices d_C, s valencies from vertices incident to the dummy vertex d and $2k$ valencies from k edges in $M \cap \binom{\ddot{V}_f}{2}$. In total these are $\overline{a}+\ddot{c}+s+2k$ valencies, which is even due to the choice of s, and hence the parity condition holds. For the connectivity condition, observe that the dummy vertices d_C imply connectivity condition (1) and the choice of s implies connectivity condition (2).

It remains to prove that the matching condition is satisfied for f. Let S denote a maximum indicator set of f (with respect to A). If $S \subseteq \overline{V}_f$ then S was already a maximum indicator set with respect to $\overline{A}_f$, and its demand is satisfied due to the choice of s. If $S \subseteq \ddot{V}_f$, then S is a joker, a pair or a triangle. For a joker the necessary valency exists due to the parity condition for f. If S is a pair, the vertices of S are in a common connected component. Since $z > 1$ and A satisfies the connectivity condition for f, there is another connected component incident to f that provides a further valency. A second additional valency is guaranteed by the parity condition for f. Hence the demand of S is satisfied, unless S is part of a 3-cycle C and the second additional valency is provided by the remaining vertex in C. This situation, however, does not occur. If C is empty, its vertices are preassigned, contradicting $S \subseteq \ddot{V}_f$. If C is not empty, at least one of its vertices is incident to an edge in M that is embedded in the other face distinct from f (otherwise there would be a dummy vertex in this face that is supposed to ensure the connectivity condition and whose demand is not satisfied). Hence the second valency cannot be provided by a common neighbor of the vertices in S. Finally, if S is a triangle, its vertices are preassigned, contradicting again $S \subseteq \ddot{V}_f$.

Conversely, let A be a node assignment that satisfies, for each face f, the parity condition, the matching condition, and the connectivity condition. We construct a perfect matching in G_A that together with $\overline{A}$ induces A. First of all, note that the preassignment $\overline{A}$ follows from necessary conditions, and thus A assigns preassigned vertices in the same unique way as $\overline{A}$. For a connected face f, a matching $M_f \subseteq E'_f$ that satisfies exactly the demands of the vertices assigned to f exists according to Proposition 3.9.

For a disconnected face f, we seek a matching $M_f \subseteq E'_f$ that further satisfies the demand of the dummy vertices associated with f. Recall that, by construction, the vertices in $\overline{V}_f$ have no demands and are not contained in G_A. Connectivity condition (1) implies that each connected component either contains a vertex in $\overline{V}_f$ or a vertex in $\ddot{V}_f$ assigned to f. We pick for each connected component C that does not contain a vertex in $\overline{V}_f$ an arbitrary assigned vertex in $C \cap \ddot{V}_f$ and match it to d_C. Thus, the demands of the dummy vertices d_C are satisfied, as well as the demands of the corresponding vertices in $\ddot{V}_f$. The matching condition implies that the number of remaining valencies at vertices in $\ddot{V}_f$ is at least δ, and connectivity condition (2) implies that at least $2z - 2 - (\overline{a} + \ddot{c})$ valencies remain at vertices in $\ddot{V}_f$. Thus, we can match arbitrary s of these valencies at vertices in $\ddot{V}_f$ to d, satisfying its demand. The remaining yet-unmatched valencies are an even number and an arbitrary pairing of the corresponding vertices in $\ddot{V}_f$ completes the matching M_f. □

Theorem 4.6. *Connected* 3-FEPRA *can be solved in $O(n^{2.5})$ time.*

Proof. We can construct $G_A = (V^{\circledast} \cup V_D, E')$, which has $O(n^2)$ edges and a linear number of vertices, in $O(n^2)$ time, with n the number of vertices in the input graph G. Checking whether G_A admits a perfect matching can be done in $O(\sqrt{|V^{\circledast} \cup V_D|}|E'|) = O(n^{2.5})$ time, using the algorithm due to Gabow [50], as introduced at the beginning. According to the characterization in Proposition 4.4 and the correspondence between perfect matchings in G_A and node assignments satisfying the characterization in Lemma 4.5, this decides whether G admits a connected planar 3-regular augmentation. By the same lemma, we obtain a node assignment A (induced by the matching in G_A and $\overline{A}$) that satisfies the parity condition, the matching condition, and the connectivity condition for each face. Using Lemma 4.3, we obtain in $O(n)$ time a node assignment A' that additionally satisfies the planarity condition for each

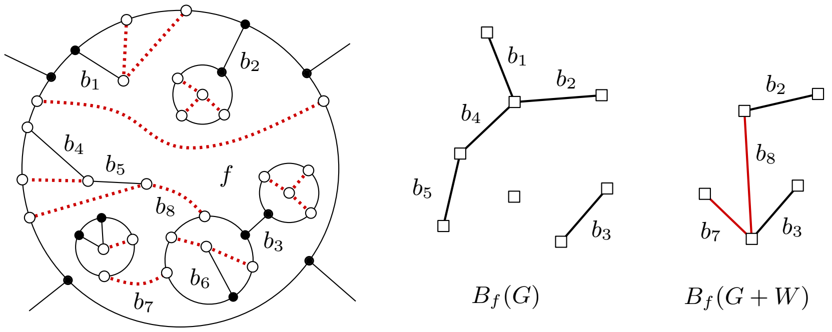


FIGURE 4.2: A face f of G (left) and its corresponding bridge forests $B_f(G)$ (middle) and $B_f(G+W)$ (right); solid (black) edges belong to G, dotted (red) edges form the augmentation W. The bridge b_6 is not incident to f and hence not contained in $B_f(G)$ and $B_f(G+W)$.

face. A corresponding connected planar realization of A' can then be obtained in $O(n)$ time by Proposition 4.2. □

4.1.2 Biconnected 3-Fepra

In this section we show that also biconnected 3-Fepra can be solved efficiently. Again, we first give a local characterization of node assignments admitting biconnected planar realizations and then construct an assignment graph whose perfect matchings correspond to such node assignments.

Let f be a face of G. We consider the *bridge forest* $B_f(G)$ of f with respect to G, which is constructed as follows. Remove all bridges from G. This yields exactly the biconnected components of G, as G has maximum degree 3. We call the biconnected components of G *blocks*. Note that deleting bridges in G does not change the faces in G, since bridges are only incident to one face. We create a node for each block that is incident to f and connect these nodes by an edge if and only if the corresponding blocks are connected by a bridge in G. We identify the nodes and the edges in $B_f(G)$ with the corresponding blocks and bridges in G. Observe that $B_f(G)$ contains a (connected) tree for each connected component of G incident to f.

For an augmented graph $G + W$, where W is a partial planar augmentation, we define the bridge forest $B_f(G+W)$ of f by only removing bridges of $G+W$. Note that $B_f(G+W)$ is still defined with respect to the face f, although f is a face of G, which might be split into several faces in $G+W$. An edge, a vertex, or a block in $G+W$ is incident to f in G if it is incident to a face f' of $G+W$ that arose by splitting f. Figure 4.2 shows an example of $B_f(G)$ and $B_f(G+W)$. Clearly, an augmentation W is connected if and only if the bridge forest $B_f(G+W)$ of each face f in G is a tree, and it is biconnected if and only if each bridge forest $B_f(G+W)$ consists of a single node. Observe further that each block in $G+W$ consists of blocks in G. In particular, each block corresponding to a leaf in $B_f(G+W)$ contains a block that corresponds to a leaf in $B_f(G)$.

Next, we study necessary and sufficient conditions for when a node assignment A admits a planar 3-regular realization W such that the resulting bridge forest $B_f(G+W)$ is a single node for each face f. Obviously, if there is more than one connected component incident to f, each of them must assign at least two valencies to f; if none is assigned, the realization will not be connected, if only one is assigned the single edge incident to this valency will form a bridge in $G+W$, and hence in $B_f(G+W)$. Additionally, each block that corresponds to a leaf in $B_f(G)$ must assign at least one valency, otherwise its incident bridge in $B_f(G)$ will remain a bridge in $B_f(G+W)$.

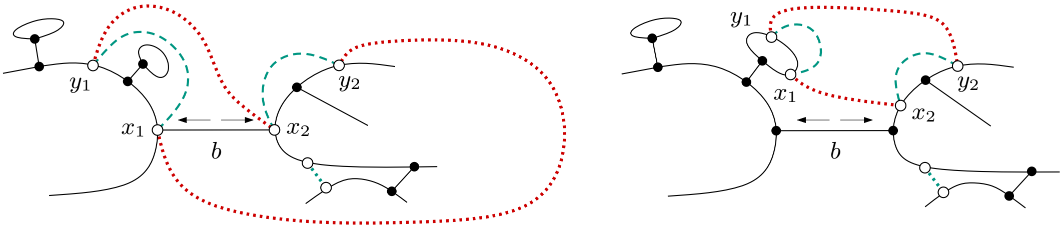


FIGURE 4.3: Illustration of the rewiring step in the proof of Proposition 4.8; on the left side x_1, x_2 are adjacent, on the right side x_1, x_2 are not adjacent. The dashed (green) edges are part of the augmentation W' and are replaced by the dotted (red) edges. Afterwards the edge b is not a bridge anymore.

Hence, the following *biconnectivity condition* is necessary for a node assignment A to admit a biconnected realization for a face f with z incident connected components.

Condition 4.7 (biconnectivity)**.** *(1) If $z > 1$, each connected component incident to f must have at least two valencies assigned to f.*

(2) Each block corresponding to a leaf in $B_f(G)$ must assign at least one valency to f.

We show that this condition is also sufficient; besides the parity and the matching condition.

Proposition 4.8. *There exists a biconnected planar realization W of A if and only if A satisfies for each face the parity, matching, biconnectivity, and planarity condition; W can be computed in $O(n)$ time.*

Proof. Clearly the conditions are necessary. Let A be a node assignment satisfying the parity condition, the matching condition, the biconnectivity condition, and the planarity condition for all faces of G. We prove the existence of a biconnected planar realization W for each face f individually. First, we ensure connectivity for f. If f is a face with $z > 1$ incident connected components, the bridge forest $B_f(G)$ of f is not connected and consists of z trees. Hence, we add a set W' of $z - 1$ edges to f such that $B_f(G + W')$ becomes connected. As biconnectivity condition (1) implies the connectivity condition, and the biconnectivity condition is preserved by adding edges in this way, this can be done by applying Phase 1 of Rule 3.14. Observe that $G+W'$ is still planar and contains the face f. For f it is now $z = 1$. The assignment A' induced by A on $G+W'$ still satisfies the biconnectivity condition as well as the parity condition, the matching condition, and the planarity condition, according to Lemma 3.15. Thus, in the following, we assume G is connected and consider only faces with $z = 1$.

Let f be a face with $z = 1$ and A a node assignment satisfying the parity condition, the matching condition, the biconnectivity condition, and the planarity condition for all faces. Then, according to Proposition 3.16, there exists a planar (not necessarily biconnected) realization W' of A. We show that it is possible to rewire some edges of W' that are embedded in f such that we obtain a biconnected planar realization of A for f, that is, no edge incident to f is a bridge.

We first observe that, if $B_f(G + W')$ is a single node, we are already done. Otherwise, let b be an edge in $B_f(G + W')$, this is, b corresponds to a bridge in $G + W'$ that is embedded in f. Since G is assumed to be connected, b is even a bridge in G. Starting from one side of b, we now traverse the facial cycle C_f of f in clockwise and counterclockwise direction until we find the first vertices x_1 and x_2 that are assigned to f. Note that these vertices lie on different sides of b, since we searched in different directions starting from b and biconnectivity condition (2) ensures that we find an assigned vertex before we traverse b again. Let f' denote the face incident to b

in $G + W'$. By the choice of x_1 as the first vertex along C_f with an incident edge in W', there is an edge x_1, y_1 in W' that is incident to f'. Similarly, there is an edge x_2, y_2 in W' incident to f'. Note that, since b is a bridge, also y_1 and y_2 lie on distinct sides of b. Hence, the only two vertices that might be adjacent in G are x_1 and x_2. In this case, it is $b = \{x_1, x_2\}$. If this is the case, we replace $\{x_1, y_1\}$ and $\{x_2, y_2\}$ by $\{x_1, y_2\}$ and $\{x_2, y_1\}$. Otherwise, we replace them by $\{x_1, x_2\}$ and $\{y_1, y_2\}$; see Fig. 4.3 for an illustration. In either case, b is not a bridge afterwards, and, moreover it can be seen that no new bridges are created by this rewiring step, since the two new edges together with any path from x_1 to y_1 and from x_2 to y_2 form a cycle. Thus, after such a step, $B_f(G+W')$ has fewer edges and iteratively applying such rewiring steps results in the desired augmentation W.

Concerning the running time, recall that, by applying Rule 3.14, $z-1$ edges can be added to each face in G in $O(n)$ time such that G becomes connected and the remaining node assignment A' still satisfies the parity, the matching, the biconnectivity and the planarity condition. A planar (possibly not biconnected) realization W' of A' for each face f in the connected graph G can be computed in linear time by Proposition 3.16. For the rewiring, we traverse the facial cycle of f once. At the beginning no bridge in $G + W'$ is marked. During the traversal, we always maintain the last vertex x_1 assigned to f we encountered, together with the edge $\{x_1, y_1\}$ in W' incident to it. Whenever we traverse a bridge that is not marked yet, we continue the traversal until we find the first vertex x_2 assigned to f after the bridge, together with the edge $\{x_2, y_2\}$ in W' incident to it. On the way from x_1 to x_2 we possibly traverse more than one (unmarked) bridge. After rewiring the edges $\{x_1, y_1\}$ and $\{x_2, y_2\}$, none of these bridges will be a bridge in $G + W'$ anymore. Hence, we mark these bridges as processed in order to ignore them when we traverse them the second time. We then apply the rewiring as described above in $O(1)$ time. After traversing the facial cycle of f in this way, no bridges are left, and we have found the claimed augmentation. Clearly, the running time is linear in the size of f. ☐

The following Lemma follows from Lemma 3.17 by showing that the reassignment that establishes the planarity condition preserves the biconnectivity condition. Similar to the proof of Lemma 4.3 (regarding the connectivity condition), it can be seen that the rewiring performed in the proof of Lemma 3.17 does not invalidate the biconnectivity condition. It reassigns vertices to other faces only if a face is assigned an insufficient number of additional valencies, which only shortens long paths of degree-2 vertices but never reduces the number of assigned valencies of a block corresponding to a leaf in $B_f(G)$ to zero or of a connected component below two.

Lemma 4.9. *Given a node assignment A that satisfies the parity, matching, and biconnectivity condition for all faces, a node assignment A' that additionally satisfies the planarity condition can be computed in $O(n)$ time.*

Proposition 4.8 together with Lemma 4.9 implies the following characterization.

Proposition 4.10. *A planar graph of maximum degree 3 with a fixed planar embedding admits a biconnected planar 3-regular augmentation if and only if it admits a node assignment that satisfies for all faces the parity, matching, and biconnectivity condition.*

Assignment Graph Construction for Biconnected 3-FEPRA. As before, we construct an assignment graph $G_A = (V^{\circledcirc} \cup V_D, E')$ such that the perfect matchings of G_A correspond to the characterized node assignments. To this end, we describe how to construct an individual edge set E'_f for each face f. In doing so, we again distinguish biconnected faces, for which

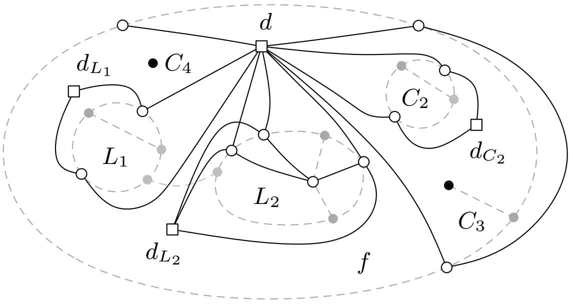


Figure 4.4: Graph (dashed lines) with one non-biconnected face f and seven biconnected faces. Degree-3 vertices are filled gray, preassigned vertices as filled black, remaining vertices with degree less than 3 as empty. The solid lines depict the corresponding assignment graph (dummy vertices as empty boxes).

the biconnectivity condition is trivially satisfied, and non-biconnected faces, for which we need to ensure the biconnectivity condition when constructing E'_f. In the context of biconnected 3-Fepra, a face is biconnected if and only if its bridge forest $B_f(G)$ consists of a single node. For biconnected faces, we again construct the set $E'_f = \binom{V_f^{\circledast}}{2} \setminus E$ as in Section 3.3.2.

The following description focuses on non-biconnected faces, which are the faces whose bridge forest $B_f(G)$ contains at least two nodes. Let f be such a face. We first determine the sets $\overline{V}_f$ and $\ddot{V}_f$ of preassigned and non-preassigned vertices and the preliminary node assignment $\overline{A}_f$. For each block that corresponds to a leaf L in $B_f(G)$ and does not contain a vertex in $\overline{V}_f$, we add a dummy vertex d_L with demand 1 and connect it to all vertices in $\ddot{V}_f \cap L$; this ensures biconnectivity condition (2). Moreover, for each connected component incident to f that contains a bridge this also ensures biconnectivity condition (1). For each connected component C that neither contains a bridge nor a vertex in $\overline{V}_f$, we add a dummy vertex d_C with demand 2 and connect it to all vertices in $\ddot{V}_f \cap C$. This completely ensures biconnectivity condition (1) for f. Note that the only connected component that does not contain a bridge but a vertex in $\overline{V}_f$ is an empty 3-cycle. An empty 3-cycle, however, provides even three vertices in $\overline{V}_f$, which clearly satisfies biconnectivity condition (1). Now let $\ddot{c}$ denote the number of valencies demanded by dummy vertices d_C and ℓ the number of valencies demanded by dummy vertices d_L. As in the construction for connected 3-Fepra, we further denote the number of valencies assigned to f by $\overline{A}_f$ by $\overline{a}$ and the maximum remaining demand of an indicator set in $\overline{A}_f$ by δ. To satisfy this remaining demand, at least δ additional vertices from $\ddot{V}_f$ need to be assigned to f (besides the $\ddot{c}+\ell+\overline{a}$ valencies that are already ensured). We thus create a dummy vertex d whose demand is set to $s = \max\{\delta, 0\}$ and connect d to all vertices in $\ddot{V}_f$. The vertices in $\ddot{V}_f$ have demand 1. If $s + \overline{a} + \ddot{c} + \ell$ is odd, we increase s by 1 to also guarantee the parity condition.

As in the connected case, we will see in the proof of the following lemma that the construction so far already assigns enough valencies to f such that assigning further vertices of $\ddot{V}_f$ to f never yields an indicator set whose demand is not satisfied. Hence, we add again all pairs of (possibly adjacent) vertices in $\ddot{V}_f$ as edges to E'_f.

For non-biconnected faces, the edges in E'_f are again only incident to dummy vertices and vertices in $\ddot{V}_f$. Figure 4.4 shows an example of an assignment graph; for clarity edges connecting vertices in $\ddot{V}_f$ are omitted for non-biconnected faces.

Lemma 4.11. *A perfect matching of G_A (together with $\overline{A}$) corresponds to a node assignment that satisfies the parity, matching, and biconnectivity condition for all faces, and vice versa.*

Proof. Let M be a perfect matching of G_A and let A denote the corresponding node assignment together with $\overline{A}$. We show that A satisfies the parity, matching and biconnectivity condition for all faces. For biconnected faces, A satisfies the biconnectivity condition by definition. The matching condition and the parity condition are satisfied since the set $E'_f \cap M$ is a perfect matching with respect to all vertices in $V_f^{\circledast}$, as we did not distinguish preassigned and non-preassigned vertices for the construction of E'_f.

Now consider a non-biconnected face f, for which $B_f(G)$ consists of at least two nodes. As in the connected case, the demand of s ensures that an even number of valencies is assigned to f, and hence A satisfies the parity condition. Moreover, the dummy vertices d_C and d_L explicitly ensure the biconnectivity condition. For the matching condition, recall that s was chosen such that the demands of all indicator sets consisting of vertices in $\overline{V}_f$ are satisfied. Thus, if there is an indicator set S whose demand is not satisfied, it must consist of vertices in $\ddot{V}_f$, and hence is a joker, a pair or a triangle. For a joker the necessary valency exists due to the parity condition. If S is a pair, the vertices of S share a block in $B_f(G)$. Since $B_f(G)$ contains at least two nodes and A satisfies the biconnectivity condition, at least one of the other nodes provides a further valency. It then follows from the parity condition that there is also a second additional valency. These two additional valencies satisfy the demand of S, unless S is part of a 3-cycle C and the second additional valency is provided by the remaining vertex in C. As in the connected case, this situation, however, does not occur. If C is empty, its vertices are preassigned, contradicting $S \subseteq \ddot{V}_f$. If C is not empty, at least one of its vertices is incident to an edge in M that is embedded in the other face distinct from f (otherwise there would be a dummy vertex in this face that is supposed to ensure the biconnectivity condition and whose demand is not satisfied). Hence, the second valency cannot be provided by a common neighbor of the vertices in S. Finally, if S is a triangle, its vertices are preassigned, contradicting again $S \subseteq \ddot{V}_f$.

Conversely, let A be a node assignment that satisfies, for each face f, the parity condition, the matching condition, and the biconnectivity condition. We construct a perfect matching in G_A that together with $\overline{A}$ induces A. We note again that, as in the connected case, the preassignment $\overline{A}$ follows from necessary conditions, and thus, A assigns preassigned vertices in the same unique way as $\overline{A}$. For a biconnected face f, a matching $M_f \subseteq E'_f$ that satisfies exactly the demands of the vertices assigned to f exists according to Proposition 3.9.

For a non-biconnected face f, we seek a matching $M_f \subseteq E'_f$ that further satisfies the demand of the dummy vertices associated with f. However, the conditions satisfied by A imply, as in the connected case, that enough vertices of $\ddot{V}_f$ are assigned to f such that we can match those to the dummy vertices satisfying their demand. The choice of the demand of the dummy vertices further implies that the number of unmatched vertices in $\ddot{V}_f$ assigned to f is even. Those can be paired arbitrarily in G_A to eventually form M_f. □

Theorem 4.12. *Biconnected* 3-FEPRA *can be solved in* $O(n^{2.5})$ *time.*

Proof. We can construct $G_A = (V^{\circledast} \cup V_D, E')$, which has $O(n^2)$ edges and a linear number of vertices, in $O(n^2)$ time, with n the number of vertices in the input graph G. Checking whether G_A admits a perfect matching can be done in $O(\sqrt{|V^{\circledast} \cup V_D|}|E'|) = O(n^{2.5})$ time, using the algorithm due to Gabow [50], as introduced at the beginning. According to the characterization in Proposition 4.10 and the correspondence between perfect matchings in G_A and node assignments satisfying the characterization in Lemma 4.11, this decides whether G admits a biconnected planar 3-regular augmentation. By the same lemma, we obtain a node assignment A (induced by the matching in G_A and $\overline{A}$) that satisfies the parity condition, the matching condition and the biconnectivity condition for each face. Using Lemma 4.9, we obtain in $O(n)$ time a node assignment A' that additionally satisfies the planarity condition for each face. A corresponding biconnected planar realization of A' can then be obtained in $O(n)$ time by Proposition 4.8. □

4.2 Improving the Running Times

In this section, we show that the running time of the above algorithms for c-connected 3-FEPRA, with $c = 0, 1, 2$, can be reduced to $O(n^{1.5})$. To achieve this, we modify the previous assignment graphs such that their perfect matchings still induce the same node assignments as before but the modified assignment graphs have size $O(n)$ (and can also be constructed in time $O(n)$) instead of $O(n^2)$. Since the matching algorithm of Gabow [50] runs in time $O(\sqrt{n}m)$, this immediately yields the claimed running time.

Recall from the previous sections that, for the construction of an assignment graph $G_A = (V^{\circledast} \cup V_D, E')$, we used the following scheme. For each face f of G we defined an edge set E'_f of edges in G_A, distinguishing two types of faces. The first type contained all faces for which the subgraph H_f induced by the vertices in G incident to the face f already provided the required (bi)connectivity. For $c = 1, 2$, we called these faces (bi)connected. For $c = 0$, all faces were of this type, since there is no connectivity required. In the following, we call the faces of this type *connectivity-satisfied*, referring to all faces if $c = 0$, to connected faces if $c = 1$, and to biconnected faces if $c = 2$. The second type contained the so-called non-(bi)connected faces, which only occurred for $c = 1, 2$ and for which we needed to enforce the required (bi)connectivity by assigning and matching the vertices in a special way. Analogous to the connectivity-satisfied faces, we now call the faces of the second type *connectivity-non-satisfied*. Whenever we consider the faces of only one problem variant ($c = 1$ or $c = 2$), we also still use the corresponding more descriptive terms of (non-)connected and (non-)biconnected faces.

For connectivity-satisfied faces the edge set E'_f just consisted of the complement edges of H_f, which directly implied the parity and matching condition but yielded size $O(n^2)$ for the complete assignment graph. Recall that we did not distinguish preassigned and non-preassigned vertices for connectivity-satisfied faces. For connectivity-non-satisfied faces, we created a certain set of dummy vertices connected to non-preassigned vertices incident to f (denoted by $\ddot{V}_f$) and the vertices in $\ddot{V}_f$ among each other. Adding the edge set $\binom{\ddot{V}_f}{2}$ to E'_f again yielded size $O(n^2)$ for G_A. Note that the number of dummy vertices for each connectivity-non-satisfied face f and their incident edges was only linear in the size of f

To overcome the quadratic size of the assignment graphs, for connectivity-non-satisfied faces, we thus need to find a way to allow the assignment of an arbitrary even number of non-preassigned vertices by only adding a linear number of edges to E'_f (instead of adding the set $\binom{\ddot{V}_f}{2}$). For connectivity-satisfied faces, we need to model the parity and the matching condition by a linear number of edges. For both types of faces we will achieve this by the help of a new linear *partial construction* replacing the set of complement edges of H_f (for connectivity-satisfied faces) and the set $\binom{\ddot{V}_f}{2}$ (for connectivity-non-satisfied faces) in the previous constructions, respectively. Furthermore, we will consider preassigned and non-preassigned vertices also for connectivity-satisfied faces, ensuring the remaining necessary conditions by the help of dummy vertices. We start with the description of the new partial construction. The sets $\overline{V}$, $\overline{V}_f$, $\ddot{V}$, $\ddot{V}_f$ and the preliminary node assignments $\overline{A}$, $\overline{A}_f$ are defined as in Section 4.1. Note that, as a consequence

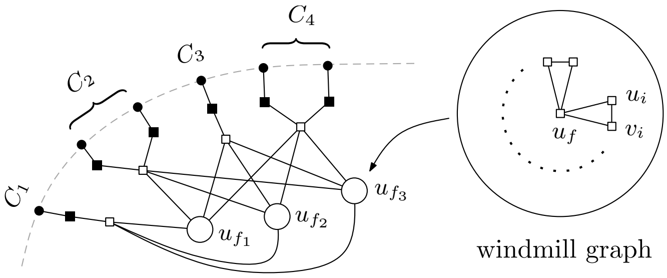


Figure 4.5: Even-Vertices-No-Critical-Pair construction using windmill graphs. Vertices in $\ddot{V}_f$ are filled disks, shadow vertices are filled boxes, remaining dummy vertices are empty boxes.

of replacing the complement edges of H_f for connectivity-satisfied faces, the vertices in $\overline{V}$ will no longer be involved in G_A. Hence, the vertex set of G_A is now $\ddot{V} \cup V_D$, instead of $V^{\circledast} \cup V_D$.

Partial Construction. We present a partial construction that will appear as a substructure in the new sparse assignment graphs allowing the assignment of an arbitrary even number of vertices from $\ddot{V}_f$ to a face f by using only a linear number of edges. In addition, this construction also allows to prevent the assignment of a single *critical pair* from $\ddot{V}_f$, which we will see becomes necessary for connectivity-satisfied faces in order to satisfy the matching condition. A pair of adjacent (degree-2) vertices in $\ddot{V}_f$ is *critical* unless there are already enough valencies assigned to f (due to preassigning or the demand of dummy vertices) to satisfy its demand, or it is adjacent to further degree-2 vertices in $\ddot{V}_f$ without being part of a 3-cycle. A pair in $\ddot{V}_f$ that is not critical is a *non-critical pair*. Note that assigning, to a face f, a single non-critical pair without further (preassigned) valencies (that is, a pair that is adjacent to further degree-2 vertices in $\ddot{V}_f$ without being part of a cycle) still violates the matching condition. Nevertheless, we will see in Lemma 4.14 that for such non-critical pairs the matching condition can be easily established by reassigning valencies from the pair and its (degree-2) neighbors. Recall from Section 4.1 that, for connectivity-non-satisfied faces, due to the parity and (bi)connectivity condition, there are always enough vertices preassigned or required by dummy vertices to satisfy the demand of all indicator sets in $\ddot{V}_f$. Thus, for these faces all pairs in $\ddot{V}_f$ are non-critical.

Even-Vertices-No-Critical-Pair construction. Let $\mathcal{C} = \{C_1, \dots, C_r\}$ be a partition of $\ddot{V}_f$ into sets of size at most 2. We call the elements of $\mathcal{C}$ *clusters*. We seek a construction that allows to assign to f an arbitrary even number of vertices in $\ddot{V}_f$ except for assigning exactly two vertices that belong to the same cluster. To achieve this, we proceed as follows. We create for each vertex $v \in \ddot{V}_f$ a corresponding shadow vertex v', and for each cluster C_i a corresponding cluster vertex c_i. We connect each vertex to its shadow vertex, and we connect each shadow vertex to its cluster vertex. The vertices in $\ddot{V}_f$ and their shadow vertices have demand 1, each cluster vertex c_i has demand $|C_i|$. Additionally, we add three identical *windmill* graphs u_{f1}, u_{f2}, u_{f3} (consisting of dummy vertices), each constructed as follows. A vertex u_f with demand $x_f = 2 \cdot \lfloor |\mathcal{C}|/2 \rfloor$ forms the *center*. Furthermore, we create $x_f/2$ pairs $u_1, v_1, \dots, u_{x_f/2}, v_{x_f/2}$ of dummy vertices with demand 1, each, and connect each such vertex to the center u_f. Afterwards, we connect the two vertices of each pair; see the sketch of a windmill graph in Figure 4.5. Finally, the centers of the windmill graphs are connected to each cluster vertex, as shown in Figure 4.5. We identify the centers with their corresponding windmill graphs thus also denoting the centers by u_{f1}, u_{f2}, u_{f3}.

Proposition 4.13. *A matching in the Even-Vertices-No-Critical-Pair construction for a face f that satisfies the demand of all dummy vertices induces a node assignment for f that assigns* *to f an even number of vertices in $\ddot{V}_f$, except for exactly two vertices that belong to the same cluster, and vice versa.*

Proof. Let M_f be an arbitrary matching in the Even-Vertices-No-Critical-Pair construction that satisfies the demand of all dummy vertices. The total demand of the dummy vertices is even, since the sum of the demand of the vertices in the windmill graphs is even, and for each cluster C in $\mathcal{C}$, we have $|C|$ shadow vertices with demand 1 and a cluster vertex with demand $|C|$, which sums to a demand of $2|C|$. Thus, certainly an even number of vertices in $\ddot{V}_f$ is incident to edges in M_f. Moreover, suppose that $C \in \mathcal{C}$ is a cluster of size 2. If M_f contains the two edges incident to the vertices in C, then the demand of the cluster vertex c of C is not satisfied by the edges incident to the shadow vertices of C, and hence M_f contains one of the three edges $cu_{f1}, cu_{f2}, cu_{f3}$, say cu_{f1}. Since the demand of u_{f1} is even and the vertices u_i, v_i of each pair in the windmill graph are either both matched to u_{f1} or both not matched to u_{f1}, M_f contains another edge $c'u_{f1}$ to a cluster vertex c' distinct from c. Let C' denote the corresponding cluster. This implies that M_f cannot contain all edges between c' and the shadow vertices of C'. Let $v \in C'$ be a vertex whose shadow vertex v' is not matched to c'. It follows that M_f contains the edge vv', and hence also v (which is not in C) is assigned to f by the construction. It follows that, with this construction, it is not possible to assign exactly two vertices to f that belong to the same cluster.

Conversely, let $X \subseteq \ddot{V}_f$ be an arbitrary even number of vertices such that $X \notin \mathcal{C}$, and let A be a node assignment that assigns exactly the vertices in X to f. We show that there exists a matching M_f in the Even-Vertices-No-Critical-Pair construction for f that matches (besides the dummy vertices) exactly the vertices in X and satisfies the demand of all dummy vertices. In a first step, we pair the vertices in X up into $X_1, \dots, X_k$, such that none of the pairs X_i is a cluster in $\mathcal{C}$. This is always possible. Next, we construct a conflict graph on the pairs X_i. We say that two pairs X_i and X_j are in conflict if there exists a cluster $C \in \mathcal{C}$ such that both $X_i \cap C$ and $X_j \cap C$ are non-empty, that is, if they contain vertices from the same cluster. Since the clusters have size at most 2, it follows that the conflict graph on the pairs has maximum degree 2, and hence is 3-colorable. We fix an arbitrary 3-coloring of the pairs with colors $\{1, 2, 3\}$. We now construct the matching M_f as follows. For each vertex $v \in \ddot{V}_f \setminus X$, we add to M_f the edge connecting the shadow vertex v' to its cluster vertex. That is, the corresponding vertex v is not matched. We treat the pairs $X_1, \dots, X_k$ as follows. Let $X_i = \{x_i, y_i\}$, let x'_i and y'_i be the corresponding shadow vertices and let c_i and c'_i denote the cluster vertices of x_i and y_i, respectively. Note that, by the choice of the pairs, we have that x_i and y_i belong to distinct clusters, and hence $c_i \neq c'_i$. We add to M_f the edges $x_i x'_i$, $c_i u_{f\ell}$, $y_i y'_i$ and $c'_i u_{f\ell}$, where ℓ denotes the color of pair X_i. Note that, due to the coloring of the pairs, no edge between a windmill graph and a cluster vertex is added twice. Moreover, by construction, the windmill centers have even degree in M_f. If the demand of a center is not yet satisfied, we further add the missing number of edges, which is even, to M_f by matching the center $u_{f\ell}$ with vertices u_i and v_i of the same pair in the windmill graph. The remaining vertices u_i, v_i of pairs in the windmill graph are matched to each other. This satisfies the demand of all vertices in the windmill graphs. The demand of the cluster vertices c_i is satisfied, as for each vertex v in C_i the matching M_f contains either the edge $c_i v'$ (if $v \notin X$) or an edge $c_i u_{f\ell}$ to one of the windmill centers (if $v \in X$). Finally, the demand of the shadow vertices is clearly satisfied. ☐

Sparse Assignment Graph Construction. For connectivity-non-satisfied faces, all pairs in $\ddot{V}_f$ are non-critical. Thus, for these faces, we define the clusters in the Even-Vertices-No-Critical-Pair construction such that each vertex in $\ddot{V}_f$ forms its own cluster. Replacing in the assignment graph of Section 4.1, for each connectivity-non-satisfied face, the set $\binom{\ddot{V}_f}{2}$ by the Even-Vertices-No-Critical-Pair construction then obviously yields a new edge set E'_f of linear size for each connectivity-non-satisfied face such that the perfect matchings in the resulting assignment graph still correspond to the same node assignments as before. In this case the Even-Vertices-No-Critical-Pair construction can be even simplified without changing its behavior, as follows. Since each cluster consist of only one vertex, we can skip the shadow vertices as well as the cluster vertices instead connecting each vertex in $\ddot{V}_f$ directly to the centers of the windmill graphs. At a second glance, we see that also one windmill graph suffices to model the required behavior of the Even-Vertices-No-Critical-Pair construction in this case.

For connectivity-satisfied faces, no (bi)connectivity condition needs to be explicitly ensured by the help of dummy vertices. Hence, for each connectivity-satisfied face f, we construct only one dummy vertex d connected to all vertices in $\ddot{V}_f$ that ensures the parity condition and the matching condition for the indicator sets in $\overline{V}_f$. The demand of d is $s := \max\{\delta, 0\}$, with δ the maximum remaining demand of an indicator set in $\overline{V}_f$ after counting the valencies already provided by vertices in $\overline{V}_f$ (as in Section 4.1). If s is odd, we set the demand of d to $s + 1$. Note that biconnected faces (for $c = 2$) are never incident to preassigned vertices. Thus, for these faces, it is $s = 0$, that is, we can even omit the dummy vertex d. Finally, we seek to prevent the assignment of single critical pairs from $\ddot{V}_f$. To this end, we define the clusters in the Even-Vertices-No-Critical-Pair construction such that each critical pair forms a cluster while the remaining vertices in $\ddot{V}_f$ form singleton clusters. In the following we show how critical pairs in $\ddot{V}_f$ can be identified depending on the problem variant. Recall that a pair in $\ddot{V}_f$ is critical unless its demand is already satisfied by preassigned valencies or valencies demanded by the dummy vertex d, or it is incident to further degree-2 vertices in $\ddot{V}_f$ without being part of a 3-cycle.

2-connected (or biconnected) 3-FEPRA : The critical pairs for a biconnected face f are exactly the pairs in $\ddot{V}_f$ that form a maximal path of length 2 in $\ddot{V}_f$. Each pair in $\ddot{V}_f$ that is incident to further degree-2 vertices in $\ddot{V}_f$ but is no part of a 3-cycle is non-critical by definition. Note that no vertices in $\ddot{V}_f$ are part of a 3-cycle, since f is incident to a single block and if this block was a 3-cycle, the side containing f would be empty and the vertices of the 3-cycle would be preassigned to the face at the opposite side.

Vice versa, since f is incident to a single block, no vertices are preassigned and no dummy vertices are added. Hence, there are no valencies already assigned to f that might satisfy the demand of a single pair in $\ddot{V}_f$. Consequently, all pairs in $\ddot{V}_f$ that are not non-critical due to their neighbors, are critical.

1-connected (or connected) 3-FEPRA : Each connected face f is incident to a single connected component, and thus, by the same argument as above, no vertex in $\ddot{V}_f$ is part of a 3-cycle. However, there may be preassigned vertices and a dummy vertex d. If there is at least one valency preassigned to f, the demand of d or further preassigned vertices guarantee a second valency. Hence, the demand of all possible pairs in $\ddot{V}_f$ is satisfied and there are no critical pairs at all. Otherwise, if no valency is preassigned, the critical pairs are again exactly the pairs in $\ddot{V}_f$ that form a maximal path of length 2.

0-connected (or arbitrary) 3-FEPRA : In this case, no (bi)connectivity condition needs to be satisfied for the faces. This is, a face f may be incident to several connected components and, in particular, the vertices in $\ddot{V}_f$ may also form 3-cycles. Hence, besides the assignment of critical pairs as defined above, we further need to prevent the assignment of (non-empty) 3-cycles whose demand is not satisfied. We reduce this to preventing again critical pairs by extending the set $\overline{V}$ of preassigned vertices as follows. Let $C = xyz$ be a non-empty 3-cycle incident to f and let f' denote the face at the other side of C. Recall that, since C is non-empty, both f and f' are incident to further vertices. Then, if G contains an even number of vertices (otherwise a 3-regular augmentation does not exist), one side of C contains an odd number of vertices. The vertices on this side thus cannot be independently augmented to a 3-regular subgraph, they need at least one valency from C. Consequently, if the odd side of C is the side containing f, at least one vertex of C must be assigned to f. Since the vertices of C are completely symmetric, we may assume that x is assigned to f, thus fixing its assignment yielding $x \in \overline{V}_f$. This is, we additionally preassign one vertex per non-empty 3-cycle in G. The remaining two vertices of the 3-cycle then form a pair in $\ddot{V}_f$ and $\ddot{V}_{f'}$. Now we distinguish four cases, depending on the number of valencies preassigned to f, in which we identify different critical pairs in $\ddot{V}_f$.

Case 1: If there are at least three valencies preassigned to f, the demand of d or further preassigned vertices guarantees a fourth valency, such that the demand of each pair in $\ddot{V}_f$ is satisfied, even if it is part of a 3-cycle whose remaining vertex is preassigned to f. Hence, there are no critical pairs at all.

Case 2: If there are only two valencies preassigned to f and none of them is part of a 3-cycle, again the demand of each pair in $\ddot{V}_f$ is satisfied and there are no critical pairs. If at least one valency of the two preassigned valencies is part of a (non-empty) 3-cycle (note that d then requires no further valencies), the remaining vertices of each 3-cycle that has its third vertex preassigned to f becomes a critical pair.

Case 3: If there is only one valency preassigned to f, this is provided by either a vertex x of a (non-empty) 3-cycle C or a vertex x that is only incident to f and adjacent to two degree-3 vertices. In the first case, we need to prevent the dummy vertex d from matching one of the remaining vertices y, z of C, since this would still not satisfy the remaining demand δ of the preassigned vertex x. Hence, we connect d only to the vertices in $\ddot{V}_f \setminus \{y, z\}$. In this way, the demand of x and all pairs in $\ddot{V}_f$ is satisfied, apart from $\{y, z\}$. The pair $\{y, z\}$ then is the only critical pair. In the second case, observe that no preassigned vertex of a non-empty 3-cycles is assigned to f. Hence, assigning to f the remaining vertices of these 3-cycles, as well as any other pair in $\ddot{V}_f$, is feasible, since the demand of d ensures a further valency that together with x satisfies the demand of any pair. Thus, there are no critical pairs at all.

Case 4: If no valency is preassigned to f, the face f is incident to only blocks. By similar arguments as in the case of biconnected 3-FEPRA, the critical pairs are exactly the pairs in $\ddot{V}_f$ that form a maximal path of length 2 (possibly being part of a non-empty 3-cycle whose third vertex is preassigned to a different face).

Replacing in the assignment graph of Section 3.3.2 (for $c = 0$) and Section 4.1 (for $c = 1, 2$), respectively, for each connectivity-satisfied face, the complement edges of H_f by the Even-Vertices-No-Critical-Pair construction and (possibly) the dummy vertex d finally yields a new set E'_f of linear size for each connectivity-satisfied face. Note that, for faces with no critical pairs, we can again simplify the Even-Vertices-No-Critical-Pair construction as described before. In the following, we will see that the perfect matchings in the resulting assignment graph G_A correspond to node assignments that satisfy for each connectivity-satisfied face the parity condition, the (bi)connectivity condition (if $c = 1, 2$) and a *relaxed matching condition*, which we introduce now. For connectivity-non-satisfied faces, we have already seen that replacing $\binom{\ddot{V}_f}{2}$ by the Even-Vertices-No-Critical-Pair construction yields an assignment graph whose perfect matchings correspond to the same node assignments (for connectivity-non-satisfied faces) as in Section 4.1. Hence, for these faces the corresponding node assignments already satisfy the parity condition, the (bi)connectivity condition and the matching condition.

Relaxed Matching Condition. We consider the following relaxation of the matching condition for a node assignment A of G. We say that A satisfies the *relaxed matching condition* for a face f if it either (i) satisfies the matching condition for f, or (ii) it assigns to f a single non-critical pair. The following lemma shows that it suffices to find assignments that satisfy the relaxed matching condition.

Lemma 4.14. *Given a node assignment A that satisfies the parity, relaxed matching, and, possibly, some connectivity condition for all faces, there also exists a node assignment A' that satisfies the matching condition instead of the relaxed matching condition, and vice versa. The assignment A' can be computed in $O(n)$ time.*

Proof. Clearly, an assignment that satisfies the matching condition trivially satisfies the relaxed matching condition. Conversely, we show that, if A is an assignment that satisfies the relaxed matching condition, then we can modify it to satisfy the matching condition while preserving all other conditions. Let A be an assignment that satisfies for f the relaxed matching condition but not the matching condition. Then A assigns to f exactly two adjacent vertices u and v of $\ddot{V}_f$ that form a non-critical pair, that is, the pair $\{u, v\}$ is not part of a 3-cycle but adjacent to further degree-2 vertices in $\ddot{V}_f$ that are not assigned to f. Thus, without loss of generality, v has a neighbor w in $\ddot{V}_f$ that is not assigned to f but to the other face f' to which u and v are incident, and u is not adjacent to w (otherwise u, v, w would form a triangle, and $\{u, v\}$ would be critical).

We modify A by reassigning v to f' and w to f. Clearly, this preserves the parity condition and establishes the matching condition for f. It further preserves (or establishes) the matching condition for f', since v is a joker in f'. Finally, since v and w belong to the same block (and thus to the same connected component), also possible connectivity conditions for f and f' are preserved. These modifications can be obviously applied in linear time and eventually yield the desired assignment. □

Lemma 4.15. *A perfect matching of G_A (together with $\overline{A}$) corresponds to a node assignment that satisfies parity, relaxed matching, and (bi)connectivity condition (for $c > 0$) for all faces, and vice versa.*

Proof. For connectivity-non-satisfied faces (which only occur for $c = 1, 2$), the perfect matchings in G_A (together with $\overline{A}$) correspond to the node assignments that satisfy the parity condition, the matching condition, and the (bi)connectivity condition, according to Section 4.1 and the choice of the clusters for the Even-Vertices-No-Critical-Pair construction.

Hence, in the remainder of this proof, we consider a connectivity-satisfied face f. Let A denote the assignment that is induced by a perfect matching M and the preliminary assignment $\overline{A}$. Since f is a connectivity-satisfied face, all possible conditions on (bi)connectivity are satisfied by definition. The parity condition as well as the demand of each indicator set in $\overline{V}_f$ is satisfied due to the choice of s. Hence, in order to prove the relaxed matching condition, we finally show that the demand of each indicator set S that possibly arises by additionally assigning vertices from $\ddot{V}_f$ to f is satisfied, unless S is a non-critical pair. Recall from Section 4.1 that for $c = 1, 2$ the vertices of an indicator set are either in $\overline{V}$ or in $\ddot{V}$. Hence, for $c = 1, 2$, all vertices of S are in $\ddot{V}_f$, and thus, S is either a joker or a pair. For $c = 0$, however, S may also be a triangle consisting of a vertex $x \in \overline{V}_f$ and two vertices $y, z \in \ddot{V}_f$, due to the extended definition of $\overline{V}$.

If S is a joker, its demand is satisfied since we already know that A satisfies the parity condition. If S is a critical pair, Proposition 4.13 ensures that there are at least two further valencies from $\ddot{V}_f$ assigned to f. Due to the construction (including the identification of critical pairs and the definition of $\overline{V}$ for $c = 0$) these additional valencies do not form a 3-cycle with S, and thus satisfy the demand of S. For a non-critical pair there is nothing to show. If S is a triangle (for $c = 0$), by construction, one valency of S is preassigned. If the remaining valencies form a non-critical pair (that is, there are at least three valencies preassigned), then the demand of S is satisfied. Otherwise, if the remaining valencies form a critical pair (that is, there are at most two valencies preassigned), Proposition 4.13 again ensures two further valencies besides S and the parity condition ensures another valency. Together these three additional valencies satisfy the demand of S.

Conversely, let A be a node assignment (including $\overline{A}$) that satisfies the parity condition, the relaxed matching condition, and the (bi)connectivity condition (if $c = 1, 2$). We construct for f a matching $M_f \subseteq E'_f$ satisfying exactly the demands of all vertices in G_A that are assigned to f and all dummy vertices associated with f. First recall that there are no dummy vertices ensuring the (bi)connectivity condition in connectivity-satisfied faces. Since critical and non-critical pairs are defined only in $\ddot{V}_f$, A provides enough valencies (also possibly from $\ddot{V}_f$) to satisfy the demand of the preassigned indicator sets in $\overline{V}_f$ (although A only satisfies the relaxed matching condition). Since A also satisfies the parity condition, there are finally enough valencies from $\ddot{V}_f$ assigned to f that can be matched to v_f satisfying its demand. To satisfy the demand of the dummy vertices in the Even-Vertices-No-Critical-Pair construction, we add edges to M_f as described in the proof of Proposition 4.13. This is obviously possible due to the choice of the clusters, since A satisfies the parity and the relaxed matching condition. □

Theorem 4.16. *Arbitrary* 3-Fepra *can be solved in $O(n^{1.5})$ time.*

Theorem 4.17. *Connected* 3-Fepra *can be solved in $O(n^{1.5})$ time.*

Theorem 4.18. *Biconnected* 3-Fepra *can be solved in $O(n^{1.5})$ time.*

Proof. We can construct the new sparse assignment graph $G_A = (\ddot{V} \cup V_D, E')$, which has only a linear number of edges and a linear number of dummy vertices, in $O(n)$ time (instead of $O(n^2)$ time), with n the number of vertices in the input graph G. Checking whether G_A admits a perfect matching can be thus done in $O(\sqrt{|\ddot{V} \cup V_D|}|E'|) = O(n^{1.5})$ time, using the algorithm due to Gabow [50]. According to Proposition 3.18, 4.4, 4.10 (characterizations), Lemma 4.14 and Lemma 4.15 this decides whether G admits the required planar 3-regular augmentation. By the latter lemma we obtain a node assignment A (induced by the matching in G_A and $\overline{A}$) that satisfies the parity condition, the relaxed matching condition and the (bi)connectivity condition for each face. Using Lemma 4.14 and one of the Lemmas 3.17, 4.3, or 4.9, we obtain in $O(n)$ time a node assignment A' that satisfies the (non-relaxed) matching condition instead, and additionally the planarity condition for each face. A corresponding planar realization of A' can then be obtained in $O(n)$ time by Proposition 3.16, Proposition 4.2, or Proposition 4.8. □

4.3 Complexity of Triconnected 3-FEPRA

We contrast the results from the previous section by showing that triconnected 3-FEPRA is NP-complete. Altogether this completely settles the complexity of c-connected 3-FEPRA.

Theorem 4.19. *Triconnected 3-*FEPRA *is NP-complete, even if the input graph is biconnected.*

Proof. First note that triconnected 3-FEPRA is in NP since, given a planar graph G with a fixed embedding, we can guess a set $W \subseteq \binom{V}{2}$ of non-edges of G and then test efficiently whether the graph $G+W$ is 3-regular, planar, and triconnected, and that W respects the given embedding of G (the latter can be checked using an algorithm due to Angelini et al. [5]). We prove NP-hardness by reducing from the monotone planar 3-satisfiability problem (MONOTONEPLANAR3SAT), which is known to be NP-hard [29]. It is a special variant of PLANAR3SAT, which we already used in Section 3.1 for the hardness proof of 3-PRA.

A monotone planar 3SAT formula is a 3SAT formula whose clauses either contain only positive or only negated literals and whose variable–clause graph is planar. A *monotone rectilinear representation* of a monotone planar 3SAT formula is a drawing of the variable–clause graph such that the variables correspond to axis-aligned rectangles on the x-axis and clauses correspond to non-crossing three-legged “combs” above the x-axis if they contain positive variables and below the x-axis otherwise; see Fig. 3.1. An instance of MONOTONEPLANAR3SAT is a monotone rectilinear representation of a monotone planar 3SAT formula φ. We now construct a biconnected graph G_φ with a fixed planar embedding that admits a planar 3-regular triconnected augmentation if and only if φ is satisfiable.

Similar to the proof of Theorem 3.1, the graph G_φ consists of so-called *gadgets*, which are subgraphs that represent the variables, literals, and clauses of φ. The reduction is illustrated in Fig. 4.6. For each gadget, we will argue that there are only a few ways to augment it to be 3-regular, triconnected and planar. Again, our construction connects variable gadgets corresponding to neighboring variables in the layout of the variable–clause graph of φ. Hence G_φ is always connected. Additionally, we identify the left boundary of the leftmost variable gadget with the right boundary of the rightmost variable gadget. In the figure vertices with degree less than 3 are highlighted by empty disks. All bends and junctions of line segments represent vertices of degree at least 3. Vertices of degree greater than 3 are actually modeled by small cycles of vertices of degree 3, as indicated in the left of Fig. 4.6. The (black) solid line segments between adjacent vertices represent the edges of G_φ; the (red) dotted line segments represent non-edges of G_φ that are candidates for an augmentation of G_φ. Gaps in the thick black line segments of the variable gadgets indicate positions where further subgraphs (which we will describe in more detail in the following) can be plugged in, depending on the number of clauses that contain the literal.

Each variable gadget consists of two symmetric parts, which correspond to the two literals. These literal (sub)gadgets are separated by thick horizontal edges. The degree-2 vertex u is incident to both literal gadgets. The thin triangle at the right side is called the *parity triangle*. Each literal gadget contains a subgraph that is attached in only two vertices to the horizontal edges separating the literals, and thus, induces a separator of size 2. We call this subgraph the *literal body.* The literal body can be considered as a path of smaller subgraphs (drawn with thin black edges) connected by thick black edges. The thin subgraphs can be characterized as a triangle at the front side (*front triangle*) that is based on another triangular shaped subgraph (*triangle basement*) and further oppositely placed *pairs of triangles.* In Fig. 4.6, we exemplarily marked a front triangle with its triangle basement and a pair of triangles. In the construction, the number of pairs of triangles in the literal body corresponds to the number of clauses containing the literal. Note that, without loss of generality, we may assume that each literal appears in at least one clause. The necessary number of pairs of triangles can be plugged in at the gaps. The corresponding clauses are attached to the outer boundary of the literal gadget, as exemplarily shown in Fig. 4.6. Each attached clause thereby requires a pair of adjacent degree-2 vertices at the boundary of the literal gadget, which are thus incident to the literal gadget and to the clause gadget. We call the corresponding valencies the *boundary valencies* of the literal gadget. Thus, each literal gadget has twice as many boundary valencies as clauses contain the literal.

Consider the graph G'_φ that we obtain by deleting the literal bodies and contracting the parity triangles and the degree-2 vertices. We claim that G'_φ is triconnected. This is true since (a) the subgraph of G'_φ induced by the variable gadgets is triconnected and (b) each subgraph induced by a clause gadget is also triconnected and is attached in twelve vertices to the subgraph of the variable gadgets. Hence, a 3-regular triconnected augmentation of G_φ only needs to care for the connectivity at the literal bodies and the parity triangles. Note that G_φ is already biconnected, since it is obtained from the triconnected graph G'_φ by subdividing edges, replacing degree-2 vertices by (parity) triangles and adding paths of biconnected subgraphs (literal bodies) between existing endpoints. In the following, we call a 3-regular, triconnected, planar augmentation a *valid* augmentation. We show two properties of G_φ:

(P1) Let W denote a valid augmentation and let x be a variable gadget. Then, for at least one literal gadget in x, the augmentation W assigns all boundary valencies to the incident literal face.

(P2) Given a literal L of a variable x and a (sub)set of clauses containing L, there exists a set of augmenting edges embedded in a planar way in the variable gadget corresponding to x such that all demands in the variable gadget of x are satisfied apart from those of the boundary vertices that are incident to the given clauses.

We start with (P1). Consider the exemplary variable gadget in Fig. 4.6. The valency of u is incident to both literal (sub)gadgets, and hence, is either assigned to x or $\neg x$ by a valid augmentation. Without loss of generality, assume that u is assigned to x. The opposite case is symmetric. The two degree-2 vertices in the triangle basement in $\neg x$ are thus connected, since the inner face of the literal body provides no further valencies. Let ℓ denote the number of clauses containing $\neg x$. The outer face of the literal body of $\neg x$ is incident to $2(2\ell + 1)$ valencies; $2\ell + 1$ stem from the triangles at the literal body, 2ℓ are boundary valencies and one additional valency is placed at the triangle to the right. We argue that the valencies at the triangles of the literal body of $\neg x$ are not connected to each other by a valid augmentation. This is true since such an edge would immediately induce a subgraph that is separated from the rest by only two vertices; namely two of the vertices where the connected triangles are attached to the literal body. Consequently, a valid augmentation must assign all 2ℓ boundary valencies of $\neg x$ to the literal face. The last valency, which is necessary due to the parity condition, is provided by the vertex at the thick triangle to the right. This concludes the proof of (P1).

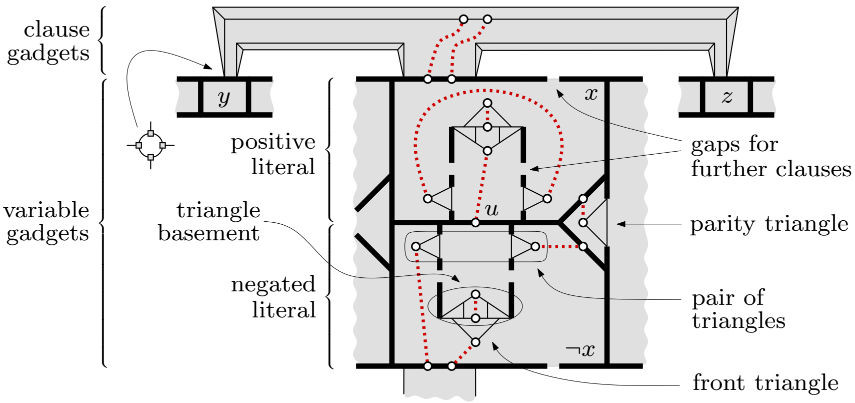


FIGURE 4.6: Variable gadget for variable x and clause gadget for clause $(y \vee x \vee z)$ in graph G_φ. The augmentation (dotted edges) corresponds to the assignment $x = \texttt{true}$, $\neg x = \texttt{false}$.

For the proof of (P2) consider again Fig. 4.6 and let (without loss of generality) x denote the given literal. The number of clauses containing x is ℓ, and s, with $0 \leq s \leq \ell$, denotes the cardinality of the given subset of the clauses containing x. In order to construct a planar augmentation in the variable gadget that uses exactly $2(\ell - s)$ boundary valencies of x, we connect u to a valency in x. This induces an augmentation of $\neg x$ as described in the proof of (P1). Note that this augmentation makes the triangle basement in $\neg x$ triconnected and all the triangles of the literal body are connected to vertices outside the literal body, which also makes the literal body triconnected. In the literal gadget of x, the only vertex that can be connected to u belongs to the triangle basement. Hence, the two remaining degree-2 vertices at the front side of the literal body are also connected. Furthermore, we connect the valency at the parity triangle to the only possible vertex at the opposite thick edge, which makes the parity triangle triconnected. Finally, we choose the s pairs of triangles of the literal body of x closest to the front triangle of x and connect each of these s pairs by an edge. In contrast to the proof of (P1), connecting opposite triangles at the literal body is feasible, since the augmenting edge incident to u ensures triconnectivity. The remaining $2(\ell - s)$ valencies at the literal body can be obviously connected in a planar way to the $2(\ell - s)$ boundary valencies that are not incident to the given clauses, which finally ensures the triconnectivity of the (partially) augmented variable gadget.

With the help of (P1) it is now easy to show that, if G_φ admits a valid augmentation, then φ is satisfiable. Assume that W is a valid augmentation for G_φ. Then W connects the two degree-2 vertices of each clause to two boundary valencies of literal gadgets, since connecting these degree-2 vertices to each other would yield a parallel edge. This *selects* a set of literal gadgets in the sense that a gadget is selected if at least one of its boundary valencies is assigned to a clause face. According to (P1), the boundary valencies of the negated literal gadget of a selected gadget are all assigned to the literal face, and hence, a literal and its negation are never selected at the same time. Thus, the literal selection induces a truth assignment of the variables, which satisfies φ since each clause selects at least one (true) literal.

Conversely, we need to show that, if φ is satisfiable, then G_φ admits a valid augmentation. Assume we have a satisfying truth assignment for φ. For each clause, we choose exactly one true literal L and connect the two degree-2 vertices of the clause to the two boundary valencies of L that are incident to the clause gadget. This ensures triconnectivity at the former degree-2 vertices of the clause and at the former degree-2 vertices providing the boundary valencies. Recall that G'_φ is already triconnected. With the help of (P2) this can be finally extended to a valid augmentation of G_φ.

Since we add, for each clause gadget, only a constant number of vertices to the attached variable gadgets and the number of the remaining vertices and edges in the clause and variable gadgets is also constant, our reduction is polynomial. Recall that the layout of the planar variable–clause graph (that is, the monotone rectilinear representation of ϕ) is already part of the instance of MONOTONEPLANAR3SAT. ☐

CHAPTER 5

Complexity of Planar 4- and 5-Regular Augmentation

In this chapter, we study the complexity of k-Pra and k-Fepra for $k \geq 4$. We show that c-connected k-Pra and c-connected k-Fepra are NP-complete for $k = 4, 5$ and $0 \leq c \leq k$. As in the previous section, we reduce from MonotonePlanar3Sat. The general structure of the proofs follows the outline of the previous hardness proofs. The main difference is that, in the previous section, triconnected 3-Fepra turned out to be NP-hard due to the special connectivity constraint, which must be provided by the augmentation. In contrast, in the following proofs any planar k-regular augmentation W implies the existence of a so-called *canonical augmentation* W' that provides k-connectivity for the resulting graph. Furthermore, the gadgets are constructed such that the input graph is already $(k-1)$-connected $(k-1 \geq 3)$. Since for a planar 3-connected input graph the embedding is unique, and thus, already fixed, the following constructions prove NP-hardness for both problem variants—variable and fixed embedding—and all possible connectivity constraints.

The fact that the input graph has a fixed embedding allows us to carry over some tools from the fixed embedding case for $k = 3$. Our construction will be such that in the hardness proof for the k-regular augmentation all vertices will have degree at least $k-1$. Thus, each vertex has at most one valency, and there is again a notion of node assignments to faces as in Section 4.3. In particular, several necessary conditions for such node assignments carry over. Namely, for each face, the number of assigned valencies must be even (*parity condition*), a pair of adjacent degree-$(k-1)$ vertices requires an additional valency (*matching condition*), and a path of p assigned valencies demands p additional assigned valencies outside that path (*planarity condition*), since the input graphs of the following proofs will be connected. The main difference to 3-Fepra is that the planarity condition cannot be ensured in a postprocessing step, since two nodes that are part of a path whose vertices are all assigned to face f do not necessarily share a second face f' that is distinct from f.

Analogous to the proofs of Theorem 3.1 and Theorem 4.19, it is easy to see that c-connected k-Pra and c-connected k-Fepra are both in NP for $k = 4, 5$. We start with the NP-hardness proof for $k = 4$.

5.1 NP-Completeness of C-connected 4-Pra/Fepra

We prove the following theorem.

Theorem 5.1. *The problems c-connected 4-*Pra *and c-connected 4-*Fepra *are NP-hard for* $0 \leq c \leq 4$, *even if the input graph is already triconnected.*

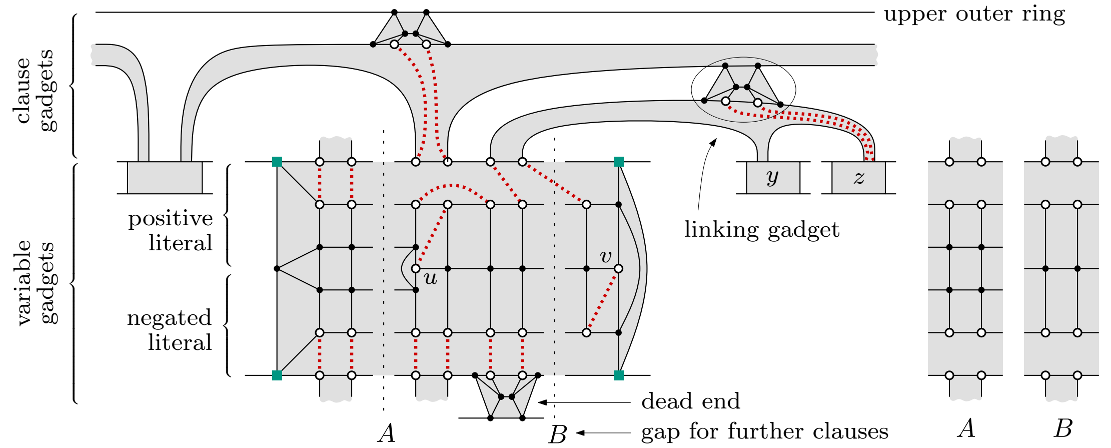


FIGURE 5.1: Variable gadget for variable x and clause gadget for clause $(x \vee y \vee z)$ in graph G_φ. The augmentation (dotted edges) corresponds to the assignment $x =$ `true`, $\neg x =$ `false`.

Proof. Let φ denote the 3SAT formula represented by an instance of MONOTONEPLANAR3SAT. In the following, we construct a triconnected graph G_φ with the following properties. First, φ is satisfiable if and only if G_φ admits a planar 4-regular augmentation. Second, if φ is satisfiable, then G_φ admits a planar 4-regular so-called *canonical augmentation* W', for which we show that $G + W'$ is 4-connected. This immediately implies hardness for all variants of the planar 4-regular augmentation problem. The proof that G_φ is triconnected and $G_\varphi + W'$ is 4-connected for each canonical augmentation W' is deferred to Lemma 5.2.

Figure 5.1 shows the variable and clause gadgets in G_φ. Vertices with degree less than 4 (that is, vertices of degree 3) are highlighted by empty disks, the small black disks depict degree-4 vertices and the green boxes mark the corners of the variable gadgets, which are also degree-4 vertices. The (black) solid line segments between adjacent vertices represent the edges of G_φ; the (red) dotted line segments represent non-edges of G_φ that are candidates for an augmentation of G_φ.

Each variable gadget consists of two symmetric parts (upper and lower part), which correspond to the two literals (positive and negated literal) of the variable. The degree-3 vertex u is incident to both *literal parts*. Vertex v at the right side is called the *parity vertex*. From another point of view, each variable gadget consists of four *corners* (green boxes in Fig. 5.1) and three horizontally embedded subgraphs, which arise from deleting the corners in the gadget. We call the inner (grid-shaped) subgraph the *variable body*. The *outer subgraphs* consist of pairs of degree-3 vertices, each pair connecting to a clause gadget or being part of a *dead end*. Dead ends are formed by *linking gadgets* that stick to the boundary of the variable gadgets. A second function of the linking gadgets is to form the clause gadgets together with some edges that imitate the "comb" structure given by the layout of the variable–clause graph of φ. We exemplarily marked a linking gadget and a dead end in Fig. 5.1.

Analogous to the previous NP-hardness proofs, the variable gadgets are not only connected to clause gadgets but form a ring called *variable ring*, which guarantees that G_φ is always connected. Two adjacent variable gadgets are linked by two edges incident to the corners. Similarly, clause gadgets are not only connected to variable gadgets but also to other clause gadgets as follows. If a clause gadget is nested in the "comb" of another clause gadget such that both gadgets are incident to a common face (see, for example, the clauses c_3 and c_2 in Fig. 3.1), the linking gadget of the nested clause gadget is linked to the gadget above, as shown in Fig. 5.1. The linking gadgets of clause gadgets that are not nested in the "comb" of any other clause gadget are linked to an *outer ring*, which we additionally introduce above all clause gadgets at each side of the variable ring. Hence, the construction contains two outer rings, an upper and a lower one. Figure 5.1 depicts a part of the upper outer ring. Linking gadgets in dead ends are analogously linked to a next higher clause gadget or an outer ring.

Each clause that contains a literal of a variable requires one pair of degree-3 vertices in the outer subgraph of the corresponding literal part of the variable. We call these pairs the *boundary vertices*, the corresponding valencies the *boundary valencies*, and the face of the variable gadget incident to the boundary vertices the *literal face* of the literal part. Note that degree-3 vertices in dead ends are not considered as boundary vertices, since they are only incident to the variable gadget. Thus, each literal part has twice as many boundary valencies as clauses contain the literal. The required number of boundary vertices can be achieved by adding copies of the subgraphs shown in the right of Fig. 5.1 at the positions A and B indicated by the two dotted vertical lines in the variable gadget. For symmetry reasons, the number of inserted subgraphs at A and B must be the same. A dead end is constructed whenever there occurs a pair of degree-3 vertices at the boundary that is not required by any clause.

If, after this construction, one of the outer rings is only linked to one linking gadget, there exists a separator of size 2 in G_φ, that is, G_φ is not triconnected. In this case, we add two further subgraphs (one at A and one at B) to one of the variable gadgets that share a face with the outer ring and construct dead ends at the outer degree-3 vertices of those subgraphs. Then, the dead end of at least one subgraph can be also linked to the outer ring. With at least two linking gadgets linked to the outer ring, now at least four vertices must be deleted in order to separate vertices of the outer ring from the remaining graph. In the following, we call a 4-regular, planar augmentation a *valid* augmentation. We show two properties of G_φ:

(P1) Let W be a valid augmentation and let x be a variable gadget. Then, for at least one literal part in x, the augmentation W assigns all boundary valencies to the incident literal face.

(P2) Given a literal L of a variable x and a (sub)set of clauses containing L, there exists a set of augmenting edges embedded in a planar way in the variable gadget corresponding to x such that all demands in the variable gadget of x are satisfied apart from those of the boundary vertices that are incident to the given clauses.

We start with (P1). Consider the exemplary variable gadget x in Fig. 5.1. The valency of u is incident to both literal parts of the variable gadget, and hence, is either assigned to x or $\neg x$ by a valid augmentation. Without loss of generality, assume that the valency at u is assigned to x. The opposite case is symmetric.

Let ℓ denote the number of clauses containing $\neg x$, let d be the number of dead ends in the literal part of $\neg x$ and let f be the literal face of $\neg x$. The vertices of the variable body provide $2(\ell+d)+1$ valencies incident to f, only one of which, namely the valency at the degree-3 vertex w that shares a common face with v, can be assigned to a face different from f (recall that the valency at u is assigned to x). The remaining $2(\ell+d)$ valencies must be assigned to f by any valid augmentation. Observe further that these valencies induce a path of degree-3 vertices, this is, by the planarity condition, they demand additional $2(\ell+d)$ valencies assigned to f. If w was additionally assigned to f, the path would be even longer, and the demand would even be $2(\ell+d)+1$, which cannot be satisfied, since there are not enough boundary valencies. It follows that the $2(\ell+d)$ boundary valencies of $\neg x$ must be also assigned to f, while w must be connected to v due to the parity condition. This concludes the proof of (P1).

For the proof of (P2), consider again Fig. 5.1 and let (without loss of generality) x denote the given literal. The number of clauses containing x is ℓ, d is the number of dead ends in the part of x, f is the literal face of x, and s, with $0 \leq s \leq \ell$, denotes the cardinality of the given subset of the clauses containing x. In order to construct a planar augmentation in the variable gadget that uses exactly $2(\ell+d-s)$ of the boundary valencies of x, we connect u to the degree-3 vertex w' in the literal part of x such that the number of remaining degree-3 vertices at the variable body incident to f is the same at both sides of w'. This is possible due to the construction, which requires that the subgraphs that are added to a variable gadget are inserted in a balanced way to the left and right of u (and w', respectively). Connecting u in this way induces an augmentation in the literal part of $\neg x$ as described in the proof of (P1). In particular, the valency of v is assigned to the literal part of $\neg x$.

Hence, we observe again that the variable body provides $2(\ell+d)$ valencies incident to f, each of which must be assigned to f. In contrast to the proof of (P1), these valencies induce two paths of degree-3 vertices of the same length, separated by w'. This is, their vertices can be pairwise connected by augmentation edges in a planar way. Hence, connecting the s pairs of left and right vertices closest to w' leaves $2(\ell+d-s)$ valencies at the variable body that can be obviously connected in a planar way to the $2(\ell+d-s)$ boundary valencies incident to f that are not incident to the given clauses. Thus, exactly the demands of the boundary vertices incident to the given clauses are not satisfied. This concludes the proof of (P2).

With the help of (P1), it is now easy to show that, if G_φ admits a valid augmentation, then φ is satisfiable. Assume that W is a valid augmentation. Then W connects the two degree-3 vertices of each clause gadget to two boundary valencies of variable gadgets since connecting those degree-3 vertices to each other would yield a parallel edge. This *selects* a set of literals in the sense that a literal is selected if at least one of the boundary valencies in the corresponding literal part of the variable gadget is assigned to a clause face. According to (P1), the boundary valencies of the negated literal of a selected literal are all assigned to a literal face in the variable gadget, and hence, a literal and its negation are never selected at the same time. Thus, the literal selection induces a truth assignment of the variables, which satisfies φ, since each clause selects at least one (true) literal.

Conversely, if φ is satisfiable, then G_φ admits a valid augmentation as follows. Assume we have a satisfying truth assignment for φ. For each clause, we choose exactly one true literal L and connect the two degree-3 vertices in the linking gadget of the clause to the two boundary valencies of L that are incident to the clause gadget. This augments the clause gadgets to 4-regularity in a planar way. With the help of (P2) this can be extended to a valid augmentation of G_φ. In particular, (P2) allows a valid augmentation that connects the remaining boundary valencies, which are not connected to degree-3 vertices in clause gadgets, to vertices in the variable bodies. We call such an augmentation a *canonical augmentation* and show in Lemma 5.2 that each canonical augmentation of G_φ is 4-connected. This finally proves that G_φ admits a planar 4-regular 4-connected augmentation if and only if φ is satisfiable. Recall that the layout of the planar variable–clause graph (that is, the monotone rectilinear representation of ϕ) is already part of the instance of MONOTONEPLANAR3SAT. Furthermore, we add, for each clause gadget, only a constant number of vertices to the attached variable gadgets and the number of the remaining vertices and edges in the clause and variable gadgets is also constant, thus our reduction is polynomial. □

Lemma 5.2. *Any canonical augmentation of G_φ is 4-connected, G_φ is triconnected.*

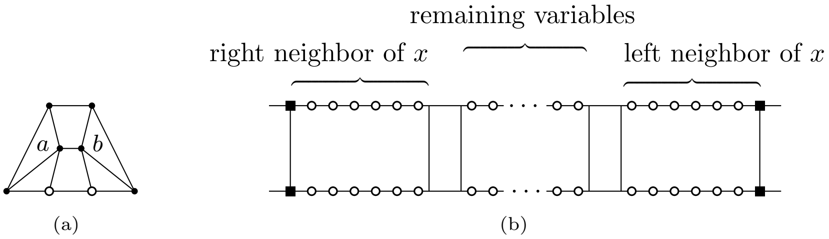


FIGURE 5.2: (a) Subgraph U induced by vertices of linking gadget. (b) Schematization of set M in the variable gadgets distinct from x. Filled boxes depict vertices adjacent to the corners of x, empty disks depict boundary vertices.

Proof. Let W' denote a canonical augmentation of G_φ. We argue in one go that G_φ is triconnected and $G_\varphi + W'$ is 4-connected. To this end, let S denote a set of vertices in G_φ such that $G_\varphi - S$ (respectively $G_\varphi + W' - S$) consists of at least two connected components. Note that, for the purpose of this proof, we consider dead ends to be part of the variable gadget they are attached to. Then, at least one of the following scenarios occurs.

1) S splits a variable gadget.
2) S separates two variable gadgets.
3) S splits a linking gadget.
4) S separates a linking gadget from the variable ring.

For each of these scenarios we prove that $|S| \geq 3$ (respectively $|S| \geq 4$).

Scenario 1: If s and t are vertices of the same variable gadget, we claim that there exist three vertex-disjoint paths between s and t in G_φ and four such paths in $G_\varphi + W'$. We defer the proof of this claim to the end. Hence, separating s and t requires the deletion of at least three (four) vertices and it follows that $|S| \geq 3$ ($|S| \geq 4$).

Scenario 2: If S separates two variable gadgets, S induces a separator $S' \subseteq S$ in the variable ring. If S' splits a variable gadget, we know from scenario 1 that $|S| \geq 3$ ($|S| \geq 4$). Otherwise, consider two variable gadgets x and y that are neighbors in the variable ring. Since x and y are connected by two disjoint edges (sharing no common vertex), separating x and y requires at least two vertices. Since S' needs to split the ring of variable gadgets twice, it is $4 \leq |S'| \leq |S|$ in G_φ and $G_\varphi + W$.

Scenario 3: Consider a linking gadget U that is split by S, and let s and t denote vertices of U that are separated by S. Then, S induces a separator $S' \subseteq S$ in U. We examine the structure of the subgraph U; see Fig. 5.2(a). We first observe that U is triconnected. Thus, it is $3 \leq |S'| \leq |S|$ in G_φ.

Furthermore, after the augmentation, the vertices a and b are the only vertices in U that are not adjacent to a vertex in $(G_\varphi + W) - U$, which is connected. Recall that the outer rings in G_φ are linked to at least two linking gadgets (or dead ends). Let C_s and C_t denote the connected components in U induced by S' containing s and t, respectively. If C_s and C_t both contain a vertex in $U \setminus \{a, b\}$, then there exists a path in $(G_\varphi + W) - U$ connecting C_s and C_t. It follows that $|S| \geq 4$. Otherwise, C_s or C_t contains only vertices in $\{a, b\}$. Then it is readily seen that $4 \leq |S'| \leq |S|$.

Scenario 4: Let C be a connected component of $G_\varphi - S$ ($G_\varphi + W' - S$) that contains a linking gadget but not the variable ring. Recall, that the dead ends are considered to be part of the variable ring. Hence, C does not contain any dead end. If C contains only a single linking gadget U, observe that U is connected to the rest of the graph by four disjoint edges (here we use the assumption that there are at least two linking gadgets incident to each of the outer rings). It follows that $|S| \geq 4$ in G_φ and $G_\varphi + W$. If C contains at least two linking gadgets, each linking gadget is connected to the variable ring by two disjoint edges of the corresponding clause gadget, and the edges of distinct clause gadgets are disjoint. Thus, it is again $|S| \geq 4$ in G_φ and $G_\varphi + W$.

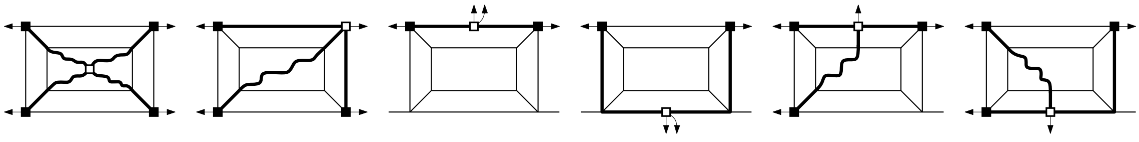
Overall, it follows from these scenarios that a vertex set S needs cardinality $|S| \geq 3$ ($|S| \geq 4$) in order to disconnect the graph, which implies the statement of the lemma. It remains to prove the claim stated in the first scenario.

Claim: Let s and t denote two vertices in the same variable gadget. There exist three vertex-disjoint paths between s and t in G_φ and four such paths in $G_\varphi + W$.

First observe that each vertex has degree 3 in G_φ and degree 4 in $G_\varphi + W$. In the following we route red tokens starting at s and green tokens starting at t through the graphs. In G_φ we route three tokens per vertex, in $G_\varphi + W$ we route four tokens per vertex. The sequence of vertices that are passed by a token yields a path between the token and its starting point. We say that a path is *partial x-y-monotone* if each subpath of vertices in the variable body visits the vertices in an x-y-monotone order in the drawing shown in Fig. 5.1. Two paths cross if they share a common vertex. Whenever a red token meets a green token, a path between s and t is found. Tokens that have neither crossed any other path nor met with a counterpart yet are *free*. Our goal is to route the tokens such that four disjoint paths are formed.

Let x denote the variable gadget containing s and t. We denote by M the set consisting of the boundary vertices of all variable gadgets besides x together with the four corner vertices outside x that are adjacent to the corners of x. Figure 5.2(b) shows a schematization of M in the variable gadgets distinct from x. Without loss of generality, we assume that there are at least three variables in φ. The vertices adjacent to the corners of x are depicted as filled (green) boxes, the boundary vertices are shown as empty disks. Now consider two red and two green tokens at vertices in M such that tokens of the same color are at different vertices. The order in which they occur at the outer face of the schematization in Fig. 5.2(b) is either red–red–green–green or red–green–red–green. In both cases the tokens can be easily routed within the schematization such that they form vertex-disjoint paths. If there is only one red and one green token in M, they can meet in M as well. Hence, we observe the following. If we place in M at most four tokens such that the number of red and green tokens is the same, then the tokens can always be routed via the remaining variable gadgets (besides x) such that they meet a correct counterpart forming vertex-disjoint paths. With this observation it suffices to route at most four tokens to vertices in M and the remaining tokens within x.

In the remainder of this proof we only consider four tokens in $G_\varphi + W$. Routing three tokens in G_φ is analogous. We first sketch the construction and give details afterwards. The idea is to route tokens of the same color in different directions to the corners, the boundary vertices or the two vertices in each dead end in x that are incident to a clause or an outer ring. The routing is done on vertex-disjoint partial x-y-monotone paths such that we can resolve the resulting crossings between paths of different colors iteratively as follows. By exploiting the monotonicity, we show that, as long as there are crossings, there exists a red path π and a green path π' such that π' is the first path (in the direction the token was routed) that is crossed by π and vice versa. However, if π and π' cross more than once, the first crossing a in the direction of the green token does not necessarily equal the first crossing b in the direction of the red token; see Fig. 5.6(b). In this case, π and π' must have the same monotonicity, and we show that no other path crosses π



Type 1 Type 2 Type 3 (upper subgraph) Type 3 (lower subgraph) Type 4 (upper subgraph) Type 4 (lower subgraph)

FIGURE 5.3: Tokens (filled boxes) routed in schematized variable gadget consisting of a variable body (rectangle in center) framed by boundary edges and linked by four edges connecting the corners to the variable body. Tokens start at vertices of different types (empty boxes).

or π' between the crossings a and b. Thus, starting from s, following π up to the first crossing and then traversing π' in opposite direction to t always yields a path between s and t that does not cross any path besides π and π'. Replacing π and π' by this new path resolves at least one crossing. We iterate this operation until all crossings are resolved. Afterwards, the tokens that did not cross another path remain at the corners, the boundary vertices or the two vertices of each dead end in x that are incident to a clause or an outer ring. We show that all but at most four of these tokens can be further routed within x. By finally routing the remaining tokens via the clause gadgets or the outer rings (if they are at a boundary vertex or a vertex in a dead end) or the connecting edges between x and its adjacent variable gadgets (if they are at a corner), we obtain at most four tokens at distinct vertices in M, which can then be matched up as described above.

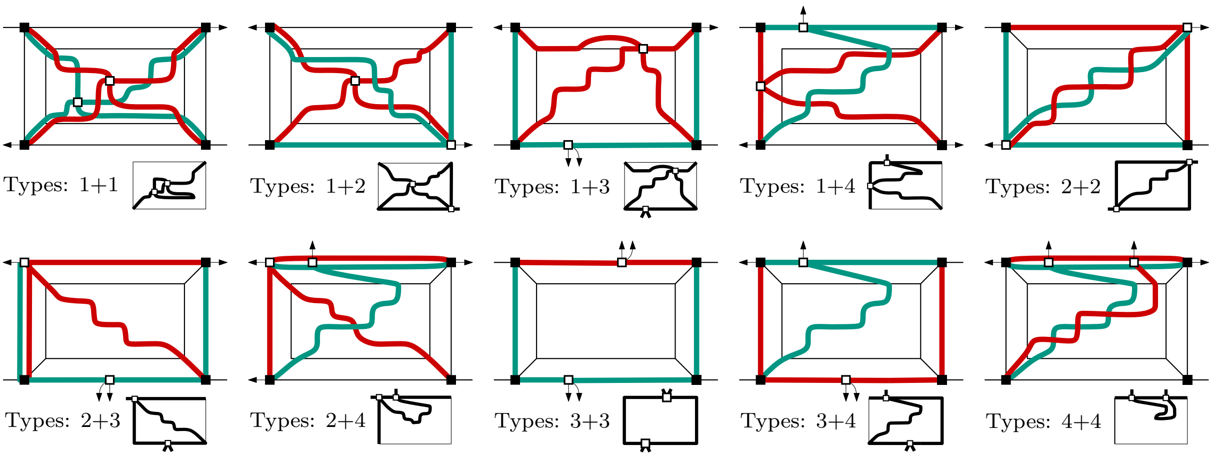
In the following, we make this more precise and show that such paths exist for each vertex pair s, t in x. We distinguish four types of vertices for s and t and give for each type a scheme that sketches how to route the tokens to the corners, the boundary vertices and the designated vertices in the dead ends of x. Following these schemes, for any two vertices in x, we can find a set of paths that admits resolving crossings as described above. Figure 5.3 outlines the schemes described in the following; the arrows indicate the further routing direction for each token in case it is not involved in a crossing.

Type 1: The vertices of the variable body of x. Starting at such a vertex, we route four tokens to the four corners of x. All tokens are routed via the variable body, except if the starting point is adjacent to a vertex outside the variable body that is not a corner. Then we route one token via an outer subgraph.

Type 2: The corners of x. Starting at a corner of x, we route three tokens within x to the remaining corners, one of them via the variable body. The fourth token remains at the starting point.

Type 3: The boundary vertices of x whose augmentation edges are embedded into a face of a clause gadget and the two vertices of each dead end that are adjacent to vertices outside x. Starting at a Type 3 vertex (in the upper or lower outer subgraph), we route two vertices along the boundary of x to the upper left and right corner. The two last tokens remain at the starting point.

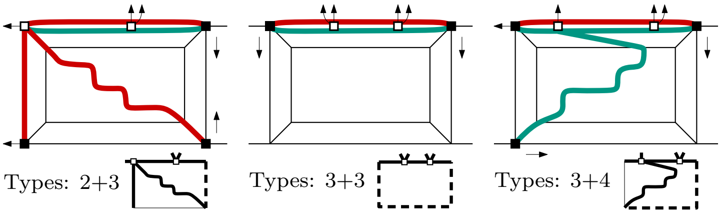
Type 4: The boundary vertices of x whose augmentation edges are embedded into a literal face and the remaining vertices of the dead ends. Starting at a Type 4 vertex (in the upper or lower outer subgraph), we route three tokens within x to all corners apart from the lower right one. In doing so, one token is routed via the augmentation edge and through the variable body. The fourth token remains at the starting point if the starting point



Types: 1+1 Types: 1+2 Types: 1+3 Types: 1+4 Types: 2+2

Types: 2+3 Types: 2+4 Types: 3+3 Types: 3+4 Types: 4+4

FIGURE 5.4: Exemplary paths resulting from routing tokens (filled boxes) for different types of s and t (empty boxes). The small icons sketch the situation after resolving the crossings.



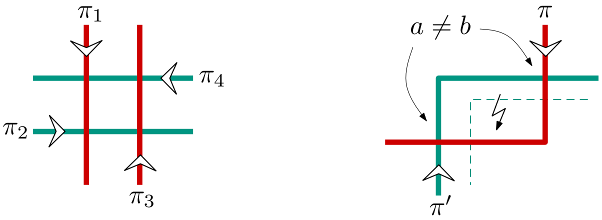
Types: 2+3 Types: 3+3 Types: 3+4

FIGURE 5.5: Exemplary paths resulting from routing tokens (filled boxes) for different types of s and t (empty boxes) in the *same outer subgraph*. In these cases, only one path between s and t arises from resolving the crossings (see small icons, dashed lines sketch how a second path can be found).

is a boundary vertex, and is routed to one of the two vertices in the dead end that are incident to a clause or an outer ring, otherwise.

Note that, when routing only three tokens in G_φ, it is not necessary to distinguish Type 3 and Type 4 vertices. For each type of s and t Fig. 5.4 and Fig. 5.5 sketch an example of a set of paths resulting from routing the tokens from s and t according to the schemes. We observe that, due to the x-y-monotonicity within the variable body and the directions dictated by the schemes, a situation as depicted in Fig. 5.6(a), where each path π that is involved in the first crossing on a paths π' also crosses another path π'' before it reaches its crossing with π', does not occur. We further see that, for the same reason, π and π' are not crossed by any other path between their first crossings if π' is the first path that is crossed by π and vice versa, but the first crossings on both paths are distinct; see Fig. 5.6(b)). Hence, resolving the crossings as described above is possible.

We observe that, following the schemes, we can find a set of paths such that resolving the crossings either yields at least two new paths between s and t (see the cases in Fig. 5.4) or, if we get only one path between s and t, this path runs in one of the outer subgraphs and we get two further free tokens of different color that can be routed along the free boundary of the variable gadget until they meet (see the cases in Fig. 5.5). Hence, there remain at most four free tokens, which, in a second step, are routed to vertices in M via clause gadgets, outer rings or the edges that connect x to its adjacent variable gadgets.



(a) (b)

FIGURE 5.6: (a) Set of paths where no paths π,π' exist such that π is the first path that crosses π' and vice versa. (b) Two paths π and π' such that π is the first path that crosses π' and vice versa, but the crossings a and b are distinct. A third path (indicated by the dashed line) that crosses one of the previous paths between a and b does not exist.

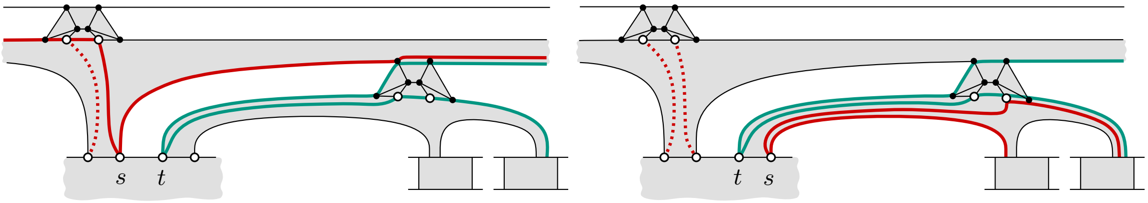


FIGURE 5.7: Exemplary paths resulting from routing tokens for s and t through clause gadgets.

It remains to argue that in this step tokens of the same color can be still routed on vertex-disjoint path. Tokens at the corners of x reach M via single edges (namely the edges that connect x to its adjacent variable gadgets), which readily form a set of vertex-disjoint path.

Within the clause gadgets of one side of the variable ring, we can route up to four tokens (one or two per color) such that paths of the same color are vertex-disjoint and crossings of paths of different colors can be again resolved. Note that tokens of the same color are always routed within the same clause gadget. Figure 5.7 shows two examples. Each token that is not involved in any crossing reaches a vertex in M. Through an outer ring we can route up to $r \leq 4$ tokens such that $\lfloor r/2 \rfloor$ tokens of different color meet, while the possibly remaining token reaches a vertex in M. Finally, the free tokens in M easily meet their counterparts, as we have seen before. $\square$

5.2 NP-Completeness of C-connected 5-PRA/FEPRA

For $k = 5$, the proof of NP-hardness for c-connected k-PRA and c-connected k-FEPRA follows exactly the previous proof for $k = 4$. We use again φ for the given monotone planar 3SAT formula, G_φ for the constructed graph and W for an augmentation. Figure 5.8 shows the gadgets of our construction. The conventions for the drawing are the same as in the previous case. The graph G_φ consisting of *variable gadgets*, which are connected to a *variable ring*, *clause gadgets* and two *outer rings*, is exactly constructed as before. Within a variable gadget there are again a *variable body*, *outer subgraphs*, *literal parts*, *corners*, a $k-1$-degree vertex u, a *parity vertex* v and *dead ends* consisting of *linking gadgets*, *boundary vertices*, *boundary valencies* and *literal faces*, all having the same function as before, although some are constructed slightly different in detail. A canonical augmentation W' is defined analogously to the proof above. We only point out the differences in the following proof sketch. The assertion of Theorem 5.3 finally follows by the same arguments as before, in particular by the help of analogous properties (P1) and (P2) of G_φ and a *valid* augmentation, which is now 5-regular and planar. The proof that G_φ is

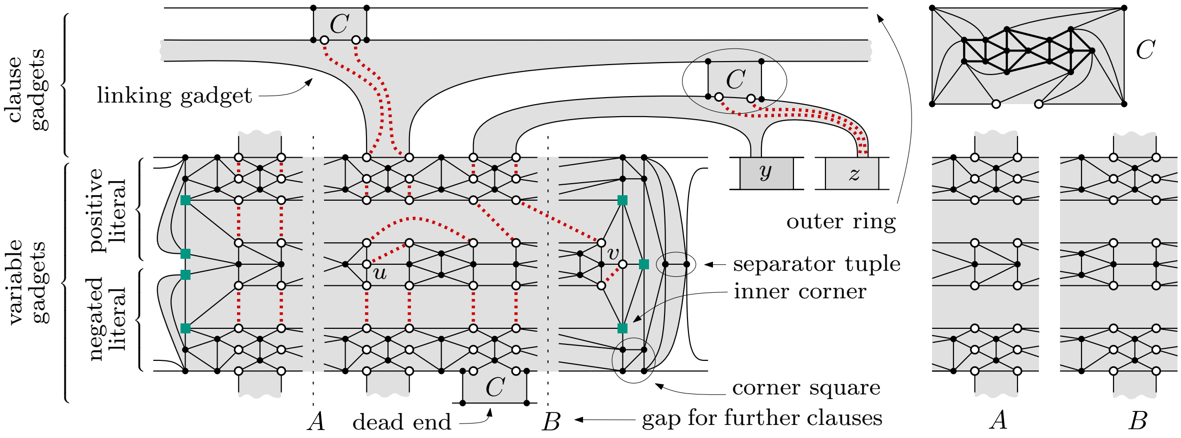


FIGURE 5.8: Variable gadget for variable x and clause gadget for clause $(x \vee y \vee z)$ in graph G_φ. The augmentation (dotted edges) corresponds to the assignment $x = \texttt{true}$, $\neg x = \texttt{false}$.

4-connected and $G_\varphi + W'$ is 5-connected for each canonical augmentation W' is again deferred to a separate lemma (Lemma 5.4).

Theorem 5.3. *The problems c-connected 5-*Pra *and c-connected 5-*Fepra *are NP-hard for* $0 \leq c \leq 5$*, even if the input graph is already 4-connected.*

Sketch of proof. In the 5-regular case, the variable body is separated from the remaining gadget by the four green vertices (boxes) to the left and the three green vertices (boxes) to the right depicted in Fig. 5.8. We call these green vertices the *body separator*. If we additionally delete the *separator tuple*, which is marked in Fig. 5.8, the rest of the gadget splits into two *outer subgraphs*. To the right, each outer subgraph contains a *corner square* of four vertices, of which one is a corner of the whole gadget. In contrast to these corners of the whole gadget, which we call the *outer corners*, we call the four corner vertices in the body separator the *inner corners* of the variable gadget.

The outer subgraphs contain triples of vertically arranged pairs of degree-4 vertices. We call these vertices the *boundary vertices* of the outer subgraph (and the corresponding literal part) and distinguish *outer, middle* and *inner* boundary vertices. The outer boundary vertices connect to a clause gadget or are part of a *linking gadget* that forms a *dead end.* Note that the linking gadgets are more complicated than in the previous case. For clarity, we replaced them by conceptional boxes in Fig. 5.8, each representing the linking gadget C depicted at the right. Note further that, if C is part of a clause gadget, the two empty discs are connected by an edge thereby becoming degree-4 vertices. If C is part of a dead end, the empty discs are connected to a common neighbor. The inner boundary vertices share the literal face with vertices of the variable body. The middle boundary vertices are the remaining ones. We call the corresponding valencies the *outer, middle* and *inner boundary valencies* of the outer subgraph or the corresponding literal part.

In order to finish the proof according to the previous structure, observe that, if an inner boundary vertex is assigned to the literal face, the associated middle and outer boundary vertices need to be connected by a valid augmentation, that is, the outer boundary vertex is not assigned to a clause face. Conversely, if an outer boundary vertex is assigned to a clause face, any valid augmentation connects the associated middle and inner boundary vertices. This is, the inner boundary vertex is not assigned to the literal face. This establishes analogs to properties (P1) and (P2) from the proof of Theorem 5.1. The remaining arguments are analogous. □

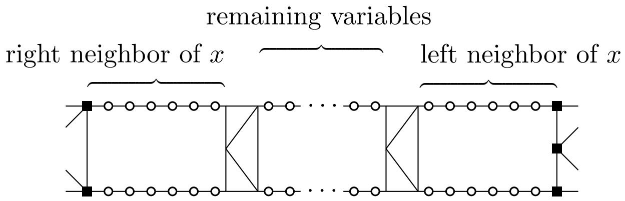


Figure 5.9: Schematization of set M in the variable gadgets distinct from x. Filled boxes depict vertices adjacent to x, empty disks depict outer boundary vertices.

Lemma 5.4. *Any canonical augmentation of G_φ is 5-connected, G_φ is 4-connected.*

Sketch of proof. The proof of Lemma 5.4 follows the proof of Lemma 5.2. We consider the same four scenarios regarding splitting and separation of substructures with respect to a separator S. According to the first scenario in the proof of Lemma 5.2, we first claim that no pair of vertices in the same variable gadget can be separated by a set S of size less than 4. Again, we defer the proof of this claim to the end.

It is again easy to see (scenario 2) that splitting the variable ring requires at least five vertices.

Now consider a linking gadget U that is no dead end, and that is split by S (scenario 3). We analyze the structure of the new linking gadget in more detail; see gadget C in Fig. 5.8. We first observe that U is 4-connected. Thus, the separator S' induced by S in U contains at least four vertices of U and it is $4 \leq |S'| \leq |S|$ in G_φ and $G_\varphi + W$. Observe further that the only separators in U that contain exactly four vertices are those that separate a single corner or a single empty vertex from the rest of U. However, in $G_\varphi + W$, the corners as well as the empty vertices are also incident to vertices in $(G_\varphi + W) - U$, which is connected. Hence, in order to expand a 4-vertex separator S' to a separator in $G_\varphi + W$ we need to add at least one further vertex, that is, separating two vertices in U requires at least five vertices.

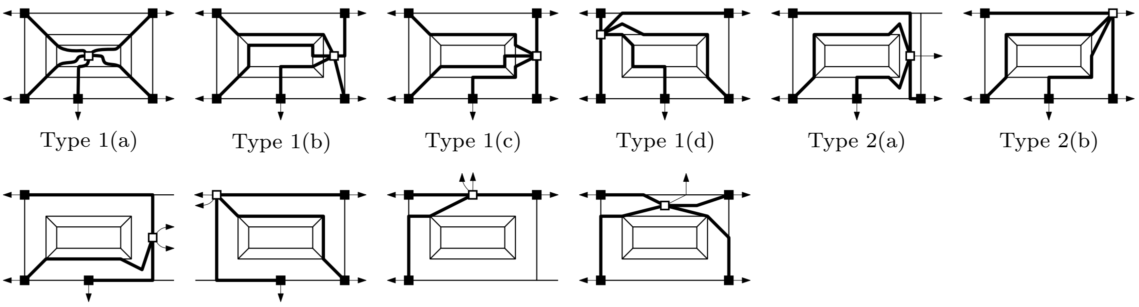
Finally, for scenario 4, observe that splitting a single linking gadget from the rest requires at least four (five) vertices in G_φ (in $G_\varphi + W$). As in the previous case, each linking gadget is further connected to the variable ring by two (four) disjoint edges. Thus, separating more than one linking gadget from the rest requires four (eight) vertices. Thus, analogous to the proof of Lemma 5.2, it follows that G_φ ($G_\varphi + W$) is 4-connected (5-connected).

We finally prove the claim stated in the first case—again following the structure of the previous claim in Lemma 5.2.

Claim: Let s and t denote two vertices in the same variable gadget. There exist four vertex-disjoint paths between s and t in G_φ and five such paths in $G_\varphi + W$.

Analogous to the 4-regular case, starting at s, we route four red tokens in G_φ and five red tokens in $G_\varphi + W$ and the same numbers of green tokens starting at t. In contrast to the 4-regular case, here tokens in the outer subgraphs of the variable gadgets in G_φ are able to visit vertices in a non-monotone order. For example, a token at an inner boundary vertex can be easily routed via the associated middle boundary vertex to the outer one on a *zig-zag path.* This is why we extend the definition of partial x-y-monotone paths to outer subgraphs. A path is *partial x-y-monotone* if its subpaths in the variable body and in the outer subgraphs, apart from zig-zag paths, visit the vertices in an x-y-monotone order.

Let x denote the variable gadget containing s and t, and let M consist of those vertices outside x that are adjacent to vertices in x and the outer boundary vertices of the variable gadgets distinct from x. Figure 5.9 shows a schematization of M. By the same arguments as in the 4-regular case, we see that, if we place at most four tokens with the same number of red and



Type 1(a) Type 1(b) Type 1(c) Type 1(d) Type 2(a) Type 2(b)

Type 3(a) Type 3(b) Type 4 Type 5

FIGURE 5.10: Tokens (filled boxes) routed in schematized variable gadget consisting of the variable body (rectangle in the center) framed by the body separator, the separator tuple and the outer subgraphs and linked by four edges connecting the corners to the variable body. Tokens start at vertices of different types (empty boxes).

green tokens at vertices in M (tokens of the same color at different vertices), we can always route these tokens via the variable gadgets distinct from x such that they meet a correct counterpart forming vertex-disjoint paths. Hence, in the following we route at most four tokens to vertices in M and the remaining tokens within x.

In the remainder of this proof we only consider five tokens in $G_\varphi + W$. Routing four tokens in G_φ is analogous. Analogous to the 4-regular case, we give a schematization on how the tokens, starting at different types of vertices, can be routed on partial x-y-monotone vertex-disjoint path such that we can resolve crossings as described in the 4-regular case and such that, after resolving the crossings, there remain at most four free tokens at vertices in M. With the help of these schemes, we finally see that there exist five vertex-disjoint paths between each vertex pair s, t in x. Figure 5.10 outlines the schemes described in the following; the arrows indicate the further routing direction for each token in case it is not involved in a crossing. We note that some of the vertex types considered in the 4-regular case are further split into subtypes, here.

Type 1: We distinguish the Type 1 vertices depending on the number of tokens that are routed via different subgraphs. For all Type 1 vertices, we seek to route one token to an outer boundary vertex and four tokens to the outer corners of x. Type 1(a): The vertices of the variable body of x, apart from v. For these vertices we route all tokens via the variable body. Due to the zig-zag paths in the outer subgraphs and since routing a token from any inner corner to the next outer corner is straightforward, routing five tokens as described above can be done once we can route five tokens to the inner corners and one inner boundary vertex. Type 1(b): The vertex v. Starting from v, we route two tokens via the variable body, and three tokens via the outer subgraphs. Type 1(c): The right vertices of the body separator. Here we route one token via the variable body and four tokens—two per subgraph—via the outer subgraphs. Type 1(d): The left vertices of the body separator and the two vertices that are adjacent to a left vertex of the body separator and a left outer corner. In this case we also route one token via the variable body and four tokens via the outer subgraphs, however, here we have three tokens in one subgraph and one token in the other subgraph.

Type 2: The left vertex of the separator tuple (a) and the vertices in the corner squares (b). Starting at a vertex of Type 2, we route one token to an outer boundary vertex, two tokens to the left outer corners and, if the Type 2 vertex is in a corner square, two tokens to the right outer corners of x. If we consider the left vertex of the separator tuple, we route one token to a right outer corner and the second one to the second vertex of the separator tuple.

Type 3: The right vertex of the separator tuple (a) and the left outer corners (b). Starting at one of these vertices, we route three tokens via the outer subgraphs; one to an outer boundary vertex and two to the outer corners at the opposite side. The two last tokens remain at the starting point.

Type 4: The vertices in the outer subgraphs that are incident to an augmentation edge that is embedded into a clause face. Starting at such a vertex we route three tokens to the left outer corners and one right outer corner. The two last tokens remain at the starting point.

Type 5: The remaining vertices in the outer subgraphs. Starting at one of the remaining vertices we route two tokens to the left outer corners and two tokens to the right outer corners. The fifth token remains at the starting point.

Note that none of the schemes routes more than two tokens via the three vertices that connect x to its adjacent variable gadget to the right. Analogously, at most two tokens are routed via the left outer corners of x. This ensures that, when routing these free tokens to M, tokens of the same color do not meet in M, and thus the paths in M can be build as described above. The details of the construction of the paths inside x and inside the clause gadgets and outer rings are analogous the proof of Lemma 5.2. □

Conclusion of Part I

In this part we have studied the problem of augmenting a planar graph (with a variable or fixed embedding) such that it becomes k-regular, c-connected and remains planar for $0 \leq c \leq k \leq 5$. We have completely determined the complexity of these problems by either giving an NP-hardness proof or an efficient algorithm. For $k \leq 2$ all problems can be solved in linear time, whereas for $k \geq 4$ all variants are NP-complete. For $k = 3$, we showed NP-completeness for the case of variable embedding and gave a simple testing algorithm for the existence of an arbitrary 3-regular augmentation with a fixed embedding. We then considered the 3-regular case with a fixed embedding in more detail and showed that for $c = 0, 1, 2$ the problem can be characterized in terms of the existence of node assignments that satisfy, for each face, certain conditions. We first gave relatively simple $O(n^{2.5})$-time algorithms to find such node assignments. The bottleneck here was finding a generalized perfect matching in an assignment graph with potentially quadratically many edges. By giving a construction for an equivalent sparse assignment graph we reduced the running time to $O(n^{1.5})$ for all cases. The proof that the problem is NP-complete for $k = c = 3$, even for the fixed embedding case, finished the treatment of the case $k = 3$.

Open Problems. Our main question is whether there are reasonable parameters or restrictions subject to which the NP-complete variants become tractable. For example, what is the complexity of augmenting a $(k-1)$-regular graph or a graph of maximum degree $k-1$ to be k-regular and c-connected? Recently we could show that for $k = 3$ all problem variants become solvable in linear time if we restrict the input to 2-regular instances. However, it is still open, whether similar improvements are possible for $k = 4, 5$. Another interesting direction is fixed-parameter tractability. Is 3-connected 3-FEPRA possibly fixed-parameter tractable with respect to the number of faces of the input graph?

Notes. For the final publication at Algorithmica, we recently also conducted a computer-aided proof of the c-connectivity ($c = 3, 4$) of the gadgets in Chapter 5, which additionally confirms the correctness of our gadgets.

Part II

All-Pairs Minimum Cut

CHAPTER 6

Introduction – Gomory-Hu Trees and Other Data Structures

In 1961 Gomory and Hu claimed to solve the multiterminal network flow problem[1] stated by Mayeda [106], which asks for the edge connectivity $\lambda(u, v)$ of each vertex pair $\{u, v\}$ in an undirected, weighted graph $G = (V, E, c)$. In fact, with their elegant construction of a Gomory-Hu tree, they even solve the *all-pairs minimum-cut problem*, which besides the connectivity value, also asks for a concrete minimum separating cut. Clearly, a solution of the all-pairs minimum-cut problem consists of at least $n - 1$ minimum separating cuts. The remarkable thing is that, according to Gomory and Hu, $n-1$ cut computations are also sufficient to represent at least one minimum separating cut for each of the $\binom{n}{2}$ different vertex pairs in the graph.

Gomory and Hu also introduced *flow-equivalent trees* for undirected, weighted graphs, which also need $n - 1$ cut computations for the construction, but just solve the multiterminal network flow problem (besides other advantages which we will see below). That is, dropping the requirement of a concrete cut does not result in a faster approach. This is comparable to the situation of a single minimum separating cut, where the computation of the connectivity $\lambda(u, v)$ of two vertices u and v also implies the construction of a minimum s-t-cut (see Section 1.3).

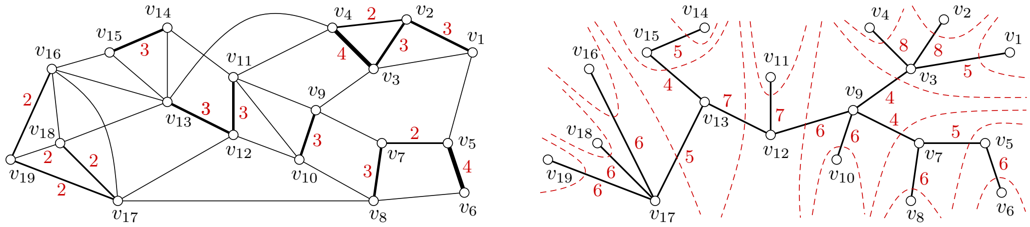
Gomory-Hu Trees. A Gomory-Hu tree $T(G) = (V, E_T, c_T)$ of an undirected, weighted graph $G = (V, E, c)$ is a weighted tree on the vertices of G that represents at least one minimum s-t-cut (and the corresponding cost) for each pair $\{s, t\} \subseteq V$ in the form of an edge. An edge induces a cut in the underlying graph G by decomposing the tree into two connected subtrees when it is deleted. The vertices of the resulting subtrees then induce the sides of a cut in G.

This property holds for any tree on the vertices of G. In a Gomory-Hu tree $T(G)$, the cut represented by an edge $e = \{s, t\} \in E_T$ is further a minimum s-t-cut in G, and the cost $c_T(e)$ of the edge e corresponds to $\lambda_G(s, t)$ in G. As a consequence, it also holds that a minimum s-t-cut for an arbitrary vertex pair $\{s, t\} \subseteq V$ is represented by a cheapest edge on the (unique simple) path between s and t in $T(G)$, that is, $\lambda_{T(G)}(s, t) = \lambda_G(s, t)$. Figure 6.1(b) exemplarily shows a Gomory-Hu tree and the cuts represented by the edges for the graph in Fig. 6.1(a), which is taken from Nagamochi [110]. Note that a Gomory-Hu tree is not necessarily a spanning tree. This is, the tree edges do not necessarily correspond to edges in G.

A Gomory-Hu tree $T(G)$ thus represents $n - 1$ non-crossing minimum s-t-cuts, one cut per edge $\{s, t\} \in E_T$, and can be also considered as a set of cuts, or even more, as a *minimum cut basis*

---

[1] This misleading name was not used by Mayeda but by Gomory and Hu and many subsequent authors. Mayeda called the resulting matrix the *terminal capacity matrix* and was interested in its characterization.



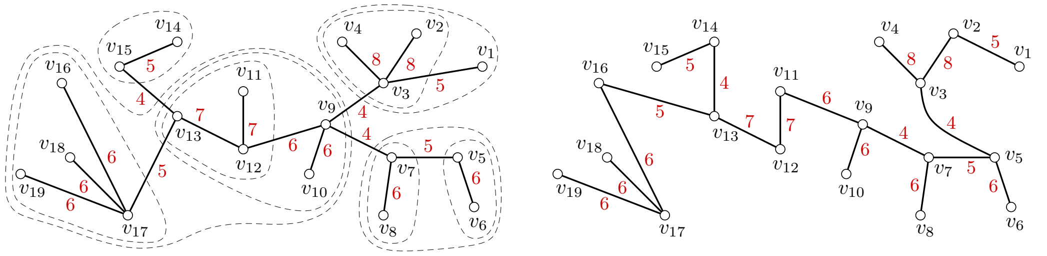
(a) Undirected, weighted graph $G = (V, E, c)$. Unlabeled edges have cost 1.

(b) Undirected, weighted Gomory-Hu tree $T(G) = (V, E_T, c_T)$ and induced cuts.

Figure 6.1: Example of an undirected, weighted graph G and a corresponding Gomory-Hu tree $T(G)$. Red numbers denote edge costs, dashed red lines indicate the minimum separating cuts represented by $T(G)$.

of G. The latter also follows from the considerations of Gomory and Hu [59]. The construction of a Gomory-Hu tree for a general undirected weighted graph $G = (V, E, c)$ needs $n-1$ minimum separating cut computations with respect to $n-1$ distinct cut pairs $\{u, v\} \subseteq V$. Hence, the running time of the construction depends on the running time of the chosen maximum-flow algorithm. In this work, we do not care about the explicit running time but quantify the running time of constructing a Gomory-Hu tree by counting cut computations. We further remark that the cut pairs $\{u, v\}$ used for computing the cuts do not necessarily correspond to the final tree edges. The tree in Fig. 6.1(b), for example, was constructed using the following sequence of cut pairs (see Algorithm 1): $\{v_1, v_2\}$, $\{v_2, v_3\}$, $\{v_3, v_4\}$, $\{v_3, v_5\}$, $\{v_5, v_6\}$, $\{v_5, v_7\}$, $\{v_7, v_8\}$, $\{v_7, v_9\}$, $\{v_9, v_{10}\}$, $\{v_9, v_{11}\}$, $\{v_{11}, v_{12}\}$, $\{v_{12}, v_{13}\}$, $\{v_{13}, v_{14}\}$, $\{v_{14}, v_{15}\}$, $\{v_{13}, v_{16}\}$, $\{v_{16}, v_{17}\}$, $\{v_{17}, v_{18}\}$, $\{v_{17}, v_{19}\}$. The edges in E_T however do not correspond to these vertex pairs. This is due to the fact that during the tree construction intermediate tree edges may change in order to guarantee that each cut is indeed represented by an edge in the end. In Section 7.1 we describe the construction of a Gomory-Hu tree in detail.

Interestingly, it is not possible to generalize Gomory-Hu trees for vertex connectivity in undirected graphs without weakening the notion of vertex connectivity, as shown by Benczúr [12]. He further showed that also a generalization to directed graphs (and edge connectivity) is impossible. In this way, he disproved the results of Schnorr [126, 127, 128] and Gusfield and Naor [67, 68], who first attempted this directions. Vahrenkamp [136] discusses how the contraction technique used in the Gomory-Hu tree construction (see Algorithm 1) can be used for directed graphs, while Gupta [65] describes a straightforward generalization at least to directed Eulerian graphs. Some special graph classes further admit adapted techniques that exploit special graph properties to construct a Gomory-Hu tree. Borradaile et al. [17] provide an algorithm that returns a Gomory-Hu tree for a planar, undirected, weighted graph in $O(n \log^5 n)$ time and $O(n \log n)$ space if shortest paths in the graph are unique. Otherwise, the running time of the algorithm increases by a $\log^2 n$ factor. Bhalgat et al. [13] consider undirected, unweighted graphs and exploit techniques for computing *Steiner cuts* in order to develop a Gomory-Hu tree algorithm with an expected running time of $\tilde{O}(mn)$. In general, undirected, weighted graphs, however, to the best of our knowledge, the original Gomory-Hu tree construction based on $n-1$ cut computations is still the best we can do. Since the actual running time of this construction depends on the used maximum-flow algorithm for computing the minimum separating cuts, developing faster maximum-flow algorithms for special graph classes is another way to speed up a Gomory-Hu construction. For graphs with bounded treewidth, for example, the construction runs in $O(n^2)$



(a) Maximal components of the graph in Fig. 6.1(a) derived from the Gomory-Hu tree in Fig. 6.1(b).

(b) Undirected, weighted flow-equivalent tree $T(G) = (V, E_T, c_T)$ of the graph G in Fig. 6.1(a).

FIGURE 6.2: All maximal components (a) of the graph G of Fig. 6.1(a), and an exemplary flow-equivalent tree of G (b). Red numbers denote edge costs, dashed black lines indicate the maximal components. The flow-equivalent tree $T(G)$ was constructed using the same sequence of cut pairs as for the Gomory-Hu tree of Fig. 6.1(b). But here the edges in E_T correspond to these cut pairs.

time, due to the linear-time maximum-flow algorithm by Hagerup et al. [71] for these special graphs. We further note that the choice of cut pairs and cuts in the original construction of a Gomory-Hu tree is not determined, and thus, leaves room for different strategies and heuristics. Goldberg and Tsioutsiouliklis [58] compare the performance of some interesting heuristics in an experimental study. In this work we will exploit these degrees of freedom in the design of our algorithms from a more theoretical point of view.

Due to their special structure, Gomory-Hu trees are however not only representations of minimum s-t-cuts, but can be also considered as a data structure that is able to answer several queries. In this light the all-pairs minimum-cut problem is a query that expects a minimum s-t-cut and $\lambda_G(s,t)$ for a given vertex pair $\{s,t\}$ as an answer. Given a Gomory-Hu tree for G, this query can be answered in $O(n)$ time. Another query example considers *maximal components* studied by Nagamochi [110]. A (proper) subset S of the vertex set V of an undirected, weighted graph G is a maximal component if the edge connectivity of each vertex pair $\{u,v\}$ with $u \in S$ and $v \in V \setminus S$ is less than the minimum edge connectivity among all pairs $\{s,t\} \subseteq S$, that is, $\forall u \in S, v \in V \setminus S : \lambda_G(u,v) < \min_{s,t \in S} \lambda_G(s,t)$. That is, separating any vertex pair $\{s,t\}$ within a maximal component S is more expensive than separating a vertex u inside of S from a vertex v outside of S, which implements a kind of cohesion property. Nagamochi shows that two maximal components are either nested or disjoint, and all maximal components can be easily derived from any Gomory-Hu tree $T(G)$ by considering the values $k \in c(E_T)$ in an increasing order. Then, deleting the edges in E_T with cost less than k, iteratively for all k, decomposes $T(G)$ into connected subtrees such that the vertices of each subtree induce a maximal component S in G with $\min_{s,t \in S} \lambda_G(s,t) \geq k$. Figure 6.2(a) exemplarily shows the maximal components of the graph given in Fig. 6.1(a) together with the Gomory-Hu tree of Fig. 6.1(b). In this way, a Gomory-Hu tree also admits to answer queries which, for example, ask for each maximal component S with $\min_{s,t \in S} \lambda_G(s,t) \geq k$, or for the inclusion-maximal maximal component that contains a given vertex, both in $O(n)$ time.

Flow-Equivalent Trees. A data structure closely related to Gomory-Hu trees are flow-equivalent trees. A weighted tree $T(G) = (V, E_T, c_T)$ on the vertices of G is a flow-equivalent tree if it is weighted by a cost function $c_T : E_T \to \mathbb{R}_0^+$ such that the cost $c_T(\{s,t\})$ of each edge $\{s,t\}$ corresponds to $\lambda_G(s,t)$. As a consequence, it also holds that the edge connectivity $\lambda_G(s,t)$ for an arbitrary vertex pair $\{s,t\} \subseteq V$ is given by the cost of a cheapest edge on the (unique simple) path between s and t in $T(G)$, that is, $\lambda_{T(G)}(s,t) = \lambda_G(s,t)$. Hence, flow-equivalent trees represent the edge connectivity of arbitrary vertex pairs $\{s,t\}$ in G in the same way as Gomory-Hu trees, but the edges do not necessarily provide any information about the shape of the minimum s-t-cuts that induce these values. That is, Gomory-Hu trees form a subclass of flow-equivalent trees, that is, each Gomory-Hu tree is also a flow-equivalent tree, and thus, admits to answer at least each query that can be answered by a flow-equivalent tree, while the converse is not true. The two queries described above, for example, cannot be answered by flow-equivalent trees. Consequently, Gomory-Hu trees are more powerful.

Flow-equivalent trees as well as Gomory-Hu trees were both introduced by Gomory and Hu [59], however, they did not introduce concise names for these structures. The terms Gomory-Hu tree and flow equivalent tree used in this work are chosen according to Nagamochi [110]. Gusfield [66, 69], who also worked a lot on this kind of data structures, uses the terms *Gomory-Hu cut tree* and *equivalent-flow tree.* In his pioneering work [66], he introduced a simplification of Gomory's and Hu's algorithm for constructing a Gomory-Hu tree, which we describe in Section 7.1.2, and gave a simple algorithm for the construction of a flow-equivalent tree, which not necessarily returns a Gomory-Hu tree. The latter is further not *solution-complete*, that is, there exist flow-equivalent trees that cannot be constructed. Nevertheless, the collection of all flow-equivalent trees of a given graph can be compactly represented, fully characterized and recognized in polynomial time [80]. We review Gusfield's algorithm in more detail and give an example for its solution-incompleteness in Section 6.2.1. The algorithm also constructs $n-1$ cuts in order to determine the necessary edge-connectivity values, but the remarkable property of this algorithm is that, in contrast to the construction of a Gomory-Hu tree, the $n-1$ distinct cut pairs used for the construction finally also form the tree edges in E_T. Figure 6.2(b) shows an example. If we now compute full maximum flows during the construction of a flow-equivalent tree, instead of just minimum separating cuts (which can be already obtained from preflows), this property admits to additionally assign these flows to the edges in E_T. From each flow assigned to an edge $\{s,t\}$ we can then derive a data structure called *Picard-Queyranne DAG* [118], which represents all minimum s-t-cuts in the underlying graph G, in $O(m)$ time. Furthermore, a DAG is able to answer whether a given vertex pair $\{u,v\}$ is separated by a minimum s-t-cut in constant time. The corresponding cut can be deduced in time $O(m)$. For a more detailed description on how to construct these DAGs see Section 6.2.3. Together with such a DAG assigned to each edge, flow-equivalent trees become very powerful. Gusfield and Naor [69] showed that such an *extended flow-equivalent tree* may now also answer the all-pairs minimum-cut problem, as the DAG for an arbitrary vertex pair $\{s,t\}$ (and a minimum s-t-cut therein) can be also derived in $O(m)$ time.

However, extended flow-equivalent trees are less adapted to queries like those related to maximal components, which strongly depend on the tree representation of minimum separating cuts, since, in contrast to Gomory-Hu trees, they do not provide the required structure directly. Instead, the structure must be first extracted from the information stored in the DAGs. In Chapter 12 we will see a further example where Gomory-Hu trees are more appropriate than extended flow-equivalent trees. In this chapter we introduce a query that asks for inclusion-maximal source-community clusterings with respect to a given community. These queries are answered by a special Gomory-Hu tree, we call it *unique-cut tree*, which is developed in Section 7.2.

Marrying Gomory-Hu Trees and Extended Flow-Equivalent Trees. In Chapter 8 we consider the all-pairs minimum-cut problem in a dynamic scenario. That is, we seek a data structure answering the corresponding query that can be efficiently updated whenever the underlying graph changes. In this context we suggest a simple, and in many cases very fast, dynamic update algorithm for Gomory-Hu trees but with a worst-case running time of $n-2$ cut computations. In Chapter 9, however, we show that (under a plausible assumption, which we call the *recomputation conjecture*, and which we discuss in Section 6.2.3) this worst-case running time is asymptotically optimal in the following sense: we marry Gomory-Hu trees and extended flow-equivalent trees to a very powerful data structure, which we call *extended Gomory-Hu trees* (see below). Then we show that, even if we have such a comprehensive data structure at hand in the current time step, there still exists a situation where, under the assumption of the *recomputation conjecture*, $n - O(1)$ cut computations are necessary in order to solve the all-pairs minimum-cut problem after a change in the underlying graph. The considered extended Gomory-Hu tree consists of an arbitrary Gomory-Hu tree $T(G)$, an arbitrary extended flow-equivalent tree $T'(G)$, and all maximum flows computed in G during the construction of the trees $T(G)$ and $T'(G)$. Hence, extended Gomory-Hu trees subsume the advantages of both concepts, the structural information on cuts given by (regular) Gomory-Hu trees and the possibility to derive a minimum s-t-cut for any vertex pair $\{s,t\} \subseteq V$ in $O(m)$ time given by extended flow-equivalent trees. Additionally they provide all flows that led to $T(G)$ and $T'(G)$, which is a true added value, since updating flows maintains more information than updating DAGs, as we will see in Section 6.2.3.

Further Data Structures Providing Information about Minimum Cuts. An interesting generalization of Gomory-Hu trees was developed by Cheng and Hu [24]. They show that $n-1$ minimum separating cuts still suffice to separate all $\binom{n}{2}$ vertex pairs in an undirected graph G even if an arbitrary cost, that does not necessarily result from a cost function on the edges, is assigned to each of the $2^{n-1} - 1$ cuts in G. The resulting tree representing these $n - 1$ cuts is a rooted binary tree (edges directed to the leaves) where the leaves correspond to the original vertices in the underlying graph, and the inner vertices represent the cuts. The costs of the cuts are assigned to the inner vertices. A minimum separating cut for two original vertices is then represented by the nearest common predecessor of the leaves representing the vertices. Another idea are *mimicking networks*, which do not represent minimum separating cuts, but the pairwise maximum-flow values for a set of $k < n$ vertices (terminals) of a weighted graph. Mimicking networks were introduced by Hagerup et al. [71] and consist of a constant number of vertices, usually depending on k. The terminals in the original graph correspond to vertices in the mimicking network such that a maximum s-t-flow in the mimicking network has the same value as in the original graph. Hagerup et al. showed that a mimicking network of at most 2^{2^k} vertices exists for each (directed and undirected) weighted graph, and presented an efficient algorithm for the construction of such a network if the given graph has bounded treewidth. Chaudhuri et al. [23] improve the bound for graphs with bounded tree width to $O(k)$ vertices in the mimicking network, and Krauthgamer and Rika [97] give a bound of $O(k^2 2^{2^k})$ vertices for planar networks. In contrast, a *cactus representation of all global minimum cuts* in an undirected (possibly weighted) graph, is a graph with $O(n)$ vertices together with a mapping that maps inclusionwise minimum cuts in the cactus to global minimum cuts in the original graph. This representation was first introduced by Dinits et al. [32]. Nagamochi and Kameda [114] developed an algorithm that runs in $O(m + \lambda n^2)$ time for unweighted graphs, while Fleischer [43] presented an approach for weighted graphs that benefits from the fast minimum-cut algorithm of Hao and Orlin [74] and achieves the same running time as this minimum-cut algorithm, which is $O(nm \log(n^2/m))$. Karger and Panigrahi [92] employ randomization to obtain a near linear time Monte Carlo algorithm for constructing a cactus representation.

6.1 Contribution and Outline

The results in Part II are either new or in parts based on joint publications with different coauthors. That is, the chapters in this part do not correspond to single publications. Hence, in this outline, we shortly discuss the new and already published parts per chapter.

Chapter 6: Introduction – Gomory-Hu Trees and Other Data Structures

In the remainder of this chapter, that is, in Section 6.2, we introduce specific notations related to the construction of Gomory-Hu trees and explain two special concepts, namely the *equivalence of cuts* in disconnected and dynamic graphs, and *reusable cuts* in dynamic graphs, which we will in particular use in Chapter 8 and Chapter 9. Furthermore, we briefly sketch the construction of flow-equivalent trees in Section 6.2.1 and introduce the classes of U-cuts and M-sets in Section 6.2.2, which will be intensively used throughout Part II and Part III. In Section 6.2.3, we briefly consider maximum flows in undirected, weighted networks and sketch how to construct Picard-Queyranne DAGs from given flows. Moreover, we discuss the task of updating maximum flows and DAGs, and state our *recomputation conjecture*, which is the main assumption in Section 9.3, where we claim the optimality of the asymptotic worst-case running time of our update procedures developed in Chapter 8.

Chapter 7: Gomory-Hu Trees in Static Graphs

In Section 7.1 we provide a compact and clearly structured description of Gomory's and Hu's algorithm that is somewhat easier to understand than the original publication of Gomory and Hu [59]. We particularly point out the fundamental ideas and techniques, the degrees of freedom, and the relation to the simpler algorithm introduced by Gusfield [66]. The latter is remarkably easy to implement, due to a special representation of intermediate trees, which we slightly generalize with respect to our purposes.

In Section 7.2 we develop a new data structure that represents a class of special cuts, namely the class of U-cuts, which consists of those minimum separating cuts that minimize one side of the cut. The new data structure, which we call *unique-cut tree*, is basically a Gomory-Hu tree representing a set of special cuts. Nevertheless, we will see that adapting the original Gomory-Hu tree construction is not straightforward. The efficient construction of maximum SC-clusterings in Part III employs unique-cut trees as underlying data structure.

Publications. Since unique-cut trees form the basis of the construction of maximum SC-clusterings, but are not directly related to the field of graph clustering, the results of Section 7.2 are only sketched in a joint publication on cut-based graph clustering with Michael Hamann and Dorothea Wagner [73].

Chapter 8: Gomory-Hu Trees in Dynamic Graphs

We consider Gomory-Hu trees also in a dynamic scenario where the underlying graph evolves due to atomic changes. This scenario is also regarded in a field called *sensitivity analysis*, where the main focus is on the costs of minimum separating cuts, that is, the goal is to understand the behavior of the local connectivity values when the underlying graph changes. In contrast, we aim at *efficiently* and *smoothly* updating the whole structure of a Gomory-Hu tree with all its special properties. To this end, in Section 8.1 we examine different conditions that admit the reuse of a cut represented in a previous tree for the construction of a new tree. Based on these insights, we then develop a fully-dynamic update algorithm for Gomory-Hu trees in Section 8.2, which provides a high potential for saving cut computations and guarantees optimal temporal smoothness. This algorithm consists of several update procedures adapted to the different changes that may occur in the underlying graph. Among these, the update procedure for the case of an edge deletion or decreasing edge cost is the most challenging one, which has been also stated by other authors like Barth et al. [11], who formulated the design of such an update procedure as an open problem. Since an updated Gomory-Hu tree already represents the all-pairs local connectivity values in the changing graph, our procedures also provide a sensitivity analysis for undirected, weighted multiterminal flow networks and solve the open problem stated by Barth et al.

In Section 8.3, we confirm the high potential of our update algorithm for saving cut computations in a brief experiment using a dynamic real-world network. Furthermore, we conduct a detailed experimental study that itemizes the savings according to different factors that influence the running time. This study confirms what is already to be expected due to the tools exploited by the algorithm in order to find reusable cuts, namely, a strong dependency of the running time on the shape of the Gomory-Hu trees in the corresponding time steps.

Publications. The main results in this chapter are published jointly with Dorothea Wagner in [79]. The detailed experimental study of the factors influencing the savings of our update approach in Section 8.3 tops these results off and has not yet been published.

Chapter 9: Optimality in Smoothness and Running Time

We finally prove that our update approach indeed provides optimal temporal smoothness, that is, we show that the number of equivalent cuts that are represented in the Gomory-Hu trees of consecutive snapshots is as large as possible. We further argue that also the worst case running time of our approach is optimal under the assumption of the reconnection conjecture introduced in Section 6.2.3.

More precisely, in Section 9.1, we prove optimal temporal smoothness for the update procedure for edge deletion and cost decrease, and improve the update procedure for edge insertion and cost increase in Section 9.2, such that optimal temporal smoothness can be also proven in this case. In order to argue for the optimality of the asymptotic worst-case running time, we describe an exemplary situation with an initial graph and an initial, even extended, Gomory-Hu tree in Section 9.3 and show that also with this additional information (recall that an extended Gomory-Hu tree provides comprehensive information on the structure of the minimum separating cuts in the graph) it is not possible to update a (regular) Gomory-Hu tree with less cut computations, given that the recomputation conjecture holds.

Publications. The results in this chapter are exclusively presented in this thesis.

We finally remark that, in contrast to many proofs, the implementation of the algorithms developed in Part II is quite easy, since all algorithms rely on the simple construction techniques introduced by Gusfield [66].

6.2 Preliminaries

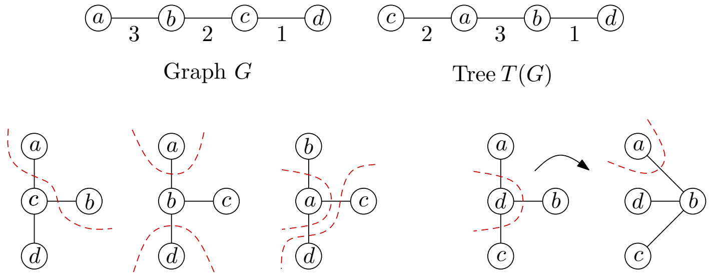
Notation. A Gomory-Hu tree is a weighted tree $T(G) = (V, E_T, c_T)$ on the vertices of an undirected (weighted) graph $G = (V, E, c)$ (with edges not necessarily in G) such that each $\{u, v\} \in E_T$ induces a minimum u-v-cut in G (by decomposing $T(G)$ into two connected components) and such that $c_T(\{u, v\})$ is equal to the cost of the induced cut. The cuts induced by $T(G)$ are non-crossing and for each $\{s, t\} \subseteq V$ each cheapest edge on the path $\pi(s, t)$ between s and t in $T(G)$ corresponds to a minimum s-t-cut in G. If G is disconnected, $T(G)$ contains edges of cost 0 between connected components. We identify the edges in a Gomory-Hu tree with the corresponding cuts without further notice. This allows for saying that a vertex is *incident* to a cut and an edge *separates* a pair of vertices. Furthermore, we consider a graph, and particularly, the path $\pi(u, v)$ between u and v in $T(G)$ and the tree $T(G)$, as the set of edges or the set of vertices it contains, as convenient. We say a cut *crosses* a set of vertices or a graph if it separates at least two vertices in the particular object (note, a set of edges also forms a graph). In few cases, we emphasize how often a cut crosses a (sub)graph, referring to the number of connected components that result from splitting. A cut crosses a connected graph k times if it decomposes the graph into $k + 1$ connected components.

Depending on the context, we also identify Gomory-Hu trees with the sets of cuts they represent. This admits to consider intersections, unions and subsets of Gomory-Hu trees and to denote the sizes of the resulting sets by $| \cdot |$. We call a set $\mathcal{H}$ of at most $n - 1$ cuts in a graph G a *(partial) Gomory-Hu set* if there exists a Gomory-Hu tree $T(G)$ of G with $T(G) \supseteq \mathcal{H}$. If $\mathcal{H} = T(G)$ we call $\mathcal{H}$ the Gomory-Hu set of $T(G)$. We say a Gomory-Hu tree $T(G)$ *contains* a cut θ if $\{\theta\} \subseteq T(G)$. We may also write $\theta \in T(G)$.

Reconnecting an edge in $T(G)$ means replacing the edge by another edge with the same attributes without creating a cycle. Recall further that a pair $\{s, t\}$ in G is called a cut pair of a cut if the cut is a minimum s-t-cut in G. In the next chapter, in the context of the construction of a Gomory-Hu tree, we will also introduce *step pairs* referring to the cut pairs that are considered for the computation of minimum separating cuts during the construction. These step pairs do not necessarily correspond to the final edges in the Gomory-Hu tree. Hence, if we talk about the step pairs of a Gomory-Hu tree $T(G)$, we mean the cut pairs considered during the construction. If we mention the cut pairs of a Gomory-Hu tree $T(G)$, we either mean any set of cut pairs with respect to the cuts in $T(G)$ or the cut pairs that are indicated by the tree edges. The particular interpretation is given by the context.

Equivalent Cuts in Disconnected Graphs and Consecutive Snapshots. Let $G = (V, E, c)$ denote an undirected, weighted graph that consists of at least two connected components. We say two cuts in G are equivalent if they cross the same set of edges. Hence, equivalent cuts in G have the same cost. Since G is disconnected, the cut sides of equivalent cuts however may differ in those connected components that are not split by the cuts. Nevertheless, a minimum s-t-cut is only equivalent to minimum s-t-cuts (not to other cuts), and two equivalent minimum s-t-cuts share the same set of cut pairs in G.

In a dynamic scenario, we can similarly compare cuts of consecutive snapshots G and $G^{\textcircled{U}}$. If $G^{\textcircled{U}}$ results from a vertex insertion or deletion, this particular vertex is a connected component in $G^{\textcircled{U}}$ in the first case and in G in the second case, while the edge set of G and $G^{\textcircled{U}}$ is the same. Hence, we define equivalent cuts in G and $G^{\textcircled{U}}$ as described above. Then it also holds that minimum s-t-cuts are only equivalent to minimum s-t-cuts, and two equivalent minimum



Graph G

Tree $T(G)$

FIGURE 6.3: Example of an undirected, weighted graph G (top left) and a flow-equivalent tree $T(G)$ (top right) that cannot be constructed by Gusfield's algorithm.

s-t-cuts in G and $G^{\textcircled{\scriptsize U}}$ share the same set of cut pairs. If $G^{\textcircled{\scriptsize U}}$ results from an edge insertion or deletion or the increase or decrease of an edge cost, two cuts in G and $G^{\textcircled{\scriptsize U}}$ are equivalent if, apart from the changing edge $\{b, d\}$, they cross the same edges and if they both either separate b and d or do not separate b and d. Note that in this case, a cut in $G^{\textcircled{\scriptsize U}}$ that is equivalent to a minimum s-t-cut in G is not necessarily a minimum s-t-cut in $G^{\textcircled{\scriptsize U}}$, and vice versa. In any case, we denote the equivalence of two cuts θ_1 and θ_2 by $\theta_1 = \theta_2$ and use the usual notation to describe relations between sets of cuts. We describe, for example, the set of equivalent cuts in a set $\mathcal{S}_1$ of cuts in G and a set $\mathcal{S}_2$ of cuts in $G^{\textcircled{\scriptsize U}}$ by $\mathcal{S}_1 \cap \mathcal{S}_2$.

Reusable Minimum Separating Cuts. In a dynamic scenario, we say a minimum separating cut in the current graph G is *reusable* or *remains valid* if it is also a minimum separating cut in $G^{\textcircled{\scriptsize U}}$ with respect to any cut pair. Some cuts can be only proven to remain valid with respect to a designated cut pair. In this case we usually also mention the cut pair or it is clear from the context that we currently consider only cuts that are reusable with respect to special cut pairs as, for example, the cut pairs given by the edges in the Gomory-Hu tree $T(G)$.

6.2.1 Constructing Flow-Equivalent Trees

Gusfield's algorithm for constructing a flow-equivalent tree $T(G) = (V, E_T, c_T)$ of a given graph $G = (V, E, c)$ is very simple. It assumes increasing indices $1, \ldots, n$ of the vertices in V and initially represents V as a star with center 1. This intermediate tree T_* is then rebuild step by step to the final flow-equivalent tree $T(G)$. To this end, the following operations are applied for $i = 2, \ldots, n$. Let c denote the (unique) neighbor of i in T_* and compute a minimum c-i-cut in G. Label the edge $\{c, i\}$ in T_* with $\lambda_G(c, i)$. Then reconnect each $j > i$ that is a neighbor of c but ends up on the same cut side as i to i, this is, replace the edge $\{c, j\}$ by the edge $\{i, j\}$ in T_*. Note that j is a leaf of T_* before and after this reconnection. The algorithm ends after $n-1$ cut computations and returns a flow equivalent tree $T_* = T(G)$ where the edges correspond to the vertex pairs $\{c, i\}$ used for computing the minimum separating cuts. The only degree of freedom provided by this simple procedure is the choice of the initial indices of the vertices. This however does not suffice to ensure solution-completeness, as the small example in Fig. 6.3 shows.

In order to understand the example, we observe the following. First, the minimum separating cuts in G are unique. Second, since the vertex pairs used for the construction of the flow-equivalent tree correspond to the edges in the tree, depending on the center 1 of the initial star T^*, we can already exclude those vertices from being indexed by 2 that are not incident to the center in the desired flow equivalent tree $T(G)$. With this kind of argument we can now show that the flow-equivalent tree $T(G)$, depicted at the top right in Fig. 6.3, cannot be constructed for the graph G, depicted at the top left in Fig. 6.3, by Gusfield's algorithm. To this end, we consider the initial stars at the bottom of Fig 6.3, each with another center. The first star has center c. Since a is the only vertex in $T(G)$ incident to c we must index a by 2 and apply the minimum a-c-cut in G depicted by the dashed red line in the star. This cut however separates all vertices from c besides d. Consequently, the algorithm must consider a minimum c-d-cut at one point, and thus, other than in $T(G)$, d becomes adjacent to c in the resulting tree. The second star has center b. By the same argument as above, we need to index either a or d by 2 and apply one of the corresponding minimum separating cuts (dashed red lines). However, none of these cuts separates c from b such that, other than in $T(G)$, in the final tree $\{b, c\}$ will occur as an edge. The third start has center a. Here, either c or b needs to be indexed by 2. With the minimum a-b-cut, however, a becomes a leaf in the final tree, while applying the minimum a-c-cut results in an edge $\{c, d\}$ in the final tree. In both cases the resulting tree is different from $T(G)$. The last star finally has center d. Hence, in a first step we need to apply the minimum b-d-cut, which separates d from all remaining vertices. Thus, in the second step b (indexed by 2) is the center. Now we need to apply the minimum a-b-cut in order to achieve the edge $\{a, b\}$ in the final tree. This cut however does not separate c from b such that in the end, we also get the edge $\{b, c\}$, which contradicts the edge structure in $T(G)$. In total, we have seen that there exists no indexing of the vertices in G such that the algorithm described above returns the flow equivalent tree $T(G)$.

6.2.2 The Classes of U-Cuts and M-Sets

Among all minimum separating cuts in a graph G, we point out the class $\mathcal{UC}(G)$ of *U-cuts*, which are used in many proofs and application due to their uniqueness. The class $\mathcal{UC}(G)$ is defined by the surjective mapping $\mathfrak{uc} : V \times V \longrightarrow \mathcal{UC}(G)$, $(s, t) \mapsto$ the unique minimum s-t-cut that minimizes the cut side that contains s. The uniqueness of these cuts follows from the Non-Crossing Lemma (7.2), which induces that at least one of two different minimum s-t-cuts with cut sides of s of the same size can be reshaped resulting in a minimum s-t-cut with a smaller cut side of s. Hence, the unique smallest cut side of s is nested in any other cut side of s with respect to t. We denote U-cuts in $\mathcal{UC}(G)$ by $\mathfrak{uc}(s, t)$. We further call the minimized cut sides of U-cuts *M-sets* and denote the M-set of a cut $\mathfrak{uc}(s, t)$ by $\mathfrak{m}(s, t)$ and the class of all M-sets by $\mathcal{MC}(G)$. This also induces a surjective mapping $\Phi : \mathcal{MC}(G) \rightarrow \mathcal{UC}(G)$. In this way, each cut pair $\{s, t\}$ is associated with two U-cuts, namely $\mathfrak{uc}(s, t)$ and $\mathfrak{uc}(t, s)$ and two M-sets, namely $\mathfrak{m}(s, t)$ and $\mathfrak{m}(t, s)$. We call $\mathfrak{uc}(s, t)$ the *opposite U-cut* of $\mathfrak{uc}(t, s)$, and $\mathfrak{m}(s, t)$ the *opposite M-set* of $\mathfrak{m}(t, s)$, and vice versa. Note further that neither $\mathfrak{uc}$ nor Φ is injective, that is, for two vertex pairs $(s, t) \neq (u, v)$ it might be $\mathfrak{uc}(s, t) = \mathfrak{uc}(u, v)$. Then it is $\Phi(\mathfrak{m}(s, t)) = \Phi(\mathfrak{m}(u, v))$, however, it is not necessarily $\mathfrak{m}(s, t) = \mathfrak{m}(u, v)$. In particular it might happen that $\mathfrak{uc}(s, t) = \mathfrak{uc}(t, s)$. In this case, the minimum s-t-cut is unique and induces two different M-sets, namely $\mathfrak{m}(s, t)$ and $\mathfrak{m}(t, s)$. We remark that a cut that is equivalent to an U-cut in a disconnected graph is not necessarily a U-cut.

We further introduce *generalized U-cuts* and *generalized M-sets*. For a vertex s and a subset $T \subsetneq V$, we call the minimum s-T-cut that minimizes the cut side of s a *generalized U-cut* and the cut side containing s a *generalized M-set*. Analogous to U-cuts and M-sets we denote generalized U-cuts by $\mathfrak{uc}(s, T)$ and generalized M-sets by $\mathfrak{m}(s, T)$. Note that a generalized U-cut $\mathfrak{uc}(s, T)$ in G corresponds to a (regular) U-cut $\mathfrak{uc}(s, \{T\})$ in the graph G' that is obtained from G by contracting T, resulting in a compound node denoted by $\{T\}$. Hence, generalized U-cuts and generalized M-sets behave like regular U-cuts and regular M-sets and can be computed just like

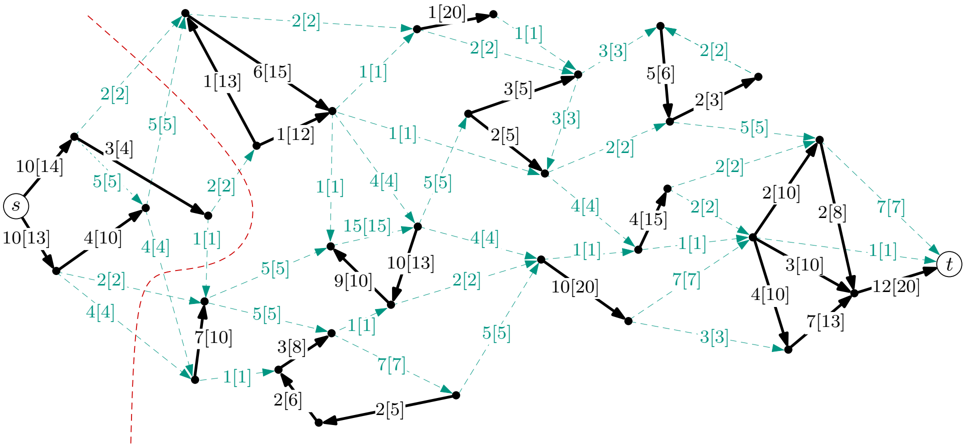


FIGURE 6.4: Graph G'' resulting from a maximum s-t-flow f with value $\mathfrak{v}(f) = 20$ in an augmented, directed graph G'. The edge costs (in G' as well as in the original undirected graph G) are denoted in brackets, the remaining numbers denote the flow per edge. Saturated edges (that is, edges e with $f(e) = c(e)$) are depicted as dashed green arrows. The minimum s-t-cut that minimizes the cut side S of s is depicted as dashed red line. Here, S consists of four vertices, the cut is crossed by seven saturated edges.

these. On the other hand, each regular U-cut is also a generalized U-cut and each regular M-set is also a generalized M-set.

6.2.3 Maximum Flows and DAGs

Classically, network flows are considered in directed graphs with capacities assigned to the edges, instead of undirected graphs. However, there are many ways to interpret an undirected, weighted graph as a directed flow network. In this work, we follow the approach of Nagamochi [110].

Maximum Flows in Undirected, Weighted Graphs. Let $G = (V, E, c)$ denote an undirected, weighted graph. Then a directed, weighted graph $G' = (V, E', c')$ can be obtained from G by replacing each edge in E with two oppositely oriented edges of the same cost. The costs defined by the function c' are considered as edge capacities. For a fixed vertex pair $\{s, t\} \subseteq V$, a *flow from* s *to* t (or an s-t-*flow* for short) in G' is then defined as in Section 1.3.1, that is, as a function $f : E' \to \mathbb{R}_0^+$ that satisfies the *flow conservation property* and the *capacity constraint*. Each flow f in G' can be further transformed into a flow f' of the same value where $f'(u, v) = 0$ or $f'(v, u) = 0$ for each edge $\{u, v\} \in E$. To this end, we define $f'(u, v) := f(u, v) - f(v, u)$ if $f(u, v) - f(v, u) > 0$ and $f'(u, v) = 0$, otherwise. Analogously, we define $f'(v, u) := f(v, u) - f(u, v)$ if $f(v, u) - f(u, v) > 0$ and $f'(v, u) = 0$, otherwise. The edges $e \in E'$ with $f'(e) > 0$ then induce a directed graph G'' without edges in opposite directions. Figure 6.4 shows such a graph G'' resulting from a maximum s-t-flow in an augmented, directed graph G'. The famous MaxFlow-MinCut Theorem finally states that $\mathfrak{v}(f) = \lambda_G(s, t)$ for each maximum flow f from s to t in G' and G'', respectively. Since the edge connectivity of $\{s, t\}$ in G does not depend on the order of s and t, it further follows that $\lambda_G(s, t) = \mathfrak{v}(f)$ also holds if f is a maximum flow from t to s.

Besides the connectivity value $\lambda_G(s, t)$, which can be easily determined in linear time, a maximum s-t-flow f in G'' (or G') provides even more information. By applying a BFS in G'' (or G') starting at s and not passing *saturated* edges, that is, edges e where $f(e) = c(e)$ holds,

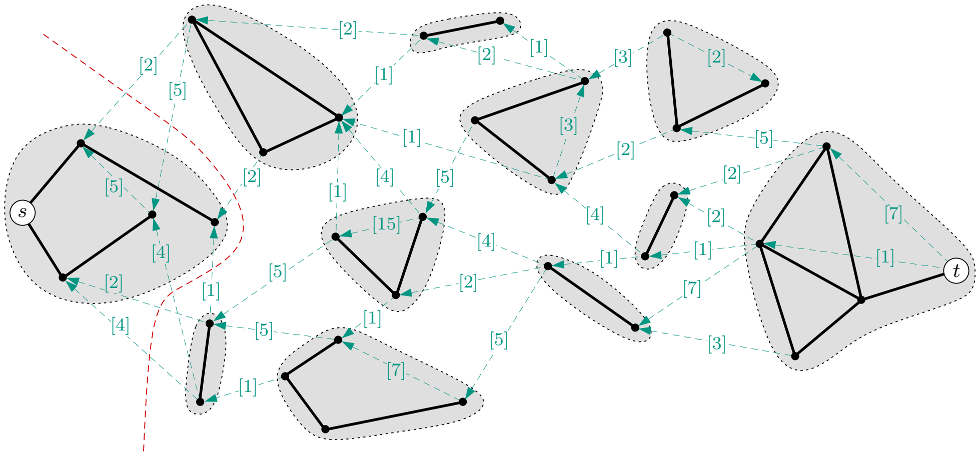


Figure 6.5: Graph H resulting from graph G'' in Fig. 6.4. The edge costs at the saturated edges are denoted in brackets. Oppositely oriented edge pairs are depicted as undirected black edges. Gray regions indicate strongly connected components.

we can also determine the side $S \ni s$ of the minimum s-t-cut in G that is crossed by exactly the saturated edges that have their head in S and their tail in $V \setminus S$, see Fig. 6.4. Hence, S, and thus, the minimum s-t-cut $(S, V \setminus S)$, can be determined in $O(m)$ time. We further remark that the cut $(S, V \setminus S)$ determined in this way is the unique cut among all minimum s-t-cuts in G that minimizes the size of the cut side that contains s. We call such a cut a U-cut. The class of U-cuts is introduced in the next paragraph. Depending on the exact definition and the exact use of the labels, some push-relabel algorithms for computing a maximum flow further allow to decide in constant time (by the help of the labels finally assigned to the vertices) to which cut side of the U-cut a given vertex belongs. One example is the pioneering algorithm of Goldberg and Tarjan [57]. Today many of the fastest maximum-flow algorithms rely on the push-relabel technique. For an introduction to push-relabel algorithms see for example [27]. Picard and Queyranne [118] showed that even all minimum s-t-cuts in G can be determined from a maximum s-t-flow in G''. The minimum s-t-cuts can be represented by a directed acyclic graph (a so-called DAG), which can be constructed in $O(m)$ time. We describe the construction of such a Picard-Queyranne DAG in more detail in one of the following paragraphs.

Picard-Queyranne DAGs. In the literature exist many different ways to construct a Picard-Queyranne DAG (or PQ-DAG for short) from a given maximum s-t-flow. Here we follow the approach of Gusfield and Naor [69].

We consider a maximum s-t-flow in a directed graph G'' that has no oppositely oriented edges resulting from an underlying undirected graph G, as described above. This graph is now again augmented by oppositely oriented edges, resulting in a directed graph H, however, this time only non-saturated edges with positive flow are replaced, while saturated edges (with positive flow) are just reversed. Furthermore, the flow values are no longer interesting in H. We only keep the edge costs at the saturated edges (which for these edges correspond to the flow values). Figure 6.5 shows the graph H resulting from the graph G'' in Fig. 6.4. The Picard-Queyranne DAG with respect to s and t, denoted by $\mathrm{DAG}_{s,t}$, is then obtained from H by contracting each *strongly connected component* in H. A set S of vertices in H is a strongly connected component if each vertex in S is reachable from every other vertex in S and S is maximal with respect to this property. The partition of a directed graph into its strongly connected components, and

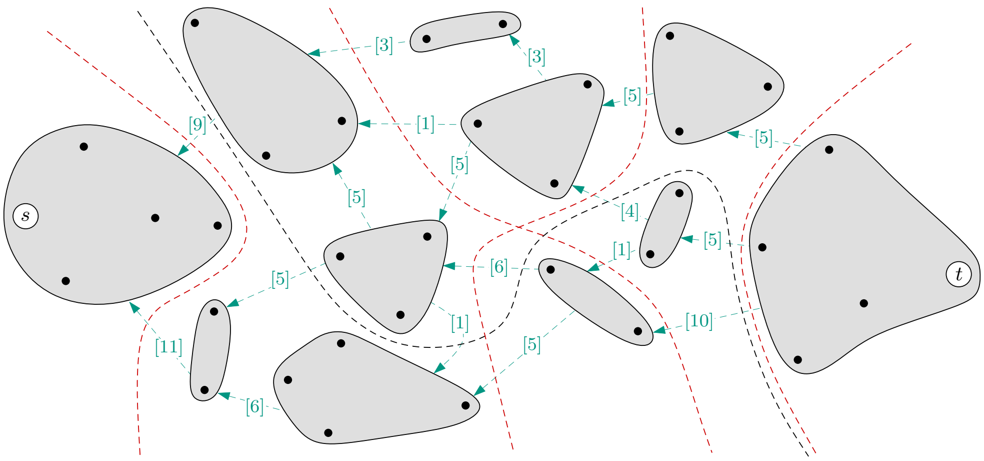


FIGURE 6.6: $\mathrm{DAG}_{s,t}$ resulting from the graph H in Fig. 6.5. The (accumulated) edge costs resulting from the contraction of strongly connected components are denoted in brackets. The U-cut $\mathfrak{uc}(s,t)$ and the opposite U-cut $\mathfrak{uc}(t,s)$ as well as two further minimum s-t-cuts are depicted as dashed red lines. The dashed black line indicates an s-t-cut that is not minimum, since it does not correspond to a closed set in the $\mathrm{DAG}_{s,t}$.

thus the $\mathrm{DAG}_{s,t}$, can be computed in $O(m)$ time. Figure 6.6 shows the $\mathrm{DAG}_{s,t}$ resulting from the graph H in Fig. 6.5. According to Picard and Queyranne [118], there is then a one-to-one correspondence between the minimum s-t-cuts in G and the *closed sets* (containing the node that contains s) in the $\mathrm{DAG}_{s,t}$. A set S of nodes in the $\mathrm{DAG}_{s,t}$ is closed if there are no outgoing edges leaving S. Figure 6.6 exemplarily depicts some minimum s-t-cuts corresponding to closed sets. Furthermore, let $\{u,v\}$ denote a pair of nodes in the $\mathrm{DAG}_{s,t}$ such that, without loss of generality, v is not reachable from u. This situation always exists, since the $\mathrm{DAG}_{s,t}$ is acyclic. Then u together with the nodes that are reachable from u form a closed set in the $\mathrm{DAG}_{s,t}$ that does not contain v. This set thus corresponds to a minimum s-t-cut that separates all original vertices (in G) that are contained in u from all original vertices (in G) that are contained in v. In particular, deciding whether two original vertices x and y are separated by a minimum s-t-cut in G equals the decision whether x and y are contained in different nodes in the $\mathrm{DAG}_{s,t}$, and thus, can be done in constant time, given that membership of the vertices in G to the nodes in the $\mathrm{DAG}_{s,t}$ is indicated for example by indices assigned to the vertices, which can be easily achieved during the construction of the $\mathrm{DAG}_{s,t}$.

Updating DAGs and Maximum Flows. In Chapter 8 we consider the all-pairs minimum-cut problem in a dynamic scenario and prove (see Chapter 9) that our update algorithm proposed for Gomory-Hu trees is optimal in terms of asymptotic worst-case running time compared to the asymptotic worst-case running time that could be achieved if, instead of a regular Gomory-Hu tree, an extended Gomory-Hu tree was available for the current snapshot. The proof relies on the *recomputation conjecture*, which we argue in the following is reasonable. The conjecture states that finding a minimum s-t-cut in $G^{\textcircled{\scriptsize U}}$ that is neither an old minimum separating cut in G (with respect to any cut pair) nor associated with a known maximum s-t-flow in G, needs at least one cut computation from scratch in $G^{\textcircled{\scriptsize U}}$. The first condition says that the minimum s-t-cut is not represented by a DAG in G. The second condition implies that there is no maximum flow that can be updated, even if efficiently updating a maximum flow was possible. In the following, we briefly discuss update techniques for maximum flows and DAGs, which we think are reasonable and not too far from the best we can do. The difficulties in updating maximum flows suggest that efficiently deducing a maximum s-t-flow in G (which then could be updated) from the known maximum flows in G is not possible. The discussion on updating DAGs will show that updating a DAG does not generate new cuts, but restricts the previous DAG to a subset of the previously represented cuts. Hence, updating DAGs maintains even less information than updating flows. Together, these observations suggest that the recomputation conjecture is plausible.

Recall that in the dynamic scenario we consider atomic edge changes, while vertex changes are only allowed for singletons. We start with the discussion on update techniques for maximum flows. In the literature many results can be found for so-called *dynamic flows in networks with transit times* [117, 44]. However, the graph models used for these dynamic flow problems and the flow problems themselves are fundamentally different from the task of efficiently updating a given maximum flow after a change in the underlying graph. The latter problem seems to be discussed only rarely although one could imagine many fields of application. Computer vision, more precisely, image segmentation in videos, is one field where such incremental flow updates are actually used. Hence, we exemplarily refer to Kohli and Torr [96] who present a fully-dynamic update algorithm for maximum flows, which seems to perform well at least in the context of image segmentation in videos. Unfortunately, it also seems that proving a concrete speedup of this and other update techniques for maximum flows is very difficult. The technique of Kohli and Torr modifies the current graph G according to the previous change in G and the previous flow value of the changing edge in constant time, such that the previous s-t-flow can be easily modified satisfying the capacity constraint in the modified graph G'. A maximum s-t-flow is then obtained from the modified s-t-flow by iteratively finding augmenting paths from s to t where the flow can be increased. The running time of this approach depends on the number of paths found during this procedure and the difference of the values of the modified s-t-flow and the resulting maximum s-t-flow. Furthermore, the resulting maximum s-t-flow in the modified graph G' is not necessarily a maximum s-t-flow in the original graph G. Just the connectivity $\lambda_G(s,t)$ and at least one minimum s-t-cut can be obtained from the maximum s-t-flow in G'. We further remark that, in contrast to DAGs, where a new $\mathrm{DAG}_{t,u}$ can be deduced form a $\mathrm{DAG}_{s,t}$ and a $\mathrm{DAG}_{u,v}$ in time $O(m)$ [69], to the best of our knowledge, generating new flows in this way is not possible with a better asymptotic worst-case running time than a maximum-flow computation from scratch. This is due to similar difficulties that prevent a fast and elegant flow update.

For updating DAGs we can give a very natural approach. First, consider the insertion of an edge $\{b,d\}$ in G or the increase of the edge cost. If for a vertex pair $\{s,t\}$ the connectivity does not change, that is, $\lambda_G(s,t) = \lambda_{G^\oplus}(s,t)$, then the new $\mathrm{DAG}_{s,t}$ represents exactly the previous minimum s-t-cuts that do not separate b and d. If the connectivity increases by exactly Δ (that is, the amount the cost of $\{b,d\}$ increases), all previous minimum s-t-cuts separate b and d but remain valid and the new $\mathrm{DAG}_{s,t}$ represents those cuts and possibly some additional new cuts. If the connectivity increases by an amount less than Δ, then the new $\mathrm{DAG}_{s,t}$ represents a completely new set of cuts. In the first case, we obtain the new $\mathrm{DAG}_{s,t}$ from the current $\mathrm{DAG}_{s,t}$ by simply contracting the nodes B and D that contain b and d and the nodes on any path between B and D, resulting in exactly one new node. Since deciding which node contains b and d, respectively, is possible in constant time, this can be done in $O(m)$ time, where m refers to the set of edges in the $\mathrm{DAG}_{s,t}$. In the second case we can either be satisfied with keeping the old $\mathrm{DAG}_{s,t}$, which at least represents some of the new cuts or we seek for a maximum flow in the new graph in order to construct the new $\mathrm{DAG}_{s,t}$ from scratch. The latter can be done by either updating (if known) or recomputing the current maximum s-t-flow. The same holds for the third case, where computing the new $\mathrm{DAG}_{s,t}$ from scratch seems to be the most natural approach.

Now consider the deletion of an edge $\{b, d\}$ in G or the decrease of the edge cost. If for a vertex pair $\{s, t\}$ the connectivity does not change, that is, $\lambda_G(s, t) = \lambda_{G^\oplus}(s, t)$, then none of the previous minimum s-t-cuts separates b and d but all cuts remain valid. Thus, the new $\mathrm{DAG}_{s,t}$ represents those cuts and possibly some additional new cuts. If the connectivity decreases by exactly Δ (that is, the amount the cost of $\{b, d\}$ decreases), the new $\mathrm{DAG}_{s,t}$ represents exactly the previous minimum s-t-cuts that separate b and d. If the connectivity decreases by an amount less than Δ, then the new $\mathrm{DAG}_{s,t}$ represents a completely new set of cuts. In the first case we can either be satisfied with keeping the old $\mathrm{DAG}_{s,t}$, which at least represents some of the new cuts or we seek for a maximum flow in the new graph in order to construct the new $\mathrm{DAG}_{s,t}$ from scratch. The latter can be again done by either updating (if known) or recomputing the current maximum s-t-flow. The same holds for the third case, where computing the new $\mathrm{DAG}_{s,t}$ from scratch seems to be the most natural approach. In the second case, without loss of generality, let B denote the node that contains b and D the node that contains d and D not reachable from B. Then we obtain the new $\mathrm{DAG}_{s,t}$ from the current $\mathrm{DAG}_{s,t}$ by simply contracting B and all nodes that are reachable from B as well as D and all nodes from which D is reachable, resulting in exactly two new nodes. Since deciding which node contains b and d, respectively, is possible in constant time, this can be done in $O(m)$ time, where m refers to the set of edges in the $\mathrm{DAG}_{s,t}$.

We conclude that with this natural update approach for DAGs the worst case running time for finding a new minimum s-t-cut in $G^{\textcircled{U}}$ is still equal to the asymptotic worst-case running time of a maximum-flow computation from scratch. Such a worst case occurs if $\lambda_{G^{\textcircled{U}}}(s, t)$ changes by less that Δ, and thus, the previous $\mathrm{DAG}_{s,t}$ in G becomes invalid.

Summing up the discussion, we see that if for a vertex pair $\{s, t\}$ in $G^{\textcircled{U}}$ there is no corresponding flow known in G that could be updated, constructing such a flow from other flows does not pay off. Furthermore, if the $\mathrm{DAG}_{s,t}$ becomes invalid in $G^{\textcircled{U}}$, we also need to invest a maximum-flow computation in order to find a new minimum cut in $G^{\textcircled{U}}$. Note that the $\mathrm{DAG}_{s,t}$ becomes invalid if and only if the $\mathrm{DAG}_{t,s}$ becomes invalid. This justifies the recomputation conjecture.

CHAPTER 7

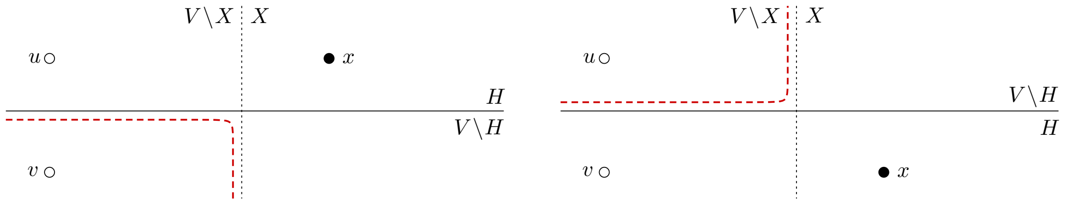
Gomory-Hu Trees in Static Graphs

In this chapter we provide the basics for Chapter 8 where we will consider Gomory-Hu trees in dynamic scenarios, and Chapter 14 where we employ the results of Chapter 8 to also dynamically update cut-based clusterings, which are closely related to Gomory-Hu trees. Furthermore, we develop the concept of unique-cut trees, which provide the base for the maximum source-community clusterings considered in Chapter 12. Moreover, unique-cut trees are also of great interest by themselves. In contrast to Gomory-Hu trees, which provide an arbitrary minimum separating cut for each vertex pair, they represent all regular U-cuts and the corresponding M-sets of an undirected, weighted graph. Unique-cut trees are presented in Section 7.2. In Section 7.1 we start with a characterization of the sets of minimum separating cuts that can be represented by a Gomory-Hu tree and a description of the original algorithm of Gomory and Hu for constructing a Gomory-Hu tree. Compared to the original paper, our description is much more compact and we introduce some special terms for important objects considered by the algorithm in order to refer to these objects in later argumentation. Furthermore, we reformulate Gomory's and Hu's algorithm based on the ideas of Gusfield, who presented an elegant simplification of the algorithm in 1990. Compared to the algorithm given by Gusfield, our reformulation is more general, that is, it provides a higher degree of freedom in choosing vertices and cuts. We will exploit this fact for the update procedures developed in Chapter 8 and Chapter 14.

7.1 Basics for Constructing Gomory-Hu Trees

In Chapter 6 we have seen that without further modification Gusfield's algorithm for constructing flow-equivalent trees is not solution-complete, that is, there exist flow-equivalent trees that cannot be constructed by this algorithm. For Gomory-Hu trees one can postulate the same question. In this section we will see that the original algorithm of Gomory and Hu as well as our formulation of Gusfield's simplification are solution-complete. To this end, we characterize Gomory-Hu sets, that is, the sets of minimum separating cuts that can be represented by a Gomory-Hu tree, in Lemma 7.1, and observe that each Gomory-Hu set allows to completely fix the degree of freedom in the algorithms, such that an execution of the algorithms returns the unique Gomory-Hu tree that represents the initial Gomory-Hu set.

Lemma 7.1. *A set $\mathcal{H}$ of $n-1$ cuts is a Gomory-Hu set if and only if the cuts in $\mathcal{H}$ are pairwise non-crossing and for each cut exists a cut pair that is exclusively separated by this cut, that is, that is not separated by any other cut in $\mathcal{H}$.*



(a) Deflected by x, the Non-Crossing Lemma (7.2) bends $(H, V \setminus H)$ downwards along X.

(b) Deflected by x, the Non-Crossing Lemma (7.2) bends $(H, V \setminus H)$ upwards along X.

Figure 7.1: Situation of the Non-Crossing Lemma (7.2). Reshaping newly found minimum separating cuts (solid black lines) along previous minimum separating cuts (dotted black lines) resulting in new cuts with the same costs (dashed red lines). Note that y is not depicted, since the statement does not depend on the position of y.

Proof. Obviously, each Gomory-Hu set satisfies the conditions of Lemma 7.1. Now let $\mathcal{H}$ denote a set of $n-1$ pairwise non-crossing cuts and $\mathcal{P}$ a set of $|\mathcal{H}|$ exclusively separated cut pairs, as described in Lemma 7.1. Since the cuts in $\mathcal{H}$ are non-crossing, their cut sides are nested, and since there are $n-1$ of these cuts, there exists exactly one vertex pair for each cut that is exclusively separated by this cut. Hence, these vertex pairs must correspond to the cut pairs in $\mathcal{P}$. Connecting cut pairs in $\mathcal{P}$ by edges then yields a Gomory-Hu tree that represents $\mathcal{H}$. ☐

7.1.1 The Algorithm of Gomory and Hu

The simple idea of this algorithm is to iteratively construct a Gomory-Hu set by choosing in each step a vertex pair (a so-called *step pair* consisting of *step vertices*) that is not yet separated and computing a minimum separating cut for this pair. The challenging part is to ensure that the found cut does not cross any of the previous cuts. Gomory and Hu solve this problem by contracting at least one cut side of each cut found so far and computing the minimum separating cut in the resulting graph. This is feasible due to the following key lemma, which we will use extensively in the remainder of this work, and thus, refer to as *Non-Crossing Lemma.* According to this lemma, there always exists a minimum separating cut that does not cross any of the previous cuts. While Gomory and Hu proved this assertion in a much more complicated way, this lemma was formulated and proven by Gusfield. It shows that any arbitrary minimum separating cut can be bent along the previous cuts without changing costs, thus resolving any potential crossings (see also Fig. 7.1)

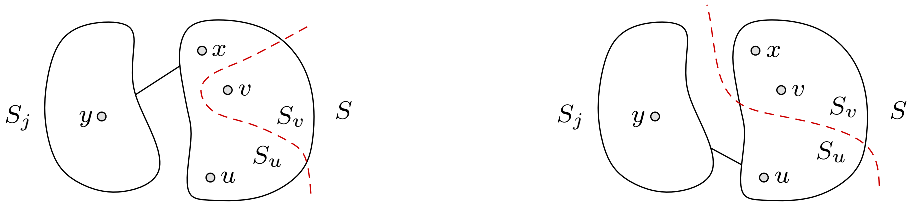
Lemma 7.2 (Non-Crossing Lemma, Lem. 1 in Gus. [66]). *Let $(X, V \setminus X)$ be a minimum x-y-cut in G, with $x \in X$. Let $(H, V \setminus H)$ be a minimum u-v-cut, with $u, v \in V \setminus X$ and $x \in H$. Then the cut $(H \cup X, (V \setminus H) \cap (V \setminus X))$ is also a minimum u-v-cut.*

We say that $(X, V \setminus X)$ *shelters* X, meaning that each minimum u-v-cut with $u, v \notin X$ can be reshaped, such that it does no longer split X.

Algorithm 1 briefly describes the tree construction of Gomory and Hu. The *intermediate tree* $T_* = (V_*, E_*, c_*)$ is initialized as an isolated, edgeless node containing all original vertices. Then, until each node of T_* is a singleton node, a node $S \in V_*$ is *split* and an edge representing a new cut is added to E_*. To this end, nodes $S' \neq S$, which are related to cut sides of previously found cuts, are dealt with by contracting in G whole subtrees N_j of S in T_*, connected to S via edges $\{S, S_j\}$, to single nodes $[N_j]$ before cutting, which yields G_S. The split of S into S_u and S_v is then defined by a minimum u-v-cut (*split cut*) in G_S (with *step pair* $\{u, v\} \subseteq S$), which

Algorithm 1: Gomory-Hu

```
Input: Graph $G = (V, E, c)$
Output: Gomory-Hu tree of $G$
Initialize tree $T_* := (V_*, E_*, c_*)$ with $V_* \leftarrow \{V\}, E_* \leftarrow \emptyset$ and $c_*$ empty
while $\exists S \in V_*$ with $|S| > 1$ do                                  // unfold all nodes
    $\{u, v\} \leftarrow$ arbitrary pair of vertices in $\binom{S}{2}$
    forall the $S_j$ adjacent to $S$ in $T_*$ do $N_j \leftarrow$ subtree of $S$ in $T_*$ with $S_j \in N_j$
    $G_S = (V_S, E_S, c_S) \leftarrow$ in $G$ contract each $N_j$ to $[N_j]$          // contraction
    $(U, V \setminus U) \leftarrow$ min-$u$-$v$-cut in $G_S$, cost $\lambda_{G_S}(u, v)$, $u \in U$
    $S_u \leftarrow S \cap U$ and $S_v \leftarrow S \cap (V_S \setminus U)$                 // split $S = S_u \dot\cup S_v$
    $V_* \leftarrow (V_* \setminus \{S\}) \cup \{S_u, S_v\}$, $E_* \leftarrow E_* \cup \{\{S_u, S_v\}\}$, $c_*(S_u, S_v) \leftarrow \lambda_{G_S}(u, v)$
    forall the former edges $e_j = \{S, S_j\} \in E_*$ do
        if $[N_j] \in U$ then $e_j \leftarrow \{S_u, S_j\}$ ;                        // reconnect $S_j$ to $S_u$
        else $e_j \leftarrow \{S_v, S_j\}$ ;                                      // reconnect $S_j$ to $S_v$
return $T_*$
```



(a) If $x \in S_u$, $\{x, y\}$ is still a cut pair of $\{S_u, S_j\}$.

(b) If $x \notin S_u$, $\{u, y\}$ is a cut pair of $\{S_u, S_j\}$.

Figure 7.2: Situation in Lemma 7.3. There always exists a cut pair of edge $\{S_u, S_j\}$ in the incident nodes S_u and S_j, independent of the shape of the split cut (dashed line).

does not cross any of the previously used cuts due to the contraction technique. Recall that this split cut in G_S also induces a minimum u-v-cut in G, due to the Non-Crossing Lemma (7.2). After the splitting, the edge $\{S_u, S_v\}$ is added to E_*, and each N_j is reconnected, again by S_j, to either S_u or S_v depending on which side of the cut $[N_j]$ ended up. This reconnection ensures that the edge $\{S_u, S_v\}$ indeed represents the split cut.

The previous split cuts represented in T_*, and in particular the cuts represented by the reconnected edges, do not change due to the reconnection. But the reconnection changes the incident nodes of the reconnected edges, and thus, in order to ensure the correctness of the algorithm, the new nodes incident to such an edge also need to contain a cut pair of the cut represented by the edge. This is guaranteed by Lemma 7.3, which states that each edge $\{S, S_j\}$ in T_* has a cut pair $\{x, y\}$ with $x \in S$, $y \in S_j$ (see Fig. 7.2). An intermediate tree satisfying this condition is *valid*. The lemma was formulated and proven by Gomory and Hu [59] and rephrased by Gusfield [66].

Lemma 7.3 (Lem. 4 in Gus. [66])**.** *Let $\{S, S_j\}$ be an edge in T_* inducing a cut with cut pair $\{x, y\}$, where $x \in S$ and $y \in S_j$. Consider step pair $\{u, v\} \subseteq S$ that splits S into S_u and S_v, without loss of generality S_j and S_u ending up on the same cut side, that is $\{S_u, S_j\}$ becomes a new edge in T_*. If $x \in S_u$, $\{x, y\}$ remains a cut pair for $\{S_u, S_j\}$. If $x \in S_v$, $\{u, y\}$ is also a cut pair of $\{S_u, S_j\}$.*

7.1.2 A Simpler Version of the Gomory-Hu Algorithm

While Gomory and Hu use contractions in G to prevent crossings of the cuts, as a simplification, Gusfield computes arbitrary minimum separating cuts directly in G (without previous contraction) and resolves potential crossings by the help of the Non-Crossing Lemma. To this end, he introduces a special representation of the intermediate tree that admits to reshape the cuts according to the Non-Crossing Lemma by just reconnecting further edges. The results on Gomory-Hu trees in this work will be all based on Gusfield's simplification instead of the original Gomory-Hu algorithm, but we slightly generalize Gusfield's representation of intermediate trees in order to make it compatible with the special intermediate trees we will use in the dynamic scenario in the next chapter.

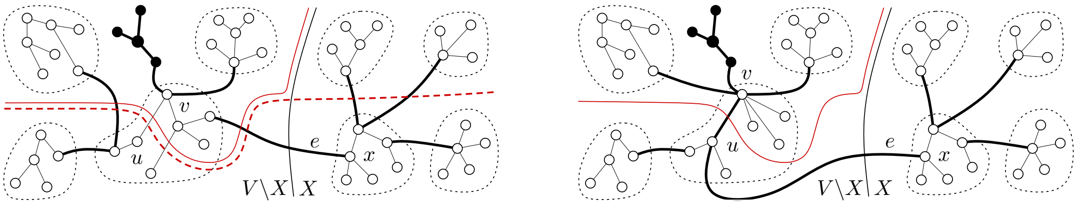
Representation of Intermediate Trees. In an intermediate tree $T_* = (V_*, E_*, c_*)$, we present each node in V_*, which consists of original vertices in V, by an arbitrary tree of *thin* edges connecting the contained vertices in order to indicate their membership to the node. An edge connecting two nodes in V_* is represented by a *fat* edge in E_*, which we connect to an arbitrary vertex in each incident node. Fat edges represent minimum separating cuts in G, and are thus associated with costs. If a node contains only one vertex, we color this vertex black. Black vertices are only incident to fat edges. The vertices in non-singleton nodes are colored white. White vertices are incident to at least one thin edge. In this way, $T_* = (V, E_t, E_f, c_f)$ can be also considered as a tree on V with two types of edges (thin edges in E_t, fat edges in E_f) and vertices (black and white), and a cost function c_f on the fat edges. For an example see the intermediate tree depicted in Fig. 7.3(a).

Description of the Algorithm. With this representation, addressing the subtrees of a node in T_* can be easily done by just addressing the vertices that are linked by a fat edge to a vertex in the node. Hence, Algorithm 1 can be reformulated as done in Algorithm 2. In order to distinguish this simpler version from the original algorithm, we refer to this algorithm as Cut Tree algorithm.
According to the representation described above, Algorithm 2 considers the intermediate tree T_* as a tree on the vertices of G consisting of thin and fat edges and omits any contraction in G. Instead, Line 12 and 13 implement the reconnection of the subtrees of the current node S with respect to this more direct representation of T_*, and additionally, also realize the reshaping of the current split cut $(U, V \setminus U)$ according to the Non-Crossing Lemma (7.2). Figure 7.3 again illustrates these reconnection steps. The current step pair $\{u, v\}$ (filled gray) is separated by the current split cut $(U, V \setminus U)$ (dashed red line), which crosses the previously found cut $(X, V \setminus X)$ represented by the edge e (solid black line), see Fig. 7.3(a). Thus, according to the Non-Crossing Lemma (7.2), $(U, V \setminus U)$ is meant to be reshaped resulting in the cut indicated by the solid red line. Furthermore, the edge $\{u, v\}$ is meant to represent this reshaped cut. Both issues are now handled by reconnecting edges, compare Fig. 7.3(b). The edge e is reconnected such that $(U, V \setminus U)$ no longer splits the cut side X. This realizes the reshaping. Furthermore, the thin edges in the current node S, that is, the node that contains u and v, are reconnected such that the edge $\{u, v\}$ indeed represents the reshaped cut.

Since the step pairs and the split cuts are chosen arbitrarily, the procedure Cut Tree is non-deterministic, but a single run of Cut Tree is characterized by a sequence $\mathcal{F}$ of $n-1$ step pairs and a sequence $\mathcal{K}$ of $n-1$ corresponding split cuts, and thus, returns a unique Gomory-Hu

Algorithm 2: CUT TREE

```
Input: Graph $G = (V, E, c)$
Output: Gomory-Hu tree of $G$
Initialize tree $T_* = (V, E_t, E_f, c_f)$ with $E_t \leftarrow$ thin edges forming an arbitrary spanning tree
  of $V$, $E_f \leftarrow \emptyset$ and $c_f$ empty
while $\exists$ thin edge $\{s, t\}$ in $E_t$ do                                  // unfold all nodes
    $S \leftarrow$ subtree of thin edges that contains $\{s, t\}$                        // choose node
    $\{u, v\} \leftarrow$ arbitrary pair of vertices in $\binom{S}{2}$                        // choose step pair
    $(U, V \setminus U) \leftarrow$ min-$u$-$v$-cut in $G$, cost $\lambda_G(u, v)$, $u \in U$             // choose split cut
    $E_t \leftarrow E_t \setminus S$
    forall the $x \in S \setminus \{u, v\}$ do                                   // split $S = S_u \dot\cup S_v$
        if $x \in U$ then $E_t \leftarrow E_t \cup \{\{u, x\}\}$;                          // reconnect $x$ to $u$
        if $x \in V \setminus U$ then $E_t \leftarrow E_t \cup \{\{v, x\}\}$;                      // reconnect $x$ to $v$
    $N \leftarrow$ all vertices $x$ that are linked by a fat edge to a vertex $v_x \in S$
    forall the $x \in N$ do                          // reconnect subtrees to $S_u$ and $S_v$
        if $x \in U$ then $E_t \leftarrow (E_t \setminus \{\{v_x, x\}\}) \cup \{\{u, x\}\}$;              // reconnect $x$ to $u$
        if $x \in (V \setminus U)$ then $E_t \leftarrow (E_t \setminus \{\{v_x, x\}\}) \cup \{\{v, x\}\}$;         // reconnect $x$ to $v$
    $E_f \leftarrow E_f \cup \{\{u, v\}\}$, $E_t \leftarrow E_t \setminus \{\{u, v\}\}$, $c_f(u, v) \leftarrow \lambda_G(u, v)$          // draw $\{u, v\}$ fat
return $T_*$
```



(a) Original and reshaped split cut in an intermediate tree T_* before the reconnection. Nodes in T_* are indicated by dotted frames.

(b) Reconnecting edges according to CUT TREE ensures the reshaping of the split cut and the representation of the reshaped cut by the new fat edge $\{u, v\}$.

FIGURE 7.3: Reconnecting edges implements splitting of the current node and reshaping of the split cut. The original split cut $(U, V \setminus U)$ (dashed red line) with respect to step pair $\{u, v\}$ is bent along the cut $(X, V \setminus X)$ represented by edge e (solid black line) resulting in a non-crossing split cut (solid red line).

tree. We call a triple $(G, \mathcal{F}, \mathcal{K})$ consisting of a graph G, a sequence $\mathcal{F}$ of $|\mathcal{F}| \leq n - 1$ step pairs, and a sequence $\mathcal{K}$ of $|\mathcal{K}| = |\mathcal{F}|$ corresponding split cuts a *(partial)* CUT TREE *execution*, and observe that each (partial) CUT TREE execution returns a unique valid intermediate tree, which is a Gomory-Hu tree for $|\mathcal{F}| = n - 1$. The set $\mathcal{K}$ of split cuts is represented by the (fat) edges of this intermediate tree and can be considered as a (partial) Gomory-Hu set, since further processing the intermediate tree would yield a Gomory-Hu tree with a Gomory-Hu set $\mathcal{H}$ such that $\mathcal{K} \subseteq \mathcal{H}$.

Vice versa, the following lemma states sufficient conditions for a set $\mathcal{K}$ of $k \leq n-1$ cuts in G, such that there exists a valid intermediate tree that exactly represents the cuts in $\mathcal{K}$.

Lemma 7.4. *Let $\mathcal{K}$ denote a set of $k \leq n-1$ pairwise non-crossing cuts in G such that for each cut exists a cut pair that is exclusively separated by this cut, that is, that is not separated by any other cut in $\mathcal{K}$. Then there exists a valid intermediate tree representing exactly the cuts in $\mathcal{K}$.*

Proof. Let $\mathcal{F}$ denote the set of $|\mathcal{K}|$ exclusively separated cut pairs, as described in Lemma 7.4. The statement follows inductively from the correctness of the procedure CUT TREE. Consider a run of CUT TREE that uses the elements in $\mathcal{F}$ as step pairs in an arbitrary order and the

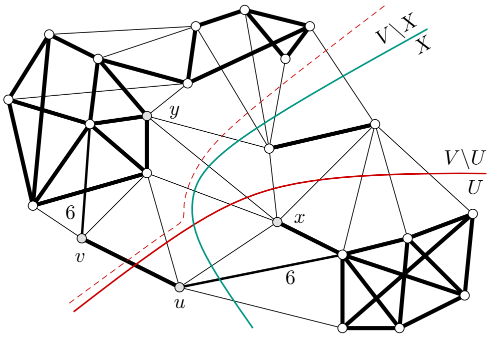


FIGURE 7.4: Crossing U-cuts in a graph G. Fat edges are very expensive, thin edges cost 1. The two remaining edges cost 6, each. Cut $(X, V \setminus X)$ (solid green line) is the minimum x-y-cut that minimizes the cut side of y. Cut $(U, V \setminus U)$ (solid red line) is the minimum u-v-cut that minimizes the cut side of u, and is bent along X by the Non-Crossing Lemma (7.2) resulting in the cut indicated by the dashed red line.

associated cuts in $\mathcal{K}$ as split cuts. Since the cuts in $\mathcal{K}$ are non-crossing, each separating exactly one cut pair in $\mathcal{F}$, splitting a node neither causes reconnections nor the separation of a pair that was not yet considered. Thus, CUT TREE reaches an intermediate tree representing the cuts in $\mathcal{K}$ with the cut pairs located in the incident nodes. □

This characterizes partial Gomory-Hu sets analogously to Gomory-Hu sets. In particular, there exists a CUT TREE execution for each Gomory-Hu set, and thus for each Gomory-Hu tree $T(G)$, that returns $T(G)$. Hence, the procedure CUT TREE (and also GOMORY-HU) is solution-complete. A Gomory-Hu tree as well as a valid intermediate tree can be further returned by several (partial) CUT TREE executions, since any permutation of $\mathcal{F}$ (and the same permutation applied to $\mathcal{H}$ and $\mathcal{K}$, respectively) yields the same tree.

7.2 Unique-Cut Trees

In the previous section we have seen that a CUT TREE execution $(G, \mathcal{F}, \mathcal{K})$ returns a unique Gomory-Hu tree $T(G)$ for a given graph G. However, the cuts in the corresponding Gomory-Hu set $\mathcal{H}$, that is, the cuts finally represented by the edges of the tree, are not necessarily equal to the split cuts given in $\mathcal{K}$. In contrast to the cuts in $\mathcal{H}$, the cuts in $\mathcal{K}$ possibly cross and are thus reshaped during the execution according to the Non-Crossing Lemma (7.2). That is, if we seek a Gomory-Hu tree that represents a special type of cuts, like, for example, U-cuts, in general, it does not suffice to just choose such cuts as split cuts, since due to the reshaping the split cuts may loose again their special properties. Figure 7.4 shows that this indeed happens for U-cuts. Here the cut $(X, V \setminus X)$ (solid green line), which corresponds to the U-cut $\mathfrak{uc}(y, x)$, does not separate u and v. Now suppose a CUT TREE execution that considers $\{u, v\}$ as next step pair with the split cut $(U, V \setminus U)$ (solid red line), which corresponds to the U-cut $\mathfrak{uc}(u, v)$. Since this cut crosses $(X, V \setminus X)$, it is reshaped according to the Non-Crossing Lemma (7.2), such that the resulting cut (dashed red line) is no U-cut anymore; neither with respect to u nor to v, since the opposite U-cut $\mathfrak{uc}(v, u)$ is given by the cut $(\{v\}, V \setminus \{v\})$.

In this section, we show that, nevertheless, there exists a Gomory-Hu tree with a Gomory-Hu set consisting of U-cuts, and present an algorithm to construct such a tree. Furthermore, we extend this tree by some additional information such that the resulting data structure represents all regular U-cuts. This data structure will play a fundamental role in Chapter 12 of Part III

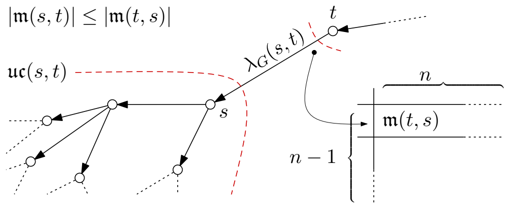


FIGURE 7.5: The data structure of unique-cut trees. Each edge (t,s) is associated with two opposite M-sets $\mathfrak{m}(s,t)$ (induced by the U-cut represented by the edge) and $\mathfrak{m}(t,s)$ (stored in a matrix as a vector indicating the vertices that are contained in the set).

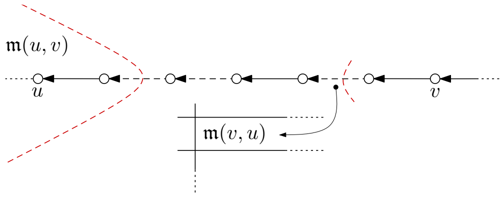
where we consider queries that ask for clusterings of special cohesive subsets. We call this data structure a *unique-cut tree.*

7.2.1 The Data Structure

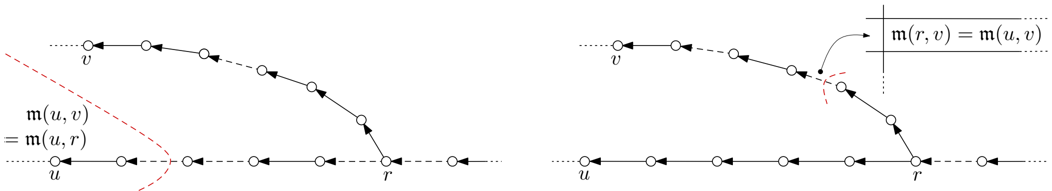
A unique-cut tree $\mathcal{T}(G) = (V, E_\mathcal{T}, c_\mathcal{T})$ of an undirected, weighted graph $G = (V, E, c)$ consists of two parts. The first part is a special Gomory-Hu tree and the second part is an $(n-1) \times n$ matrix that stores additional information associated with the edges in the tree. The special Gomory-Hu tree has the following form. It is a directed, weighted tree where each edge (t,s) represents a regular U-cut $\mathfrak{uc}(s,t)$, that is, a regular U-cut that minimizes the cut side that contains the head s of the edge. Furthermore, the opposite U-cut $\mathfrak{uc}(t,s)$ is stored in the additional matrix in form of a vector that indicates which vertices belong to the corresponding M-set $\mathfrak{m}(t,s)$, and is also associated with the edge (t,s), besides the connectivity $\lambda_G(s,t)$ (see Fig. 7.5). In this way, each edge of $\mathcal{T}(G)$ is associated with opposite U-cuts, which induce opposite M-sets, and vice versa. For the opposite M-sets associated with an edge (t,s), it further holds that the M-set $\mathfrak{m}(t,s)$, that is, the M-set that is stored in the matrix, contains at least as many vertices as the M-set $\mathfrak{m}(s,t)$, which is induced by the edge (t,s). In this way, a unique-cut tree decomposes the set of M-sets that are associated with its edges into two subsets; the M-sets in the matrix, and the remaining M-sets. For a simpler notation in the later proofs, we call the M-sets that are stored in the matrix the *matrix sets.* We further remark that, although opposite U-cuts may be identical, they always induce two different opposite M-sets. Different edges, however, may have the same associated matrix set. We will see that during the construction of a unique-cut tree most of these cases can be easily identified, such that avoiding the storing of duplicate matrix sets is often possible. Furthermore, there exists at least one matrix set, since each regular M-set has a different opposite M-set. From the existence of a unique-cut tree it thus follows that there exist at least $n-1$ different regular U-cuts (one per edge in $\mathcal{T}(G)$) and at least n different regular M-sets (one per edge in $\mathcal{T}(G)$ plus one matrix set) in an undirected, weighted graph G. We show in the next section how a unique-cut tree for a given graph G can be efficiently constructed by computing at most $2(n-1)$ maximum flows.

We further claim that each regular U-cut and each regular M-set in an undirected, weighted graph G is associated with a unique edge in $\mathcal{T}(G)$ and can be deduced from the tree structure as described in the following. We restrict the description to M-sets, since this already implies the description for U-cuts. Let $\{u,v\} \subseteq V$ denote an arbitrary vertex pair in G. We distinguish two cases regarding the position of u and v in the unique-cut tree $\mathcal{T}(G)$.

Case 1: u is a successor of v. In this case, the M-set $\mathfrak{m}(u,v)$ is represented by the cheapest edge on the (undirected) path $\pi(u,v)$ that is closest to u, that is, that is found first when traversing $\pi(u,v)$ from u to v. The opposite M-set $\mathfrak{m}(v,u)$ is the matrix set that is associated



(a) If u is a successor of v.



(b) If no edge on $\pi(r,v)$ is cheaper than the cheapest edge on $\pi(r,u)$.

(c) If all edges on $\pi(r,u)$ are more expensive than the cheapest edge on $\pi(r,v)$.

Figure 7.6: Illustration of rules to deduce the M-set of an arbitrary vertex pair $\{u,v\}$ from a unique-cut tree. Solid edges are more expensive than dashed edges. All dashed edges have all the same cost.

with the cheapest edge on the path $\pi(u,v)$ that is closest to v (see Fig. 7.6(a)). The case where v is a successor of u is symmetric.

Case 2: u is neither a successor nor a predecessor of v. In this case, also v is neither a successor nor a predecessor of u, and we denote the nearest common predecessor of u and v by r. If no edge on the path $\pi(r,v)$ is cheaper than the cheapest edge on $\pi(r,u)$, then the M-set $\mathfrak{m}(u,v)$ equals the M-set $\mathfrak{m}(u,r)$, which can be deduced as in Case 1 (see Fig. 7.6(b)). Otherwise, it equals the M-set $\mathfrak{m}(r,v)$, which can be also deduced as in Case 1 (see Fig. 7.6(c)). The opposite M-set $\mathfrak{m}(v,u)$ can be obtained symmetrically, changing the roles of u and v.

Proving this claim immediately yields that a unique-cut tree represents all regular U-cuts and all regular M-sets in an undirected, weighted graph G. Since each edge in $\mathcal{T}(G)$ is associated with at most two different U-cuts or M-sets, it follows that there are at most $2(n-1)$ U-cuts in the class $\mathcal{UC}(G)$ and at most $2(n-1)$ M-sets in the class $\mathcal{MS}(G)$. Taking further the construction of a unique-cut tree into account, we can state the following theorem.

Theorem 7.5. *For an undirected, weighted graph $G=(V,E,c)$ it is $(n-1) \leq |\mathcal{UC}(G)| \leq 2(n-1)$ and $n \leq |\mathcal{MS}(G)| \leq 2(n-1)$. Constructing the sets $\mathcal{UC}(G)$ and $\mathcal{MS}(G)$ needs at most $2(n-1)$ maximum-flow computations.*

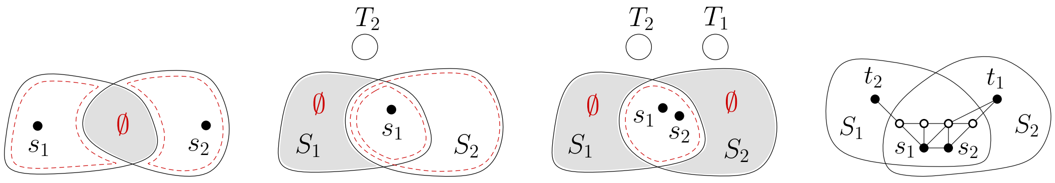
We will prove Theorem 7.5 by proving the claim above. This proof will be based on a central lemma (Lemma 7.6) that characterizes the nesting behavior of generalized M-sets. The different cases considered in this lemma are depicted in Fig. 7.7. Lemma 7.6 will also be the key to many further proofs regarding unique-cut trees and M-sets in this section and also in Part III.

Lemma 7.6. *Consider two generalized M-sets $S_1 := \mathfrak{m}(s_1,T_1)$ and $S_2 := \mathfrak{m}(s_2,T_2)$.*

(1) If $\{s_1,s_2\} \cap (S_1 \cap S_2) = \emptyset$, then $S_1 \cap S_2 = \emptyset$.

(2) If $T_2 \cap S_1 = \emptyset$ and $s_1 \in S_2$, then $S_1 \subseteq S_2$ (i). If further $T_1 \cap S_2 = \emptyset$ and $s_2 \in S_1$, then $S_1 = S_2$ (ii).

(3) Otherwise, S_1 and S_2 are neither nested nor disjoint.



(a) Case (1). (b) Case (2i). (c) Case (2ii). (d) Case (3).

FIGURE 7.7: Situation of Lemma 7.6. Filled areas are proven to not exist.

Proof of Lemma 7.6. We prove the cases (1), (2i), (2ii) and (3) separately. The proof is very technical. It exploits the properties of generalized U-cuts and generalized M-sets and compares costs of cuts by representing the costs as sums of cheaper costs with respect to subsets of the cut sides.

Proof of (1): If at least one of the two generalized M-sets is a singleton, (1) immediately holds. Otherwise, we observe the following. First recall that $S_1 \cap T_1 = S_2 \cap T_2 = \emptyset$. Now suppose $S_1 \cap S_2 \neq \emptyset$ (see Figure 7.7(a)). Since $U := (S_1 \cap S_2) \subseteq S_1$ with $s_1 \notin U$ it is $c(U, V \setminus S_1) < c(U, S_1 \setminus U)$, since otherwise, $(S_1 \setminus U, V \setminus (S_1 \setminus U))$ would be either a cheaper s_1-T-cut than $(S_1, V \setminus S_1)$ or a cut of the same cost but inducing a side of s_1 smaller than S_1. Since $(S_2 \setminus U) \subseteq (V \setminus S_1)$, it thus follows

$$c(U, S_2 \setminus U) \quad \leq \quad c(U, V \setminus S_1) < c(U, S_1 \setminus U). \tag{7.1}$$

We apply inequality (7.1) in order to show that the cut $(S_2 \setminus U, V \setminus (S_2 \setminus U))$, which also separates s_2 and T_2, is cheaper than the generalized U-cut $\mathfrak{uc}(s_2, T_2) = (S_2, V \setminus S_2)$. This leads to a contradiction.

We represent the costs of the two cuts as follows:

$$\begin{aligned} c(S_2, V \setminus S_2) = & \\ c(S_2 \setminus U, S_1 \setminus U) \quad & + \quad c(U, S_1 \setminus U) + c(S_2, V \setminus (S_1 \cup S_2)) && (7.2) \\ c(S_2 \setminus U, V \setminus (S_2 \setminus U)) = & \\ c(S_2 \setminus U, S_1 \setminus U) \quad & + \quad c(S_2 \setminus U, U) + c(S_2 \setminus U, V \setminus (S_1 \cup S_2)) && (7.3) \end{aligned}$$

Since $(S_2 \setminus U) \subseteq S_2$ it is $c(S_2 \setminus U, V \setminus (S_1 \cup S_2)) \leq c(S_2, V \setminus (S_1 \cup S_2))$ and with (7.1) we see that (7.3) < (7.2). □

Proof of (2), general case (i): If S_1 is a singleton, (2i) immediately holds. If S_2 is a singleton, it is $s_1 = s_2$ and since T_2 is supposed to be outside of S_1, it must hold $S_1 = \{s_1\} = \{s_2\} = S_2$. That is, (2i) holds. In any other case, we observe the following.

Suppose $S_1 \setminus S_2 \neq \emptyset$ (see Fig. 7.7(b)). Then it is $c(U, S_2) \leq c(U, V \setminus (S_1 \cup S_2))$, since otherwise $(S_1 \cup S_2, V \setminus (S_1 \cup S_2))$ would be a cheaper s_2-T_2-cut than $\mathfrak{uc}(s_2, T_2) = (S_2, V \setminus S_2)$. Since $(S_1 \setminus U) \cup (S_2 \setminus S_1) = S_2$, it is

$$c(U, S_1 \setminus U) + c(U, S_2 \setminus S_1) \quad \leq \quad c(U, V \setminus (S_1 \cap S_2)). \tag{7.4}$$

We apply inequality (7.4) in order to show that the cut $(S_1 \setminus U, V \setminus (S_1 \setminus U))$, which also separates s_1 and T_1, is at most as expensive as $\mathfrak{uc}(s_1, T_1) = (S_1, V \setminus S_1)$, which leads to a contradiction, since $|S_1 \setminus U| < |S_1|$.

We represent the costs of the two cuts as follows:

$$\begin{aligned}
&c(S_1, V \setminus S_1) = \\
&c(S_1 \setminus U, S_2 \setminus S_1) \quad + \quad c(U, S_2 \setminus S_1) + c(S_1 \setminus U, V \setminus (S_1 \cup S_2)) \\
&\qquad + \quad c(U, V \setminus (S_1 \cup S_2)) \qquad (7.5) \\
&c(S_1 \setminus U, V \setminus (S_1 \setminus U)) = \\
&c(S_1 \setminus U, S_2 \setminus S_1) \quad + \quad c(S_1 \setminus U, U) + c(S_1 \setminus U, V \setminus (S_1 \cup S_2)) \qquad (7.6)
\end{aligned}$$

If we add $c(U, S_2 \setminus S_1)$ to (7.6) and apply (7.4) we get a result that is at most as expensive as (7.5). Hence, (7.6) $\leq$ (7.5). But $S_1 \setminus U$ is smaller than S_1 contradicting the fact that S_1 is a generalized M-set. □

Proof of (2), special case (ii): If $S_1 = \{s_1\}$ is a singleton, we already know from (2i) that $s_1 \in S_2$. Now it is additionally required that T_1 is outside of S_2. Hence, it must be $S_1 = S_2$ and (2ii) holds. If S_2 is a singleton, we already know from (2i) that $S_1 = \{s_1\} = \{s_2\} = S_2$, and thus (2ii) holds. In any other case, we argue as follows.

Since the general case applies, it is $S_1 \subseteq S_2$ (see Fig. 7.7(c)). Furthermore, with S_1 also separating s_2 and T_2 and S_2 also separating s_1 and T_1 we get $\lambda(s_1, T_1) = \lambda(s_2, T_2)$, and thus, $(S_1, V \setminus S_1)$ is also a minimum s_2-T_2-cut with $|S_1| \leq |S_2|$. Hence, it must be $S_1 = S_2$, otherwiseS_2 would not be the source community of s_2 with respect to T_2. □

Proof of (3): In the remaining cases, at least one source is in $S_1 \cap S_2$, that is, $S_1 \cap S_2 \neq \emptyset$. Hence, S_1 and S_2 are not disjoint. Furthermore, it is either $T_1 \cap S_2 \neq \emptyset$ and $T_2 \cap S_1 \neq \emptyset$ or $T_1 \cap S_2 \neq \emptyset$ and $s_1 \in S_1 \setminus S_2$. Thus, S_1 and S_2 are not nested. Figure 7.7(d) shows an example proving that there exist S_1 and S_2 that are neither nested nor disjoint. □

Proof of Theorem 7.5. We prove that the regular M-sets of an arbitrary vertex pair $\{u, v\}$ can be deduced from the tree structure of a unique-cut tree $\mathcal{T}(G)$ as claimed above. The corresponding statement for regular U-cuts then holds implicitly. First observe that if u is a successor of v, it follows directly from the structure of $\mathcal{T}(G)$ that the M-set $\mathfrak{m}(u, v)$ is given by the cheapest edge on the path between u and v that is closest to u. We further show that (i) the opposite M-set $\mathfrak{m}(v, u)$ is the matrix set associated with the cheapest edge on the path from v to u that is closest to v.

If u and v are not in successor-predecessors relation with r the nearest common predecessor, we prove that the M-set $\mathfrak{m}(u, v)$ (ii) equals the M-set $\mathfrak{m}(u, r)$ if no edge on the path from r to v is cheaper than the cheapest edge on the path from r to u, and (iii) equals the M-set $\mathfrak{m}(r, v)$, otherwise.

Proof of (i): Let $(t, s) \in E_{\mathcal{T}}$ denote the cheapest edge on $\pi(v, u)$ that is closest to v. Obviously it is $\lambda_G(v, u) = c_{\mathcal{T}}(t, s)$. Since (t, s) is closest to v it is further $\lambda_G(v, t) > c_{\mathcal{T}}(t, s)$, and thus, neither the U-cut inducing $\mathfrak{m}(t, s)$ separates v and t nor the U-cut inducing $\mathfrak{m}(v, u)$. Hence, we have $\{v, t\} \subseteq \mathfrak{m}(v, u) \cap \mathfrak{m}(t, s)$, while $u \notin \mathfrak{m}(t, s)$. If $s \notin \mathfrak{m}(v, u)$, we thus get the situation of Lemma 7.6(2ii), which yields $\mathfrak{m}(v, u) = \mathfrak{m}(t, s)$. If $s \in \mathfrak{m}(v, u)$, we get $\mathfrak{m}(t, s) \subseteq \mathfrak{m}(v, u)$, according to Lemma 7.6(2i). However, since $\lambda_G(v, u) = \lambda_G(t, s)$ and $\mathfrak{m}(t, s)$ also separates u and v, it must hold $|\mathfrak{m}(v, u)| = |\mathfrak{m}(t, s)|$, which contradicts the assumption that $s \in \mathfrak{m}(v, u)$. □

Proof of (ii): If no edge on $\pi(r,v)$ is cheaper than the cheapest edge on $\pi(r,u)$, it is $\lambda_G(u,v) = \lambda_G(r,u)$, and any cheapest edge on $\pi(r,u)$ also induces a minimum u-v-cut. In particular the U-cut of $\mathfrak{m}(u,r)$ is a minimum u-v-cut, and thus it is $v \notin \mathfrak{m}(u,r)$, while clearly $u \in \mathfrak{m}(u,v) \cap \mathfrak{m}(u,r)$. If $r \notin \mathfrak{m}(u,v)$, we thus get the situation of Lemma 7.6(2ii), which yields $\mathfrak{m}(u,v) = \mathfrak{m}(u,r)$. If $r \in \mathfrak{m}(u,v)$, we get $\mathfrak{m}(u,r) \subseteq \mathfrak{m}(u,v)$, according to Lemma 7.6(2i). However, since $\lambda_G(u,v) = \lambda_G(u,r)$ and $\mathfrak{m}(u,r)$ also separates u and v, it must hold $|\mathfrak{m}(u,v)| = |\mathfrak{m}(u,r)|$, which contradicts the assumption $r \in \mathfrak{m}(u,v)$. □

Proof of (iii): If all edges on $\pi(r,u)$ are more expensive than the cheapest edge on $\pi(r,v)$, it is, $\lambda_G(u,v) = \lambda_G(r,v)$, and neither the U-cut inducing $\mathfrak{m}(u,v)$ separates u and r nor the U-cut inducing $\mathfrak{m}(r,v)$. That is, $u \in \mathfrak{m}(u,v) \cap \mathfrak{m}(r,v)$. Since v is neither in $\mathfrak{m}(u,v)$ nor in $\mathfrak{m}(r,v)$, we get the situation of Lemma 7.6(2ii), and it follows $\mathfrak{m}(u,v) = \mathfrak{m}(r,v)$. □

7.2.2 Constructing a Unique-Cut Tree

For the construction of the special Gomory-Hu tree that forms the basis of a unique-cut tree, we simply choose the regular U-cuts as split cuts in the procedure Cut Tree that are supposed to be represented by the directed edges in the final tree. That is, for the step pair $\{s,t\}$, we choose the U-cut with the smallest M-set. Furthermore, we direct the resulting tree edge to the chosen M-set and store the opposite M-set in the matrix. At first, this approach might sound contradictory to the explanations at the beginning and the example with the crossing U-cuts in Fig. 7.4. However, we can show that simply comparing the sizes of the M-sets already admits to choose non-crossing U-cuts, which prevents a reshaping of the original split cuts and guarantees that the cuts represented in the final tree are the same cuts as chosen for the construction. In Fig. 7.4, we observe that choosing the opposite U-cut $\mathfrak{uc}(v,u) = (\{v\}, V \setminus \{v\})$ instead of $\mathfrak{uc}(u,v) = (U, V \setminus U)$, whose corresponding M-set $\mathfrak{m}(u,v) = U$ is clearly larger than $\mathfrak{m}(v,u) = \{v\}$, already resolves the crossing.

However, reconnecting edges in the intermediate tree during the construction might be still necessary in order to ensure that each final edge indeed represents the correct cut. It is therefore necessary to implement a statement similar to Gomory's and Hu's Lemma 7.3, which ensures that after a reconnection the new compound nodes incident to the reconnected edge still contain a cut pair of the edge, also for U-cuts and M-sets. The key lemmas to our construction of unique-cut trees, which we introduce in the next section, thus consist of two parts, each. The first part ensures that the chosen U-cuts do not cross, the second part guarantees that after the reconnection of a (directed) edge in the intermediate unique-cut tree the induced M-set is also an M-set with respect to the new tail and head of the reconnected edge. For the matrix sets associated with reconnected edges, the lemmas show that at least in most cases the previous sets remain valid. Thus, explicitly recomputing a matrix set is rarely necessary and avoiding the storing of duplicate matrix sets is often possible. So the construction of a unique-cut tree needs at least $(n-1)$ maximum-flow computations, but is often possible with much less than $2(n-1)$ maximum-flow computations. In this way, the lemmas prove the correctness of Algorithm 3, which presents the modification of Cut Tree as described above, and thus, the existence of unique-cut trees.

Theorem 7.7. *For an undirected, weighted graph $G = (V,E,c)$ there exists a rooted Gomory-Hu tree $T(G) = (V, E_T, c_T)$ with edges directed to the leaves such that each edge $(t,s) \in E_T$ represents the U-cut $\mathfrak{uc}(s,t)$ with $|\mathfrak{m}(s,t)| \leq |\mathfrak{m}(t,s)|$. Such a tree can be constructed by $n-1$*

maximum-flow computations. Additionally associating the opposite M-sets with the edges costs at most $n-1$ additional maximum-flow computations.

The Unique-Cut Tree Algorithm. In the modification of the procedure Cut Tree presented in Algorithm 3, we use stars to represent nodes in the intermediate tree, choose star edges as step pairs, and use U-cuts that induce the smallest M-set of the current step pair as split cuts. Furthermore, we compute complete maximum flows in line 4, while for the construction of general Gomory-Hu trees, the computation of minimum cuts suffices, which can be already deduced from preflows. Here, complete maximum flows are necessary in order to compare the sizes of the M-sets and to choose the U-cut with the smallest M-set. Recall that a maximum s-t-flow induces a DAG that represents all minimum s-t-cuts. Hence, the two opposite M-sets with respect to s and t can be deduced from this DAG. Note further that, for the sake of simplicity, Algorithm 3 does not deal with the matrix sets of the unique-cut tree. However, these M-sets can be easily handled according to the cases given by the key lemmas (see Section 7.2.3).

Algorithm 3: Unique-Cut Tree

```
Input: Graph $G = (V, E, c)$
Output: Gomory-Hu tree of $G$
1  Initialize tree $T_* = (V, E_t, E_f, c_f)$ with $E_t \leftarrow$ thin edges forming a star of $V$, $E_f \leftarrow \emptyset$ and $c_f$
   empty
2  while $\exists$ thin edge $\{u, v\}$ in $E_t$ do                                   // unfold all nodes
3      $S \leftarrow$ star of thin edges with center $v$ that contains $\{u, v\}$           // choose node
4      compute max-$u$-$v$-flow with value $\lambda_G(u, v)$
5      $X \leftarrow \operatorname{argmin}\{|\mathfrak{m}(u, v)|, |\mathfrak{m}(v, u)|\}$                // choose split cut
6      $x \leftarrow X \cap \{u, v\}$, $y \leftarrow (V \setminus X) \cap \{u, v\}$
7      $E_t \leftarrow E_t \setminus S$
8      forall the $s \in S \setminus \{x, y\}$ do                    // split $S = S_u \uplus S_v = S_x \uplus S_y$
9          if $s \in X$ then $E_t \leftarrow E_t \cup \{\{s, x\}\}$;                    // reconnect $s$ to $x$
10         if $s \in V \setminus X$ then $E_t \leftarrow E_t \cup \{\{s, y\}\}$;                // reconnect $s$ to $y$
11     center of $S_x \leftarrow x$, center of $S_y \leftarrow y$
12     $N \leftarrow$ all vertices $s$ that are linked by a fat edge to a vertex $v_s \in S$
13     forall the $s \in N$ do                     // reconnect subtrees to $S_x$ and $S_y$
14         if $s \in X$ then                                           // reconnect $s$ to $x$
15             if $s$ is a head then $E_t \leftarrow (E_t \setminus \{(v_s, s)\}) \cup \{(x, s)\}$
16             if $s$ is a tail then $E_t \leftarrow (E_t \setminus \{(s, v_s)\}) \cup \{(s, x)\}$
17         if $s \in (V \setminus X)$ then                                 // reconnect $s$ to $y$
18             if $s$ is a head then $E_t \leftarrow (E_t \setminus \{(v_s, s)\}) \cup \{(y, s)\}$
19             if $s$ is a tail then $E_t \leftarrow (E_t \setminus \{(s, v_s)\}) \cup \{(s, y)\}$
20     $E_f \leftarrow E_f \cup \{(y, x)\}$, $E_t \leftarrow E_t \setminus \{\{x, y\}\}$, $c_f(y, x) \leftarrow \lambda_G(u, v)$            // draw $(y, x)$ fat
21 return $T_*$
```

7.2.3 The Key Lemmas to the Construction of Unique-Cut Trees

Lemma 7.8 states that if we chose a step pair $\{s, x\}$ in an M-set that is the smallest M-set with respect to a cut pair $\{s, t\}$, the U-cut that induces the smallest M-set with respect to $\{s, x\}$ does not cross the previous U-cut of $\{s, t\}$ (see also Fig. 7.8). Furthermore, the considered U-cuts can be represented by directed edges and the opposite M-sets can be associated with the edges.

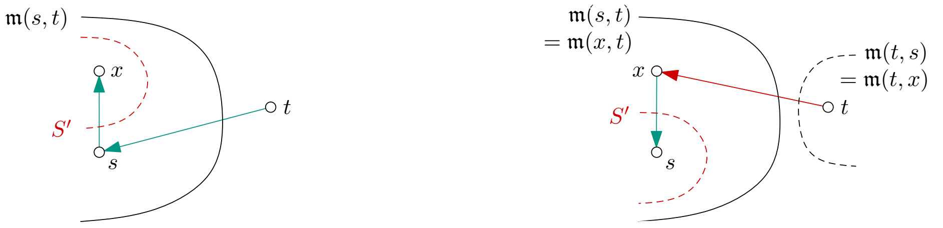
Lemma 7.8. *For a cut pair* $\{s,t\}$*, let* $|\mathfrak{m}(s,t)| \leq |\mathfrak{m}(t,s)|$*, and let* S' *denote a smallest M-set with respect to a cut pair* $\{s,x\} \subseteq \mathfrak{m}(s,t)$*. Then* $S' \subsetneqq \mathfrak{m}(s,t)$ *(Fig. 7.8). If* $S' = \mathfrak{m}(s,x)$ *(Fig. 7.8(b)), then* $\mathfrak{m}(s,t) = \mathfrak{m}(x,t)$ *is the smallest M-set with respect to* $\{x,t\}$ *and* $\mathfrak{m}(t,s) = \mathfrak{m}(t,x)$*.*

Besides choosing step pairs inside of previous M-sets as in Lemma 7.8, the construction of a Gomory-Hu tree also requires the possibility of choosing step pairs outside of previous M-sets. This possibility is given by the next lemma.

Lemma 7.9. *For a cut pair* $\{u,s\}$*, let* $|\mathfrak{m}(u,s)| \leq |\mathfrak{m}(s,u)|$*, and let* S' *denote a smallest M-set with respect to a cut pair* $\{s,x\} \subseteq V \setminus \mathfrak{m}(u,s)$*. Then* $\mathfrak{m}(u,s) \subsetneqq S'$ *(Fig. 7.9) or* $\mathfrak{m}(u,s) \cap S' = \emptyset$ *(Fig. 7.10).*

If $\mathfrak{m}(u,s) \subsetneqq S'$ *and* $S' = \mathfrak{m}(x,s)$ *(Fig. 7.9(b,c)), then* $\mathfrak{m}(u,s) = \mathfrak{m}(u,x)$ *is the smallest M-set with respect to* $\{u,x\}$*. If further* $x \in \mathfrak{m}(s,u)$ *(Fig. 7.9(b)), then* $\mathfrak{m}(s,u) = \mathfrak{m}(x,u)$*. Otherwise, if* $x \notin \mathfrak{m}(s,u)$ *(Fig. 7.9(c)), then* $\mathfrak{m}(s,u) = \mathfrak{m}(s,x)$*, but the opposite M-set* $\mathfrak{m}(x,u)$ *of the reconnected edge must be computed additionally.*

If $\mathfrak{m}(u,s) \cap S' = \emptyset$ *and* $S' = \mathfrak{m}(s,x)$ *(Fig. 7.10(b,c,d)), then* $\mathfrak{m}(u,s) = \mathfrak{m}(u,x)$ *is again the smallest M-set with respect to* $\{u,x\}$*. If further* $x \in \mathfrak{m}(s,u)$ *(Fig. 7.10(b)), then* $\mathfrak{m}(s,u) = \mathfrak{m}(x,u)$*. Otherwise, if* $x \notin \mathfrak{m}(s,u)$ *(Fig. 7.10(c,d)), then* $\mathfrak{m}(s,u) = S'$ *and if further* $u \notin \mathfrak{m}(x,s)$ *(Fig. 7.10(c)), then* $\mathfrak{m}(x,s) = \mathfrak{m}(x,u)$*. Otherwise (Fig. 7.10(d)), the opposite M-set* $\mathfrak{m}(x,u)$ *of the reconnected edge must be computed additionally.*



(a) $S' = \mathfrak{m}(x,s)$, reconnecting edges is not necessary. (b) $S' = \mathfrak{m}(s,x)$, t needs to be reconnected to x.

FIGURE 7.8: Situation in Lemma 7.8. In order to represent U-cuts by directed tree edges, depending on the shape of S' (dashed red line) a reconnection becomes necessary; see (*b*) where t is reconnected to x. This reconnection is feasible, since the second part of Lemma 7.8 guarantees that the new vertices incident to the reconnected edge are still a cut pair of the represented cut and the opposite M-set possibly assigned to the edge (dashed black line) is also an opposite M-set with respect to this new cut pair.

Proof of Lemma 7.8. We distinguish two cases, $S' = \mathfrak{m}(x,s)$, that is, $x \in S'$ (Fig. 7.8(a)) and $S' = \mathfrak{m}(s,x)$, that is, $s \in S'$ (Fig. 7.8(b)). The cases are (almost) symmetric with respect to the first assertion that $S' \subset \mathfrak{m}(s,t)$. Hence, we prove $S' \subset \mathfrak{m}(s,t)$ just for the first case. The second part of Lemma 7.8 refers to the second case, where the edge (t,s) needs to be reconnected, and we thus need to show that the M-set to which the reconnected edge points and the opposite M-set assigned to the tail remain valid.

First assertion $S' \subset \mathfrak{m}(s,t)$:

Consider the case where $S' = \mathfrak{m}(x,s)$, that is, $x \in S'$ (Fig. 7.8(a)). If $t \notin S'$, it is $S' \subset \mathfrak{m}(s,t)$ according to Lemma 7.6(2i). Hence, suppose $t \in S'$. We show that this case yields a contradiction, and thus, does not occur.

Consider the opposite M-set $\mathfrak{m}(s,x)$ of S'. Note that $t \notin \mathfrak{m}(s,x)$, since $\mathfrak{m}(s,x) \cap S' = \emptyset$. Thus, it must hold $\mathfrak{m}(s,x) \subset \mathfrak{m}(s,t)$, since otherwise t would deflect the corresponding U-cut $\mathfrak{uc}(s,x)$

according to the Non-Crossing Lemma (7.2) resulting in a smaller cut side containing s, and hence, $\mathfrak{m}(s,x)$ would not be an M-set. In the following we argue that if $t \in S'$ then $|\mathfrak{m}(s,x)| < |S'|$ and hence, S' is not the smallest M-set with respect to s and x, which is the final contradiction. Hence, the case $t' \in S'$ does not occur.

We first observe that if $t' \in S'$, then $\mathfrak{uc}(s,t)$ is also a minimum x-t-cut, since $\mathfrak{uc}(x,s)$ (which is the cut associated with S') could be bent along $\mathfrak{m}(s,t)$ deflected by t (according to the Non-Crossing Lemma (7.2)), such that it separates t and s. Hence, according to the correctness of the Gomory-Hu algorithm (Lemma 7.3), $\{x,t\}$ is also a cut pair of $\mathfrak{uc}(s,t)$. Thus, deflected by s, $\mathfrak{uc}(s,t)$ can be bent along (the original) S' (which contains x and t) yielding a cut side $T \ni t$ of a minimum s-t-cut. Since $\mathfrak{m}(s,t)$ is the smallest M-set with respect to s and t, it follows $|\mathfrak{m}(s,t)| \leq |T|$, while $T \subset S'$ and $\mathfrak{m}(s,x) \subset \mathfrak{m}(s,t)$. Finally it is $|\mathfrak{m}(s,x)| < |\mathfrak{m}(s,t)| \leq |T| < |S'|$.

Second part regarding $S' = \mathfrak{m}(s,x)$:
Now we know that the first assertion $S' \subset \mathfrak{m}(s,t)$ holds for both cases $S' = \mathfrak{m}(x,s)$ and $S' = \mathfrak{m}(s,x)$. We show next that if $S' = \mathfrak{m}(s,x)$, that is, $s \in S'$ and the edge (t,s) is reconnected becoming the edge (t,x) (Fig. 7.8(b)), $\mathfrak{m}(s,t)$ is also the smallest M-set with respect to x and t and $\mathfrak{m}(t,s) = \mathfrak{m}(t,x)$.

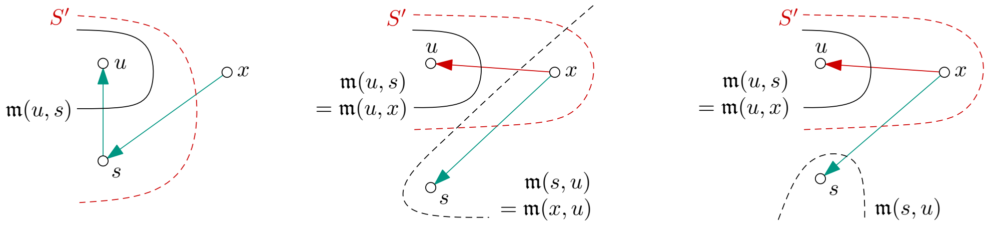
Analogously to the observation above, we observe again that, according to the correctness of the Gomory-Hu algorithm (Lemma 7.3), $\mathfrak{uc}(s,t)$ is also a minimum x-t-cut, since S' again separates s and t. Consequently, it is $\lambda(x,t) = \lambda(s,t)$ and $\mathfrak{m}(s,t) = \mathfrak{m}(x,t)$, since otherwise, if there was a smaller M-set $\mathfrak{m}(x,t)$, this would not contain s, and thus, separate x and s. This means that $\lambda(s,x) \leq \lambda(s,t)$. Then, however, S' contradicts the assumption that $\mathfrak{m}(s,t)$ is an M-set. Hence, it must be $\mathfrak{m}(s,t) = \mathfrak{m}(x,t)$.

Furthermore, for the opposite M-sets it is $\mathfrak{m}(t,s) = \mathfrak{m}(t,x)$, as $x \in \mathfrak{m}(s,t)$, and together with $\mathfrak{m}(s,t) = \mathfrak{m}(x,t)$, $\mathfrak{m}(s,t)$ is the smallest M-set with respect to x and t, since $|\mathfrak{m}(t,s)| \geq |\mathfrak{m}(s,t)|$.

Proof of Lemma 7.9. We distinguish two cases, $S' = \mathfrak{m}(s,t)$, that is, $s \in S'$ (Fig. 7.9) and $S' = \mathfrak{m}(x,s)$, that is, $x \in S'$ (Fig. 7.10). The first assertion, which is $U \cap S' = \emptyset$ or $U \subset S'$, follows for both cases directly from Lemma 7.6(1),(2i), depending on whether or not $u \in S'$. The second part of the lemma again focuses on the situations where a reconnection of an edge becomes necessary (see all but the first cases in Fig. 7.9 and Fig. 7.10). In the following we proof the assertions of this second part.

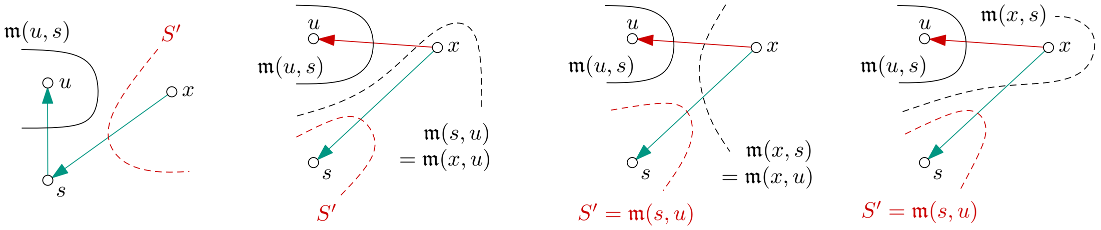
Second part if $\mathfrak{m}(u,s) \subsetneqq S'$ and $S' = \mathfrak{m}(x,s)$ (Fig. 7.9(b,c)):
We first observe that due to the correctness of the Gomory-Hu algorithm (Lemma 7.3), $\mathfrak{uc}(u,s)$ is also a minimum u-x-cut, and hence, it is $\lambda(u,x) = \lambda(u,s)$ and $\mathfrak{m}(u,s) = \mathfrak{m}(u,x)$, since a smaller M-set $\mathfrak{m}(u,x)$ would also induce a smaller M-set $\mathfrak{m}(u,s)$, which is a contradiction. That $\mathfrak{m}(u,x)$ is further the smallest M-set with respect to u and x, we will see later.

Before we continue to prove further assertions regarding opposite M-sets, at this point, we prove by contradiction that $s \in \mathfrak{m}(x,u)$ (which is not depicted in Fig. 7.9(b,c)). This will admit to apply Lemma 7.6, and thus, help to prove the remaining assertions of the second part in the case that $\mathfrak{m}(u,s) \subsetneqq S'$ and $S' = \mathfrak{m}(x,s)$. Suppose $s \notin \mathfrak{m}(x,u)$. Then $\mathfrak{uc}(x,u)$ would also separate x from s, and thus, it would be $\lambda(x,s) \leq \lambda(x,u)$. However, since $\lambda(x,u) = \lambda(s,u)$ and S' also separates s and u, it must be also $\lambda(x,u) \leq \lambda(x,s)$. Consequently, it would be $\lambda(x,u) = \lambda(x,s)$ if $s \notin \mathfrak{m}(x,u)$. Furthermore, according to the Non-Crossing Lemma (7.2), $\mathfrak{uc}(x,u)$ could be bent along S' (deflected by s) such that $\mathfrak{m}(x,u) \subsetneqq S'$. Recall that we assumed $s \notin \mathfrak{m}(x,u)$, and hence, $\mathfrak{m}(x,u) \subsetneqq S'$ contradicts the fact that S' is the M-set $\mathfrak{m}(x,s)$. Hence, we know that $s \in \mathfrak{m}(x,u)$.



(a) (b) (c)

FIGURE 7.9: Situation in Lemma 7.9 if $\mathfrak{m}(u,s) \subsetneq S'$. In order to represent U-cuts by directed tree edges, depending on the shape of S' (dashed red line) a reconnection becomes necessary. (a) $S' = \mathfrak{m}(s,x)$, reconnecting edges is not necessary, (b,c) $S' = \mathfrak{m}(x,s)$, x needs to be reconnected to u. This reconnection is feasible, since Lemma 7.9 guarantees that the new vertices incident to the reconnected edge are still a cut pair of the represented cut. Furthermore, in (b), the opposite M-set possibly assigned to the reconnected edge (dashed black line) is also an opposite M-set with respect to this new cut pair, since x is contained in this opposite M-set. In (c), the opposite M-set for the reconnected edge must be computed additionally.



(a) (b) (c) (d)

FIGURE 7.10: Situation in Lemma 7.9 if $\mathfrak{m}(u,s) \cap S' = \emptyset$. In order to represent U-cuts by directed tree edges, depending on the shape of S' (dashed red line) a reconnection becomes necessary. (a) $S' = \mathfrak{m}(x,s)$, reconnecting edges is not necessary, (b,c,d) $S' = \mathfrak{m}(s,x)$, x needs to be reconnected to u. This reconnection is feasible, since Lemma 7.9 guarantees that the new vertices incident to the reconnected edge are still a cut pair of the represented cut. Furthermore, in (b), the opposite M-set possibly assigned to the reconnected edge (dashed black line) is also an opposite M-set with respect to this new cut pair, since x is contained in this opposite M-set. In (b), the opposite M-set of the edge that is not reconnected (dashed black line) is also an opposite M-set with respect to the new cut pair of the reconnected edge, since u is contained in this opposite M-set. In (c), the opposite M-set $\mathfrak{m}(x,u)$ for the reconnected edge must be computed additionally.

Now we continue to prove the remaining assertions regarding the opposite M-sets. We still consider the case where $\mathfrak{m}(u,s) \subsetneq S'$ and $S' = \mathfrak{m}(x,s)$ (Fig. 7.9(b,c)). If $x \in \mathfrak{m}(s,u)$ (Fig. 7.9(b)), together with $s \in \mathfrak{m}(x,u)$, it follows directly from Lemma 7.6(2ii) that the opposite M-set of the reconnected edge is $\mathfrak{m}(x,u) = \mathfrak{m}(s,u)$.

Based on this insight, we can now also prove for $x \in \mathfrak{m}(s,u)$ (Fig. 7.9(b)) the missing assertion from above, namely that $\mathfrak{m}(u,x)$ is the *smallest* M-set with respect to u and x. We have already seen that $\mathfrak{m}(u,s) = \mathfrak{m}(u,x)$, and we know that $|\mathfrak{m}(u,s)| \leq |\mathfrak{m}(s,u)|$. Hence, it is $|\mathfrak{m}(u,x)| = |\mathfrak{m}(u,s)| \leq |\mathfrak{m}(s,u)| = |\mathfrak{m}(x,u)|$.

If $x \notin \mathfrak{m}(s,u)$ (Fig. 7.9(c)), it is $\lambda(s,x) \leq \lambda(s,u)$. Since S' also separates s and u, it is further $\lambda(s,x) \geq \lambda(s,u)$, and thus, $\lambda(s,x) = \lambda(s,u)$ and $\mathfrak{m}(s,u) = \mathfrak{m}(s,x)$. However, the opposite M-set $\mathfrak{m}(x,u)$ for the reconnected edge must be computed separately.

Finally, we again prove the missing assertion from above, namely that $\mathfrak{m}(u,x)$ is the *smallest* M-set with respect to u and x, now for the case that $x \notin \mathfrak{m}(s,u)$ (Fig. 7.9(c)). We exploit that we showed above that $s \in \mathfrak{m}(x,u)$. Together with $x \notin \mathfrak{m}(s,u)$ it follows directly from

Lemma 7.6(2i) that $\mathfrak{m}(s,u) \subsetneq \mathfrak{m}(x,u)$. We have further seen that $\mathfrak{m}(u,s) = \mathfrak{m}(u,x)$ and we know that $|\mathfrak{m}(u,s)| \leq |\mathfrak{m}(s,u)|$. Hence, it is $|\mathfrak{m}(u,x)| = |\mathfrak{m}(u,s)| \leq |\mathfrak{m}(s,u)| < |\mathfrak{m}(x,u)|$.

Second part if $\mathfrak{m}(u,s) \cap S' = \emptyset$ and $S' = \mathfrak{m}(s,x)$ (Fig. 7.10(b,c,d)):
Analogously to the second part where $\mathfrak{m}(u,s) \subsetneq S'$ and $S' = \mathfrak{m}(x,s)$ (Fig. 7.9(b,c)), we observe again that due to the correctness of the Gomory-Hu algorithm (Lemma 7.3), $\mathfrak{uc}(u,s)$ is also a minimum u-x-cut, and hence, it is $\lambda(u,x) = \lambda(u,s)$ and $\mathfrak{m}(u,s) = \mathfrak{m}(u,x)$, since a smaller M-set $\mathfrak{m}(u,x)$ would also induce a smaller M-set $\mathfrak{m}(u,s)$, which is a contradiction. That $\mathfrak{m}(u,x)$ is further the smallest M-set with respect to u and x, we will see later.

For the case that $x \in \mathfrak{m}(s,u)$ (Fig. 7.10(b)), we further argue again (by contradiction) that $s \in \mathfrak{m}(x,u)$, in order to apply later Lemma 7.6. Suppose $s \notin \mathfrak{m}(x,u)$. Then it would be $\lambda(s,x) \leq \lambda(u,x)$. Since S' also separates u and s it is further $\lambda(u,s) \leq \lambda(s,x)$. Since we already know that $\lambda(u,s) = \lambda(u,x)$, this would yield $\lambda(s,x) = \lambda(u,x)$. Furthermore, it holds $S' \subset \mathfrak{m}(s,u)$, according to Lemma 7.7(2i). Together, S' would thus contradict the fact that $\mathfrak{m}(s,u)$ is an M-set. Hence, we know that $s \in \mathfrak{m}(x,u)$ if $x \in \mathfrak{m}(s,u)$ (Fig. 7.10(b)).

Now we see that for the opposite M-set of the reconnected edge it follows directly from Lemma 7.7(2ii) that $\mathfrak{m}(s,u) = \mathfrak{m}(x,u)$. The still missing assertion from above saying that $\mathfrak{m}(u,x)$ is the *smallest* M-set with respect to u and x now also follows, since we already know that $\mathfrak{m}(u,s) = \mathfrak{m}(u,x)$ and $|\mathfrak{m}(u,s)| \leq |\mathfrak{m}(s,u)|$.

For the case that $x \notin \mathfrak{m}(s,u)$ (Fig. 7.10(c,d)), we note that $S' = \mathfrak{m}(s,u)$ follows directly from Lemma 7.7(2ii). If further $u \notin \mathfrak{m}(x,s)$, again by Lemma 7.6(2ii), it follows for the opposite M-set of the reconnected edge that $\mathfrak{m}(x,u) = \mathfrak{m}(x,s)$. The still missing assertion from above saying that $\mathfrak{m}(u,x)$ is the *smallest* M-set with respect to u and x follows again, since we already know that $\mathfrak{m}(u,s) = \mathfrak{m}(u,x)$ and $S' = \mathfrak{m}(s,u)$ (and $S' = \mathfrak{m}(s,x)$). Hence, we get $|\mathfrak{m}(u,x)| = |\mathfrak{m}(u,s)| \leq |\mathfrak{m}(s,u)| = |\mathfrak{m}(s,x)| \leq |\mathfrak{m}(x,s)| = |\mathfrak{m}(x,u)|$.

CHAPTER 8

Gomory-Hu Trees in Dynamic Graphs

Gomory-Hu trees are a powerful data structure that admits to answer a variety of queries related to edge connectivity, minimum s-t-cuts, and interesting subgraphs of undirected, weighted graphs. Hence, seeking for methods to efficiently maintain a Gomory-Hu tree also in a dynamic scenario, where the underlying graph changes over time, is a natural further step, all the more in the light of existing approaches in the context of *sensitivity analysis of multiterminal flow networks*. Shortly after the introduction of Gomory-Hu trees [59] in 1961, in 1964, Elmaghraby [36] already studied the sensitivity of minimum separating cuts to changes in the underlying graph. He considers an undirected, weighted graph and a designated edge whose cost continuously changes. For this scenario he proposes an iterative approach that detects all breakpoints in the range of the varying edge cost where the corresponding Gomory-Hu tree changes. Whenever he finds such a breakpoint, he accordingly adapts the edge cost in the underlying graph and computes a new Gomory-Hu tree from scratch. Hence, a more efficient update approach for Gomory-Hu trees would already speed up this method. Barth et al. [11] already improved this method and showed that Elmaghraby's breakpoints can be found by constructing only two Gomory-Hu trees in total. They further extend the sensitivity analysis to several varying edges and point out some first relations to dynamic Gomory-Hu trees in undirected, weighted graphs. Nevertheless, they do not solve the latter problem. Instead they in particular declare the update after the decrease of an edge cost as a difficult open problem. For Gomory-Hu trees in undirected, unweighted graphs (the authors call them 0-1 undirected networks) Lin and Ma [100] already solved this problem with worst-case running time of $n-1$ cut computations. Their approach exploits the fact that the cost of an edge changes by exactly 1. This allows some nice conclusions on reusable cuts after a change, which do not apply to general weighted graphs. Cuts that cannot be proven to remain valid are computed from scratch. A sensitivity analysis for only one cut is further realized by *parametric maximum flows*. This problem considers a flow network with only two terminals s and t but with several parametric edge capacities. The goal is again to determine the breakpoints in the parameter ranges of the edge capacities, and to give corresponding maximum s-t-flows (or minimum s-t-cuts). Parametric maximum flows are studied, for example, by Gallo et al. [54] and Scutellà [131].

In this work we do not seek for breakpoints in parameter ranges, but for efficient updates of Gomory-Hu trees in undirected, weighted graphs after the change of an edge or a vertex. More precisely, we consider a dynamic scenario in which the underlying graph evolves due to atomic changes, and we aim at efficiently and smoothly maintaining an initial Gomory-Hu tree over the time. An atomic change is either the increase or decrease of an edge cost, the insertion or deletion of an edge, or the insertion or deletion of a vertex, where the latter only occur for isolated vertices. Hence, inserting or deleting a vertex changes neither the cost function nor the connectivity of the underlying graph. We can immediately describe update procedures for these cases as follows. Recall that, according to our definition, a Gomory-Hu tree $T(G)$ may also contain edges of cost 0 in case the underlying graph G is disconnected. A Gomory-Hu tree is thus always connected. Furthermore, observe that after reconnecting an edge of cost 0 in $T(G)$ each edge represents a cut that is still equivalent to the cut it previously represented. Consequently, the resulting tree is again a Gomory-Hu tree of G. Moreover, deleting or inserting a leaf in $T(G)$ that is incident to an edge of cost 0 results in a Gomory-Hu tree of $G^{\textcircled{\scriptsize U}}$. According to these observations, an isolated vertex can be inserted by simply connecting it to an arbitrary vertex in $T(G)$ via an edge of cost 0. An isolated vertex in G that is supposed to be deleted forms a connected component in G, and is thus only incident to edges of cost 0 in $T(G)$. Reconnecting these edges such that the vertex becomes a leaf and then deleting the leaf yields again a Gomory-Hu tree of $G^{\textcircled{\scriptsize U}}$. These update procedures trivially abandon cut computations. In this chapter we thus focus on the non-trivial cases of edge insertion and deletion, and cost increase and decrease.

The Main Idea and the Hidden Problems Within. The main idea of our approach is to determine cuts in a given Gomory-Hu tree $T(G)$ that are still minimum separating cuts in the graph $G^{\textcircled{\scriptsize U}}$ after a change, and thus, can be reused for the construction of a new Gomory-Hu tree $T(G^{\textcircled{\scriptsize U}})$; the missing cuts are constructed from scratch. This idea however requires that the reusable cuts form a partial Gomory-Hu set. Otherwise, not all of the independently reusable cuts can be reused in the same construction. A naive method to check whether an old cut is reusable is to choose a cut pair of the old cut and to compute a new minimum separating cut for this cut pair in $G^{\textcircled{\scriptsize U}}$. Comparing the costs of both cuts then admits to identify old cuts that are reusable with respect to the checked cut pair. However, if the old cut is not reusable with respect to the checked but with respect to another cut pair, this cannot be detected. In this case, the hope is that we can at least use the newly computed cut as a split cut in the construction of the new tree. This, however, is only possible if the checked cut pair is not yet separated by a previously considered cut. Otherwise, we would have performed cut computation that unnecessarily increases the running time. Conversely, if the newly found cut separates cut pairs of old cuts that have been not yet checked, these old cuts cannot be reused together with the new cut in a Gomory-Hu tree construction; at least not with respect to the separated cut pairs. This, however, is an obstacle to temporal smoothness.

In this work, we state several lemmas that admit to detect reusable cuts in at most linear time without any cut computation. Furthermore, we show that for the remaining cuts it is yet possible to choose the cut pairs for checking in a clever way, such that the newly computed cuts can be always used for the construction of a new tree in case the old cut is not reusable, and at the same time, no possibly reusable cut is missed. In this way, our update approach provides a high potential for saving cut computations, which is also confirmed by our experiments in Section 8.3, and additionally guarantees optimal temporal smoothness (see Chapter 9). The key to these results is provided by two lemmas (one for edge deletions and decreasing edge costs (Lemma 8.9) and one for edge insertions and increasing edge costs (Lemma 9.4)) that, similar to the Non-Crossing Lemma (7.2), admit to bend newly found cuts in $G^{\textcircled{\scriptsize U}}$ along old cuts in G, such that the edges in $T(G)$ representing cut pairs of old cuts are not crossed.

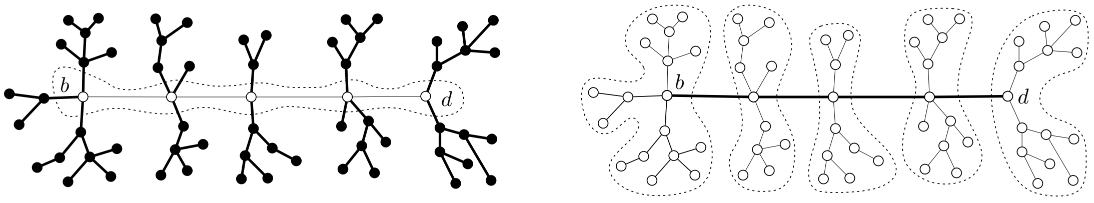
Obviously Reusable Cuts. Coming from the direction of sensitivity analysis, Barth et al. [11] already stated that after an cost increase the path $\pi(b,d)$ in $T(G) = (V, E_T, c_T)$ is the only part of a given Gomory-Hu tree that needs to be recomputed. Recall that the vertices b and d define the changing edge. This result is rather obvious according to the fundamental insight given by the following Lemma 8.1 and the observation that the cuts represented by the edges that are not on $\pi(b,d)$ form a partial Gomory-Hu set $\mathcal{K}$ in $G^{\oplus}$ (as they are clearly non-crossing and, according to Lemma 8.1, the vertices incident to the edges represent cut pairs of the corresponding cuts). Hence, $\mathcal{K}$ induces a valid intermediate tree (by Lemma 7.4), which can be further processed by Cut Tree resulting in a Gomory-Hu tree that contains $\mathcal{K}$. This directly shows the result of Barth et. al. [11] without any further proof.

Lemma 8.1. *If $c(b,d)$ changes by $\Delta > 0$, then each $\{u,v\} \in E_T$ remains a minimum u-v-cut (i) in $G^{\oplus}$ with cost $\lambda_G(u,v)$ if $\{u,v\} \notin \pi(b,d)$, (ii) in $G^{\ominus}$ with cost $\lambda_G(u,v) - \Delta$ if $\{u,v\} \in \pi(b,d)$.*

Proof. The edges on $\pi(b,d)$ are the only edges in E_T that represent cuts separating b and d. Thus, these edges represent the only cuts with changing costs in $T(G)$. The costs of those edges change exactly by Δ. If $c(\{b,d\})$ decreases, let $\{u,v\} \in \pi(b,d)$ and observe that the connectivity of any vertex pair decreases by at most Δ. Hence, $\{u,v\}$ is a minimum u-v-cut in $G^{\ominus}$, since $\lambda_{G^{\ominus}}(u,v) = \lambda_G(u,v) - \Delta = c_*(u,v) - \Delta$. If the cost of $\{b,d\}$ increases, the cuts whose costs do not change obviously remain minimum separating cuts in $G^{\oplus}$. □

For the case of decreasing edge costs, Barth et al. stress the difficulty of dynamically updating a Gomory-Hu tree, since in this case, Lemma 8.1 admits to only reuse the edges on the path $\pi(b,d)$. They leave this issue open.

In order to yet solve this problem and to give an efficient update approach for all types of changes, we start with the construction of two valid intermediate trees that represent the reusable cuts of Lemma 8.1 for the case of increasing edge costs and the case of decreasing edge costs. Note that for the latter such a tree exists by the same arguments as for the case of increasing costs. We stick with the convention for representing intermediate trees introduced in the previous section. The resulting tree for the increase of an edge cost is shown in Fig. 8.1(a). In this case, all but the edges on $\pi(b,d)$ can be reused. Hence, we draw these edges fat. The remaining edges are thinly drawn. The vertices are colored according to the compound nodes indicated by the thickness of the edges. Vertices incident to a fat edge correspond to a cut pair. For the decrease of an edge cost, the edges on $\pi(b,d)$ are fat, while the edges that do not lie on $\pi(b,d)$ are thin (see Fig. 8.1(b)). Furthermore, the costs of the fat edges decrease by Δ, since they all cross the changing edge $\{b,d\}$ in G. Compared to a construction from scratch, starting the Cut Tree procedure from these intermediate trees already saves $n-1-|\pi(b,d)|$ cut computations in the first case and $|\pi(b,d)|$ cut computations in the second case, where $|\pi(b,d)|$ counts the edges on $\pi(b,d)$. Hence, in scenarios with only little varying path lengths and a balanced number of increasing and decreasing costs, we can already save about half of the cut computations. In the following we want to use even more information from the previous Gomory-Hu tree $T(G)$ when executing Cut Tree unfolding the intermediate tree to a final Gomory-Hu tree of $(n-1)$ fat edges. To this end, we introduce some further lemmas that allow the reuse of old cuts given by $T(G)$. By the help of these lemmas, we enhance the obvious approach for increasing edge costs by a special treatment of newly inserted bridges, and give an update algorithm for the case of an edge deletion or a decreasing edge cost that provides much better changes to save cut computations that the obvious approach considered so far. In Chapter 9 we will see that



(a) Intermediate tree for $G^{\oplus}$.

(b) Intermediate tree for $G^{\ominus}$.

FIGURE 8.1: Intermediate trees representing reusable cuts detected by Lemma 8.1. Fat edges represent (reused) minimum cuts in $G^{\textcircled{\scriptsize U}}$, thin edges indicate compound nodes. Contracting thin edges yields nodes of white vertices (indicated by dotted lines). Black vertices correspond to singletons.

the latter also provides optimal temporal smoothness, that is, the number of cuts that occur in both Gomory-Hu sets of consecutive trees is maximum. In contrast, the approach for increasing edge costs and edge insertions nicely enforces temporal smoothness by reusing all edges that are not on the path $\pi(b, d)$, but it does not provide any smoothness guarantee for the remaining edges and cuts. Hence, in Chapter 9, we further develop this approach, such that updating the path $\pi(b, d)$ can be also done in an optimally smooth way.

8.1 Finding Reusable Cuts

We now focus on the reuse of those cuts that are still represented by thin edges in Fig. 8.1. If $\{b, d\}$ is inserted or the cost increases, the following corollary obviously holds, since the edge $\{b, d\}$ crosses each minimum b-d-cut.

Corollary 8.2. *If $\{b, d\}$ is newly inserted with $c^{\oplus}(b, d) = \Delta$ or $c(b, d)$ increases by Δ, any minimum b-d-cut in G remains valid in $G^{\oplus}$ with $\lambda_{G^{\oplus}}(b, d) = \lambda_G(b, d) + \Delta$.*

Note that reusing a valid minimum b-d-cut as split cut in the procedure CUT TREE separates b and d such that $\{b, d\}$ cannot be used again as step pair in a later iteration step. That is, we can reuse only one minimum b-d-cut, even if there are several such cuts represented in $T(G)$. Together with the following corollary, Corollary 8.2 directly allows the reuse of the whole Gomory-Hu tree $T(G)$ if $\{b, d\}$ is an existing bridge in G with increasing cost.

Corollary 8.3. *An edge $\{u, v\}$ is a bridge in G if and only if $c(u, v) = \lambda_G(u, v) > 0$. Then $\{u, v\}$ is also an edge in $T(G)$.*

Proof. The first part of Corollary 8.3 is obvious. In order to see that $\{u, v\}$ is also an edge in $T(G)$, first observe the following. The minimum u-v-cut in G is unique (up to equivalences if G is disconnected). Furthermore, none of the minimum separating cuts of a vertex pair $\{x, y\}$ that is no cut pair of the unique minimum u-v-cut θ in G separates u and v.

Now assume, $\{u, v\}$ is no edge in $T(G)$. Then the minimum u-v-cut is represented by another edge $\{w, z\}$, with w on $\pi(u, z)$. Hence, each cheapest edge on $\pi(u, w)$ in $T(G)$ represents a minimum u-w-cut that separates u and v, although $\{u, w\}$ is no cut pair of θ. This contradicts the observation above. □

If b and d are in different connected components in G and $\{b, d\}$ is a new bridge in $G^{\oplus}$, according to the next lemma, reusing the whole Gomory-Hu tree is also possible by replacing a single edge. This case can be easily detected having the Gomory-Hu tree $T(G)$ at hand, since $\{b, d\}$ is a new bridge if and only if $\lambda_G(b, d) = 0$. New bridges particularly occur if newly inserted vertices are connected for the first time.

Lemma 8.4. *Let* $\{b, d\}$ *be a new bridge in* $G^{\oplus}$. *Then replacing an edge of cost* 0 *on* $\pi(b, d)$ *in* $T(G)$ *by* $\{b, d\}$ *with cost* $c^{\oplus}(b, d)$ *yields a new Gomory-Hu tree* $T(G^{\oplus})$.

Proof. Since $\{b, d\}$ is a new bridge, b and d are in different connected components in G, and $\pi(b, d)$ in $T(G)$ contains at least one edge of cost 0. Replacing this edge by $\{b, d\}$ (of cost 0) does not create a cycle and is thus feasible. The resulting tree is again a Gomory-Hu tree for G, according to the observation we have already made in the context of vertex changes. According to Corollary 8.2, the minimum b-d-cut represented by $\{b, d\}$ in the new Gomory-Hu tree remains valid after the insertion of $\{b, d\}$ in G, and thus, it is feasible to simply increase the cost of $\{b, d\}$ in the tree. □

If $\{b, d\}$ is deleted or the cost decreases, handling bridges (always detectable by Corollary 8.3) is also easy, as stated by Lemma 8.5.

Lemma 8.5. *If* $\{b, d\}$ *is a bridge in* G *and the cost decreases by* Δ *(or* $\{b, d\}$ *is deleted), decreasing all edge costs on* $\pi(b, d)$ *in* $T(G)$ *by* Δ *yields a new cut tree* $T(G^{\ominus})$.

Proof. Since $\{b, d\}$ is a bridge in G, according to Corollary 8.3 and the observations in the proof thereof, the path $\pi(b, d)$ in $T(G)$ consists of only $\{b, d\}$ and represents the unique minimum b-d-cut θ in G. By Lemma 8.1, $\{b, d\}$, and thus θ, further remains a valid cut with cost $\lambda_G(b, d) - \Delta$ in $G^{\ominus}$. All other edges in $T(G)$ are incident to vertices that lie on a common cut side of the cut induced by $\{b, d\}$. Hence, those vertex pairs are no cut pairs of θ in $G^{\ominus}$, however, $\{b, d\}$ is still a bridge (possibly with cost 0) in $G^{\ominus}$. By the observation in proof of Corollary 8.3, none of the minimum separating cuts of these vertex pairs in $G^{\ominus}$ thus separates b and b, and the old cuts represented by these edges are still valid with respect to the vertices incident to the edges. □

If $\{b, d\}$ is no bridge, at least other bridges in G can be reused in case of an edge deletion or the decrease of an edge cost. Observe that a minimum separating cut in G only becomes invalid in $G^{\ominus}$ if there is a cheaper cut in $G^{\ominus}$ that separates the same vertex pair. Such a cut must cross the changing edge $\{b, d\}$ in G, since otherwise it would have been already cheaper in G. Hence, an edge in E_T corresponding to a bridge in G cannot become invalid, since any cut in $G^{\ominus}$ that crosses $\{b, d\}$ besides the bridge would be more expensive. In particular, this also holds for edges of cost 0 in E_T.

Corollary 8.6. *Let* $\{u, v\}$ *denote an edge in* $T(G)$ *with* $c_T(u, v) = 0$ *or an edge that corresponds to a bridge in* G. *Then* $\{u, v\}$ *is still a minimum* u-v-*cut in* $G^{\ominus}$.

The next lemma shows how a cut that remains valid in $G^{\ominus}$ may allow the reuse of all edges in E_T that lie on a common cut side. Figure 8.2 shows an example.

Lemma 8.7. *Let* $(U, V \setminus U)$ *be a minimum* u-v-*cut in* $G^{\ominus}$ *with* $\{b, d\} \subseteq V \setminus U$ *and let* $\{g, h\} \in E_T$ *be an edge in* $T(G)$ *with* $g, h \in U$. *Then* $\{g, h\}$ *is a minimum separating cut in* $G^{\ominus}$ *for all its previous cut pairs within* U.

Proof. Suppose there exists a minimum g-h-cut in $G^{\ominus}$ that is cheaper than the cut represented by $\{g, h\}$. Note that the cut $\{g, h\}$ costs the same in G and $G^{\ominus}$. Such a cheaper minimum g-h-cut in $G^{\ominus}$ would separate b and d in $V \setminus U$. At the same time, the Non-Crossing Lemma (7.2) would allow to bend such a cut along $V \setminus U$ such that it induces a minimum g-h-cut that does not separate b and d. The latter would have been already cheaper in G. □

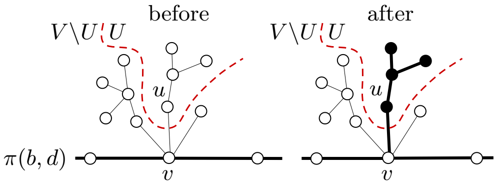


FIGURE 8.2: Situation of Lemma 8.7 in an intermediate tree for $G^{\ominus}$. Cut $(U, V\setminus V)$ (dashed red line) remains valid, edges of the subtree in U can be reused.

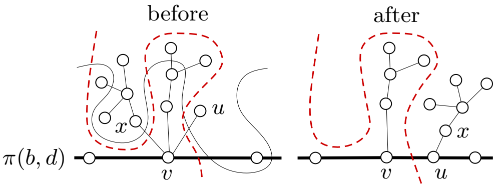


FIGURE 8.3: New cheaper cut for $\{u, v\}$ (solid black line) can be reshaped by Theorem 8.9 (dashed red line), such that the subtrees rooted at x and x' are not split. Reshaping is realized by reconnecting edges, $\{u, v\}$ becomes fat edge on $\pi(b, d)$.

Finally we see that a cut that is cheap enough cannot become invalid in $G^{\ominus}$. Note that the bound considered in this context depends on the current intermediate tree.

Lemma 8.8. *Let* $T_* = (V, E_*, c_*)$ *denote a valid intermediate tree for* $G^{\ominus}$*, where all edges on* $\pi(b, d)$ *are fat and let* $\{u, v\}$ *be a thin edge with* v *on* $\pi(b, d)$ *such that* $\{u, v\}$ *represents an old minimum u-v-cut in* G*. Let* N_π *denote the set of neighbors of* v *on* $\pi(b, d)$*. If* $\lambda_G(u, v) \leq \min_{x\in N_\pi}\{c_*(x, v)\}$*, then* $\{u, v\}$ *is a minimum u-v-cut in* $G^{\ominus}$*.*

Proof. The edges on $\pi(b, d)$ incident to v are fat, and thus, already represent minimum separating cuts in $G^{\ominus}$. Any new u-v-cut in $G^{\ominus}$ that is cheaper than the cut represented by $\{u, v\}$, must separate b and d, that is, must separate two adjacent vertices on $\pi(b, d)$. However, the fat edges incident to v on $\pi(b, d)$ shelter the remaining path edges from being separated by a new cut (due to the Non-Crossing Lemma (7.2)). Thus, there would exist a new cheaper cut that separates v from exactly one of its neighbors on the path, denoted by x. That is, the new cut must be at least as expensive as the cost of a minimum x-v-cut in $G^{\ominus}$, which is not possible if $\lambda_G(u, v)$ in G is already lower or equal. □

8.2 The Dynamic Gomory-Hu Tree Algorithm

We introduce two update procedures for the different edge changes, one for edge insertions and increasing costs, and one for edge deletions and decreasing costs. These procedures rely on the static iterative approach CUT TREE, but involve the results from Section 8.1 in order to save cut computations. The update procedures for vertex insertion and vertex deletion have been already discussed at the beginning of this chapter. We again represent intermediate trees by fat and thin edges, which simplifies the reshaping of cuts.

The procedure for an increasing edge cost or the insertion of an edge first checks if $\{b, d\}$ is a (maybe newly inserted) bridge in G. In this case, it adapts $c_T(b, d)$ according to Corollary 8.2 if $\{b, d\}$ already exists in G, and rebuilds $T(G)$ according to Lemma 8.4 otherwise. Both requires no cut computation. If $\{b, d\}$ is no bridge, the procedure constructs the intermediate tree shown in Figure 8.1(a), reusing all edges that are not on $\pi(b, d)$. Furthermore, it chooses one edge on $\pi(b, d)$ that represents a minimum b-d-cut in $G^{\oplus}$ and draws this edge fat (see Corollary 8.2). The resulting tree is then further processed by the procedure CUT TREE, which costs $|\pi(b, d)|-1$ cut computations and is correct according to Lemma 7.4.

The procedure for decreasing an edge cost or deleting an edge is given by Algorithm 4. We assume G and $G^{\ominus}$ to be available as global variables. Whenever the intermediate tree T_*

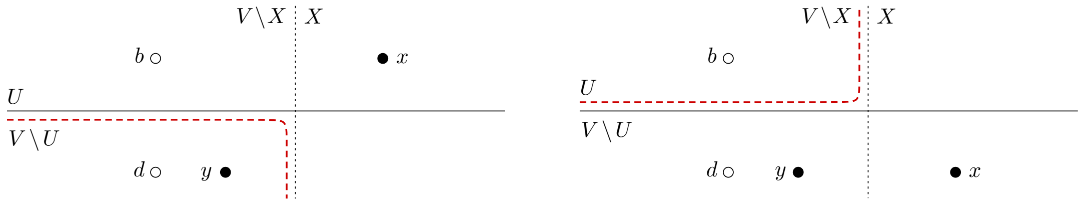
Algorithm 4: DECREASE OR DELETE

```
Input: $T(G)$, $b, d$, $c(b,d)$, $c^{\ominus}(b,d)$, $\Delta := c(b,d) - c^{\ominus}(b,d)$
Output: $T(G^{\ominus})$
1  $T_* \leftarrow T(G)$
2  if $\{b,d\}$ is a bridge then apply Lemma 8.5; return $T(G^{\ominus}) \leftarrow T_*$
3  Construct intermediate tree according to Figure 8.1(b)
4  $Q \leftarrow$ thin edges non-increasingly ordered by their costs
5  while $Q \neq \emptyset$ do
6      $\{u,v\} \leftarrow$ most expensive thin edge with $v$ on $\pi(b,d)$
7      $N_\pi \leftarrow$ neighbors of $v$ on $\pi(b,d)$; $L \leftarrow \min_{x \in N_\pi}\{c_*(x,v)\}$
8      if $L \geq \lambda_G(u,v)$ or $\{u,v\} \in E$ with $\lambda_G(u,v) = c(u,v)$ then    // Lem. 8.8 and Cor. 8.6
9          draw $\{u,v\}$ as a fat edge
10         consider the subtree $U$ rooted at $u$ with $v \notin U$,    // Lem. 8.7 and Fig. 8.2
11         draw all edges in $U$ fat, remove fat edges from $Q$
12         continue loop
13     $(U, V \setminus U) \leftarrow$ minimum $u$-$v$-cut in $G^{\ominus}$ with $u \in U$
14     draw $\{u,v\}$ as a fat edge, remove $\{u,v\}$ from $Q$
15     if $\lambda_G(u,v) = c^{\ominus}(U, V \setminus U)$ then goto line 10    // old cut still valid
16     $c_*(u,v) \leftarrow c^{\ominus}(U, V \setminus U)$    // otherwise
17     $N \leftarrow$ neighbors of $v$
18     forall the $x \in N$ do    // bend split cut by Theo. 8.9 and Lem. 7.2
19         if $x \in U$ then reconnect $x$ to $u$
20 return $T(G^{\ominus}) \leftarrow T_*$
```

changes during the run of Algorithm 4, the path $\pi(b,d)$ is implicitly updated without further notice. Thin edges are weighted by the old connectivity, fat edges by the new connectivity of their incident vertices. Whenever an edge is reconnected (by reconnecting one of its incident vertices), the newly occurring edge inherits the cost and the thickness from the disappearing edge. Algorithm 4 starts by checking if $\{b,d\}$ is a bridge (line 2) and reuses the whole Gomory-Hu tree $T(G)$ with adapted cost $c_T(b,d)$ (see Lemma 8.5) in this case. Otherwise (line 3), it constructs the intermediate tree shown in Figure 8.1(b), reusing all edges on $\pi(b,d)$ with adapted costs. Then it proceeds with iterative steps similar to CUT TREE. However, the difference is, that the step pairs are not chosen arbitrarily, but according to the edges in $T(G)$, starting with those edges that are incident to a vertex v on $\pi(b,d)$ (line 6). In this way, each edge $\{u,v\}$ which is found to remain valid in line 8 or line 15 allows to retain a maximal subtree (see Lemma 8.7), since $\{u,v\}$ is as close as possible to $\pi(b,d)$. The problem however is that cuts that are no longer valid, must be replaced by new cuts, which not necessarily respect the tree structure of $T(G)$. This is, a new cut possibly separates adjacent vertices in $T(G)$, which hence cannot be used as a step pair in a later step. Thus, we potentially miss valid cuts and the chance to retain further subtrees.

We solve this problem by reshaping the new cuts in the spirit of Gusfield, similar to the Non-Crossing Lemma (7.2). Lemma 8.9 shows how arbitrary cuts in $G^{\ominus}$ can be bent along old minimum separating cuts in G without becoming more expensive (see Figure 8.4).

Lemma 8.9. *Let $(X, V \setminus X)$ denote a minimum x-y-cut in G with $x \in X$, $y \in V \setminus X$ and $\{b,d\} \subseteq V \setminus X$. Let $(U, V \setminus U)$ denote a further cut. If (i) $(U, V \setminus U)$ separates x, y with $x \in U$, then $c^{\ominus}(U \cup X, V \setminus (U \cup X)) \leq c^{\ominus}(U, V \setminus U)$. If (ii) $(U, V \setminus U)$ does not separate x, y with $x \in V \setminus U$, then $c^{\ominus}(U \setminus X, V \setminus (U \setminus X)) \leq c^{\ominus}(U, V \setminus U)$.*



(a) Deflected by x, Lemma 8.9(i) bends $(U, V \setminus U)$ downwards along X.

(b) Deflected by x, Lemma 8.9(ii) bends $(U, V \setminus U)$ upwards along X.

FIGURE 8.4: Situation of Lemma 8.9. Reshaping cuts in $G^{\ominus}$ (solid black lines) along previous cuts in G (dotted black lines) resulting in new cuts in $G^{\ominus}$ (dashed red lines). Since we will apply Lemma 8.9 to cuts $(U, V \setminus U)$ that separate b and d, without loss of generality b is depicted in $(V \setminus X) \cap U$, and d is depicted in $(V \setminus X) \setminus U$ in this figure.

Proof. We prove Lemma 8.9(i) by contradiction, using the fact that $(X, V \setminus X)$ is a minimum x-y-cut in G. We show that $(U \cap X, V \setminus (U \cap X))$ would have been cheaper than the minimum x-v-cut $(X, V \setminus X)$ in G if $c^{\ominus}(U, V \setminus U)$ was cheaper than $c^{\ominus}(U \cup X, V \setminus (U \cup X))$ in $G^{\ominus}$. We express the costs of $(U \cap X, V \setminus (U \cap X))$ and $(X, V \setminus X)$ with the help of $(U, V \setminus U)$ and $(U \cup X, V \setminus (U \cup X))$ considered in Lemma 8.9(i). Note that $(U \cap X, V \setminus (U \cap X))$ and $(X, V \setminus X)$ do not separate b and d. Thus, their costs are unaffected by the deletion and it makes no difference whether we consider the costs in G or $G^{\ominus}$. We get

$$\begin{array}{llll} \textit{(i)} & c(U \cap X, V \setminus (U \cap X)) & = c^{\ominus}(U, V \setminus U) & \\ & & - \ c^{\ominus}(U \setminus X, V \setminus U) & + \ c^{\ominus}(U \setminus X, U \cap X) \\ \textit{(ii)} & c(X, V \setminus X) & = c^{\ominus}(U \cup X, V \setminus (U \cup X)) & \\ & & - \ c^{\ominus}(U \setminus X, V \setminus (U \cup X)) & + \ c^{\ominus}(U \setminus X, X) \end{array}$$

Since $V \setminus (U \cup X) \subseteq V \setminus U$, it is $c^{\ominus}(U \setminus X, V \setminus (U \cup X)) \leq c^{\ominus}(U \setminus X, V \setminus U)$. From $U \cap X \subseteq X$ further follows that $c^{\ominus}(U \setminus X, U \cap X) \leq c^{\ominus}(U \setminus X, X)$; together with the assumption that $c^{\ominus}(U, V \setminus U) < c^{\ominus}(U \cup X, V \setminus (U \cup X))$, by subtracting *(ii)* from *(i)*, we get:

$$\begin{array}{rl} c(U \cap X, V \setminus (U \cap X)) & - \ \ c(X, V \setminus X) \\ & = \ \ [c^{\ominus}(U, V \setminus U) - c^{\ominus}(U \cup X, V \setminus (U \cup X))] \\ & - \ \ [c^{\ominus}(U \setminus X, V \setminus U) - c^{\ominus}(U \setminus X, V \setminus (U \cup X))] \\ & + \ \ [c^{\ominus}(U \setminus X, U \cap X) - c^{\ominus}(U \setminus X, X)] < 0 \end{array}$$

This contradicts the fact that $(X, V \setminus X)$ is a minimum x-y-cut in G.

We prove Lemma 8.9(ii) with the help of the same technique. We show that $(X \setminus U, V \setminus (X \setminus U))$ would have been cheaper than the minimum x-y-cut $(X, V \setminus X)$ in G if $c^{\ominus}(U, V \setminus U)$ was cheaper than $c^{\ominus}(U \setminus X, V \setminus (U \setminus X))$ in $G^{\ominus}$. We express the costs of $(X \setminus U, V \setminus (X \setminus U))$ and $(X, V \setminus X)$ with the help of $(U, V \setminus U)$ and $(U \setminus X, V \setminus (U \setminus X))$ considered in Lemma 8.9(ii). Note that $(X \setminus U, V \setminus (X \setminus U))$ and $(X, V \setminus X)$ do not separate b and d. Thus, their costs are unaffected by the deletion and it makes no difference whether we consider the costs in G or $G^{\ominus}$. We get

$$\begin{array}{llll} \textit{(i)} & c(X \setminus U, V \setminus (X \setminus U)) & = c^{\ominus}(U, V \setminus U) & \\ & & - \ c^{\ominus}(U, V \setminus (X \cup U)) & + \ c^{\ominus}(X \setminus U, V \setminus (X \cup U)) \\ \textit{(ii)} & c(X, V \setminus X) & = c^{\ominus}(U \setminus X, V \setminus (U \setminus X)) & \\ & & - \ c^{\ominus}(U \setminus X, V \setminus (X \cup U)) & + \ c^{\ominus}(X, V \setminus (X \cup U)) \end{array}$$

Since $U \setminus X \subseteq U$, it is $c^{\ominus}(U \setminus X, V \setminus (X \cup U)) \leq c^{\ominus}(U, V \setminus (X \cup U))$. From $X \setminus U \subseteq X$ further follows that $c^{\ominus}(X \setminus U, V \setminus (X \cup U)) \leq c^{\ominus}(X, V \setminus (X \cup U))$; together with the assumption that $c^{\ominus}(U, V \setminus U) < c^{\ominus}(U \setminus X, V \setminus (U \setminus X))$, by subtracting *(ii)* from *(i)*, we get:

$$\begin{aligned} c(X \setminus U, V \setminus (X \setminus U)) \quad &- \quad c(X, V \setminus X) \\ &= \quad [c^{\ominus}(U, V \setminus U) - c^{\ominus}(U \setminus X, V \setminus (U \setminus X))] \\ &- \quad [c^{\ominus}(U, V \setminus (X \cup U)) - c^{\ominus}(U \setminus X, V \setminus (X \cup U))] \\ &+ \quad [c^{\ominus}(X \setminus U, V \setminus (X \cup U)) - c^{\ominus}(X, V \setminus (X \cup U))] < 0 \end{aligned}$$

This contradicts the fact that $(X, V \setminus X)$ is a minimum x-y-cut in G. $\square$

Whenever a new cheaper cut is found in line 13, which separates b and d, we apply Lemma 8.9 to this cut regarding the old cuts that are induced by the other thin edges $\{x, v\}$ incident to v. As a result, the new cut gets reshaped without changing its cost such that each subtree rooted at a vertex x is completely assigned to either side of the reshaped cut (line 19), depending on whether or not the new cut separates x and v. Figure 8.3 shows an example. Here the subtree at x is on the side of u, while the subtree at x' is on the side of v. That is, similar to the situation of the Non-Crossing Lemma (7.2), the old minimum separating cuts shelter the subtrees rooted at the neighbors of v. Furthermore, the Non-Crossing Lemma (7.2) allows the reshaping of the new cut along the cuts induced by the fat edge on $\pi(b, d)$ that are incident to v. This ensures that the new cut does not cross parts of T_* that are sheltered by the cuts induced by these flanking fat edges. Since after the reshaping exact one vertex adjacent to v on $\pi(b, d)$ ends up on the same cut side as u, u finally becomes a part of $\pi(b, d)$.

It remains to show that after the reconnection the reconnected edges are still incident to one of their cut pairs in $G^{\ominus}$ (for fat edges) and G (for thin edges), respectively. For fat edges this holds according to Lemma 7.3. For thin edges the order in line 4 guarantees that an edge $\{x, v\}$ that will be reconnected to u in line 19 is at most as expensive as the current edge $\{u, v\}$, and thus, also induces a minimum u-x-cut in G. This allows to apply Lemma 8.7 and Lemma 8.8 as well as the comparison in line 15 also to reconnected thin edges. In the end all edges in T_* are fat, since each edge is either a part of a reused subtree or was considered in line 6. We finally note that reconnecting a thin edge in line 19 makes this edge incident to a vertex on $\pi(b, d)$ and decrements the height of the related subtree.

8.3 Running Times of the Update Procedures

The running times of our fully-dynamic update procedures are clearly dominated by the computation of the new minimum separating cuts in $G^{\textcircled{\scriptsize U}}$. Thus, in the following, we evaluate the performance of our procedures by the number k of applied minimum-cut computations. Compared to the static algorithm CUT TREE, which in each time step constructs a Gomory-Hu tree from scratch, this yields $(n-1) - k$ saved cut computations. In this light, the procedures for vertex insertion and vertex deletion save $n - 1$ cut computations, which is 100% of effort saving. The saving for increasing edge cost and edge insertions simply depends on the length of the path $\pi(b, d)$ in the considered Gomory-Hu tree $T(G)$. Hence, in most cases one would expect high savings, unless the cut structure of the underlying graph is that much degenerated that the shape of the Gomory-Hu tree is close to a path. In fact, Gomory-Hu trees rather tend to a star-like shape, particularly for very regular graph structures. While a star yields optimal savings in the case of edge insertion or cost increase, namely $(n-1)-1=n-2$ saved computations, it constitutes the worst-case with respect to an edge deletion or cost decrease. Figure 8.5 shows an example based on a regular (unweighted) grid. In this example each vertex has degree 4, and we assume that the (global) edge connectivity is also 4 (imagine for example a grid of at least 4 rows and 4 columns on a torus). Then, the edge connectivity $\lambda_G(u,v)$ is also 4 for each vertex pair $\{u,v\} \subseteq V$, and $(\{u\}, V \setminus \{u\})$ is a minimum u-v-cut. Hence, any star with an arbitrary vertex c as center and edges of cost 4 is a Gomory-Hu tree of G. Now assume the edge $\{b,d\}$ depicted in Fig. 8.5 is deleted. For each vertex pair that contains at least b or d, this decreases the connectivity to 3, however, the deletion has no effect on the connectivity of the remaining vertex pairs. Thus, decreasing the costs at the edges $\{c,b\}$ and $\{c,d\}$ in $T(G)$ would suffice to update the Gomory-Hu tree. Instead, Algorithm 4 checks each edge, apart from the two edges on $\pi(b,d)$, by computing a new minimum separating cut in $G^{\ominus}$ in line 13. Consequently, it needs $n-3$ cut computations, which is the worst possible running time (recall that, according to Lemma 8.5, maintaining the whole tree is possible if $\pi(b,d)=\{b,b\}$, that is, if $\{b,d\}$ is a bridge). In Chapter 9 we will see that, with respect to the recomputation conjecture, this worst-case running time is still asymptotically optimal in the sense that even with the help of an extended Gomory-Hu tree, which provides comprehensive information about all minimum separating cuts in G, we cannot achieve a better asymptotic worst-case running time for solving the all-pairs minimum cut problem. Nevertheless, a first experimental proof of concept involving all types of changes promises high practicability, particularly on graphs with less regular structures. In the next section, we further present some experiments on non-consecutive edge changes, which we conducted in order to investigate the impact of the different factors that potentially help saving cut computations in the case of edge deletion and cost decrease.

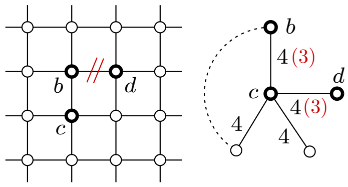


FIGURE 8.5: Grid graph G with star-shaped tree $T(G)$.

8.3.1 Experimental Proof of Concept

In the following brief experiment we use an evolving network of email communications within the Department of Informatics at KIT, obtained from data described in Section 1.4. In this instance, vertices represent members, and edges correspond to email contacts, weighted by the number of emails sent between two individuals during the last 72 hours. We process a queue of 924 900 atomic changes, which indicate the time steps in Fig. 8.6, and 923 031 of which concern edges. We start with an empty graph, constructing the network from scratch. Figure 8.6 shows the ratio of cuts computed by the update procedures and cuts needed by the static approach accumulated over all time steps so far. The ratio is shown in total, that is, for all procedures together (red line), as well as broken down to edge insertions (151 169 occurrences), increasing costs (310 473 occurrences), edge deletions (151 061 occurrences) and decreasing costs (310 328 occurrences). The trend of the curves follows the evolution of the graph, which slightly densifies around time step 100 000 due to a spam-attack; however, the update approach needs less than 4% of the static computations even during this period. These results are not least due to the less regular graph structures of the snapshots of this real world instance (see also Section 1.4). We further remark that for decreasing costs and edge deletions, Lemma 8.9 together with the Non-Crossing Lemma (7.2) allows to contract all subtrees incident to the current vertex v on $\pi(b,d)$

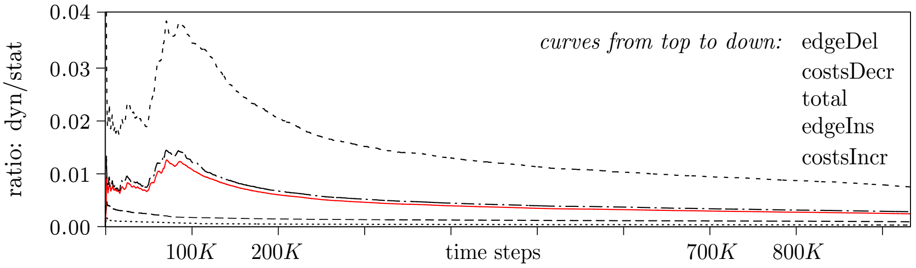


FIGURE 8.6: Cumulative ratio of dynamic and static minimum-cut computations over 924 900 time steps in the email-communication network of the Department of Informatics at KIT.

in line 6 of Algorithm 4, which shrinks the underlying graph to $deg_*(v)$ vertices, with $deg_*(v)$ the unweighted degree of v in the intermediate tree T_*. Such contractions could further speed up the single cut computations. Similar shrinkings can obviously be done for increasing costs and edge insertions, as well.

8.3.2 Experiments on Non-Consecutive Edge Changes

The performance of our update algorithm depends on two main factors, the changing edge and the shape of the current Gomory-Hu tree, where the latter highly depends on the structure of the underlying graph. Together these factors determine the length of the path $\pi(b, d)$ in the tree and the size of the subtrees that are potentially reusable according to Lemma 8.7. In a dynamic scenario, however, where we consider consecutive changes over the time, we face a slightly different graph structure before each change, which makes it difficult to see how the different factors affect the algorithm's performance.

In this experimental setup, we thus apply each possible change in a given graph independently, that is, with respect to the same initial graph structure, and aggregate the results as described in the following subsections. We give a detailed analysis in order to better understand the behavior of the different tools—Lemma 8.5, Corollary 8.6, Lemma 8.7 and Lemma 8.8—that allow to save cut computations in the case of edge deletion and decreasing cost.

The initial Gomory-Hu tree is constructed by applying the following version of Gusfield's algorithm (Algorithm 5) where the choice of the step pairs is determined by the non-increasing order of the vertices according to their weighted degrees. In line 6, Algorithm 5 uses the unique minimum u-v-cut that yields a smallest cut side for the source u, that is, the U-cut $\mathfrak{uc}(u, v)$ in G.

We use real world instances as well as generated instances. Most instances are taken from the testbed of the 10th DIMACS Implementation Challenge, which provides benchmark instances for partitioning and clustering. Additionally, we consider a snapshot of the linked wiki pages at www.dokuwiki.org, which we gathered ourselves. For more details see Section 1.4.

8.3.2.1 Experimental Setup

In the following setup each edge in the given graph is deleted once. After each deletion the initial Gomory-Hu tree is updated by the procedure DECREASE OR DELETE (Algorithm 4), counting the applied cut computations. Before deleting the next edge, we return to the initial graph and the initial Gomory-Hu tree. The results for the single edge deletions are aggregated and normalized yielding one value per instance G, as described below. We restrict ourselves to deletions, since deleting an edge that is, decreasing the edge cost as much as possible, intuitively

Algorithm 5: Cut Tree (Gusfield)

```
Input: Graph $G = (V, E, c)$
Output: Cut tree $T(G)$
$Q \leftarrow$ vertices non-increasingly ordered by their weighted degrees
$root \leftarrow$ first vertex in $Q$
$T_* \leftarrow$ all $u \in V \setminus \{root\}$ linked to $root$ by thin edges          // root is predecessor of $u$
forall the $u \in Q \setminus \{root\}$ do
    $\{u, v\} \leftarrow$ unique thin edge incident to $u$ in $T_*$                      // $u$ is a leaf in $T_*$
    $(U, V \setminus U) \leftarrow \mathfrak{uc}(u, v)$ in $G$
    $c_*(u, v) \leftarrow c(U, V \setminus U)$
    draw $\{u, v\}$ as fat edge
    $N \leftarrow$ neighbors of $v$ in $T_*$
    forall the $x \in N$ do
        if $x \in U$ then reconnect $x$ to $u$
return $T(G) \leftarrow T_*$
```

affects the minimum separating cuts the most, and thus, can be seen as a worst case scenario, which yields a lower bound on the performance of our algorithm.

The following tables provide the experimental results per instance. The key words *local* and *global* distinguish two different versions of the update algorithm. The local version uses the local bound L (as given in line 7 of Algorithm 4 and Lemma 8.8) to decide whether a cut can be reused, while the global version considers the new connectivity $\lambda_{G^\ominus}(b, d)$ instead, which is a global bound that does not depend on a special vertex on $\pi(b, d)$. Note that Lemma 8.8 also holds for $\lambda_{G^\ominus}(b, d)$. Since $L \geq \lambda_{G^\ominus}(b, d)$, the local bound allows to save at least as many cut computations as the global one. Our experiment shows how much better the performance actually is due to the local bound.

8.3.2.2 Overview on Savings (Table 8.1)

The values in Table 8.1 are decreasingly ordered by column 7. We consider actual savings and potential savings for single edge deletions and for the whole instance G. The *actual saving for one edge deletion* in G is the difference between the $n - 1$ cut computations, which are necessary in order to construct a Gomory-Hu tree for $G^\ominus$ from scratch, and the number of cut computations actually applied by the update algorithm after the edge deletion. However, the update algorithm may also recompute cuts that turn out to remain valid (see line 15, Algorithm 4). Such cut computations are needless, since they yield no additional information compared to the previous Gomory-Hu tree. A cut computation that is not needless is *necessary*. The *potential saving for one edge* in G is the saving that would be possible if we could decide for each cut whether it remains valid or not, applying a cut computation only if the cut does not remain valid. Note that the potential saving is the same for the local and the global version of the update algorithm, since the total number of necessary cut computations does not dependent on whether the local or the global bound is used. The *total saving for an instance* G (column 4 and 5) is the sum of the actual savings for all edge deletions. The *potential saving for* G (column 6) is the sum of the potential savings for all edge deletions. The values in column 4 to 6 are further normalized by $100/(m(n-1))$, yielding percentage savings with respect to the total number of cut computations needed for m Gomory-Hu tree constructions from scratch.

TABLE 8.1: Saved cut computations compared to computations from scratch.

graph	n	m	total local	total global	potential	worst edge local	worst edge global	potential worst edge
netscience	1589	2742	99.65	99.62	99.98	96.91	95.15	99.75
power	4941	6594	98.33	98.33	99.98	88.62	88.62	99.80
lesmis	77	254	91.64	90.12	99.77	67.11	60.53	97.37
dokuwiki_org	4416	12914	86.52	83.75	100.00	51.98	46.66	99.93
karate	34	78	80.50	71.33	98.80	48.48	6.06	93.94
dolphins	62	159	75.36	73.43	99.66	44.26	18.03	95.08
as-22july06	22963	48436	93.94	76.27	100.00	42.60	41.33	99.98
polblogs	1490	16715	76.85	76.21	100.00	27.60	27.54	99.87
email	1133	5451	66.49	64.28	100.00	14.22	14.13	99.82
adjnoun	112	425	59.33	57.23	99.96	10.81	9.91	99.10
polbooks	105	441	73.38	72.42	99.93	9.62	4.81	99.04
celegansneural	297	2148	50.76	49.03	99.99	8.78	8.45	99.32
data	2851	15093	50.26	50.26	100.00	7.54	7.54	99.96
celegans_metabolic	453	2025	60.54	58.58	99.99	6.64	6.19	99.78
jazz	198	2742	54.56	54.52	100.00	5.08	4.57	99.49
football	115	613	22.87	22.87	99.98	1.75	1.75	99.12
delaunay_n10	1024	3056	27.55	27.55	100.00	0.20	0.20	99.90
delaunay_n11	2048	6127	28.18	28.18	100.00	0.10	0.10	100.00
delaunay_n12	4096	12264	27.79	27.79	100.00	0.05	0.05	100.00

The *worst edge* of an instance G is the edge whose deletion causes the most cut computations during the update procedure. The actual saving for this deletion, normalized by $100/(n-1)$ (column 7 and 8), is a lower bound on the algorithm's performance on G. The *potential worst edge* is the edge whose deletion requires the most necessary computations. Column 9 shows the potential saving for this edge deletion, also normalized by $100/(n-1)$.

Discussion. At a first glance we observe that the potential saving for all instances is very high. That is, the cut structure of the graphs is very robust with respect to single edge deletions. Even by deleting the potential worst edges only few cuts become invalid. Note that in column 6 some instances have 100% saving, while deleting the potential worst edge in column 9 may save less than 100%. This is because the total number of cuts that change due to all edge deletions is that small that it is not visible in the normalized potential saving in column 6.

The total saving, as well as the lower bound on the algorithm's performance given by the worst edge, varies very much. At the top we have netscience, which can be updated by saving at least 96.91% of the $n-1$ cut computations needed by the static algorithm. At the bottom we see delaunay_n12, for which the update algorithm still computes 95% of the $n-1$ cut computations that are needed by the static algorithm. In most cases, however, the update algorithm could save at least half of the static cut computations. Considering Table 8.2 we will see how the actual saving depends on the structure of the Gomory-Hu tree.

We finally observe that a difference between the local and the global version of the update algorithm is visible, however, the global version competes well, which will be also confirmed by the results in Table 8.2. Nevertheless, with respect to a single edge deletion it might be a considerable advantage using the local version. See, for example, the results for the worst edges of karate and dolphins.

TABLE 8.2: Saved cut computations by causative factors.

graph	n	m	total		bDel	path	bridge	cost		subT
			local	global				local	global	
netscience	1589	2742	99.65	99.62	7.80	0.10	0.04	0.20	0.18	91.50
power	4941	6594	98.33	98.33	24.43	0.03	0.04	3.50	3.50	70.32
as-22july06	22963	48436	93.94	76.27	16.51	0.01	1.38	37.76	20.09	38.29
lesmis	77	254	91.64	90.12	7.09	2.48	3.36	10.09	8.57	68.62
dokuwiki_org	4416	12914	86.52	83.75	12.68	0.04	0.19	23.51	20.73	50.11
karate	34	78	80.50	71.33	1.28	5.17	1.67	35.04	25.87	37.33
polblogs	1490	16715	76.85	76.21	0.84	0.13	0.13	48.63	47.99	27.12
dolphins	62	159	75.36	73.43	5.66	3.04	1.07	22.27	20.34	43.31
polbooks	105	441	73.38	72.42	0.00	1.86	0.00	23.71	22.75	47.81
email	1133	5451	66.49	64.28	2.84	0.17	0.13	49.74	47.53	13.61
celegans_metabolic	453	2025	60.54	58.58	0.40	0.41	0.01	51.25	49.29	8.47
adjnoun	112	425	59.33	57.23	2.35	1.68	1.15	46.52	44.42	7.62
jazz	198	2742	54.56	54.52	0.18	1.00	0.02	49.16	49.11	4.20
celegansneural	297	2148	50.76	49.03	0.70	0.66	3.38	41.04	39.30	4.99
data	2851	15093	50.26	50.26	0.00	0.07	0.00	27.93	27.93	22.26
delaunay_n11	2048	6127	28.18	28.18	0.00	0.10	0.00	28.08	28.08	0.00
delaunay_n12	4096	12264	27.79	27.79	0.00	0.05	0.00	27.74	27.74	0.00
delaunay_n10	1024	3056	27.55	27.55	0.00	0.20	0.00	27.36	27.36	0.00
football	115	613	22.87	22.87	0.00	1.74	0.00	21.13	21.13	0.00

8.3.2.3 Savings by Causative Factors (Table 8.2)

The values in Table 8.2 are decreasingly ordered by column 4. Column 1 to 5 review the total saving per instance G from Table 8.1, while the remaining columns itemize the different arguments or insights that lead to the saving. Note that column 6 to 11 add up to column 4 and 5, respectively. All values are again normalized by $100/(m(n-1))$. The first argument concerns edges that are bridges in G. After the deletion of such an edge in G, the Gomory-Hu tree can be updated without computing any cut (see Lemma 8.5). Column 6 (*bDel*) shows the saving for G that is due to deleted bridges.

In any other case, that is, if the deleted edge is no bridge, cut computations are saved due to further arguments regarding special edges in the intermediate tree T_*. Column 7 (*path*) lists the saving due to tree edges on $\pi(b,d)$, which can be reused according to Lemma 8.1. Column 8 (*bridge*) lists the saving due to tree edges in T_* that are bridges in G and thus can be reused as stated by Corollary 8.6. Column 9 and 10 (*cost*) list the saving due to tree edges in T_* that cost at most L (*local* bound) or $\lambda_{G^\ominus}(b,d)$ (*global* bound) and are reusable according to Lemma 8.8. Finally, column 11 (*subT*) lists the saving due to subtrees that are linked by a reusable tree edge, and thus, can be reused according to Lemma 8.7.

Discussion. Here the different Gomory-Hu tree structures become obvious. Since the cut structures of all instances turned out to be quite robust, we know that most of the existing subtrees are reused according to Lemma 8.7. Thus, high values in column 11 indicate that the Gomory-Hu trees have a substantial subtree structure (see netscience). If the value is low, the Gomory-Hu tree must be close to a star (see delaunay_n12). In a star-like tree exist no substantial subtrees, since the diameter is small and most edges link only few vertices (that is, tiny subtrees) to the rest of the tree. Gomory-Hu trees close to stars occur for graphs with a very regular edge

TABLE 8.3: Edge frequencies by applied/necessary cut computations.

graph	none_appl		majority_appl		none_nec	majority_nec
	local	global	local	global		
netscience	12.29	12.22	87.71 (0-1/8]	87.78 (0-1/8]	69.62	30.38 (0-1/8]
power	25.08	25.08	74.92 (0-1/8]	74.92 (0-1/8]	52.64	47.36 (0-1/8]
lesmis	7.48	7.48	70.47 (0-1/8]	64.17 (0-1/8]	84.25	15.75 (0-1/8]
as-22july06	16.52	16.52	68.71 (0-1/8]	37.49 (0-1/8]	63.44	36.56 (0-1/8]
dokuwiki_org	12.70	12.69	43.19 (0-1/8]	37.59 (0-1/8]	90.92	9.08 (0-1/8]
karate	2.56	2.56	35.90 (0-1/8]	30.77 (1/8-2/8]	61.54	38.46 (0-1/8]
polblogs	0.84	0.84	28.39 (0-1/8]	31.87 (1/8-2/8]	98.76	1.24 (0-1/8]
polbooks	0.68	0.68	28.12 (1/8-2/8]	26.98 (1/8-2/8]	92.52	7.48 (0-1/8]
dolphins	6.29	6.29	27.67 (1/8-2/8]	29.56 (1/8-2/8]	84.28	15.72 (0-1/8]
email	2.86	2.86	22.89 (2/8-3/8]	22.11 (2/8-3/8]	95.60	4.40 (0-1/8]
adjnoun	2.59	2.59	21.41 (2/8-3/8]	21.18 (2/8-3/8]	95.06	4.94 (0-1/8]
data	0.00	0.00	56.79 (3/8-4/8]	56.79 (3/8-4/8]	99.74	0.26 (0-1/8]
jazz	0.26	0.26	19.37 (3/8-4/8]	19.33 (3/8-4/8]	99.85	0.15 (0-1/8]
football	0.00	0.00	50.41 (5/8-6/8]	50.41 (5/8-6/8]	98.04	1.96 (0-1/8]
celegans_metabolic	0.59	0.59	22.27 (5/8-6/8]	22.07 (5/8-6/8]	95.70	4.30 (0-1/8]
celegansneural	0.74	0.74	18.81 (5/8-6/8]	18.62 (5/8-6/8]	95.90	4.10 (0-1/8]
delaunay_n11	0.00	0.00	37.23 (6/8-7/8]	37.23 (6/8-7/8]	100.00	0.00 (0-1/8]
delaunay_n12	0.00	0.00	53.42 (7/8-8/8]	53.42 (7/8-8/8]	100.00	0.00 (0-1/8]
delaunay_n10	0.00	0.00	53.21 (7/8-8/8]	53.21 (7/8-8/8]	99.61	0.39 (0-1/8]

structure (see the example in Figure 8.6). In such structures the cheapest way to separate two vertices is mostly to cut off the vertex of cheaper degree. The Delaunay triangulation exemplarily shown in Fig. 1.5(b) confirms a regular structure of the graph `delaunay_n12`. We observe that for instances with a substantial subtree structure, Lemma 8.7 is also the most powerful tool to speedup the Gomory-Hu tree calculation. For instances with Gomory-Hu trees close to stars, the comparison to the costs of path edges (Lemma 8.8) saves the most computations, while the savings due to the path $\pi(b, d)$ are marginal. The savings due to bridges directly correspond to the amount of bridges in the instances. In our test instances most bridges are edges incident to degree-1 vertices.

8.3.2.4 Edge Frequencies by Applied/Necessary Computations (Table 8.3)

The total range of savings for a single edge deletion reaches from no computations applied/necessary ($\equiv$ maximum saving) to $n-3$ computations applied/necessary ($\equiv$ minimum saving). We split this total range into the following intervals. Each interval is defined by a factor $\beta_i := i/8$, with $i \in \{1, \ldots, 8\}$, that fixes the following bounds for interval i. The lower bound is $\beta_{i-1}(n-1)$, the upper bound is $\beta_i(n-1)$. We assign each edge deletion, that is, each single run of the update algorithm that applies/necessarily needs at least one cut computation, to the interval corresponding to applied/necessarily needed amount of cut computations. The values are decreasingly ordered by column 4 and 5.

Column 2 and 3 (*none_appl*) list the percentage of edge deletions for which no cut computation is applied. Column 4 and 5 (*majority_appl*) show in brackets the interval to which most of the remaining edge deletions are assigned and list the number of edge deletions in this interval in %

with respect to m. Column 6 (*none_nec*) and 7 (*majority_nec*) show the analogous results with respect to necessary cut computations.

Discussion. For most instances the values in column 2 and 3 are equal, although the savings that depend on the used bound (local or global) differ (see column 9 and 10, Table 8.2). Furthermore, the ranking of the instances with respect to column 2 closely corresponds to the ranking given by column 6 in Table 8.2. This indicates that most of the edge deletions where no cut computation is applied are deletions of bridges in G. Nevertheless, a closer look at the data shows that there are two instances, netscience and dokuwiki_org, where also deleting an edge that is no bridge allows an update without cut computations.

The majorities in column 4 and 5 further show that good total savings in Table 8.2 are clearly due to many edges, each of which saving a many computations. Other instances have only few total savings in Table 8.2, since the majority of their edges causes many cut computations. All in all we see that the total saving of an instance in Table 8.2 is the better the more cut computations are saved by the majority of edges. The majorities with respect to necessary cut computations correlate in a similar way to the total potential saving in Table 8.1.

CHAPTER 9

Optimality in Smoothness and Running Time

In the context of dynamic updates, the two main tasks are temporal smoothness and the improvement of the running time compared to a computation from scratch. The idea of temporal smoothness is that the update results for consecutive time steps are as similar as possible. That is, the updated object or structure evolves smoothly over time. The formal definition of similarity is individual for each particular object or structure. In this work, we consider Gomory-Hu trees in dynamic scenarios and proposed an update algorithm for this structure in Chapter 8. In Section 9.1 and Section 9.2 of this chapter, we now prove that Gomory-Hu trees can be updated with optimal temporal smoothness. To this end we define the similarity of two Gomory-Hu trees of consecutive snapshots G and $G^{\textcircled{U}}$ as follows. Let $T(G)$ and $T(G^{\textcircled{U}})$ denote the Gomory-Hu trees of G and $G^{\textcircled{U}}$. We consider both trees as sets of cuts and measure the similarity by the number of cuts that appear in both sets. Recall the definition of equivalent cuts in G and $G^{\textcircled{U}}$ in Section 6.2. For the sake of simplicity, we refrain from a normalization of this similarity measure. Hence, it ranges between 0 and $n-1$. That is, $T(G)$ and $T(G^{\textcircled{U}})$ are identical if $|T(G) \cap T(G^{\textcircled{U}})| = n-1$ and they are completely different if $|T(G) \cap T(G^{\textcircled{U}})| = 0$. The temporal smoothness of $T(G)$ and $T(G^{\textcircled{U}})$ is optimal if $|T(G) \cap T(G^{\textcircled{U}})|$ is as large as possible, where large as possible means there exists no other Gomory-Hu tree $T'(G^{\textcircled{U}})$ such that $|T(G) \cap T'(G^{\textcircled{U}})| > |T(G) \cap T(G^{\textcircled{U}})|$ for the fixed tree $T(G)$. We show that in all cases of atomic edge changes in G, the new Gomory-Hu tree resulting from our update algorithm contains a maximum number of *old cuts*, that is, cuts that have been already represented by the Gomory-Hu tree before the change. This in particular implies the *stability* of our update approach, saying that a Gomory-Hu tree that can be preserved indeed is preserved in the next time step. In Section 9.1, we start with the cases of edge deletion and cost decrease, which we consider as one case, since our algorithm handles them by the same dynamic update procedure. The same holds for the cases of edge insertion and cost increase, which we consider in Section 9.2. In contrast to the case of edge deletion or cost decrease, where optimal smoothness is already guaranteed by the update procedure presented in Chapter 8, in the case of edge insertion or cost increase, we still need to modify the update procedure presented in Chapter 8 in order to achieve optimal temporal smoothness. The modified procedure is also described in Section 9.2. We further remark that the temporal smoothness is trivially optimal in the cases of vertex insertion and vertex deletion, since only singletons are inserted or deleted.

In Section 9.3, we finally show that our update algorithm is also optimal in terms of asymptotic worst-case running time. More precisely, we give an example that shows that providing further information about the cut structure of the current snapshot (beyond the information provided by a Gomory-Hu tree) does not achieve any better asymptotic worst-case running time for solving the all-pairs minimum-cut problem in a dynamic scenario. This result is based on the assumption that finding a minimum s-t-cut in $G^{\textcircled{U}}$ that is neither an old minimum separating cut in G (with respect to any cut pair) nor associated with a known maximum s-t-flow in G, needs at least one cut computation from scratch in $G^{\textcircled{U}}$. According to the discussion on updating flows and DAGs in Section 6.2.3, this *recomputation conjecture* is a reasonable assumption.

Before we start with Section 9.1, we stress the following fact that is implicitly used in the following proofs. According to the Non-Crossing Lemma (7.2), cuts that are represented by fat edges in any intermediate tree always shelter a whole cut side. That is, reconnecting such an edge does not change the represented cut. This further guarantees that each old cut that once became a fat edge in an intermediate tree for $G^{\textcircled{U}}$ definitely contributes to the temporal smoothness.

9.1 Optimal Temporal Smoothness in Case of Edge Deletion or Cost Decrease

In this section we prove that the update procedure Decrease or Delete (Algorithm 4) presented in Chapter 8 satisfies the following *smoothness condition* and argue that this condition is sufficient to guarantee optimal temporal smoothness.

Condition 9.1 (smoothness)**.** *Let θ denote a cut in $T(G)$ and let Q denote the set of cut pairs of θ in $G^{\ominus}$. An update procedure for $T(G)$ (for the case of edge deletion or cost decrease) satisfies the* smoothness condition *if the following holds. If $Q \neq \emptyset$ (that is, θ is also a minimum separating cut in $G^{\ominus}$), the updated Gomory-Hu tree $T(G^{\ominus})$ contains again a set $\mathcal{O}$ of old cuts that together separate all cut pairs in Q. This set $\mathcal{O}$ either consists of only the cut θ or does not contain θ. In the latter case, the set $\mathcal{O}$ further has the following properties:*

- *(i) Let $\widetilde{Q}$ denote the set of all cut pairs of the cuts in $\mathcal{O}$ in $G^{\ominus}$. Each cut in $\mathcal{O}$ shares each of its cut pairs in $\widetilde{Q}$ with at least one other cut in $\mathcal{O} \cup \{\theta\}$.*
- *(ii) Let θ' be another cut in $T(G)$ with $Q' \neq \emptyset$ and $\mathcal{O}' \neq \{\theta'\}$. Then it is $\mathcal{O}' \cap \mathcal{O} = \emptyset$.*

Theorem 9.2. *Any update procedure (for the case of edge deletion or cost decrease) that satisfies the smoothness condition guarantees optimal temporal smoothness.*

Proof. Let $T(G^{\ominus})$ denote a Gomory-Hu tree that results from an update procedure that satisfies the smoothness condition, let $T'(G^{\ominus})$ denote another Gomory-Hu tree for $G^{\ominus}$ such that $T(G) \cap T'(G^{\ominus}) \neq T(G) \cap T(G^{\ominus})$. That is, $T'(G^{\ominus})$ contains at least one old cut θ (from $T(G)$) that is not in $T(G^{\ominus})$. We show that, nevertheless, $T(G^{\ominus})$ contains at least as many old cuts as $T'(G^{\ominus})$, and thus, contains a maximum set of old cuts proving that the update procedure guarantees optimal temporal smoothness.

Let $Q \neq \emptyset$ denote the set of cut pairs of θ in $G^{\ominus}$. Since θ is not in $T(G^{\ominus})$, the set $\mathcal{O}$ of old cuts in $T(G^{\ominus})$ that separate the cut pairs in Q does not consist of θ, that is, $\mathcal{O} \neq \{\theta\}$. Instead, $\mathcal{O}$ has properties (i) and (ii). From (i) follows, due to the characterization of partial Gomory-Hu sets (see Lemma 7.4), that $\mathcal{O} \cup \{\theta\}$ is no partial Gomory-Hu set of $T'(G^{\ominus})$, and hence, at least one old cut in $\mathcal{O}$ is contained in $T(G^{\ominus})$ but not in $T'(G^{\ominus})$. More generally, for each old cut θ_i in $T'(G^{\ominus})$ that is not in $T(G^{\ominus})$ there is at least one other old cut $\gamma_i \in \mathcal{O}_i \not\ni \theta_i$ in $T(G^{\ominus})$ that is not in $T'(G^{\ominus})$. Since all these sets satisfy $\mathcal{O}_i \neq \{\theta_i\}$, it follows from property (ii) that $\mathcal{O}_i \cap \mathcal{O}_j = \emptyset$ for $i \neq j$. Thus, all the cuts γ_i are distinct and $T(G^{\ominus})$ contains at least as many old cuts as $T'(G^{\ominus})$. ☐

The rest of this section now focuses on the proof of the following key theorem.

Theorem 9.3. *The update procedure* Decrease or Delete *(Algorithm 4) presented in Chapter 8 satisfies the smoothness condition.*

Proof of Theorem 9.3

Let $\{u, v\}$ denote the edge in $T(G)$ that represents $\theta = (U, V \setminus U)$ with $u \in U$ and assume $Q \neq \emptyset$. We distinguish whether or not $\{u, v\}$ is an edge on the path $\pi(b, d)$ in $T(G)$ between the vertices b and d of the changing edge in G.

Claim: If $\{u, v\}$ is an edge on $\pi(b, d)$, it is $\mathcal{O} = \{\theta\}$. If $\{u, v\}$ is an edge on $\pi(b, d)$, the cut $(U, V \setminus U)$ is also a minimum u-v-cut in $G^{\ominus}$, but with reduced cost. This is $Q \neq \emptyset$. We argue that the construction of $T(G^{\ominus})$ preserves an edge (not necessarily incident to u and v in $T(G^{\ominus})$) on $\pi(b, d)$ that still represents $(U, V \setminus U)$. This follows directly from the following key observation: during the construction of $T(G^{\ominus})$, tree edges in U are only reconnected within U and tree edges in $V \setminus U$ are only reconnected within $V \setminus U$. This can be easily seen, since edges are always reconnected to a neighbor of their center that is not on $\pi(b, d)$ before the reconnection (recall the reconnection operation illustrated in Fig. 8.3. Note that u and v denote other vertices there). Furthermore, we observe that, besides the edge representing θ, each edge on $\pi(u, v)$ in $T(G^{\ominus})$ represents a new cut, since, besides θ, no old cut in $T(G)$ separates u and v. Thus, no other old cut in $T(G^{\ominus})$ separates u and v, and it is $\mathcal{O} = \{\theta\}$. Hence, Theorem 9.3 holds.

If the edge $\{u, v\}$ is not on $\pi(b, d)$ in $T(G)$, we distinguish whether or not Algorithm 4 considers this edge in line 6 (at this point the edge is possibly incident to new vertices), and if it is considered, whether or not Algorithm 4 finds the cut $\theta = (U, V \setminus U)$ to be a minimum separating cut in $G^{\ominus}$ with respect to the vertices currently incident to the edge representing $(U, V \setminus U)$. More precisely, if the edge $\{u, v\}$ is not on $\pi(b, d)$ in $T(G)$ the situation in Algorithm 4 is as follows. After the construction of the intermediate tree according to Fig. 8.1(b) (line 3), the edge $\{u, v\}$ is a thin edge. During the construction of $T(G^{\ominus})$, it is then possibly reconnected several times still remaining a thin edge (but incident to new vertices) and still representing the cut $(U, V \setminus U)$.

Claim: If the edge representing $\theta = (U, V \setminus U)$ is not considered in line 6, it is $\mathcal{O} = \{\theta\}$. If the edge is not considered in line 6, it must have become a fat edge due to the reuse of a subtree according to Lemma 8.7, that is, $Q \neq \emptyset$. Then however it was never incident to a vertex on the path $\pi(b, d)$ in the intermediate tree, and thus, it was never reconnected. Hence, $\{u, v\}$ is a final edge in $T(G^{\ominus})$ and no other old cut in $T(G^{\ominus})$ separates u and v, which still form a cut pair of $(U, V \setminus U)$. That is, $\mathcal{O} = \{\theta\}$ and Theorem 9.3 holds.

Otherwise, at some time, the procedure finally considers the edge that represents $(U, V \setminus U)$ in line 6 checking if it represents a minimum w-z-cut in $G^{\ominus}$ with respect to their currently incident vertices w and z. Observe that at this point the fat edges on $\pi(u, v)$ in the intermediate tree result from only new cuts. Without loss of generality, we assume z is the center and $w \in U$. Now we distinguish the two cases described above.

Claim: If Algorithm 4 finds the cut $(U, V \setminus U)$ to be a minimum w-z-cut in $G^{\ominus}$, it is $\mathcal{O} = \{\theta\}$. In this case, it is $Q \neq \emptyset$ and the edge becomes a thick edge, still representing $(U, V \setminus U)$. Since any further reconnection of this thick edge does not change the represented cut, and thick edges become edges in $T(G^{\ominus})$ in the end, it is $\theta \in T(G^{\ominus})$. Moreover, each further edge on $\pi(u, v)$ in $T(G^{\ominus})$ represents a new cut, and thus, no other old cut in $T(G^{\ominus})$ separates u and v. Hence, it is $\mathcal{O} = \{\theta\}$ and Theorem 9.3 holds.

Claim: If Algorithm 4 finds that the cut $(U, V \setminus U)$ *is no minimum* w-z*-cut in* $G^{\ominus}$*, it is* $\theta \notin \mathcal{O}$. Since we assume that $Q \neq \emptyset$, in this case, $(U, V \setminus U)$ is no minimum separating cut with respect to w and z, but with respect to another cut pair $\{x, y\} \in Q$. However, as it is not found to be reusable by the algorithm, it is $\theta \notin \mathcal{O}$.

In the following, we explicitly define the set $\mathcal{O}$ of old cuts in $T(G^{\ominus})$ that separate the cut pairs in Q and show that $\mathcal{O}$ satisfies the properties (i) and (ii) given in Theorem 9.3. This will finish the proof of Theorem 9.3.

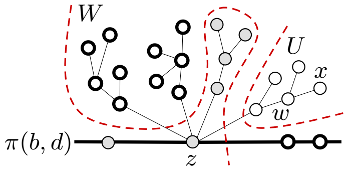


FIGURE 9.1: Partition of V induced by cut $(W, V \setminus W)$ and cut $(U, V \setminus U)$. Bold-framed vertices represent set A, (gray) filled vertices represent set B, and empty vertices represent set U.

Since $(U, V \setminus U)$ is a minimum x-y-cut in $G^{\ominus}$, with $x \in U$, but no minimum separating cut with respect to $\{w, z\}$, there is a cheaper minimum w-z-cut $(W, V \setminus W)$ in $G^{\ominus}$ with a special shape according to Lemma 8.9 and the Non-Crossing Lemma (7.2) (see Fig. 9.1). In particular we may assume $U \subset W$. This minimum w-z-cut together with the cut $(U, V \setminus U)$ induces a partition $V = A \cup B \cup U$ of V with $A := W \setminus U$ and $B := V \setminus W$. In a first step, we show that the second vertex y of the cut pair $\{x, y\}$ of $(U, V \setminus U)$ must be in $A \setminus \pi(b, d)$ (while $x \in U$). There will follow three further steps.

Step 1: $y \in A \setminus \pi(b, d)$. Obviously, $y \notin U$, since otherwise, $(U, V \setminus U)$ would not separate x and y. If $y \in B$, the cut $(W, V \setminus W)$ separates x and y, which contradicts the assumption that $(U, V \setminus U)$ is a minimum x-y-cut in $G^{\ominus}$, since $c^{\ominus}(W, V \setminus W) < c^{\ominus}(U, V \setminus U)$. If $y \in A \cap \pi(b, d)$, let $\{r, z\}$ denote the edge on $\pi(b, d)$ that is crossed by $(W, V \setminus W)$. The cut induced by this edge separates U from all vertices in $A \cap \pi(b, d)$ and in particular x from y. Since this edge is a thick edge crossed by $(W, V \setminus W)$, the corresponding cut in $G^{\ominus}$ is at most as expensive as $(W, V \setminus W)$, which is cheaper than $(U, V \setminus U)$. This again contradicts the assumption that $(U, V \setminus U)$ is a minimum x-y-cut in $G^{\ominus}$. Hence, it must be $y \in A \setminus \pi(b, d)$.

Step 2: Preliminaries for the definition of $\mathcal{O}$ *in* $T(G^{\ominus})$. Now consider $y \in A \setminus \pi(b, d)$, and let $\{r, z\}$ denote the edge incident to the center z such that the subtree rooted at r contains y. Observe that $\{r, z\}$ is crossed by $(W, V \setminus W)$, since otherwise y would have been in B. Moreover, we observe that the cut induced by $\{r, z\}$ separates x and y. If $\{r, z\}$ was a thick edge, the corresponding cut in $G^{\ominus}$ would be at most as expensive as $(W, V \setminus W)$ (otherwise it would not have been crossed by $(W, V \setminus W)$), which is cheaper than $(U, V \setminus U)$, again yielding a contradiction. Hence, $\{r, z\}$ must be a thin edge. Due to the sorting in line 4 it costs at most $c(U, V \setminus U) = c^{\ominus}(U, V \setminus U)$, as otherwise it would have been considered before the edge $\{w, z\}$, and thus, would have been a thick edge. If $\{r, z\}$ was cheaper than $c^{\ominus}(U, V \setminus U)$, it follows that in $G^{\ominus}$ the cut represented by $\{r, z\}$ (which separates x and y) is cheaper than $(U, V \setminus U)$, contradicting the assumption that $(U, V \setminus U)$ is a minimum x-y-cut in $G^{\ominus}$. Hence, the thin edge $\{r, z\}$ costs exactly $c^{\ominus}(U, V \setminus U) = c(U, V \setminus V)$.

Moreover, it is $\lambda_{G^{\ominus}}(r, w) = c^{\ominus}(U, V \setminus U)$. This can be seen as follows. Obviously, $(U, V \setminus U)$ separates r and w and thus it is $\lambda_{G^{\ominus}}(r, w) \geq c^{\ominus}(U, V \setminus U)$. On the other hand, a minimum r-w-cut in $G^{\ominus}$ either separates r form z or w from z, and thus, can be always reshaped according to Lemma 8.9, such that it does not split the subtrees rooted at r and w. Then, however, it needs to separate x and y. Consequently, $\lambda_{G^{\ominus}}(r, w) < c^{\ominus}(U, V \setminus U)$ is not possible.

In Section 8.2, we have seen that after reconnecting $\{r, z\}$ according to cut $(W, V \setminus W)$ (see

Fig. 9.1 and recall the reconnection operation illustrated in Fig. 8.3), the (reconnected) thin edge $\{r, w\}$ represents a minimum separating cut in G, denoted by $(R, V \setminus R)$, with respect to $\{r, w\}$ and $\{x, y\}$. Since $c(R, V \setminus R) = c^{\ominus}(R, V \setminus R) = c^{\ominus}(U, V \setminus U) = c(U, V \setminus U)$, and since we know that $\lambda_{G^{\ominus}}(r, w) = c^{\ominus}(U, V \setminus U)$, it finally holds that the cut $(R, V \setminus R)$ represents a minimum separating cut in $G^{\ominus}$ with respect to it incident vertices.

Step 3: Definition of $\mathcal{O}$ *in* $T(G^{\ominus})$ *and proof of* (ii). At this point, we define $\mathcal{O}$ as the set consisting of all such cuts $(R, V \setminus R)$ associated with a cut pair $\{x, y\} \in Q$ of $\theta = (U, V \setminus U)$ at the time the edge representing $(U, V \setminus U)$ is considered by Algorithm 4. These cuts together separate all cut pairs in Q and are all represented in $T(G^{\ominus})$, since all edges that now (after the reconnection according to $(W, V \setminus W)$) are incident to w (apart from $\{w, z\}$) represent minimum separating cuts in $G^{\ominus}$ with respect to their incident vertices. Note that for the edges that do not represent a cut in $\mathcal{O}$ this holds by Lemma 8.7, since $(U, V \setminus U)$ is still a minimum separating cut in $G^{\ominus}$. Hence, all edges incident to w are no more reconnected before they become thick edges in $T(G^{\ominus})$. From this observation it follows directly that different sets $\mathcal{O}$ and $\mathcal{O}'$ regarding different cuts θ and θ' (considered at different times) are disjoint, since thick edge are never chosen by a set $\mathcal{O}'$. Hence, $\mathcal{O}$ satisfies (ii).

Step 4: Proof of (i). It remains to show (i), that is, each cut in $\mathcal{O}$ shares each of its cut pairs in $\widetilde{Q}$ with at least one cut in $\mathcal{O} \cup \{\theta\}$. To this end, consider cut $(R, V \setminus R)$ (associated with the cut pair $\{x, y\}$ of $(U, V \setminus U)$ in $G^{\ominus}$, $u \in U$ and $y \in R$) in $\mathcal{O}$ and let $\{h, \ell\}$, with $h \in R$, denote an arbitrary cut pair of $(R, V \setminus R)$ in $G^{\ominus}$. If $\ell \in U$, $(R, V \setminus R)$ shares its cut pair with $\theta = (U, V \setminus U)$. If $\ell \notin U$, by the same arguments as before, ℓ must be in $A \setminus (\pi(b, d) \cup R)$ (see Fig. 9.1). In this case, let $\{s, z\}$ denote the edge incident to the center z such that the subtree rooted at s contains ℓ. Observe that $\{s, z\}$ is crossed by $(W, V \setminus W)$, since otherwise ℓ would have been in B. Moreover, we observe that the cut $(S, V \setminus S)$ induced by $\{s, z\}$ separates h and ℓ. In the following we show that $(S, V \setminus S)$ is in $\mathcal{O}$ and is a minimum h-ℓ-cut. This basically follows from the symmetry incorporated in $(U, V \setminus V)$ and $(R, V \setminus R)$, since $(W, V \setminus W)$ is also a minimum r-z-cut in $G^{\ominus}$. This is because if $\lambda_{G^{\ominus}}(r, z)$ was smaller than $c^{\ominus}(W, V \setminus W) \leq \lambda_G(r, z)$, the corresponding r-z-cut would not separate z and w, but x and y due to reshaping, which is a contradiction to the assumption that $(U, V \setminus U)$ is a minimum x-y-cut in $G^{\ominus}$ (since $c^{\ominus}(U, V \setminus U) = \lambda_G(r, z)$, as we know from above). Hence, $(R, V \setminus R)$ gets the role of $(U, V \setminus U)$ and $(S, V \setminus S)$ gets the role of $(R, V \setminus R)$. Then by the same arguments as before it follows that $(S, V \setminus S)$ also costs $c^{\ominus}(U, V \setminus U)$ and is a minimum h-ℓ-cut.

In order to see that $(S, V \setminus S)$ is also in $\mathcal{O}$ we observe the following. The cut $(S, V \setminus S)$ is also a minimum ℓ-x-cut, since any cheaper ℓ-x-cut would neither separate x and y nor h and ℓ. But h and y are both in R, and thus, due to the Non-Crossing Lemma (7.2), there exists a minimum ℓ-x-cut that either separates x and y or h and ℓ. By implication, this means that $\{\ell, x\}$ is also a cut pair of $\theta = (U, V \setminus U)$ in $G^{\ominus}$, since $c^{\ominus}(S, V \setminus S) = c^{\ominus}(U, V \setminus U)$. Hence, $\{\ell, x\} \in Q$ and $(S, V \setminus S) \in \mathcal{O}$. This is, the cut $(R, V \setminus R)$ shares its cut pair $\{h, \ell\}$ with another cut in $\mathcal{O}$, which finally proves (i).

9.2 Optimal Temporal Smoothness in Case of Edge Insertion or Cost Increase

Since the very simple and obvious update procedure for the case of edge insertion or cost increase presented in Section 8.2 provides a large number of degrees of freedom in choosing step pairs and split cuts, this procedure is not about to guarantee any smoothness property. Hence, we first modify this update procedure such that it chooses designated split cuts with respect to designated step pairs, which finally allows to prove optimality in terms of temporal smoothness. In order to point out where exactly the modification steps in, we briefly review the main steps of the update procedure as described in Section 8.2.

The procedure starts with checking whether $\{b,d\}$ is a (maybe newly inserted) bridge in G. If the answer is yes, it adapts $c_T(b,d)$ according to Corollary 8.2 if $\{b,d\}$ already exists in G, and rebuilds $T(G)$, without computing further cuts, according to Lemma 8.4, otherwise. In the first case, $T(G^\oplus)$ obviously contains exactly the same cuts as $T(G)$, and thus, guarantees optimal temporal smoothness. In the second case, the cuts represented by $T(G^\oplus)$ equal those represented by $T(G)$ in the sense that they induce the same cuts in the connected components of G, which also guarantees optimal temporal smoothness. If $\{b,d\}$ is no bridge, the update procedure constructs the intermediate tree shown in Figure 8.1(a), reusing all edges that are not on $\pi(b,d)$. These edges are thus fat edges in the intermediate tree. Additionally, it chooses one edge on $\pi(b,d)$ that represents a minimum b-d-cut in $G^\oplus$ and draws this edge also fat, according to Corollary 8.2. Finally, it processes the resulting compound nodes by applying the procedure Cut Tree, which is correct since the fat edges represent a partial Gomory-Hu set. In this section, however, we skip the choice of this last edge and directly apply Cut Tree to the intermediate tree shown in Figure 8.1(a). In this tree the thin edges on $\pi(b,d)$ induce the only compound node of vertices that still need to be separated. We denote this compound node by S_π.

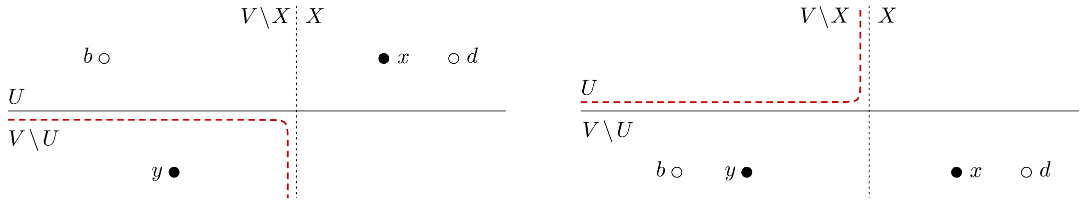
Since Cut Tree admits an arbitrary sequence of step pairs and the use of arbitrarily shaped split cuts, processing S_π by an arbitrary run of Cut Tree, however, does not guarantee any temporal smoothness. In the following we thus aim at controlling the choice of the step pairs and the shape of the split cuts during the run of Cut Tree, in order to achieve optimal temporal smoothness. More precisely, we aim at finding split cuts that cross $\pi(b,d)$ as few times as possible, thus preserving as many thin edges as possible for a potential reuse during the construction of a new Gomory-Hu tree for $G^\oplus$.

For the modified update procedure we will then show that it finds a maximum set of reusable cuts on $\pi(b,d)$ in $T(G)$, that is, no other Gomory-Hu tree of $G^\oplus$ contains more old cuts from $\pi(b,d)$ than the tree constructed by our procedure. Since a Gomory-Hu tree constructed by our procedure additionally contains all old cuts that are not on $\pi(b,d)$ in $T(G)$, it directly follows that no other Gomory-Hu tree contains more old cuts in total, which finally proves optimality in terms of temporal smoothness. The next two sections, however, first focus on finding appropriate split cuts and modifying the update procedure accordingly.

9.2.1 Reshaping Split Cuts

In a first step we describe how potential split cuts can be reshaped according to the aim proclaimed above. To this end, we introduce Lemma 9.4, which, similar to Lemma 8.9, allows to bend cuts in $G^\oplus$ along old minimum separating cuts in G without becoming more expensive. The situation of Lemma 9.4 is shown in Fig. 9.2.

Lemma 9.4. *Let* $(X, V \setminus X)$ *denote a minimum* x-y-*cut in* G *with* $x \in X$ *and* $y \in V \setminus X$ *that also separates* b *and* d. *Let* $(U, V \setminus U)$ *denote a further cut. If (i)* $(U, V \setminus U)$ *separates* x *and* y *with* $x \in U$ *and either* b *or* d *in* $U \cap X$, *then* $c^\oplus(U \cup X, V \setminus (U \cup X)) \leq c^\oplus(U, V \setminus U)$.



(a) Deflected by x, Lemma 9.4(i) bends $(U, V \setminus U)$ downwards along X.

(b) Deflected by x, Lemma 9.4(ii) bends $(U, V \setminus U)$ upwards along X.

FIGURE 9.2: Situation of Lemma 9.4. Reshaping cuts in $G^{\oplus}$ (solid black lines) along previous cuts in G (dotted black lines) resulting in new cuts in $G^{\oplus}$ (dashed red lines). Since we will apply Lemma 9.4 to cuts $(U, V \setminus U)$ that do not separate b and d, without loss of generality, b is depicted in $U \setminus X$ and $(V \setminus U) \setminus X$, respectively, in this figure.

If (ii) $(U, V \setminus U)$ *does not separate* x *and* y *with* $x \in V \setminus U$ *and either* b *or* d *in* $(V \setminus U) \cap X$, *then* $c^{\oplus}(U \setminus X, V \setminus (U \setminus X)) \leq c^{\oplus}(U, V \setminus U)$.

We remark that in this section we will only need case (i). However, in Part III, where we consider dynamic unrestricted cut clusterings, we will also make use of (ii).

Proof. The Proof of Lemma 9.4 is based on the same idea as the proof of Lemma 8.9. Using the fact that $(X, V \setminus X)$ is an old minimum x-y-cut in G, we prove Lemma 9.4(i) by contradiction. The idea is to show that $(U \cap X, V \setminus (U \cap X))$ would have been cheaper than the minimum x-y-cut $(X, V \setminus X)$ in G if $c^{\oplus}(U, V \setminus U)$ was cheaper than $c^{\oplus}(U \cup X, V \setminus (U \cup X))$ in $G^{\oplus}$. Since $(U \cap X, V \setminus (U \cap X))$ and $(X, V \setminus X)$ both separate b and d, it holds that $c(U \cap X, V \setminus (U \cap X)) = c^{\oplus}(U \cap X, V \setminus (U \cap X)) - \Delta$ and $c(X, V \setminus X) = c^{\oplus}(X, V \setminus X) - \Delta$. Thus, for the contradiction, it also suffices to show that $c^{\oplus}(U \cap X, V \setminus (U \cap X))$ would have been cheaper than $c^{\oplus}(X, V \setminus X)$. We express the costs of $(U \cap X, V \setminus (U \cap X))$ and $(X, V \setminus X)$ in $G^{\oplus}$ with the help of $(U, V \setminus U)$ and $(U \cup X, V \setminus (U \cup X))$ considered in Lemma 9.4(i). In doing so, we get

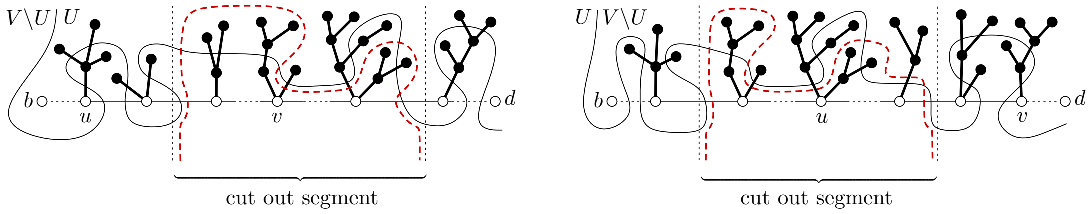
$$\begin{array}{llcll}
\textit{(i)} & c^{\oplus}(U \cap X, V \setminus (U \cap X)) & = & c^{\oplus}(U, V \setminus U) & \\
 & & \text{-} & c^{\oplus}(U \setminus X, V \setminus U) & + \; c^{\oplus}(U \setminus X, U \cap X) \\
\textit{(ii)} & c^{\oplus}(X, V \setminus X) & = & c^{\oplus}(U \cup X, V \setminus (U \cup X)) & \\
 & & \text{-} & c^{\oplus}(U \setminus X, V \setminus (U \cup X)) & + \; c^{\oplus}(U \setminus X, X)
\end{array}$$

Since $V \setminus (U \cup X) \subseteq V \setminus U$, it is $c^{\oplus}(U \setminus X, V \setminus (U \cup X)) \leq c^{\oplus}(U \setminus X, V \setminus U)$. From $U \cap X \subseteq X$ further follows that $c^{\oplus}(U \setminus X, U \cap X) \leq c^{\oplus}(U \setminus X, X)$; together with the assumption that $c^{\oplus}(U, V \setminus U) < c^{\oplus}(U \cup X, V \setminus (U \cup X))$, by subtracting *(ii)* from *(i)*, we get:

$$\begin{aligned}
c^{\oplus}(U \cap X, V \setminus (U \cap X)) - c^{\oplus}(X, V \setminus X) \;&=\; [c^{\oplus}(U, V \setminus U) - c^{\oplus}(U \cup X, V \setminus (U \cup X))] \\
&-\; [c^{\oplus}(U \setminus X, V \setminus U) - c^{\oplus}(U \setminus X, V \setminus (U \cup X))] \\
&+\; [c^{\oplus}(U \setminus X, U \cap X) - c^{\oplus}(U \setminus X, X])] < 0
\end{aligned}$$

This contradicts the fact that $(X, V \setminus X)$ is a minimum x-y-cut in G.

We prove Lemma 9.4(ii) with the help of the same technique. We show that $(X \setminus U, V \setminus (X \setminus U))$ would have been cheaper than the minimum x-y-cut $(X, V \setminus X)$ in G if $c^{\oplus}(U, V \setminus U)$ was cheaper than $c^{\oplus}(U \setminus X, V \setminus (U \setminus X))$ in $G^{\oplus}$. Since $(X \setminus U, V \setminus (X \setminus U))$ and $(X, V \setminus X)$ both separate b and d, it holds $c(X \setminus U, V \setminus (X \setminus U)) = c^{\oplus}(X \setminus U, V \setminus (X \setminus U)) - \Delta$ and $c(X, V \setminus X) = c^{\oplus}(X, V \setminus X) - \Delta$. Thus, for the contradiction, it also suffices to show that $c^{\oplus}(X \setminus U, V \setminus (X \setminus U))$ would have been



(a) If $\{u, b, d\} \subseteq U$, cut out segment contains v. (b) If $\{v, b, d\} \subseteq V \setminus U$, cut out segment contains u.

FIGURE 9.3: Reshaping cuts in $G^{\oplus}$ (solid lines) according to Lemma 9.4 and the Non-Crossing Lemma (7.2). Resulting cuts (dashed lines) cross $\pi(b, d)$ exactly twice and respect the subtrees.

cheaper than $c^{\oplus}(X, V \setminus X)$. We express the costs of $(X \setminus U, V \setminus (X \setminus U))$ and $(X, V \setminus X)$ with the help of $(U, V \setminus U)$ and $(U \setminus X, V \setminus (U \setminus X))$ considered in Lemma 9.4(ii). In doing so, we get

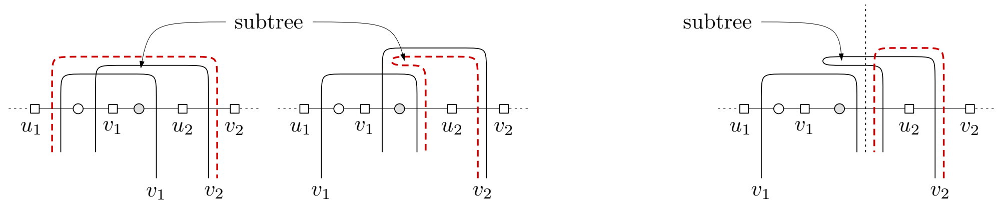
$$\begin{array}{llcll} \textit{(i)} & c^{\oplus}(X \setminus U, V \setminus (X \setminus U)) & = & c^{\oplus}(U, V \setminus U) & \\ & & - & c^{\oplus}(U, V \setminus (X \cup U)) & + \; c^{\oplus}(X \setminus U, V \setminus (X \cup U)) \\ \textit{(ii)} & c^{\oplus}(X, V \setminus X) & = & c^{\oplus}(U \setminus X, V \setminus (U \setminus X)) & \\ & & - & c^{\oplus}(U \setminus X, V \setminus (X \cup U)) & + \; c^{\oplus}(X, V \setminus (X \cup U)) \end{array}$$

Since $U \setminus X \subseteq U$, it is $c^{\oplus}(U \setminus X, V \setminus (X \cup U)) \leq c^{\oplus}(U, V \setminus (X \cup U))$. From $X \setminus U \subseteq X$ further follows that $c^{\oplus}(X \setminus U, V \setminus (X \cup U)) \leq c^{\oplus}(X, V \setminus (X \cup U))$; together with the assumption that $c^{\oplus}(U, V \setminus U) < c^{\oplus}(U \setminus X, V \setminus (U \setminus X))$, by subtracting *(ii)* from *(i)*, we get:

$$\begin{aligned} c^{\oplus}(X \setminus U, V \setminus (X \setminus U)) \; - \; & c^{\oplus}(X, V \setminus X) \\ = \; & [c^{\oplus}(U, V \setminus U) - c^{\oplus}(U \setminus X, V \setminus (U \setminus X))] \\ - \; & [c^{\oplus}(U, V \setminus (X \cup U)) - c^{\oplus}(U \setminus X, V \setminus (X \cup U))] \\ + \; & [c^{\oplus}(X \setminus U, V \setminus (X \cup U)) - c^{\oplus}(X, V \setminus (X \cup U))] < 0 \end{aligned}$$

This contradicts the fact that $(X, V \setminus X)$ is a minimum x-y-cut in G. □

Together with the Non-Crossing Lemma (7.2), Lemma 9.4 provides the key to reshape split cuts in $G^{\oplus}$ in an appropriate way. Consider the following situation in the intermediate tree shown in Figure 8.1(a). As we will refer to this special tree many further times, we denote this fixed tree structure by $\hat{T}$. Let $\{u, v\}$ denote a step pair of vertices on $\pi(b, d)$ in $\hat{T}$, and let $(U, V \setminus U)$ denote a minimum u-v-cut in $G^{\oplus}$ with $u \in U$. Assume further that $\lambda_{G^{\oplus}}(u, v) < \lambda_G(u, v) + \Delta$, that is, old minimum u-v-cuts that separate b and d (and in particular those induced by edges on $\pi(b, d)$) are no minimum u-v-cuts in $G^{\oplus}$ anymore. Consequently, $(U, V \setminus U)$ does not separate b and d, and thus, crosses $\pi(b, d)$ at least twice, decomposing $\pi(b, d)$ into several subpaths or segments. For fixed vertices u and v, we now distinguish two cases; first, $\{u, b, d\} \subseteq U$; second, $\{v, b, d\} \subseteq V \setminus U$. In the first case, we focus on the segment of $\pi(b, d)$ that is cut out by $(U, V \setminus U)$ and contains v (see Fig. 9.3(a)). At the beginning and the end of this segment $(U, V \setminus U)$ crosses an edge on $\pi(b, d)$, which represents a minimum separating cut with respect to its incident vertices in G, each (see dotted vertical lines in Fig. 9.3(a)). According to Lemma 9.4, $(U, V \setminus U)$ can thus be bent along both of these cuts (in the role of $(X, V \setminus X)$) such that the resulting cut (see dashed line in Fig. 9.3(a)) crosses $\pi(b, d)$ exactly twice. In the second case, we consider the segment of $\pi(b, d)$ that is cut out by $(U, V \setminus U)$ and contains u (see Fig. 9.3(b)). Here $(U, V \setminus U)$ can be bent analogously along the old cuts bounding the segment.



(a) Second arch is stretched (left), and clinched (right). The latter yields an overhanging subtree.

(b) Clinched arch can be truncated by Lemma 9.4, removing the overhang.

Figure 9.4: Exemplary sequence $\{u_1, v_1\}, \{u_2, v_2\}$ of step pairs and corresponding arch-shaped split cuts (solid lines), possibly reshaped (dashed lines). Vertices u_i, v_i of the step pairs are depicted as squares, corresponding split cuts are also labeled by v_i, the base vertex of the indicated subtree is filled gray.

Finally, we observe that the subtrees that are connected to $\pi(b, d)$ by fat edges are sheltered by these edges such that $(U, V \setminus U)$ can be additionally reshaped by applying the Non-Crossing Lemma (7.2) resulting in a cut that respects the subtrees, that is, each subtree is completely assigned to one cut side, as shown in Fig. 9.3. As we will refer to these special subtrees in $\hat{T}$ many further times, we call them *fat subtrees* and the (fat) edges connecting them to $\pi(b, d)$ *linking edges*. A linking edge of a fat subtree is incident to a *base vertex* on $\pi(b, d)$ and the *root* of the fat subtree and shelters the fat subtree against any minimum separating cut that has a cut pair on $\pi(b, d)$. In the following we imply the reshaping steps described so far without further notice, and depict split cuts that do not separate b and d as arch-shaped curves that are open at the bottom. We call the total reshaping so far the *arch-reshaping*. Assuming further that $\pi(b, d)$ is horizontally embedded with a fixed order of the vertices from left to right, we talk about a *right arch* if the cut out segment is to the right of the cut vertex that shares a cut side with b and d, as in Fig. 9.3(a), and a *left arch* if the cut out segment is to the left of the cut vertex that shares a cut side with b and d, as in Fig. 9.3(b).

Applying again the Non-Crossing Lemma (7.2), crossing arches can further be stretched or clinched such that they become either disjoint or nested. Such an adjustment, however, may blur the arch-shaped appearance of the cuts, as shown on the right part of Fig. 9.4(a). Consider two crossing arches and a fat subtree whose linking edge is crossed by only one of these arches, while the second arch separates the fat subtree from b and d. Then, depending on whether the second arch is stretched or clinched, it happens that the fat subtree causes an overhang, since the second arch spans the subtree but not its base vertex. We call such a fat subtree an *overhanging subtree*. However, the resulting cut is of the same form as the cut $(U, V \setminus U)$ in Fig. 9.3, and thus, can be again truncated by applying Lemma 9.4, removing the overhang. In this way we get again a properly arch-shaped split cut (see Fig 9.4(b)).

9.2.2 Modifying the Update Procedure

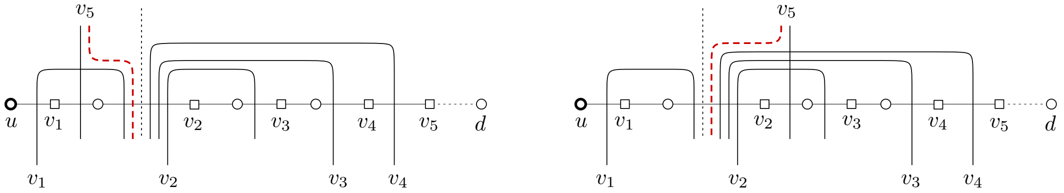
In order to guarantee temporal smoothness, we now modify the update procedure—more precisely, the subroutine Cut Tree. To this end we proceed as follows. We consider the intermediate tree $\hat{T}$ and its compound node S_π induced by the thin edges on $\pi(b, d)$. We note that splitting S_π according to a split cut would change the intermediate tree such that the old cuts formally represented by edges on $\pi(b, d)$ loose their easily recognizable form making it more difficult to see how new cuts can be reshaped. Hence, for the sake of simplicity, in a first phase, we will defer the splitting of S_π choosing the step pairs and reshaping the newly found cuts with respect to the fixed tree $\hat{T}$. The aim of this first phase is to find a set of split cuts in S that contains as many reusable cuts as possible. In a second phase, we split S according to the found cuts and finally process the resulting compound nodes by the help of the procedure Cut Tree. The latter does not affect the temporal smoothness, since we will see that a maximum set of reusable cuts is already found in the first phase. However, dividing the Gomory-Hu tree construction into a cut computing phase and a splitting phase, instead of splitting the current compound node immediately after each cut computation, requires the storing of the found cuts and is not very convenient. Hence, in Section 9.2.4, we present a pseudo-code that, in the spirit of Gomory and Hu, immediately splits the current compound node in each step, thus avoiding that the found split cuts need to be stored until the splitting phase. In the remainder of this section, however, we stick with the two phase approach, focusing on the description of the first phase and the proof showing that in this phase already a maximum set of reusable cuts is found.

In the first phase, we seek a set O of old cuts on $\pi(b,d)$ that can be reused as split cuts for the construction of $T(G^{\oplus})$. Following the notion of arches that represent newly found split cuts, reusable split cuts on $\pi(b,d)$ can be seen as vertical pillars. Arches that are possibly found during the search for O are stored in a set A for later use. The following operations are repeated until all vertices in S_{π} are separated from d by cuts in $A \cup O$. The vertices not yet separated from d are called *free*. At the beginning, S_{π} consists of only free vertices.

In each step we do the following. We assume that d is the rightmost vertex on $\pi(b,d)$ and consider the free vertices in S_{π} according to the order induced by their positions on $\pi(b,d)$. We choose the free vertex u furthest from d and the free vertex v closest to u as step pair and compute a minimum u-v-cut $(U, V \setminus U)$ in $G^{\oplus}$.

If $\lambda_{G^{\oplus}}(u,v) < \lambda_G(u,v) + \Delta$, then $(U, V \setminus U)$ is a new split cut in A, which does not separate b and d, and we apply the arch-reshaping (recall Fig. 9.3) such that $(U, V \setminus U)$ becomes a right or left arch depending on whether u or v shares a cut side with b and d. If the resulting arch crosses a previously found cut in $O \cup A$, it is further adjusted (clinched or stretched) according to the Non-Crossing Lemma (7.2) and Lemma 9.4, removing possible overhangs. The vertices cut out from $\pi(b,d)$ by the resulting arch are now separated from d, and hence, no longer free. Thus, they are not considered when choosing the next step pair. We observe that, as long as only right arches are found in consecutive steps, u remains free and the same vertex in each step pair (see Fig. 9.6(a)). In contrast, as soon as a left arch is found, u is immediately separated from d, while v takes the role of u in the next step, as it becomes the free vertex furthest from d (see Fig. 9.6(b)).

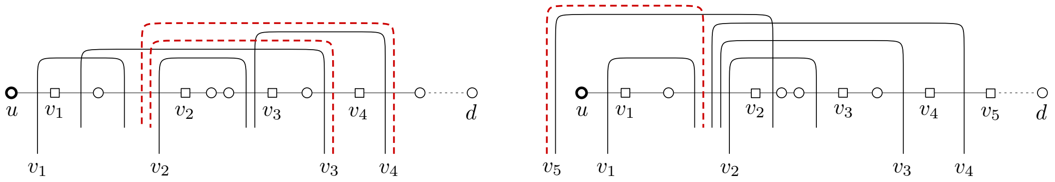
If $\lambda_{G^{\oplus}}(u,v) \geq \lambda_G(u,v) + \Delta$, the old minimum u-v-cuts on $\pi(b,d)$ remain valid and we chose a corresponding vertical pillar. If this pillar does not cross any previously found cut in $O \cup A$ we add it to O. Otherwise, it crosses a (non-nested) arch. However, we can show that in this case there exists another pillar of the same cost that does not cross the cuts in $O \cup A$, and thus can be added to O instead. Suppose the chosen pillar crosses a (non-nested) arch. Then, it can be bent along this arch, according to the Non-Crossing Lemma (7.2), resulting in a cut of the same cost that does not cross any cut in $O \cup A$, still separates b and d, and crosses an edge e on $\pi(b,d)$ that is also crossed by the non-nested arch (see Fig. 9.5). The pillar induced by e thus also crosses no cut in $O \cup A$, but separates u and v since neither u nor v is spanned by an arch. Hence, e costs at least $\lambda_{G^{\oplus}}(u,v)$ in $G^{\oplus}$. On the other hand, the reshaped cut separates the two vertices incident to e, which are a cut pair of e in G. Since both, the reshaped cut and e, separate b and d, e costs at most $\lambda_G(u,v)$ in G and $\lambda_G(u,v) + \Delta = \lambda_{G^{\oplus}(u,v)}$ in $G^{\oplus}$. Thus, e is



(a) Deflected by v_1, the minimum u-v_5-cut is bent to the right, according to the Non-Crossing Lemma (7.2). The dotted pillar finally separates u from d, v_5 becomes free vertex furthest from d in next step.

(b) Deflected by v_4, the minimum u-v_5-cut is bent to the left, according to the Non-Crossing Lemma (7.2). The dotted pillar finally separates u from d, v_5 becomes free vertex furthest from d in next step.

Figure 9.5: Exemplary sequence of step pairs $\{u, v_1\} \dots \{u, v_5\}$ and corresponding split cuts (solid lines), where the last split cut is an old cut on $\pi(b, d)$, adjusted (dashed line) such that it does not cross any arches. The vertical pillar finally added to O is drawn as dotted line. Vertices v_j are depicted as squares, corresponding split cuts are also labeled v_j.



(a) Sequence of right arches. Minimum u-v_3-cut and minimum u-v_4-cut adjusted according to the Non-Crossing Lemma (7.2) and Lemma 9.4.

(b) Minimum u-v_5-cut is a left arch, adjusted according to the Non-Crossing Lemma (7.2) and Lemma 9.4. It separates u from d, v_5 becomes free vertex furthest from d.

Figure 9.6: Exemplary sequence of step pairs $\{u, v_1\} \dots \{u, v_5\}$ and corresponding arch-shaped split cuts (solid lines), possibly adjusted (dashed lines). Vertices v_j are depicted as squares, corresponding split cuts are also labeled v_j.

a minimum u-v-cut in $G^{\oplus}$, and we finally add it to O. In doing so, u is separated from d and is thus no longer free, v becomes the free vertex furthest from d, and thus, takes the role of u in the next step.

Again summarizing, we remark that u is separated from d by either a vertical pillar or a left arch, and vice versa, as long as u is not separated from d only right arches are found. A right arch thus never crosses a previously found vertical pillar since such a pillar would have already separated u and v. At the beginning, b is chosen as the first vertex having the role of u. In this special situation, u can only be separated from d by a pillar since arches never separate b and d. Furthermore we observe that, whenever u is separated from d, the free vertices in the next step form a subpath of $\pi(b, d)$, and thus, can be iteratively handled in the way described above, until no free vertices (apart from d) remain.

9.2.3 Proving Optimality

In order to prove optimality in terms of temporal smoothness, we show that the set O found by our modified update routine is a maximum set of reusable cuts on $\pi(b, d)$, that is, there exists no Gomory-Hu tree of $G^{\oplus}$ that contains more old cuts from $\pi(b, d)$ in $T(G)$ than the tree constructed by our update routine. This key assertion follows from the next theorem. Since a Gomory-Hu tree constructed by our update routine additionally contains all old cuts that are not on $\pi(b, d)$ in $T(G)$, it directly follows that no other Gomory-Hu tree contains more reusable cuts in total, thus proving optimal temporal smoothness.

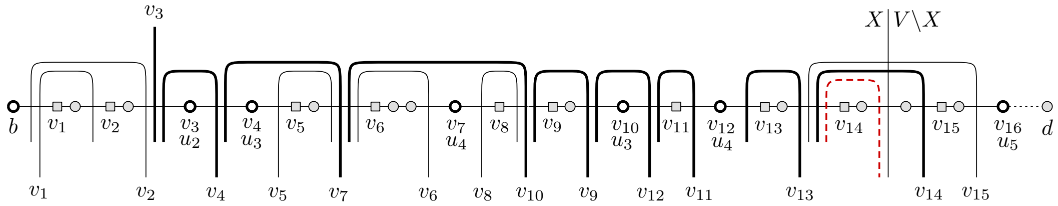


FIGURE 9.7: Exemplary sequence of step pairs $\{u_i, v_j\}$ and corresponding split cuts (solid lines). Vertices v_j taking the role of u are bold-framed, remaining vertices v_j are depicted as squares, corresponding split cuts are also labeled v_j. Step pair $\{u_4, v_{16} = u_5\}$ would be next in the sequence resulting in a split cut (not shown in this figure) that separates u_4 from d. Reusable cut $(X, V \setminus X)$ is on $\pi(u_i, u_{i+1})$ for $i = 4$. Besides v_{15}, all split cuts have their step pairs in $X \cap \pi(b, d)$. Split cuts drawn as fat lines already suffice to isolate $u_i = u_4$ in a compound node of a notional run of CUT TREE. Note that v_{14} can be bent along $(X, V \setminus X)$ (dashed line).

Theorem 9.5. *Let u_i and u_{i+1} denote two vertices in $\pi(b, d)$ such that u_{i+1} is the next vertex after u_i that takes the role of u during the modified update routine. Let $\pi(u_i, u_{i+1})$ denote the subpath of $\pi(b, d)$ induces by these vertices, and let Q denote the set of old cuts on $\pi(u_i, u_{i+1})$ that are also minimum separating cuts in $G^{\oplus}$ with respect to any cut pair. If $Q \neq \emptyset$, Q consists of minimum u_i-u_{i+1}-cuts, each of which share all its cut pairs with all other cuts in Q.*

Theorem 9.5 partitions the old cuts in $\pi(b, d)$ into (disjoint) subsets of cuts (given by the subpaths $\pi(u_i, u_{i+1})$) such that no two cuts of the same subset can be realized at the same time in a Gomory-Hu tree, as all reusable cuts in each subset have the same cut pairs. Vice versa, for each step pair $\{u_i, u_{i+1}\}$ the update routine saves a reusable cut (that is, a vertical pillar) in O if there exists a reusable minimum u_i-u_{i+1}-cut. Hence, O contains a maximum number of reusable cuts on $\pi(b, d)$.

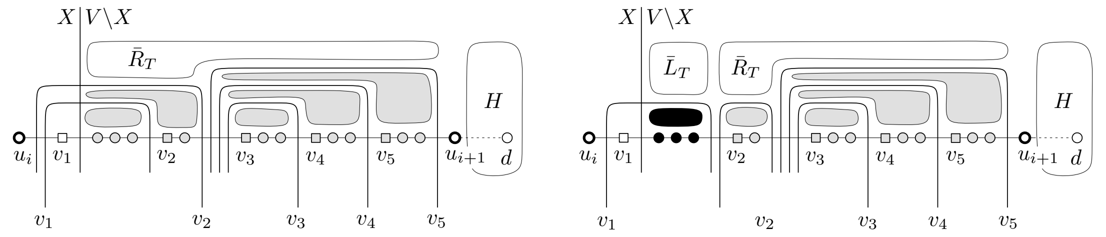
Proof of Theorem 9.5. Let $(X, V \setminus X)$ with $u_i \in X$ denote a cut in Q and $\{x, y\}$ with $x \in X$ and $y \in V \setminus X$ a cut pair of $(X, V \setminus X)$ in $G^{\oplus}$. In order to show Theorem 9.5, we will examine which vertices in X and $V \setminus X$ are candidates for x and y. We start with the candidates for x.

Claim (1): If $\{x, y\}$ is a cut pair of $(X, V \setminus X) \in Q$, then $\{u_i, y\}$ is also a cut pair of $(X, V \setminus X)$.

Proof. We show this claim by applying a notional run of CUT TREE to find further cut pairs, which result from the correctness of CUT TREE established by Lemma 7.3. We will also use this technique in the context of other claims in this proof.

We apply the notional run of CUT TREE starting at the intermediate tree resulting from V by splitting V according to $(X, V \setminus X)$ and $\{x, y\}$, which results in the compound nodes X and $V \setminus X$. In the following we argue that we can further split X until it only consists of u_i and is connected to $V \setminus X \ni y$ by a fat edge that represents $(X, V \setminus X)$. Then, due to Lemma 7.3, $\{u_i, y\}$ is also a cut pair of $(X, V \setminus X)$.

First we consider the fat subtrees of $\hat{T}$ contained in X. Splitting X according to the linking edges of these subtrees results in a compound node $X \cap \pi(b, d)$ that still contains u_i and one further compound node per subtree. Now we consider the cuts in $O \cup A$ (found during the modified update routine) that result from step pairs in $X \cap \pi(b, d)$. Note that none of these cuts separates u_i from d since the first cut that separates u_i from d appears with respect to the step pair $\{u_i, u_{i+1}\} \not\subseteq X$. Due to the choice of the step pairs and the reshaping of the cuts in $O \cup A$, these cuts separate at least the vertices in $(X \cap \pi(b, d)) \setminus \{u_i\}$ from u_i (and d). Figure 9.7



(a) Sequence of right arches where $V \setminus X$ only consists of $R \cup R_T \cup \bar{R}_T \cup H \cup \{u_{i+1}\}$.

(b) Sequence of right arches where $V \setminus X$ also contains vertices in $L \cup L_T \cup \bar{L}_T$.

FIGURE 9.8: Exemplary sequence of step pairs $\{u_i, v_1\} \ldots \{u_i, v_5\}$ and corresponding split cuts. Vertices v_j are depicted as squares, corresponding split cuts are also labeled v_j. Step pair $\{u_i, v_6 = u_{i+1}\}$ would be next in the sequence resulting in a split cut (not shown in this figure) that separates u_i from d. Reusable cut $(X, V \setminus X)$ induces sets $R, L, R_T, L_T, \bar{R}_T, \bar{L}_T, H$. Vertices in R are filled gray, vertices in L are filled black. Subtrees in R_T are indicated by gray areas, those in L_T by black areas. The remaining sets $\bar{L}_T$, $\bar{R}_T$ and H are labeled accordingly.

exemplarily shows the situation. Since $(X, V \setminus X)$ shelters $V \setminus X \ni y$, these cuts can be further bent along $(X, V \setminus X)$ such that they exactly separate the vertices in $(X \cap \pi(b, d)) \setminus \{u_i\}$ from u_i (and also from y). Hence, splitting $X \cap \pi(b, d)$ according to these cuts (in the order given by the step pairs) is feasible and yields a final compound node that only contains u_i and is linked (by a fat edge that represents $(X, V \setminus X)$) to the compound node $V \setminus X$ containing y. This finishes the proof of claim (1). □

Regarding the possible candidates for y, we decompose $V \setminus X$ into several disjoint subsets. To this end, we observe the following. At the time $\{u_i, u_{i+1}\}$ becomes the step pair in the update routine, the vertices in $((V \setminus X) \cap \pi(u_i, u_{i+1})) \setminus \{u_{i+1}\}$ are all separated from d (and u_i and u_{i+1}) by right arches (see Fig. 9.8). We denote the set of vertices in $((V \setminus X) \cap \pi(u_i, u_{i+1})) \setminus \{u_{i+1}\}$ that are separated from d by an arch whose step pair $\{u_i, v_j\}$ is separated by $(X, V \setminus X)$, that is, $v_j \in V \setminus X$, by R. The set of the remaining vertices in $((V \setminus X) \cap \pi(u_i, u_{i+1})) \setminus \{u_{i+1}\}$ is denoted by L. A vertex in L is thus separated from d only by arches whose step pairs are not separated by $(X, V \setminus X)$. Note that such an arch is necessarily crossed by $(X, V \setminus X)$, since v_j is in X and the vertex in L is in $V \setminus X$, whereas both v_j and the vertex in L are spanned by the arch. Furthermore, consider the fat subtrees that are linked to $((V \setminus X) \cap \pi(u_i, u_{i+1})) \setminus \{u_{i+1}\}$ and recall that these subtrees are completely assigned to either side of the cuts induced by the arches. Those subtrees that are separated from d (and u_i and u_{i+1}) by an arch whose step pair is separated by $(X, V \setminus X)$ contribute to the vertex set R_T. The vertices of the remaining subtrees separated from d form the set L_T. A vertex in L_T thus belongs to a fat subtree that is separated from d only by arches whose step pairs are not separated by $(X, V \setminus X)$. The subtrees that are not separated from d, but from their base vertex in $((V \setminus X) \cap \pi(u_i, u_{i+1})) \setminus \{u_{i+1}\}$ contribute to the vertex sets $\bar{R}_T$ and $\bar{L}_T$, respectively, depending on whether or not they are separated from their base vertex by an arch whose step pair is separated by $(X, V \setminus X)$. A vertex in $\bar{R}_T$ or $\bar{L}_T$ thus belongs to a fat subtree whose linking edge is crossed by at least one arch. The remaining vertices in $(V \setminus X) \setminus \{u_{i+1}\}$ are denoted by H. With this notation each vertex in $V \setminus X$ (that is, each candidate for y) is either in L, R, L_T, R_T, $\bar{L}_T$, $\bar{R}_T$, H or $\{u_{i+1}\}$. The following two claims now exclude some of these candidate sets for y.

Claim (2): There exists no cut pair $\{x, y\}$ *of* $(X, V \setminus X) \in Q$ *with* $y \in R \cup R_T \cup \bar{R}_T$.

Proof. We prove this claim by contradiction. Suppose a cut pair $\{x, y\}$ with $y \in R \cup R_T \cup \bar{R}_T$. Then, by definition, there exists an arch whose step pair $\{u_i, v_j\}$ is separated by $(X, V \setminus X)$, and that separates y from u_i if $y \in R \cup R_T$, and crosses the linking edge of the fat subtree containing y if $y \in \bar{R}_T$. Since $(X, V \setminus X)$ separates u_i and v_j, $c(X, V \setminus X) + \Delta$ must be more expensive than $\lambda_{G^\oplus}(u_i, v_j)$, otherwise such an arch would not have been added to A by the update routine. We now distinguish $y \in R \cup R_T$ and $y \in \bar{R}_T$, considering an arch as described above.

If $y \in R \cup R_T$, the arch separates u_i and y, and it is $\lambda_{G^\oplus}(u_i, y) \leq \lambda_{G^\oplus}(u_i, v_j) < c(X, V \setminus X) + \Delta$. On the other hand, $(X, V \setminus X)$ is a minimum u_i-y-cut in $G^\oplus$, according to claim (1). Hence, the above observation contradicts the assumption that $(X, V \setminus X)$ is a minimum x-y-cut with y in $R \cup R_T$.

If $y \in \bar{R}_T$, the linking edge of the subtree that contains y is crossed by the arch, and thus, costs at most $\lambda_{G^\oplus}(u_i, v_j) < c(X, V \setminus X) + \Delta$ in $G^\oplus$. Furthermore, the linking edge separates y from u_i. Since $(X, V \setminus X)$ is a minimum u_i-y-cut, according to claim (1), this is again a contradiction. ☐

Claim (3): There exists no cut pair $\{x, y\}$ of $(X, V \setminus X) \in Q$ wit $y \in L \cup L_T$.

Proof. We prove this claim again by contradiction. Suppose a cut pair $\{x, y\}$ with $y \in L \cup L_T$. Then, by definition, there exists an arch whose step pair $\{u_i, v_j\}$ is not separated by $(X, V \setminus X)$ and that separates y from u_i. As a consequence, this arch is crossed by $(X, V \setminus X)$. Since the step pair $\{u_i, v_j\}$ of the arch is in X, we can however apply the Non-Crossing Lemma (7.2) bending the arch upwards along $(X, V \setminus X)$ (deflected by y) resulting in a cut of the same cost that still separates u_i and v_j, but further separates b and d. This contradicts the fact that $\lambda_{G^\oplus}(u_i, v_j) < \lambda_G(u_i, v_j) + \Delta$, which ensured that the arch was added to A by the update routine. ☐

The next two claims show that the remaining candidates of y, which are either in H or in $\bar{L}_T$ always induce the cut pair $\{u_i, u_{i+1}\}$ for $(X, V \setminus X)$. *Claim (4): If $\{x, y\}$ with $y \in H$ is a cut pair of $(X, V \setminus X) \in Q$, then $(X, V \setminus X)$ is a minimum u_i-u_{i+1}-cut in $G^\oplus$.*

Proof. This claim is quite obvious. Suppose there is a cut pair $\{x, y\}$ with $y \in H$, but $(X, V \setminus X)$ is no minimum u_i-u_{i+1}-cut in $G^\oplus$. According to claim (1), $(X, V \setminus X)$ is a minimum u_i-y-cut in $G^\oplus$. On the other hand, the minimum u_i-u_{i+1}-cut found by the update routine will be either a left arch or a vertical pillar, and will separate u_i from d, and even more, u_i from y. Consequently, $(X, V \setminus X)$ costs at most $\lambda_{G^\oplus}(u_i, u_{i+1})$ in $G^\oplus$. Vice versa, $(X, V \setminus X)$ separates u_i and u_{i+1}, and we get $c(X, V \setminus X) + \Delta = \lambda_{G^\oplus}(u_i, u_{i+1})$, which contradicts the assumption that $(X, V \setminus X)$ is no minimum u_i-u_{i+1}-cut in $G^\oplus$. ☐

Claim (5): If $\{x, y\}$ with $y \in \bar{L}_T$ is a cut pair of $(X, V \setminus X) \in Q$, then $(X, V \setminus X)$ is a minimum u_i-u_{i+1}-cut in $G^\oplus$.

Proof. We prove this claim by applying the same technique as in the proof of claim (1), that is, we apply a notional run of CUT TREE in order to find a cut pair $\{x', y'\}$ of $(X, V \setminus X)$ with $y' \in H \cup \{u_{i+1}\}$. According to claim (1) and (4), $(X, V \setminus X)$ is then a minimum u_i-u_{i+1}-cut in $G^\oplus$.

In a first step, we split V according to $(X, V \setminus X)$ and $\{u_i, y\}$, which is a cut pair of $(X, V \setminus X)$, according to claim (1). This results in two compound nodes X and $V \setminus X$. In the following, we further split $V \setminus X$. To this end, we consider the fat subtrees in $V \setminus X$. Splitting $(V \setminus X)$

according to the linking edges of these subtrees results in the compound node $(V \setminus X) \cap \pi(b,d) = L \cup R \cup (H \cap \pi(b,d)) \cup \{u_{i+1}\}$ and several further compound nodes corresponding to the subtrees. Since one of these subtrees contains y (as $y \in \bar{L}_T$), the edge that represents $(X, V \setminus X)$ in the intermediate tree is finally incident to X and $L \cup R \cup (H \cap \pi(b,d)) \cup \{u_{i+1}\}$. Hence, according to Lemma 7.3, which establishes the correctness of the procedure Cut Tree, there exists a cut pair $\{x', y'\}$ of $(X, V \setminus X)$ with $y' \in L \cup R \cup (H \cap \pi(b,d)) \cup \{u_{i+1}\}$. As y' is not in $L \cup R$ (according to claim (2) and (3)), y' must be in $H \cup \{u_{i+1}\}$. □

From the claims so far it follows that each cut in Q is a minimum u_i-u_{i+1}-cut in $G^{\oplus}$, which proves the first assertion of Theorem 9.5. The final claim now states that at most one cut in Q can be realized in a Gomory-Hu tree for $G^{\oplus}$, since all cuts in Q share all their cut pairs with all other cuts in Q. This proves the second statement of Theorem 9.5, and thus, finishes the proof.

Claim (6): If $\{x, y\}$ is a cut pair of $(X, V \setminus X) \in Q$, it is also a cut pair of any further cut in Q.

Proof. Let $(U, V \setminus U)$, with $u_i \in U$, denote another cut in Q. According to claim (1)-(5), $(U, V \setminus U)$ is a minimum u_i-u_{i+1}-cut. We prove claim (6) by contradiction, applying again a notional run of Cut Tree. Without loss of generality, suppose the vertical pillar $(X, V \setminus X)$ is closer to u_i than the vertical pillar $(U, V \setminus U)$, and $\{x, y\}$ is a cut pair of $(X, V \setminus X)$ that is no cut pair of $(U, V \setminus U)$. That is, $\{x, y\} \subseteq U$. Hence, we can split V according to $(U, V \setminus U)$ and $\{u_i, u_{i+1}\}$ and then split U according to $(X, V \setminus X)$ and $\{x, y\}$. The edge that represents $(X, V \setminus X)$ in the intermediate tree is now incident to the compound nodes X and $(V \setminus X) \cap U$, which contains y. More precisely, y belongs to one of the fat subtrees in $(V \setminus X) \cap U$, since otherwise it would be in $L \cup R$ contradicting claim (2) and (3). However, in the following we argue that there exists another cut pair $\{x', y'\}$ of $(X, V \setminus X)$ with y' in $L \cup R$. This also contradicts claim (2) and (3), and thus, finishes the proof.

We split $(V \setminus X) \cap U$ according to the linking edges of the fat subtrees in $(V \setminus X) \cap U$. Since one of these subtrees contains y, the edge that represents $(X, V \setminus X)$ in the intermediate tree is finally incident to X and $L \cup R$. Hence, according to Lemma 7.3, which establishes the correctness of the procedure Cut Tree, there exists a cut pair $\{x', y'\}$ of $(X, V \setminus X)$ with $y' \in L \cup R$. □

9.2.4 A Simple Implementable Algorithm

For the description of the modified update routine in the previous section, we considered the split cuts with respect to the fixed intermediate tree $\hat{T}$, since this allows a better understanding of the applied reshaping steps. However, reshaping the split cuts with respect to a fixed intermediate tree in a first phase and storing the resulting cuts until the compound nodes are split in a second phase, is not much applicable. Instead it would be nice to have an algorithm that, following the procedure Cut Tree, immediately splits the current compound node with respect to the (possibly reshaped) split cut computed in the current step. In this section, we provide a simple implementable algorithm for the modified update routine (see Algorithm 6) that proceeds in this way, realizing the reshaping of the split cuts by reconnecting edges in the intermediate tree.

In Algorithm 6, we assume G and $G^{\oplus}$ are available as global variables. The connectivity values for vertex pairs in G are provided by $T(G)$. The first lines handle the cases that $\{b, d\}$ is a bridge in G and $G^{\oplus}$, respectively, and are just listed for the sake of completeness, since in these cases, Algorithm 6 implements exactly what is described in Section 8.2. The interesting part of Algorithm 6, namely the modification of the update routine, starts at line 4.

Algorithm 6: Increase or Insert

```
   Input: $T(G)$, $b, d$, $c(b,d)$, $c^{\oplus}(b,d)$, $\Delta := c^{\oplus}(b,d) - c(b,d)$
   Output: $T(G^{\oplus})$
1  $T_* \leftarrow T(G)$
2  if $\{b,d\}$ is a bridge in $G$ then apply Corollary 8.2; return $T(G^{\oplus}) \leftarrow T_*$
3  if $\{b,d\}$ is a bridge in $G^{\oplus}$ then apply Lemma 8.4; return $T(G^{\oplus}) \leftarrow T_*$
4  $T_* \leftarrow$ intermediate tree shown in Figure 8.1(a)
5  $\hat{P} \leftarrow$ subgraph of $\hat{T}$ consisting of $\pi(b,d)$ and the roots of the fat subtrees, edges on $\pi(b,d)$
   unmarked, roots of the fat subtrees colored red
6  $S_\pi \leftarrow$ vertices on $\pi(b,d)$                                        // free vertices
7  $i \leftarrow 0$; $j \leftarrow 0$; $u_i \leftarrow b$; $v_j \leftarrow$ neighbor of $b$ on $\pi(b,d)$;      // step pair
8  while $S_\pi \neq \{d\}$ do
                                  // compute split cut --------------------------------
9      $(U, V \setminus U) \leftarrow$ minimum $u_i$-$v_j$-cut in $G^{\oplus}$ with $u_i \in U$
10     if $\lambda_{G^{\oplus}}(u_i,v_j) < \lambda_G(u_i,v_j) + \Delta$ then                        // arch
11         if $\{u_i,b,d\} \subseteq U$ then $M \leftarrow V \setminus U$; $m \leftarrow v_j$ ;                // right arch
12         if $\{v_j,b,d\} \subseteq U$ then $M \leftarrow U$; $m \leftarrow u_i$ ;                     // left arch
13         $C_e, C_v \leftarrow$ edges, vertices on segment cut out from $\pi(b,d)$ containing $m$
14         $C_v \leftarrow C_v \cup \{$vertices $r$ in $\hat{P}$ incident to a vertex in $C_v$ with $r \in M\}$
15         if $\{u_i,b,d\} \subseteq U$ then markSpannedEdges($C_e$, $\hat{P}$, $i$, $j$)
16     if $\lambda_{G^{\oplus}}(u_i,v_j) \geq \lambda_G(u_i,v_j) + \Delta$ then                   // vertical pillar
17         $(U, V \setminus U) \leftarrow$ reusable cut ($u_i \in U$) induced by unmarked edge on $\pi(u_i,v_j)$ in $\hat{P}$
18         $C_v \leftarrow U$
                                  // reconnect edges --------------------------------
19     draw $\{u_i,v_j\}$ as a fat edge in $T_*$; $c_*(u_i,v_j) \leftarrow c^{\oplus}(U, V \setminus U)$
20     $F \leftarrow S_\pi \cap C_v$
21     $N \leftarrow$ all vertices that are linked to a vertex in $F$ by a fat edge in $T_*$
22     if $\{u_i,b,d\} \subseteq U$ then                                            // right arch
23         $\bar{N} \leftarrow$ all neighbors of $u_i$ in $T_*$
24         $T_* \leftarrow$ reconnect($C_v$, $T_*$, $N$, $\bar{N}$, $u_i$, $v_j$)
25         $S_\pi \leftarrow S_\pi \setminus C_v$; $v' \leftarrow$ vertex closest to $u_i$ on $\pi(b,d) \cap S_\pi$; reconnect $v'$ to $u_i$
26         $j \leftarrow j+1$; $v_j \leftarrow v'$                                    // next step pair
27     if $d \in V \setminus U$ then                               // left arch, vertical pillar
28         $T_* \leftarrow$ reconnect($C_v$, $T_*$, $N$, $\emptyset$, $v_j$, $u_i$)
29         $i \leftarrow i+1$; $u_i \leftarrow v_j$; $S_\pi \leftarrow S_\pi \setminus C_v$                          // next step pair
30         $j \leftarrow j+1$; $v_j \leftarrow$ vertex incident to $u_i$ on $\pi(b,d) \cap S_\pi$          // next step pair
31 apply Cut Tree to remaining compound nodes in $T_*$
32 return $T_*$
```

The modification uses a fixed substructure $\hat{P}$ of the tree $\hat{T}$, which consists of the path $\pi(b,d)$ together with the root vertices of the fat subtrees in $\hat{T}$. The edges on $\pi(b,d)$ are initially unmarked (marks on these edges will be set in `markSpannedEdges` and used in line 17), the roots of the subtrees are marked red (these marks will be used in `reconnect`). Whenever $\pi(b,d)$ occurs in the pseudo-code, this refers to the corresponding (fixed) path in $\hat{P}$. In contrast, the intermediate tree T_* changes during the run of Algorithm 6, as it is rebuilt by reconnecting edges and assigning new costs to the edges, finally providing the new Gomory-Hu tree $T(G^{\oplus})$. The free vertices on $\pi(b,d)$ are stored in S_π, the first step pair $\{u_i, v_j\}$ is initialized in line 7. The indices for u_i and v_j are chosen consecutively, as in Fig. 9.7. The modification repeats the computation of split cuts and the reconnection of edges until d is the only free vertex (see line 8).

Procedure markSpannedEdges(C_e, $\hat{P}$, i, j)

```
on $\pi(b,d)$ in $\hat{P}$, iterate edges in $C_e$ from the vertex closest to $d$ towards $b$
if current edge $e$ is already marked with $(i',j')$ then
    if $(U, V \setminus U)$ separates $u_{i'}$ and $v_{j'}$ then                    // stretching
        mark $e$ with $(i,j)$
        if $e$ is the last edge in $C_e$ then
            mark also all following edges that are already marked with $(i',j')$ with $(i,j)$
    if $u_{i'}$ and $v_{j'}$ are not separated by $(U, V \setminus U)$ then          // clinching
        stop iterating
```

Procedure reconnect(C_v, T_*, N, $\bar{N}$, c, $\bar{c}$)

```
forall the $x \in N \setminus \{c\}$ do                                // reconnect to outer side
    if $(x \in V \setminus C_v)$ or $(x \in C_v$ and marked with $y$ and $y \in V \setminus C_v)$ then
        $T_* \leftarrow$ reconnect $x$ to $c$ in $T_*$
        if $x$ is red then mark $x$ with $\bar{c}$
forall the $x \in \bar{N} \setminus \{\bar{c}\}$ do                    // reconnect to inner side
    if $x \in C_v$ and [$(x$ is marked with $y$ and $y \in C_v)$ or ($x$ is unmarked)] then
        $T_* \leftarrow$ reconnect $x$ to $\bar{c}$ in $T_*$
return $T_*$
```

Computing the Split Cut. After the computation of a minimum u_i-v_j-cut (with respect to step pair $\{u_i, v_j\}$) in line 9, the algorithm decides whether the final split cut will be an arch or a vertical pillar (see line 10 and line 16). In case of an arch it further decides whether it will be a left or a right arch. Note that here right and left refer to the determination in the previous section that $\pi(b,d)$ is horizontally embedded with b on the left and d on the right side. If we swap b and d, right arches become left and left become right arches.

The arch-reshaping of the found minimum u_i-v_j-cut (recall Fig. 9.3), is now realized by storing the cut side that is spanned by the resulting arch in a variable C_v (line 13 and line 14). Note that this cut side is already characterized by the cut out segment on $\pi(b,d)$ and the roots of the fat subtrees that are part of the cut side. Storing the whole subtrees on the cut side is not necessary, since the corresponding linking edges already shelter the trees such that a later reconnection of edges (according to Gusfield [66]) will automatically assign these subtrees completely to the correct cut side. The variable C_e (line 13) stores only the edges of the cut out segment. If the current split cut is a right arch, that is, u_i is still free, this information is used to mark the edges between u_i and v_j on $\pi(b,d)$ that are already spanned by a (right) arch, and thus, are no candidates for a reusable vertical pillar in a later step (see `markSpannedEdges` in line 15). The edges are marked by a tuple (i,j) referring to the step pair of the non-nested arch that spans them. However, marking just the edges in C_e, that is, the edges on the cut-out segment, would result in wrong marks if the current split cut needs to be clinched later in order to prevent crossings with previously found arches. We will see later that the clinching and stretching also happens automatically during the reconnection of edges according to Gusfield. However, when marking the edges on $\pi(b,d)$, we need to explicitly distinguish the cases of clinching and stretching, as it is done in `markSpannedEdges`. Note that removing a possible overhang after the clinching does not affect the set of spanned vertices on $\pi(b,d)$.

If the final split cut will be a vertical pillar (see line 16), an appropriate reusable cut can be easily found due to the marks previously set on $\pi(b, d)$; any unmarked edge between u_i and v_j on $\pi(b, d)$ with cost $\lambda_G(u_i, v_j)$ can be chosen.

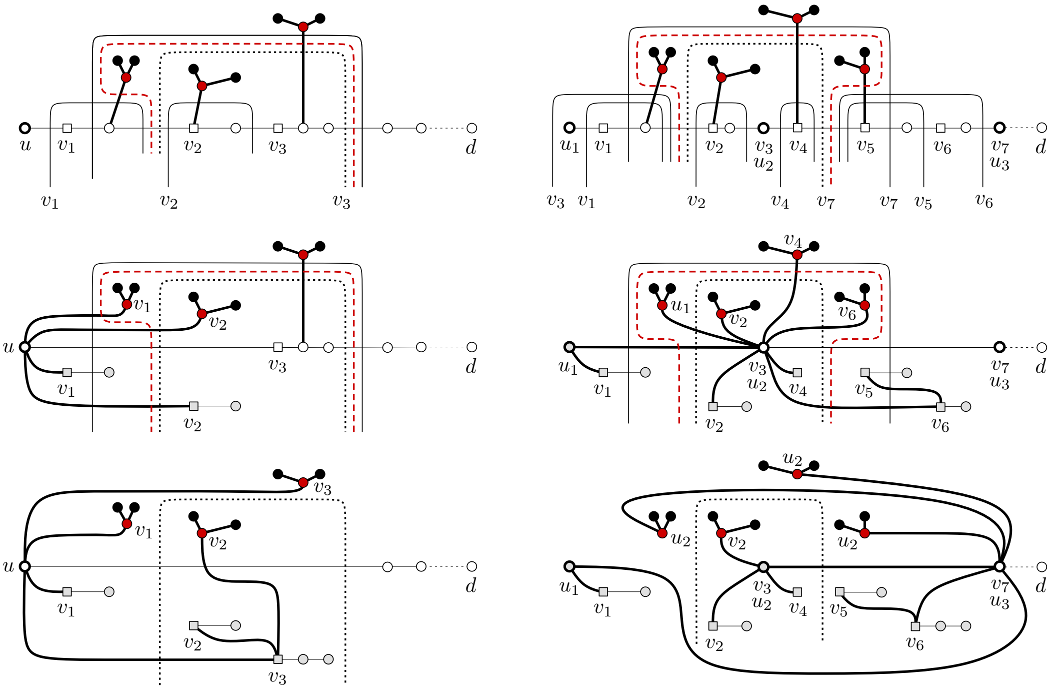
The edge $\{u_i, v_j\}$ in T_* that will later represent the final split cut is created in line 19. Note that $\{u_i, v_j\}$ is already a thin edge in T_* due to the choice of the step pair after a left arch or a vertical pillar has occurred as split cut (see line 30), and due to the choice of v' in line 25 if the split cut is a right arch; v' is reconnected to the current vertex u_i and becomes v_{j+1} in the next step.

Reconnecting Edges. The reconnection in `reconnect` is done according to Gusfield [66], where the free vertices form the compound node containing the current step pair, and the set C_v represents the cut side that is spanned by the split cut. In this context, we define that a vertical pillar $(U, V \setminus U)$ in line 17 spans all vertices in U. In order to prevent crossings with previously found split cuts, the general procedure is to appropriately reconnect the linking edges of the subtrees of the current compound node in T_*. In our modified update routine we further use the reconnection of such edges to establish the final form of the split cuts as described in the previous section. Recall that if the current split cut is an arch, it is already arch-reshaped, but not yet stretched or clinched and possible overhangs are not yet removed. While stretching and clinching directly follows from preventing crossings with previous split cuts, possible overhangs must be removed explicitly. This is described in the next paragraph. In that paragraph, we show how the subtrees of the compound node can be found. If the current split cut is a vertical pillar, no further reshaping is necessary, that is, C_v already induces the final cut.

The linking edges of the subtrees of the current compound node can be easily determined by the help of the structure of the current intermediate tree T_*. We observe that a subtree of the current compound node in T_* either corresponds to a fat subtree (in $\hat{T}$) or results from a previous split cut, and thus, corresponds to the cut side that is spanned by this cut (in $\hat{T}$). A subtree of the latter form is always linked to u_i, while a fat subtree may be linked to u_i as well as to another free vertex in T_*. For an example, see Fig 9.9(a). The upper drawing shows a sequence of right arches in $\hat{T}$. In the middle, the split cuts induced by the step pairs $\{u, v_1\}$ and $\{u, v_2\}$ are already processed resulting in the intermediate tree T_* right before the next split cut with respect to step pair $\{u, v_3\}$ is considered. The solid arch-shaped line indicates this split cut right before line 19 in Algorithm 6, that is, the cut that is induced by C_v so far. In this example, four subtrees (two of which are fat subtrees) are linked to u and one subtree, which is also a fat subtree, is linked to a free vertex in C_v.

If a fat subtree is linked to a free vertex that is not in C_v, that is, a free vertex that is not spanned by the current split cut, the subtree is also not spanned by the split cut, and thus, there is no need for reconnecting. Hence, we may skip these subtrees of the compound node in the remaining considerations, and restrict ourselves to the subtrees that are linked to a free vertex in C_v or to u_i. Note that for a vertical pillar or a left arch, the only free vertex in C_v is u_i, and thus, both cases describe the same set of subtrees. In Algorithm 6, the subtrees that are linked to a free vertex in C_v are stored in N (more precisely, the roots are stored), while the remaining subtrees are stored in $\bar{N}$. The latter set is empty for vertical pillars and left arches.

Removing Overhanging Subtrees. In the following, we show how `reconnect` removes overhanging subtrees, while reconnecting edges according to Gusfield. We observe that an overhanging subtree is not spanned by any previously found split cut, but is separated from its base



(a) Sequence of right arches. Minimum u-v_3-cut, which is a right arch, is the current split cut.

(b) Sequence of right and left arches. Minimum u_2-v_7-cut, which is a left arch, is the current split cut.

Figure 9.9: Exemplary reconnection of edges in the intermediate tree T_*. Top: Exemplary sequences of arches. Current split cut is only arch-reshaped (solid line), but not yet clinched, which will result in overhangs (dashed line). Removing the overhangs will result in the dotted line. Middle: Intermediate tree before the current split cut is considered. Bottom: Intermediate tree after reconnecting edges for the current split cut. The edge $\{u_i, v_j\}$ now represents the split cut without any overhanging subtrees. Free vertices are empty and become gray when they are separated from d, fat subtrees are red colored and possibly marked by $\bar{c}$. Step vertices u_i are bold-framed, step vertices v_j are depicted as squares.

vertex (in $\hat{T}$) by at least one previous split cut. Thus, it is must be linked to u_i in T_*. Hence, the fat subtrees linked to u_i are the candidates for overhanging subtrees with respect to the current split cut if this split cut is an arch. Moreover, all candidates were separated from their base vertices by at least one previous split cut; otherwise they would not have been connected to u_i in T_*. In the middle part of Fig. 9.9(a), the red dashed line depicts the current split cut with an overhanging subtree that is linked to u, due to the reconnection caused by the step pair $\{u, v_1\}$.

In T_*, the base vertex of each candidate is part of a subtree that is linked to the current compound node, and the linking edge of this subtree represents the latest split cut that separates the candidate from its base vertex. The root of this subtree corresponds to the latest step vertex on the same cut side as the base vertex. We denote this root by $\bar{c}$, and observe that $\bar{c}$ was assigned to the candidate in line 4 of `reconnect`, in addition to the red color that all fat subtrees get at the beginning of Algorithm 6 in line 5. In the middle of Fig. 9.9(a), the fat subtrees that are linked to u are marked with v_1 and v_2, while the third subtree is not yet marked, since it is no candidate for an overhanging subtree with respect to the current split cut. However, after the splitting, the third subtree becomes a candidate with respect to the next split cut and is thus marked with v_3 at the bottom of Fig. 9.9(a). By the help of these marks, we can now remove possibly overhanging subtrees from the current split cut as follows.

Consider a fat subtree that is linked (in T_*) to a free vertex in C_v, and is thus represented in N by its root. Such a subtree is an overhanging subtree if and only if it is in C_v, that is, it is spanned by the current split cut, but its associated vertex $\bar{c}$ is not in C_v (see second condition in line 2 in `reconnect`). We note that this situation only occurs for left arches and vertical pillars, where the only free vertex in C_v is u_i, since any candidate for an overhanging subtree is linked to u_i. An example of overhanging subtrees at a left arch is shown in Fig. 9.9(b). The overhanging subtree then needs to be reconnected to the opposite step vertex of the current split cut, that is, the step vertex that is not spanned by the split cut. Any further subtree of the compound node that is linked to a free vertex in C_v is reconnected, according to Gusfield, if and only if it is not spanned by the current split cut, that is, if its root is in $V \setminus C_v$ (see first condition in line 2 in `reconnect`). Consider the third subtree in Fig. 9.9(a) or the subtree marked with v_4 in the middle of Fig. 9.9(b) for an example.

If the current split cut is a right arch, N does not contain any candidate for an overhanging subtree, since $u_i \notin C_v$. Thus, the subtrees of the compound node that are linked to u_i in T_* are additionally stored in $\bar{N}$, represented by their roots. A fat subtree in $\bar{N}$ is an overhanging subtree if and only if it is in C_v, but its associated vertex $\bar{c}$ is not in C_v. This subtree then must not be reconnected. Instead, the remaining subtrees in $\bar{N}$ that have their root in C_v (see line 6 in `reconnect`) need to be reconnected to the current step vertex in C_v, that is, the step vertex of the current split cut that is spanned by the split cut. Consider the subtree marked with v_2 in Fig. 9.9(a) for an example.

Finishing the Gomory-Hu Tree for $G^{\oplus}$. After the reconnection in line 24 and line 28 of Algorithm 6, the set of free vertices is updated (line 25 and 29), the indices for the step pairs are increased and the new step pair is chosen. In line 31, the intermediate tree resulting from the reconnection is finally processed by Cut Tree until all compound nodes are singletons and all edges are fat edges. In this phase, arbitrary step pairs and arbitrary split cuts are allowed, since a further processing does not change the cuts that are already represented by fat edges in the tree.

9.3 Optimality of Asymptotic Worst-Case Running Time

As demonstrated by the example in Fig. 8.5, there exist instances where our dynamic algorithm proposed in Section 8.2 still needs as many cut computations as a computation from scratch. In this section we argue that there also exist instances, for which we probably cannot do any better if the task is to solve the all-pairs minimum-cut problem. To this end, we compare the asymptotic worst-case running time of our algorithm to the asymptotic running time needed in the worst case for dynamically solving the all-pairs minimum-cut problem under the assumption of the recomputation conjecture (Section 6.2.3) and assuming that we have an extended Gomory-Hu tree, which provides comprehensive information about the cut structure in the current graph G (see Chapter 6), available in the current time step. We will see that even in this situation solving the all-pairs minimum cut problem in the next snapshot $G^{\copyright}$ may require at least $n-6$ cut computations.

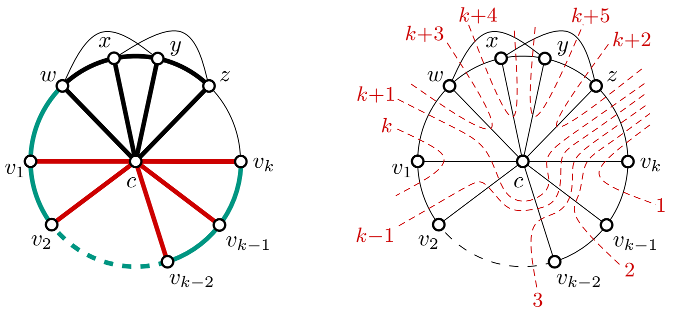
Suppose we are given an extended Gomory-Hu tree for the current graph G, that is, a Gomory-Hu tree together with $n-1$ maximum flows resulting from the construction of the tree and an extended flow-equivalent tree again with $n-1$ maximum flows resulting from the construction (which are not necessarily distinct form the flows of the Gomory-Hu tree construction). That is, besides the two tree structures, we may assume that we know $\binom{n}{2}$ DAGs, which we can derive from the extended flow-equivalent tree, $n-1$ maximum flows associated with the edges in the flow-equivalent tree and further maximum flows resulting from the Gomory-Hu tree construction. After a change in G, we aim at a new data structure that again admits to answer the queries of the all-pairs minimum-cut problem. Due to the nature of the all-pairs minimum-cut problem, the new data structure thus needs to know at least $n-1$ minimum separating cuts in $G^{\textcircled{\tiny U}}$, independent from its actual form.

Based on our preliminary discussion about different possibilities for updating given maximum flows and DAGs, we assume in the following the recomputation conjecture, that is, we assume that finding a minimum s-t-cut in $G^{\textcircled{\tiny U}}$ that is not represented in one of the $\binom{n}{2}$ DAGs provided by an extended Gomory-Hu tree, and for which further no minimum s-t-flow is known that could be updated, costs a full cut computation. With this assumption, we can imagine the following situation, where we definitely need $n-O(1)$ cut computations in order to construct the desired data structure for $G^{\textcircled{\tiny U}}$. Let G denote the current graph for which we know an extended Gomory-Hu tree. After a change in G, suppose that only a constant number c of the minimum separating cuts represented in the known DAGs remain valid in $G^{\textcircled{\tiny U}}$, that is, are still minimum separating cuts in $G^{\textcircled{\tiny U}}$ with respect to any cut pair. In other words, at least $n-O(1)$ minimum separating cuts in $G^{\textcircled{\tiny U}}$ cannot be deduced from the known DAGs. Suppose further that updating the known maximum flows also results in new maximum flows that together just induce the few minimum separating cuts which we already know from the DAGs. In this situation, $n-O(1)$ cut computations from scratch is the best we can do. In the following we will show an example where exactly this situation occurs. This proves the following theorem.

Theorem 9.6. *The asymptotic worst-case running time of $n-O(1)$ cut computations of our fully dynamic update algorithm for Gomory-Hu trees is optimal, under the assumption of the recomputation conjecture, in the sense that also providing further information about the cut structure of the current snapshot (in form of an extended Gomory-Hu tree) does not achieve any better asymptotic worst-case running time for solving the all-pairs minimum-cut problem in a dynamic scenario.*

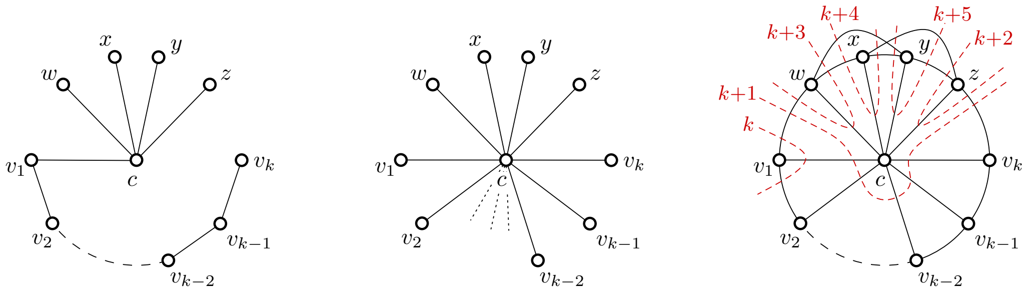
9.3.1 Outline of the Example

For $n \geq 10$, we construct a graph $G_n = (V, E, c)$ with $n = |V|$ as follows. The base of G_n is a wheel graph W_n consisting of a center c and an $(n-1)$-cycle $v_1 \dots v_k zyxw$ that is extended by two further edges $\{w, y\}$ and $\{x, z\}$ (see Fig. 9.10(a)). Due to these additional edges, the vertices w, x, y, z, c induce a K_5-minor such that G_n is not planar. The main ingredient of the construction is a carefully chosen (initial) cost function $c : E \to \mathbb{N}$, which we define in the next section, together with a change of the cost of a particular edge. The initial cost function is chosen such that the set of all minimum separating cuts in G_n consists of exactly n cuts, which are depicted in Fig. 9.10(b). That is, the $\binom{n}{2}$ DAGs provided by the extended Gomory-Hu tree in this example together represent exactly these minimum separating cuts. The numbers associated with these cuts indicate a non-decreasing order of the cuts by costs, where the cuts θ_k and θ_{k+1} as well as θ_{k+4} and θ_{k+5} are the only pairs of cuts with identical costs. Figure 9.10(c) shows a Gomory-Hu tree $T(G_n)$ of G_n, which we assume to result from the Cut Tree execution $(G_n, \mathcal{F}, \mathcal{K})$ with $\mathcal{F} = \{v_k, c\}, \dots, \{v_1, c\}, \{z, c\}, \{w, c\}, \{x, c\}, \{y, c\}$ and $\mathcal{K} = \theta_1, \dots, \theta_{k-1}, \theta_{k+1}, \dots, \theta_{k+5}$. Furthermore, we assume that the sequence $v_k, \dots, v_1, z, w, x, y$ of vertices and the sequence $\mathcal{K}$ of cuts was used to construct the flow-equivalent tree (starting



(a) Graph G_n.

(b) All minimum separating cuts in G_n.



(c) A Gomory-Hu tree $T(G_n)$.

(d) A flow-equivalent tree $T'(G_n)$.

(e) Minimum separating cuts in G_n that remain minimum separating cuts in $G_n^{\ominus}$.

FIGURE 9.10: Outline of the example proving Theorem 9.6. The colors in (a) will be used to define the precise cost function. Dashed red lines indicate minimum separating cuts.

with a star with center c); see Section 6.2.1 for a description of the algorithm. The resulting flow-equivalent tree is shown in Fig. 9.10(d). Hence, the extended Gomory-Hu tree in this example provides $n-1$ maximum flows with respect to the vertex pairs in $\mathcal{F}$. We remark that, apart from the maximum v_1-c-flow, which represents θ_k and θ_{k+1}, each of the remaining flows represents exactly one cut. That is, the minimum v-c-cut depicted in Fig. 9.10(b) is unique for each vertex pair $\{v, c\}$ with $v \in V \setminus \{c, v_1\}$

In the next time step the cost of the cycle edge $\{w, v_1\}$ decreases. As a consequence, the cut θ_{k+1} becomes the unique minimum v_i-c-cut for all vertices v_i with $i \in \{2, \dots, k\}$. Furthermore, θ_k and θ_{k+1} remain the only minimum v_1-c-cuts, and $\theta_{k+2}, \dots, \theta_{k+5}$ remain unique minimum v-c-cuts for $v = z, w, x, y$. That is, updating the $n-1$ maximum flows provided by the extended Gomory-Hu tree yields just the cuts depicted in Fig. 9.10(e). Note, that this is only a constant number of cuts. Furthermore, we will see that the cuts in Fig. 9.10(e) are the only minimum separating cuts in $G_n^{\ominus}$ that have been already presented by a DAG of the extended Gomory-Hu tree in the previous time step (or the other way round, these cuts are the only minimum separating cuts of the known DAGs that are also minimum separating cuts in $G_n^{\ominus}$). Hence, our example satisfies exactly the conditions claimed above.

In the following we give the precise cost function for G_n and define the change of the cost of the cycle edge $\{w, v_1\}$. Then, we show that the order given by the numbers in Fig. 9.10(b) is correct, and that θ_k and θ_{k+1} are the only minimum v_1-c-cuts in G_n and $G_n^{\ominus}$ and that each of the remaining cuts depicted in Fig. 9.10(b) is a unique minimum v-c-cut in G_n for the vertex $v \in V \setminus \{c, v_1\}$ it separates from c. To this end, we generally compare the costs of the depicted cuts to the costs of other possible minimum v-c-cuts ($v \in V \setminus \{c\}$) in G_n and in $G_n^{\ominus}$. Based on this comparison, we will further see that in $G_n^{\ominus}$ the cut θ_{k+1} is the unique minimum v_i-c-cut for all vertices v_i with $i \in \{2, \ldots, k\}$ and that $\theta_{k+2}, \ldots, \theta_{k+5}$ are still unique minimum v-c-cuts for $v = w, x, y, z$.

In a final step, we will then argue that each of the $\binom{n}{2}$ DAGs provided by the extended Gomory-Hu tree either equals one DAG or is a union of two DAGs in the set $\{\mathrm{DAG}_{v,c} \mid v \in V \setminus \{c\}\}$ (considering DAGs as sets of cuts). Together with the considerations above, this yields that the cuts depicted in Fig. 9.10(a) are indeed the only minimum separating cuts in G_n and that our example behaves as described in the outline, which finally proves Theorem 9.6.

9.3.2 Cost Function and Change of Edge Cost

In order to define the initial cost function $c : E \to \mathbb{N}_0$, we define partial cost functions on subsets of edges, yielding c by combining these functions. We distinguish the subsets of edges by colors, as indicated in Fig. 9.10(a). The change of the edge cost in the next time step is defined as a decrease of the cost of the cycle edge $e_{0,1} = \{w, v_1\}$ from A to 1.

- *red spokes:* We denote the red spokes by $\{v_i, c\} =: e_i$, $i = 1, \ldots, k$. Note that by assumption G_n has at least 10 vertices and thus $k \geq 5$. The idea of the partial cost function defined for these spokes is that each spoke e_i is at least as expensive as all the spokes $e_1, \ldots, e_{i-1}$ together. Thus, we define the costs as follows: $c(e_1) := k$, $c(e_i) := \sum_{j=1}^{i-1} c(e_j)$ for $i = 2, \ldots, k$. This yields $c(e_1) = c(e_2) = k$ and $c(e_i) = 2^{(i-2)}k$ for $i = 3, \ldots, k$. The total costs of all spokes are denoted by $A := \sum_{j=1}^{k} c(e_j) = 2c(e_k) = 2^{(k-1)}k$.
- *green cycle edges:* The green cycle edges are denoted by $\{v_i, v_{i+1}\} =: e_{i,i+1}$ for $i = 1, \ldots, k-1$ and $\{w, v_1\} =: e_{0,1}$. We define $c(e_{0,1}) := A$ and the partial cost function on the remaining edges such that each cycle edge $e_{i,i+1}$ together with the spokes $e_1, \ldots, e_i$ costs exactly $A+i$, for $i = 1, \ldots, k-1$. This yields $c(e_{i,i+1}) := A - \sum_{j=1}^{i} c(e_j) + i$ for $i = 1, \ldots, k-1$. Note that due to the choice of the costs of the red spokes it further holds for $i = 1, \ldots, k-1$ that each cycle edge $e_{i,i+1}$ together with the single spoke e_{i+1} also costs $A + i$.
- *thin cycle edge $\{v_k, z\}$:* We define $c(v_k, z) := 1$.
- *remaining black edges:* The edges $\{w, y\}$ and $\{x, z\}$ cost 1, each. The remaining black edges are very expensive, close to infinity. We denote the cost of such an expensive spoke by I.

9.3.3 Comparing Cut Costs

In order to compare the costs of the cuts considered in this example, which we claim to be minimum v-c-cuts (for $v \in V \setminus \{c\}$), to the costs of other possible minimum v-c-cuts, we state a central observation regarding the shape of possible minimum v-c-cuts in G_n. Due to the wheel structure of G_n, the candidates for minimum v-c-cuts in both time steps are all of the form $(\pi, V \setminus \pi)$ with π a path in the $(n-1)$-cycle containing v. In order to prove that a given cut is a minimum v-c-cut, it thus suffices to compare its cost to the costs of cuts that separate a path $\pi \ni v$ from c. Referring to the end points of $\pi = \bar{p} \ldots \bar{q}$, we denote such a cut by $\theta_{\bar{p},\bar{q}}$ with $\bar{q}$ following $\bar{p}$ in a counterclockwise direction in Fig. 9.10(a). If $\bar{p} = \bar{q}$ we also write $\theta_{\bar{p}}$.

We further observe that, due to the high costs of the spokes connected to w, x, y and z, in both time steps, the path π of a minimum v_i-c-cut $(\pi, V \setminus \pi)$ with $i \in \{1, \ldots, k\}$ contains no vertex from the set $\{w, x, y, z\}$. If $(\pi, V \setminus \pi)$ is a minimum v-c-cut with $v \in \{w, x, y, z\}$, then v is the only vertex from the set $\{w, x, y, z\}$ that is contained in π. This yields directly that the cuts θ_{k+4} and θ_{k+5} are the unique minimum separating cuts for the vertex pairs $\{x, c\}$ and $\{y, c\}$ in both time steps.

Comparing Costs of $\boldsymbol{\theta_{k+2}, \ldots, \theta_{k+5}}$. In this paragraph we will see the following.
(1) it is $c(\theta_{k+2}) < c(\theta_{k+3}) < c(\theta_{k+4}) = c(\theta_{k+5})$ in G_n
(2) each cut in $\{\theta_{k+2}, \ldots, \theta_{k+5}\}$ is the unique minimum v-c-cut in G_n and $G_n^{\ominus}$ with respect to the vertex $v \in \{w, x, y, z\}$ it separates from c

For θ_{k+4} and θ_{k+5} we have already seen above that (2) holds. For θ_{k+2} and θ_{k+3} the following table lists the costs in both time steps. The first column is associated with the initial time step, the second column represents the costs after the change in G_n. Due to the change in G_n, the cost of θ_{k+3} decreases by $A - 1$, while the cost of θ_{k+2} remains the same.

	initially	after change
θ_{k+2}	$2I + 2$	
θ_{k+3}	$2I + A + 1$	$2I + 2$

Now we compare the cost of θ_{k+2} and θ_{k+3} to the costs of all further candidates of minimum v-c-cuts with $v \in \{z, w\}$. According to the notation introduced above, the latter are denoted by $\theta_{v_p,z}$ and θ_{w,v_q} with $p, q \in \{1, \ldots, k\}$. Recall that the indices indicate the endpoints of paths in the $(n-1)$-cycle of G_n.

The costs of $\theta_{v_p,z}$ and θ_{w,v_q} are given for each time step in the following two tables.

$\theta_{v_p,z}$	initially	after change
$p > 1$	$2I + A + p + \sum_{j=p+1}^{k} c(e_j)$	
$p = 1$	$2I + 2A + 1$	$2I + A + 2$

θ_{w,v_q}	initially	after change
$q < k$	$2I + A + q + 1$	
$q = k$	$2I + A + 2$	

We observe that for $p > 1$ the cut $\theta_{v_p,z}$ does not cross the changing cycle edge $e_{0,1} = \{w, v_1\}$. Hence, the cost remains stable in both time steps. For $p = 1$ the cost of $\theta_{v_p,z}$ decreases by $A - 1$ after the change, since in this case $\theta_{v_p,z}$ crosses $e_{0,1} = \{w, v_1\}$. The cost of θ_{w,v_q} remains stable in all cases and time steps, since θ_{w,v_q} never crosses the changing cycle edge $e_{0,1} = \{w, v_1\}$.

To further verify the values in the tables, observe that the cost of $\theta_{v_p,z}$ can be determined by $c(\theta_{v_p,z}) = c(e_{p-1,p}) + \sum_{j=p}^{k} c(e_j) + 2I + 1$ and the cost of θ_{w,v_q} is given by $c(\theta_{w,v_q}) = 1 + 2I + \sum_{j=1}^{q} c(e_j) + c(e_{q,q+1})$, and recall the principles of the initial cost function. In the first table we exploit the fact that $c(e_{p-1,p}) + c(e_p) = A + (p - 1)$ for $p = 2, \ldots, k$. The values of the second table are based on the fact that $\sum_{j=1}^{q} c(e_j) + c(e_{q,q+1}) = A + q$ for $q = 1, \ldots, k$.

Finally comparing the costs to the costs of θ_{k+2} and θ_{k+3} yields the following. The minimum cost of $\theta_{v_p,z}$ over all choices of $p \in \{1, \ldots, k\}$ in any time step is $2I + A + 2$. With $A \geq k$ this is still more expensive than $2I + 2$, which is the cost of θ_{k+2} (see the first row in the previous table). Hence, θ_{k+2} is the unique minimum z-c-cut in both time steps, that is, (2) also holds for θ_{k+2}. The minimum cost of θ_{w,v_q} over all choices of $q \in \{1, \ldots, k\}$ in any time step is $2I + A + 2$. This is again still more expensive than $2I + A + 1$, which is the maximum cost of θ_{k+2} (see the second row in the previous table). Hence, θ_{k+3} is the unique minimum w-c-cut in both time steps, that is, (2) finally holds for θ_{k+3}, and thus, (2) is totally proven.

In order to prove (1), we finally consider the costs of θ_{k+4} and θ_{k+5} and compare them to the costs of θ_{k+2} and θ_{k+3} presented in the first column of the first table. Obviously, θ_{k+4} and θ_{k+5} have cost $3I + 1$ in G_n, which is more expensive than $c(\theta_{k+3}) = 2I + A + 1$. The latter is again more expensive than $c(\theta_{k+2}) = 2I + 2$. Thus, (1) holds. In the same manner, we now consider the costs of the remaining cuts depicted in Fig. 9.10(a).

Comparing Costs of $\boldsymbol{\theta_1, \ldots, \theta_{k+1}}$. In this paragraph we will see the following.

(3) it is $c(\theta_1) < \cdots < c(\theta_{k-1}) < c(\theta_k) = c(\theta_{k+1})$ in G_n
(4) each cut in θ_j is the unique minimum v_{k+1-j}-c-cut in G_n for $j = 1, \ldots, k-1$
(5) θ_k and θ_{k+1} are the only minimum v_1-c-cuts in G_n and $G_n^{\ominus}$
(6) θ_{k+1} is the unique minimum v_i-c-cut for $i = 2, \ldots, k$ in $G_n^{\ominus}$

We compare again the costs of $\theta_1, \ldots \theta_{k+1}$ in each time step to the costs of all candidates for minimum v_i-c-cuts with $i = 1, \ldots, k$. According to the notation introduced above, the latter are denoted by θ_{v_p,v_q} with $p, q \in \{1, \ldots, k\}$ and $p \le i \le q$. We also allow $p = q = i$. Note that this classes of cuts represent all candidates for minimum v_i-c-cuts, including $\theta_1, \ldots \theta_{k+1}$.

The next two tables list the costs of all these candidates (including $\theta_1, \ldots \theta_{k+1}$) for each time step. For the sake of clarity regarding the choice of parameters, we present the costs of the candidates θ_{v_p,v_q} with $p = q = i$ in the first table and the costs of all remaining candidates in the second table.

θ_{v_i}	initially	after change
$i \in \{2, \ldots, k-1\}$	$2A + 2i - 1 - \sum_{j=1}^{i} c(e_j)$	
$i = k$	$A + k$	
$i = 1$	$2A + 1$	$A + 2$

$\theta_{v_p,v_q}(p < q)$	initially	after change
$p > 1, q < k$	$2A + (p-1) - \sum_{j=1}^{p} c(e_j) + q$	
$p > 1, q = k$	$2A + p - \sum_{j=1}^{p} c(e_j)$	
$p = 1, q < k$	$2A + q$	$A + q + 1$
$p = 1, q = k$	$2A + 1$	$A + 2$

We observe that for $1 < p \le q < k$ none of the cuts θ_{v_p,v_q} crosses the changing cycle edge $e_{0,1} = \{w, v_1\}$. Hence, the costs remain stable in both time steps. For the remaining parameter values the costs of the cuts θ_{v_p,v_q} decrease according to the decrease of the costs of the changing cycle edge $e_{0,1} = \{w, v_1\}$, which is $A - 1$.

To further verify the values in the tables, observe that the cost of any cut θ_{v_p,v_q} is given by $c(\theta_{v_p,v_q}) = c(e_{p-1,p}) + \sum_{j=p}^{q} c(e_j) + c(e_{q,q+1})$, where the second term becomes $c(e_i)$ for $p = q = i$, and recall the principles of the initial cost function. In both tables, we exploit the fact that $c(e_{p-1,p}) + c(e_p) = A + (p-1)$ for $p = 2, \ldots, k$, and $c(e_{q,q+1}) = A - \sum_{j=1}^{q} c(e_j) + q$ for $q = 0, \ldots, k$.

In order to prove (3), we explicitly extract the costs of $\theta_1, \ldots, \theta_{k+1}$ in G_n from the tables or directly from the cost function. First we see in the tables that $\theta_1 = \theta_{v_k}$ has cost $A + k$.

Furthermore, for $j = 2, \dots, k-1$ $\theta_j = \theta_{v_{k+1-j},v_k}$ has cost $2A + p - \sum_{j=1}^{p} c(e_j)$ with $p = k+1-j$. For $i = 2$ this yields $c(\theta_2) = A + (k-2) + 2^{k-2}k + 1$ (by the principles of the cost function), which is more expensive than $c(\theta_1) = A + k$. For increasing j, we observe that $p = k+1-j$ decreases by 1 in each step, while the sum that is subtracted in each step increases by $c(e_p) > 1$. Thus, the costs increase with increasing j. For $j = k-1$ this yields $c(\theta_{k-1}) = 2A - 2k + 1$ (by the principles of the cost function). Finally, $\theta_k = \theta_{v_1}$ has cost $2A + 1$, which is again more expensive than $c(\theta_{k-1}) = 2A - 2k + 1$. For $\theta_{k+1} = \theta_{v_1,v_k}$ we see again in the table that $c(\theta_{k+1}) = 2A + 1 = c(\theta_k)$. This proves (3).

In order to prove (4), (5) and (6), we seek the cheapest minimum v_i-c-cut for each vertex v_i, $i \in \{1, \dots, k\}$ in each time step. We consider both time steps separately.

Initial time step. First observe that, due to the parameters, each row in tables above represents a class of cuts. We first prove (4). In (4) we claim that $\theta_1 = \theta_{v_k}$ is the unique minimum v_k-c-cut. Hence, we must prove that $c(\theta_{v_k}) = A + k$ is cheaper than any other cut that separates v_k from c. The latter cuts are hidden in second table in the second and the last row. At a second glance, we see that the cuts in these rows are exactly the cuts $\theta_2, \dots, \theta_{k-1}, \theta_{k+1}$. For these cuts we have already proven (when proving (3)) that they are all more expensive than θ_1. Hence, for θ_1 (4) holds.

For θ_j with $j = 2, \dots, k-1$, we claim in (4) that $\theta_j = \theta_{v_i,v_k}$ is the unique minimum v_i-c-cut with $i = k+1-j$. Any cut θ_j is hidden in the second table in the second row (with $p = i$) and any other cut that also separates v_i from c is either hidden in the first table in the first row or in any row of the second table. Hence, we compare the cost of cut θ_j, which we denote by I, to the cost in the first table in the first row, the minimum cost in second table in the first and second row, and the costs in the second table in the third and fourth row. We denote the latter costs by Q_1, P_1, P_2, P_3 and P_4 and prove that for all $P \in \{Q_1, P_1, P_2, P_3, P_4\}$ it is $P - I > 0$ for all $i \in \{2, \dots, k-1\}$. For Q_1 this directly yields

$$Q_1 - I = [2A + 2i - 1 - \sum_{j=1}^{i} c(e_j)] - [2A + i - \sum_{j=1}^{i} c(e_j)] = i - 1 > 0$$

For P_1 and P_2, that is, for the first and second row in the second table, we observe that the costs among all cuts that separate v_i from c become minimum (assuming a fixed q) for $p = i$. Recall that for $p > i$ the corresponding cut θ_{v_p,v_q} is no v_i-c-cut anymore. Hence, we consider P_1 and P_2 assuming that $p = i$. For P_1 this yields

$$P_1 - I = [2A + (i-1) - \sum_{j=1}^{i} c(e_j) + q] - [2A + i - \sum_{j=1}^{i} c(e_j)] = q - 1 > 0$$

For P_2 we get exactly the cuts $\theta_j = \theta_{v_i,v_k}$ with $i = k+1-j$. Hence, there is nothing to prove. For P_3 and P_4 we simply get

$$\begin{aligned} P_3 - I &= [2A + q] - [2A + i - \sum_{j=1}^{i} c(e_j)] = q - i + \sum_{j=1}^{i} c(e_j) > 0 \\ P_4 - I &= [2A + 1] - [2A + i - \sum_{j=1}^{i} c(e_j)] \geq 1 - i + k > 0 \end{aligned}$$

The equations above show that for $j = 2, \dots, k-1$ the cut $\theta_j = \theta_{v_i,v_k}$ with $i = k+1-j$ is indeed the unique minimum v_i-c-cut in G_n. Hence, we have proven (4).

Now we prove (5) for G_n. In (5) we claim that $\theta_k = \theta_{v_1}$ and $\theta_{k+1} = \theta_{v_1,v_k}$ are the only minimum v_1-c-cuts in G_n. We already know from (1) that $c(\theta_{v_1}) = 2A+1 = c(\theta_{v_1,v_k})$. The only row that remains for comparing in this case is the third row of the second table. In doing so we get $c(\theta_{v_1,v_q}) - c(\theta_{v_1}) = [2A+q] - [2A+1] = q-1 > 0$. Thus, θ_{v_1} and θ_{v_1,v_k} are the only minimum v_1-c-cuts in G_n. Regarding the next time step, after the change in G_n, we still need to prove (5) for $G_n^\ominus$ and (6).

After the change. We first prove (5) for $G_n^\ominus$. In (5) we claim that $\theta_k = \theta_{v_1}$ and $\theta_{k+1} = \theta_{v_1,v_k}$ are the only minimum v_1-c-cuts also in $G_n^\ominus$. The tables directly tell us that also after the change it is $c(\theta_{v_1}) = A+2 = c(\theta_{v_1,v_k})$. The only row that again remains for comparing is the third row of the second table. In doing so we get $c(\theta_{v_1,v_q}) - c(\theta_{v_1}) = [A+q+1] - [A+2] = q-1 > 0$. Thus, θ_{v_1} and θ_{v_1,v_k} are also the only minimum v_1-c-cuts in $G_n^\ominus$, which totally proves (5).

Now we prove (6). In (6) we claim that $\theta_{k+1} = \theta_{v_1,v_k}$ is the unique minimum v_i-c-cut in $G_n^\ominus$ for $i = 2, \dots, k$. Similar to the initial time step, the cut θ_{k+1} corresponds to the last row in the second table and any other cut that also separates v_i from c is either hidden in the first table in the first row or in one of the remaining rows of the second table. Hence, we compare the cost of cut θ_{k+1}, which we denote by I, to the cost in the first table in the first row, the minimum cost in second table in the first and second row, and the costs in the second table in the third row. We denote the latter costs by Q_1, P_1, P_2, and P_3 and prove that for all $P \in \{Q_1, P_1, P_2, P_3, P_4\}$ it is $P - I > 0$ for all $i \in \{2, \dots, k\}$. For Q_1 this directly yields

$$Q_1 - I = [2A + 2i - 1 - \sum_{j=1}^{i} c(e_j)] - [A+2] > 2i - 3 > 0$$

For P_1 and P_2, that is, for the first and the second row in the second table, we already observed before that the costs among all cuts that separate v_i from c become minimum (assuming a fixed q) for $p = i$. Hence, we consider P_1 and P_2 assuming again $p = i$. For P_1 this yields

$$P_1 - I = [2A + (i-1) - \sum_{j=1}^{i} c(e_j) + q] - [A+2] > q + i - 3 > 0$$

For P_2 we get exactly the cuts $\theta_j = \theta_{v_i,v_k}$ with $i = k+1-j$ and $j = 1, \dots, k-1$. Since the costs of these cuts do not change in $G_n^\ominus$, we already know from the proof of (3) that the minimum costs are $c(\theta_1) = A + k$. Hence, we get

$$P_2 - I = [A+k] - [A+2] = k - 2 > 0$$

For P_3 we simply get

$$P_3 - I = [A+q+1] - [A+2] = q - 1 > 0$$

The equations above show that the cut $\theta_{k+1} = \theta_{v_1,v_k}$ is indeed the unique minimum v_i-c-cuts for all vertices v_i with $i \in \{2, \dots, k\}$ in $G_n^\ominus$. Hence, we have proven (6).

9.3.4 Concluding the Proof

Since in G_n the cost of θ_{k+2}, which is $2I+2$, is more expensive than the cost of θ_{k+1}, which is $2A+1$, we know by (1) and (3) that the non-decreasing order of the cuts given by the numbers in Fig. 9.10(b) is correct. From (2), (4), and (5) we further know that the cuts depicted in Fig. 9.10(b) are the only minimum v-c-cuts in G_n for $v \in V \setminus \{c\}$. Due to the order and the shape of the cuts in Fig. 9.10(b) it further holds that each pair $\{x, y\} \subseteq V \setminus \{c\}$ is separated by a cheapest cut among all minimum x-c-cuts and minimum y-c-cuts. Vice versa, we observe that each minimum x-y-cut in G_n with $x, y \in V \setminus \{c\}$ either separates x or y from c. Thus, it must be also a minimum x-c-cut or a minimum y-c-cut, respectively.

Regarding these facts, it can be seen that each of the $\binom{n}{2}$ DAGs provided by the extended Gomory-Hu tree either equals one DAG or is a union of two DAGs in the set $\{\mathrm{DAG}_{v,c} \mid v \in V \setminus \{c\}\}$ (considering DAGs as sets of cuts). More precisely, besides $\mathrm{DAG}_{x,y} = \mathrm{DAG}_{x,c} \cup \mathrm{DAG}_{y,c}$, all remaining DAGs provided by the extended Gomory-Hu tree form the equivalence classes $\{\mathrm{DAG}_{v_i,c},\ \mathrm{DAG}_{v_i,v_{i-1}}, \ldots, \mathrm{DAG}_{v_i,v_1},\ \mathrm{DAG}_{v_i,w},\ \mathrm{DAG}_{v_i,x},\ \mathrm{DAG}_{v_i,y},\ \mathrm{DAG}_{v_i,z}\}$ for $i = k, \ldots, 1$, as well as $\{\mathrm{DAG}_{z,c},\ \mathrm{DAG}_{z,w},\ \mathrm{DAG}_{z,x},\ \mathrm{DAG}_{z,y}\}$, $\{\mathrm{DAG}_{w,c},\ \mathrm{DAG}_{w,x},\ \mathrm{DAG}_{w,y}\}$, and $\{\mathrm{DAG}_{x,c}\}$, $\{\mathrm{DAG}_{y,c}\}$. Hence, the cuts depicted in Fig. 9.10(b) are indeed the only minimum separating cuts in G_n. Finally, we know from (6) that after the change the previous minimum separating cuts in G_n behave as claimed before. According to the outline of this example this finally proves Theorem 9.6.

Conclusion of Part II

We discussed the nature of Gomory-Hu trees as a data structure in comparison with similar data structures in the field of minimum cuts and connectivity, thereby emphasizing the outstanding role of Gomory-Hu trees regarding the all-pairs minimum-cut problem. We first considered Gomory-Hu trees in a static context in Chapter 7.

Results for Static Scenarios. Since Gomory's and Hu's idea to represent all-pairs minimum cuts in a tree [59] is rather old, notations and descriptions in their pioneering work appear a bit outlandish in some points. Thus, we provided a more compact description of the Gomory-Hu tree construction pointing out the degrees of freedom, the fundamental technique of edge reconnection, and the relation to the simpler approach introduced by Gusfield [66]. The aim was to give a formulation that is clearly structured and condensed to main techniques such that similarities to other cut-based approaches become obvious. We will see that, for example, the cut-clustering algorithm considered in Part III strongly relies on Gomory's and Hu's tree construction. Furthermore, we developed the new data structure of unique-cut trees, which represents all U-cuts of an undirected, weighted graph, that is, all minimum separating cuts that minimize one side of the cut. In contrast to other attempts to represent cut structures in form of Gomory-Hu trees (recall that Gomory-Hu trees for vertex cuts do not exist), this specialization to U-cuts is possible, although in general U-cuts may cross, and the shape of the split cuts originally used during the Gomory-Hu tree construction is not guaranteed to be the same in the final tree.

Results for Dynamic Scenarios. In Chapter 8, we considered Gomory-Hu trees also in a dynamic scenario. With our dynamic update algorithm developed for Gomory-Hu trees of undirected, weighted graphs, we solve a problem that other authors also call sensitivity analysis in undirected, weighted multiterminal flow networks and that was already claimed to be challenging by authors like Barth et al. [11]. In the light of a dynamic scenario, the task is not only, as in sensitivity analysis, to understand how minimum separating cuts and their values behave when the underlying graph changes, but also to *efficiently* and *smoothly* maintain the Gomory-Hu data structure with all its special properties over the time. We proved in Chapter 9 that all update procedures for the different types of changes occurring in the dynamic graph guarantee optimal temporal smoothness. Interestingly, it turned out that obtaining a simple update algorithm in the case of an edge insertion or increasing edge cost is almost trivial. However, this procedure is far from temporal smoothness. Designing an algorithm that additionally guarantees temporal smoothness is surprisingly difficult in this case. On the other hand, efficiently updating a Gomory-Hu tree in the case of an edge deletion or decreasing edge cost is a priori more difficult, but the resulting algorithm is still easy to implement and can be proven to already guarantee temporal smoothness.

In the experiments in Chapter 8.3, our update approach showed a high potential for saving cut computations compared to a Gomory-Hu tree construction from scratch, and performed very well on the tested instance of an email-communication network. A more detailed analysis focused on the main factors that are responsible for potential savings showed that the shape of a Gomory-Hu tree significantly influences the number of saved cut computations. Moreover, we have seen that the cut structure of many graphs is very robust against changes, and hypothetically, even more cut computations could be saved. The asymptotic worst-case running time of our update algorithm, however, cannot be improved even by providing more information about the cut structure of the current graph in form of an extended Gomory-Hu tree, which additionally has access to several maximum flows and the DAGs that represent all existing minimum separating cuts in the graph. In this sense, the asymptotic worst-case running time of our algorithm is optimal.

Future Work and Further Notes. Due to the robustness of the cut structure of many graphs, there is still potential for saving even more cut computations despite the already good performance of our update algorithm. The search for additional techniques to also exploit this potential could be addressed in future work. In more general dynamic scenarios, where the consecutive snapshots differ in more than one edge or vertex, the cut structure is probably less robust. An extension of our update algorithm to more general dynamic scenarios is also an interesting challenge.

We finally remark that for both cases, edge deletion and cost decrease and edge insertion and cost increase, the proof of optimal temporal smoothness in Chapter 9 implies that each old cut in a Gomory-Hu tree that remains valid with respect to the vertices incident to its corresponding edge also appears in the new Gomory-Hu tree returned by our update algorithm. This even strengthens our smoothness result.

Part III

Cut-Based Clustering

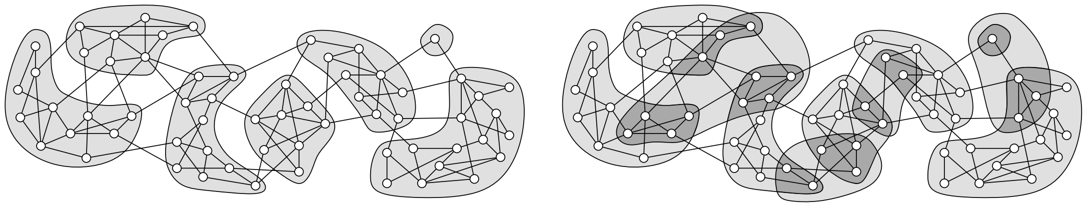
CHAPTER 10

Introduction –
Graph Clustering and Cohesive Subsets

In contrast to augmentation problems as considered in Part I, which seek for increasing the connectivity of a given graph by adding additional edges, graph clustering problems, roughly speaking, aim at decomposing a given graph along sparse cuts into somehow dense subgraphs, so-called clusters. This general aim is also expressed by the clustering paradigm of *intra-cluster density and inter-cluster sparsity* (Fig. 10.1). The vague terms sparse and dense refer to the edge structure of the underlying graph and are filled with different formal definitions, depending on the application and the particular algorithm that claims to find such clusters. In this way, almost every clustering algorithm induces its own formal problem definition. This is why comparing different clustering algorithms is very difficult.

On a high level, the literature distinguishes between algorithms that seek for pairwise disjoint clusters as depicted in Fig. 10.1(a) and algorithms that aim at detecting overlapping clusters as depicted in Fig. 10.1(b). Many of the former algorithms further allow to compute also cluster hierarchies, where the clusters of each level are still disjoint, but the clusters of different levels are nested, which depicts the nesting behavior of the clusters in a particular clear way. Considering several such hierarchies with different hierarchy levels of the same network further provides insight into the structure of overlapping clusters. Since the idea of graph clustering is originally motivated by the assumption that the structure of real-world networks naturally inheres significant groups, which are usually not disjoint but intersect, clustering algorithms that find nested or overlapping clusters became in particular important.

The techniques used by these clustering algorithms are as diverse as the applications they are designed for. Some algorithms are especially geared to the needs of a particular application, exploiting properties and possibly additional information provided by the networks typically occurring in this application. Other algorithms are designed to find dense clusters in general graphs just based on the edge structure. The results of these algorithms usually depend on how well the formal problem definition solved by the particular algorithm fits the hidden cluster structure in the given network; an issue that is, for example, also discussed in [143]. This is one of the main reasons why there exists no commonly used framework for evaluating clustering results on general graphs and comparing different clustering approaches. The overwhelming majority of algorithms that seek for good clusters in general graphs relies on heuristics for some NP-hard optimization problems regarding different formal definitions of density and sparsity. These algorithms are often evaluated just with respect to their special objective. Nevertheless,



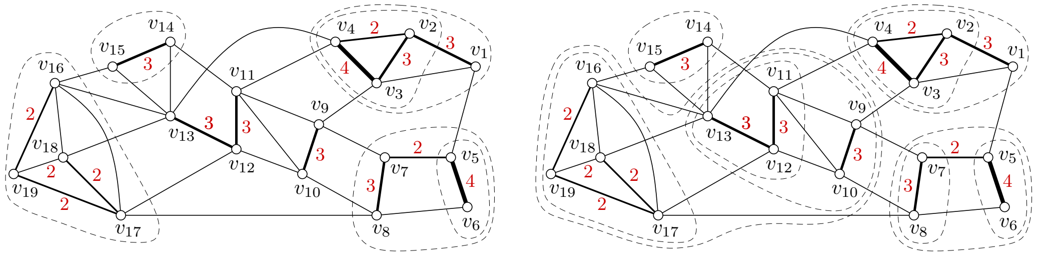
(a) Disjoint subgraphs. (b) Overlapping subgraphs.

Figure 10.1: Decompositions of a graph into subgraphs according to the intra-cluster density and inter-cluster sparsity paradigm, that is, into subgraphs that are characterized by dense internal connections while they are only sparsely connected to each other.

a few objectives, like *modularity* [116], became so popular that they are meanwhile also used as general quality measures for clusters. For an overview on further quality indices see, for example, the book of Brandes and Erlebach [21]. Other algorithms use special techniques that guarantee in advance that the found clusters satisfy some special cohesion properties or exceed a certain quality limit with respect to a numerical measure. The latter are in particular interesting for measures that are even NP-hard to compute. Popular examples are *expansion* and *conductance* [87, 55, 8]. In contrast to other measures, like modularity, these measures cannot be handled by a greedy heuristic, since it is already NP-hard to determine the exact quality of a single cluster. Instead, the corresponding algorithms employ, for example, spectral or cut-based clustering techniques. For an overview on different clustering techniques see the surveys of Schaeffer [125] and Fortunato et al. [46], as well as the thesis of Görke [60].

In this work, we will focus on a cut-based clustering algorithm originally presented by Flake et al. [42], which detects hierarchies of clusters of a guaranteed expansion. We will intensively investigate theoretical aspects of this algorithm and further develop this approach including an adaption to evolving graphs based on our results for dynamic Gomory-Hu trees in Part II.

Cohesive Subsets. The quality guaranteed by algorithms that find clusters of special cohesion properties is not measured by a numeric value. Instead, these algorithms guarantee the detection of groups that already satisfy the clustering paradigm of intra-cluster density and inter-cluster sparsity due to their formal definition. The idea of cohesive subsets originates from the analysis of social networks, where they have a particularly nice interpretation. In this context, a cohesive subset represents a community of predominantly connected entities, that is, a community whose members are in some sense stronger connected to other members of the community than to entities that are excluded from the community. One example of cohesive subsets in an undirected, weighted graph $G = (V, E, c)$ are *LS-sets*, first introduced by Luccio and Sami [103]. A set $S \subsetneq V$ is an LS-set if $c(U, S) > c(U, V \setminus S)$ for each proper subset $U \subsetneq S$. That is, the members of S are stronger connected to other members of S in terms of edge costs than to vertices that do not belong to S. Borgatti et al. [15] extensively discuss further properties of LS-sets following from this definition and the benefit of these properties in the context of network analysis and graph clustering applications. The fact that LS-sets are hierarchically nested constitutes only one of these properties. Borgatti et al. also propose an algorithm for enumerating all LS-sets in a given graph based on *λ-sets*. Interestingly, the λ-sets of Borgatti et al. are equivalent to the maximal components of Nagamochi [110], who considers this kind of sets from a theoretical point of view in the light of graph connectivity (recall Chapter 6 of Part II). The algorithm of Borgatti et al. extracts the λ-sets from a Gomory-Hu tree, which needs at least $n - 1$ cut



(a) LS-sets of the graph in Fig. 6.1(a).

(b) λ-sets of the graph G in Fig. 6.1(a).

Figure 10.2: Set of all LS-sets (extreme sets) (a) and set of all λ-sets (maximal components) (b) of the graph G in Fig. 6.1(a). Red numbers denote edge costs different from 1, dashed black lines indicate LS-sets and λ-sets. The set of LS-sets forms a proper subclass of the set of λ-sets.

computations. Nagamochi, however, presents a simple and very elegant approach to enumerate all LS-sets (Nagamochi calls them *extreme sets*) based on MA-orderings [110] that runs in $O(nm+n^2 \log n)$ time. Figure 10.2(a) depicts the LS-sets (or extreme sets) of the graph in Fig. 6.1(a) in comparison with the λ-sets (or maximal components) of the same graph (Fig. 10.2(b)). Note that LS-sets form a proper subclass of λ-sets.

In this work, we will see that the cut-based clustering algorithm of Flake et al. [42] returns not only clusters of a guaranteed expansion, but also special cohesive subsets. We will bring these special cohesive subsets into line with existing concepts of cohesive subsets as discussed by Borgatti et al. and Nagamochi, and we will show how inclusion-maximal decompositions of a graph into such subsets can be obtained from the unique-cut tree introduced in Part II, Chapter 7.

Other Significant Subgraphs. Besides λ-sets and LS-sets, many further significant substructures have been considered in graphs. A very fundamental structure are, for example, *cliques* [104]. A set of vertices in a graph is a clique if each vertex is adjacent to any other vertex in the set. The clique percolation method (PCM) of Derényi et al. [31] detects groups of maximal overlapping cliques in an undirected, unweighted graph that can be interpreted as (overlapping) clusters. This method is a special case of a more general clique-clustering framework proposed by Everett and Borgatti [40], which applies an arbitrary clustering algorithm to a weighted auxiliary graph H that represents the overlap of inclusion-maximal cliques in the input graph. In the special case considered by Derényi et al., the auxiliary graph H encodes if two inclusion-maximal cliques (of at least size k) overlap in at least $k-1$ vertices. More precisely, H is a graph where inclusion-maximal cliques in the input graph are represented by vertices and two vertices are connected by an edge if and only if the corresponding cliques share at least $k-1$ vertices. As clustering algorithm on H, Derényi et al. simply employ a *depth-first search* (DFS), which returns the connected components of H. In the original graph this still induces overlapping clusters, since inclusion-maximal cliques may also overlap in less than $k-1$ vertices. The running time of this approach is dominated by the computation of inclusion-maximal cliques, which is exponential in the number of vertices [22].

Another very fast approach, called Orca [30], relies on dense subgraphs similar to a clique. Roughly speaking, it considers a dense subgraph as a set of vertices within distance d of some center, such that each vertex in the subgraph is within distance at most d of at least a given fraction of the neighbors of the center. If the distance is set to 1, we obtain the center together with those neighbors that are connected to enough other neighbors. Mishra et al. [108] introduce overlapping (α, β)-clusters based on parameters that regulate the external sparsity and internal density. More precisely, α denotes the maximum fraction of internal vertices each external vertex connected to, while β denotes the minimum fraction of internal vertices each internal vertex is connected to. The authors investigate combinatorial properties of these clusters and show in an experiment that their algorithm detects around 90% of all $(\alpha, 1)$-clusters, which correspond to cliques. Borgatti et al. [15] introduce further relaxations of LS-sets and λ-sets and discuss several relations between cliques and further structures like k-plexes [132], n-clans, n-clubs, and n-cliques [109], which can be also seen as generalizations of cliques. We remark that the structures listed above are primarily defined for unweighted graphs. However, most of them possess a straightforward adaption to weighted graphs.

10.1 Contribution and Outline

The results in Part III are based on different publications with different coauthors. Since the arrangement of topics in these publications does not necessarily agree with the structure of chapters in this part, in this outline, we shortly discuss the underlying publications per chapter.

Chapter 10: Introduction – Graph Clustering and Cohesive Subsets

We introduce specific notations related to graph clustering and briefly present the quality measures *modularity* and *expansion* at the end of this chapter.

Chapter 11: Static Hierarchies of Cut Clusterings

We introduce the static cut-based clustering algorithm of Flake et al. [42], which plays the central role in Part III, in Section 11.1. This elegant algorithm attracts our attention, since it returns clusters with provable quality in terms of a quality measure (namely expansion) that is NP-hard to compute, it enables the user to choose from clusters of different granularity, as it can be iteratively applied in order to obtain hierarchically nested clusters, and it performed well detecting reasonable clusters in the experimental study of the authors. We further introduce *source communities*, which are a concept of cohesive subsets, and bring this concept in line with existing cohesive subsets like LS-sets, systematically listing interchangeable names, definitions, and different nesting behaviors. A full characterization of source communities based on generalized M-sets as well as a full characterization of the clusters that can be found by the algorithm of Flake et al. based on source communities are finally provided at the end of Section 11.1. These characterizations show that the clusters detected by the clustering algorithm form a proper subclass of source communities, that is, additionally to the quality guarantee proven by Flake et al. the clusters provide desirable cohesion properties.

In Section 11.2, we improve the cut-based clustering algorithm by the help of a parametric search technique. The algorithm takes an input parameter α that controls the coarseness of the resulting clusters and varying the parameter values prompts the algorithm to return a cluster hierarchy. Flake at al. refer to Gallo et al. [54] for the question how to choose α such that all possible hierarchy levels are found. However, they give no further description how to extend the approach of Gallo et al., which finds all breakpoints of α for a single parametric flow, to a fast construction of a complete hierarchy. While they simply propose a binary-search approach to find good values for α, we introduce a parametric-search approach that guarantees the completeness of the resulting hierarchy and clearly exceeds the running time of a binary-search-based approach, whose running time strongly depends on the discretization of the parameter range.

In Section 11.3 we conduct an experimental study on the guaranteed expansion of the clusters found by the cut-based clustering algorithm. This study reveals that, compared to a trivial bound, the given guarantee indeed allows for a deeper insight. Our experiment further documents that the true expansion even exceeds the guaranteed bound. In a second experiment we investigate the quality of the clusters with respect to the widely used measure modularity. In this study the clustering algorithm competes surprisingly well with a greedy modularity-based heuristic, although it is not designed to optimize modularity. This attests a high trustability of the cut-based clustering approach, confirming that the algorithm returns plausible clusters if such clusters are clearly indicated by the graph structure.

Publications. Section 11.1 and Section 11.2 are based on [73], while the characterization of the clusters as a subclass of source communities is new and not yet published. The results of Section 11.3 are published in [72]. Both publications are joint work with Michael Hamann and Dorothea Wagner.

Chapter 12: Maximum Source-Community Clusterings

The experimental evaluation of Flake et al. [42] showed that the cut-based clustering algorithm finds meaningful clusters in real-world instances, but yet, it often happens that (even in a complete hierarchy) non-singleton clusters are only found for a subgraph of the initial network, while the remaining vertices stay unclustered even on the coarsest non-trivial hierarchy level. The latter observation is also confirmed by the experiments in Section 11.3. Motivated by this observation, in Section 12.1 and Section 12.2 we develop a framework that efficiently answers the following queries: (i) Given an arbitrary source community S, what does a clustering $\Omega(S)$ (that is, a partition of the network into (disjoint) clusters) look like that consists of S and further source communities such that any source community not intersecting with S is nested in a cluster of $\Omega(S)$? In particular, $\Omega(S)$ is maximum in the sense that any clustering of source communities that contains S is hierarchically nested in $\Omega(S)$. We show that $\Omega(S)$ can be determined in linear time. (ii) Given k disjoint source communities $S_1, \dots, S_k$, which is the maximal clustering $\Omega(S_1, \dots, S_k)$ that contains the given source communities, is nested in each $\Omega(S_i)$, $i = 1, \dots, k$, and guarantees that any clustering of source communities that also contains the given source communities is nested in $\Omega(S_1, \dots, S_k)$? Computing $\Omega(S_1, \dots, S_k)$ takes $O(kn)$ time. These queries allow to examine the community structure of a given network beyond the complete clustering hierarchy found by the cut-based approach. The framework relies on precomputing a unique-cut tree (see Section 7.2), which represents all regular M-sets and can be constructed by at most $2(n-1)$ maximum-flow computations. In Section 12.3, we exemplarily apply both queries to a small real world network, thereby finding a new clustering beyond the hierarchy that contains all non-singleton clusters of the best clustering in the hierarchy but far less singletons.

Publications. The results of this chapter already appeared in a joint publication with Michael Hamann and Dorothea Wagner [73].

Chapter 13: The Unrestricted Cut-Clustering Algorithm

The hierarchical version of the cut-based clustering algorithm, as proposed by Flake et al. [42], uses U-cuts for the construction of each hierarchy level. Due to this restriction the returned clusterings are unique. However, the algorithm possibly misses convenient clusters in graphs where minimum separating cuts are not unique. We show that a restriction to U-cuts is not necessary in order to construct a clustering hierarchy, that is, permitting arbitrary minimum separating cuts in the construction is a feasible degree of freedom, which still maintains the

guaranteed quality of the clusters in terms of expansion. This makes the method more powerful, since it may choose the most appropriate cuts with respect to some special application or other objectives like, for example, temporal smoothness in a dynamic scenario.
Publications. This short chapter is an extract of a joint publication with Christof Doll and Dorothea Wagner [34].

Chapter 14:
Fully-Dynamic Hierarchies of Unrestricted Cut Clusterings

Employing the results on dynamic Gomory-Hu trees (see Chapter 8), we adapt the unrestricted cut-based clustering approach to a dynamic scenario with atomic changes. This task has been already considered by Saha and Mitra [124], however, we found their attempt erroneous beyond straightforward correction. In Section 14.1 we discuss the structural problems of their approach and give counterexamples proving its incorrectness. In Section 14.2 we propose update procedures for maintaining a single clustering of an evolving graph. In Section 14.3 we exploit the degree of freedom gained by the loss of the restriction to U-cuts in order to prove optimal temporal smoothness of the clusterings returned by our procedures. Furthermore, our procedures provide a high potential for saving cut computations, which we discuss in Section 14.4 and confirm by a brief experiment. In Section 14.5 we extend our update approach to hierarchies of clusterings, resulting in update procedures that provide even more potential for savings. These procedures in parts employ the update procedures of the previous section inheriting the guarantee of temporal smoothness.
Publications. Sections 14.1 to 14.4 are based on joint work with Robert Görke and Dorothea Wagner [61, 62], however, in this thesis we are able to describe the update procedures much simpler by referring to the comprehensive results on dynamic Gomory-Hu trees. Furthermore, we modified the procedures such that they indeed guarantee optimal temporal smoothness. In the previous publications only *stability* is proven. We call an update procedure stable if it returns the previous clusterings whenever the previous clustering remains valid after a change in the underlying graph. We note that some aspects of this sections also appear in the thesis of Robert Görke [60]. Section 14.5 is based on joint work with Christof Doll and Dorothea Wagner [34], while the procedures for maintaining whole hierarchies are again simplified and improved such that temporal smoothness is guaranteed.

10.2 Preliminaries

At this point we introduce only notation that is exclusively used in the context of graph clustering. More general notations, which also appear in the following chapters, are introduced in Section 6.2 if they are related to Gomory-Hu trees, U-cuts or M-sets, or in Section 1.2 if they concern general concepts like static or dynamic graphs, cuts or connectivity. Our understanding of a *clustering* $\Omega(G)$ of an undirected weighted graph $G = (V, E, c)$ is a partition of the vertex set V into proper subsets $C^1, \dots, C^k$, which define vertex-induced subgraphs, called *clusters*, usually conforming to the paradigm of intra-cluster density and inter-cluster sparsity. We also say that $\Omega(G)$ *contains* the clusters $C^1, \dots, C^k$. In the context of dynamic graphs and atomic edge changes of an edge $\{b, d\}$ we particularly designate C^b, C^d and $C^{b,d}$ containing b and d, respectively. A set of disjoint subsets $C^1, \dots, C^k$ with $\bigcup_{i=1}^k C^i \neq V$ can be always extended to a clustering by simply identifying the missing vertices as singleton clusters. A cluster is called *trivial* if it corresponds to a connected component. Note that V is not considered as a cluster, not even if G is connected, and thus, V forms a connected component. A vertex that forms a non-trivial singleton cluster we consider as *unclustered.* A clustering is *trivial* if it consists of trivial clusters or if $k = n$, that is, all vertices are unclustered. A *hierarchy of clusterings* is a sequence $\Omega_1(G) \leq \cdots \leq \Omega_r(G)$ such that $\Omega_i(G) \leq \Omega_j(G)$ implies that each cluster in $\Omega_i(G)$ is a subset of a cluster in $\Omega_j(G)$. We say $\Omega_i(G) \leq \Omega_j(G)$ are *hierarchically nested.* In order to stress that the clusters of $\Omega_i(G)$ are proper subsets of the clusters in $\Omega_j(G)$, we write $\Omega_i(G) < \Omega_j(G)$. Similarly, we call a set of subsets of V *hierarchically nested* if every two subsets are either nested or disjoint. Note that the indices indicating the hierarchy level of a clustering are denoted at the bottom, while the indices that distinguish the clusters in a clustering are denoted at the top. Accordingly, we denote a cluster C^i in a clustering $\Omega_j(G)$ by C^i_j. A clustering $\Omega(G)$ is *inclusion-maximal* with respect to a property $\mathcal{P}$ if there is no other clustering $\Omega'(G)$ with property $\mathcal{P}$ and $\Omega(G) \leq \Omega'(G)$. It is further *maximum* if it is the only inclusion-maximal clustering among a given set of clusterings. A *quality measure* for clusterings is a mapping to real numbers. Depending on the measure, either high or low values correspond to high quality. In this work we consider three quality measures, modularity, intra-cluster expansion and inter-cluster expansion. The former two indicate high quality by high values. Inter-cluster expansion indicates good quality by low values.

Modularity. Modularity was first introduced by Newman and Girvan [116] as a quality measure for disjoint clusters in an undirected, unweighted graph $G = (V, E)$. The modularity of the corresponding clustering is based on the total edge cost in G that is covered by clusters. The values range from -0.5 to 1, with 1 indicating the best quality. The measure expresses the significance of a given clustering in G compared to the same clustering in a random graph of the vertex set V. Formally, the modularity $\mathcal{M}(\Omega)$ of a clustering Ω is defined as

$$\mathcal{M}(\Omega) := \sum_{C \in \Omega} c(E_C)/c(E) - \sum_{C \in \Omega} (\sum_{v \in C} \deg(v))^2 / 4c(E)^2$$

where E_C denotes the set of edges with both endpoints in C, and the cost of a set E' of edges is denoted by $c(E') := \sum_{e \in E'} c(e)$. The first sum is also known as *coverage* of Ω in G, since it describes the ratio of edges covered by clusters. The second sum describes the expected coverage of Ω in a random graph on V whose expected vertex degrees corresponds to the degrees in G.

Modularity can be easily adapted to undirected, weighted graphs. Furthermore, the change of modularity that occurs if Ω slightly varies, can be expressed in a closed form and computed efficiently. Hence, modularity is particularly suitable for being optimized by a greedy heuristic. An exact maximization of modularity is NP-hard [20]. One of the fastest greedy approaches, often called the Louvain method, was proposed by Blondel et al. [14]. This method improves the modularity of a clustering by merging given subgraphs and moving vertices, thus building new subgraphs representing a clustering of higher modularity. It finally returns a hierarchy of clusterings where the top level represents a local maximum of all possible clusterings with respect to modularity. The authors also provide an excellent free accessible implementation, which might be one reason why today the Louvain method is one of the most popular and widespread graph clustering algorithms. As a consequence, modularity became a very popular quality measure, which nowadays is often used to evaluate also clusterings resulting from algorithms that do not explicitly maximize this measure. This trend is further fanned by the fact that modularity is close to human intuition of clustering quality. However, it also has some specific drawbacks as, for example, the resolution limit explored in [47]. That is, modularity-based methods tend to detect subgraphs of specific size categories depending on the size of the network. Further heuristic algorithms to optimize modularity are based on greedy agglomeration [116, 25], spectral division [115, 139], simulated annealing [64, 120] or extremal optimization [35].

Expansion and Conductance. Originally, expansion and conductance are measures for cuts in weighted graphs. Considering a cluster C as a subgraph of an underlying graph $G = (V, E, c)$, the expansion $\psi(U, C \setminus U)$ of a cut $(U, C \setminus U)$ in C is defined as

$$\psi(U, C \setminus U) := \frac{c(U, C \setminus U)}{\min\{|U|, |C \setminus U|\}}$$

and expresses the cost of the cut in relation to the smaller cut side. The conductance $\phi(U, C \setminus U)$ is a generalization of the expansion that considers weighted degrees instead of single vertices in the sets of the denominator:

$$\phi(U, C \setminus U) := \frac{c(U, C \setminus U)}{\min\{\deg(U), \deg(C \setminus U)\}}.$$

In order to derive a measure for the cohesion of a (sub)graph, we simply consider the minimum expansion value or the minimum conductance value of all cuts in the (sub)graph. The expansion of a (sub)graph or cluster C is then denoted by $\psi(C)$, the conductance of C by $\phi(C)$. Unfortunately, this leads to measures that are already NP-hard to compute for a given graph. Nevertheless, Kannan et al. [87] develop the following bicriteria measure for clusterings of disjoint clusters based on the conductance of subgraphs. They say the quality of a clustering Ω is the better

- the higher $\phi(\Omega) := \min_{C \in \Omega} \phi(C)$ (intra-cluster quality) and
- the lower $\overline{\phi}(\Omega) := \sum_{C \in \Omega} c(C, V \setminus C)/2c(E)$ (inter-cluster quality)

is. Unsurprisingly, it is also NP-hard to find, for a given value α, a clustering Ω with $\phi(\Omega) \geq \alpha$ that minimizes $\overline{\phi}(\Omega)$. However, with the help of spectral clustering techniques, the solutions of this problem can be approximated with good guarantees [87]. The cut-based algorithm of Flake et al. [42], which we consider in this work, provides a similar quality guarantee regarding expansion. The difference is, that it is no approximation algorithm but a parametric algorithm, and the quality guarantee depends on the parameter. More precisely, for each clustering Ω found by this algorithm it holds that

- $\psi(\Omega) := \min_{C \in \Omega} \psi(C) \geq \alpha$ (intra-cluster quality) and
- $\overline{\psi}(\Omega) := \max_{C \in \Omega} c(C, V \setminus C)/|V \setminus C| \leq \alpha$ (inter-cluster quality),

where α is the input parameter. Moreover, the clusterings for different values of α are hierarchically nested.

CHAPTER 11

Static Hierarchies of Cut Clusterings

Although the cut-based clustering algorithm of Flake et al. [42] provides already many desirable features—it returns clusters of guaranteed quality, admits to find hierarchically nested clusters, and, last but not least, proved successful in the experiments conducted by the authors—we will see that there is still room for improvement and further development. In this chapter, we analyze the algorithm of Flake et al. from a theoretical and a practical point of view in the context of static graphs and present an improved strategy for detecting even better hierarchies of clusterings of good quality.

11.1 Cut Clusterings and Source Communities

In the following, we describe the cut-based clustering algorithm of Flake et al. [42] and show that, besides the elegant quality guarantee, the clusters also represent special cohesive subsets. We also discuss the relations of these subsets to other cohesive subsets like LS-sets and λ-sets.

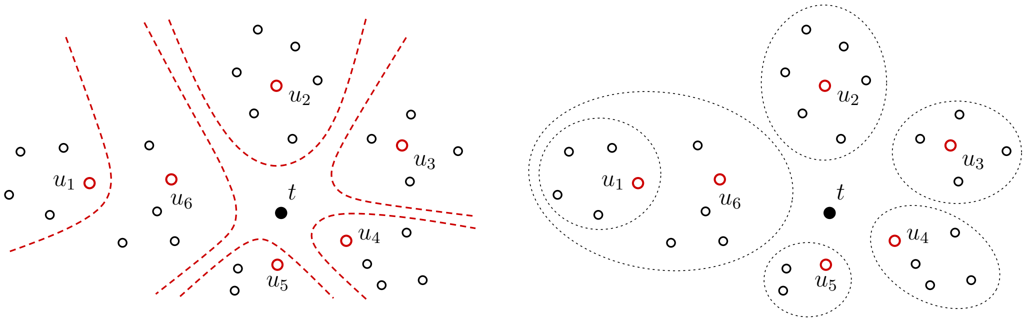
11.1.1 The Static Cut-Clustering Algorithm

Inspired by the work of Kannan et al. [87], Flake et al. [42] presented a cut-based clustering algorithm that exploits the properties of minimum separating cuts together with an input parameter α in order to find hierarchically nested clusterings where each cluster C has the following very reasonable bottleneck-property: For each $U \subsetneq C$ it holds

$$\frac{c(C, V \setminus C)}{|V \setminus C|} \leq \quad \alpha \quad \leq \frac{c(U, C \setminus U)}{\min\{|U|, |C \setminus U|\}}. \tag{11.1}$$

According to the left side of this inequality, which we denote by $\overline{\psi}(C)$, separating a cluster C from the rest of the graph costs at most $\alpha|V \setminus C|$, which guarantees a certain inter-cluster sparsity. The right side further guarantees a good intra-cluster density in terms of the expansion $\psi(U, C \setminus U)$ of an arbitrary cut $(U, C \setminus U)$, saying that splitting a cluster C into U and $C \setminus U$ costs at least $\alpha \min\{|U|, |C \setminus U|\}$. Hence, a subgraph C that is supposed to be a candidate for a cluster must be very tight, providing an expansion $\psi(C)$ that exceeds a given bound. For a clustering Ω this implies the guaranteed inter- and intra-cluster quality introduced Chapter 10

$$\overline{\psi}(\Omega) \leq \alpha \leq \psi(\Omega)$$



(a) Sequence $\mathfrak{uc}(u_1,t),\dots,\mathfrak{uc}(u_6,t)$ of U-cuts in G_α computed by CutC in order to find related M-sets.

(b) Related M-sets $\mathfrak{m}(u_1,t),\dots,\mathfrak{m}(u_6,t)$. Note that $\mathfrak{m}(u_1,t)$ is no cluster, since it is nested in $\mathfrak{m}(u_6,t)$.

Figure 11.1: Schematic illustration of U-cuts and M-sets computed by CutC.

with $\overline{\psi}(\Omega) := \max_{C\in\Omega}\overline{\psi}(C)$ and $\psi(\Omega) := \min_{C\in\Omega}\psi(C)$. In the following, we call $\overline{\psi}(C)$ and $\overline{\psi}(\Omega)$ the *inter-cluster expansion* of a cluster C and a clustering Ω, respectively, and analogously, $\psi(C)$ and $\psi(\Omega)$ the *intra-cluster expansion*. Flake et al. further call their algorithm *cut-clustering algorithm*, and according to this, we call a clustering that is found by the cut-clustering algorithm a *cut clustering*. We will see that, although the cut-clustering algorithm is not deterministic, it returns a unique cut clustering for each parameter value.

Flake et al. develop their parametric cut-clustering algorithm step-by-step, each time simplifying a previous version. They start with a version that involves the computation of a Gomory-Hu tree [59], however, their final approach just uses U-cuts with respect to cut pairs that all contain a common vertex. Hence, in most cases the algorithm computes only a partial Gomory-Hu tree by the use of special split cuts. We describe the cut-clustering algorithm more directly based on U–cuts and corresponding M–sets and refer to this method as CutC. Given a graph $G = (V,E,c)$ and a parameter $\alpha \in \mathbb{R}_0^+$, as a preprocessing step, augment G by inserting an artificial vertex t and connecting t to each vertex in G by an edge of cost α. Denote the resulting graph by $G_\alpha = (V_\alpha, E_\alpha, c_\alpha)$. Then apply CutC (Algorithm 7) by iterating V and computing the M-set $\mathfrak{m}(u,t)$ for each vertex u not yet contained in a previously computed cluster. The vertex u becomes the *representative* of the newly computed cluster $\mathfrak{m}(u,t)$ (line 4). Note that $\mathfrak{m}(u,t)$ might be equivalent to an M-set $\mathfrak{m}(u',t)$ with respect to another vertex $u' \neq u$. Hence, if the algorithm chooses u' instead of u the same cluster is found, and u' is also a representative of this cluster. Consequently, each cluster can have several representatives. Since M-sets with respect to a common vertex t are either disjoint or nested (Lemma 7.6(1),(2i)), we finally get a set of M-sets in G_α, in which the inclusion-maximal M-sets together decompose V. This induces a unique clustering $\Omega(G)$ with respect to α (see also Fig. 11.1).

Algorithm 7: CutC

```
Input: Graph $G_\alpha = (V_\alpha, E_\alpha, c_\alpha)$
1 $\Omega \leftarrow \emptyset$
2 while $\exists\, u \in V_\alpha \setminus \{t\}$ do
3     $C^u \leftarrow \mathfrak{m}(u,t)$ in $G_\alpha$
4     $r(C^u) \leftarrow u$
5     forall the $C^i \in \Omega$ do
6         if $r(C^i) \in C^u$ then $\Omega \leftarrow \Omega \setminus \{C^i\}$
7     $\Omega \leftarrow \Omega \cup \{C^u\}$ ; $V_\alpha \leftarrow V_\alpha \setminus C^u$
8 return $\Omega$
```

Applying CutC iteratively with decreasing α yields a hierarchy of at most n different clusterings (see Figure 11.2), where the clustering quality on level i depends on α_i. This is due to a special nesting property for different parameter values that is also used by Gallo et al. [54]

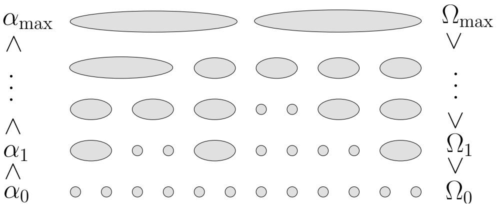


FIGURE 11.2: Hierarchy of cut clusterings returned by CUTC for decreasing α. Note that $\alpha_{\max} < \alpha_0$ whereas $\Omega_{\max} > \Omega_0$.

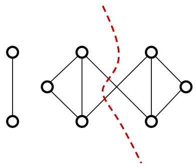


FIGURE 11.3: Graph decomposed into two web communities by the dashed red line. Left web community is disconnected.

in the context of parametric maximum-flow algorithms. Let C_1 denote the cluster $\mathfrak{m}(u,t)$ with respect to G_{α_1} and C_2 the cluster $\mathfrak{m}(u,t)$ with respect to G_{α_2}. Then it is $C_1 \subseteq C_2$ if $\alpha_1 \geq \alpha_2$. The hierarchy is bounded by two trivial clusterings, which we already know in advance. The clustering at the top consists of the connected components of G and is returned by CUTC for $\alpha_{\max} = 0$, the clustering at the bottom consists of singletons and occurs if we choose α_0 equal to the maximum edge cost in G. If G is connected, the algorithm returns $\{V\}$ at the top level, which we do not consider as a cluster, since it is no proper subset of the vertices in G.

11.1.2 Source Communities and Other Cohesive Subsets

In the context of large web-based graphs, Flake et al. [41] consider cohesive subsets in which each vertex u is predominantly connected to the remaining vertices of the subset, that is, for each vertex $u \in S \subsetneqq V$ holds $c(\{u\}, S \setminus \{u\}) > c(\{u\}, V \setminus S)$. Flake et al. call these subsets *web communities* and claim that the clusters found by the cut-clustering algorithm are almost web communities in the sense that all vertices apart from the particular representative are predominantly connected. However, this is not a full characterization of the clusters. In the following, we thus introduce a further type of cohesive subsets, which we call *source communities*, and based on these, we fully characterize the clusters that can be found by the cut-clustering algorithm. We will see that the clusters form a class of special source communities whose inter-cluster quality $\overline{\psi}$ is not only bounded by α but also by values independent from α, depending on the graph structure of G and the individual cluster. Before we prove this characterization of the clusters in Section 11.1.3, in this section, we fully characterize the class of source communities and bring the concept of source communities in line with other existing definitions of cohesive subsets like LS-sets, which we have already seen in the introduction of this part (Chapter 10).

Source Communities. A set $S \subsetneqq V$ is a source community with *source* $s \in S$ if $c(U, S \setminus U) > c(U, V \setminus S)$ for all $U \subsetneqq S \setminus \{s\}$ and $c(\{s\}, S) > c(\{s\}, V \setminus S)$. Obviously, if G is disconnected, the connected component that contains s is the largest possible source community with source s. Note that even if G is connected, V is no source community, since the definition requires $S \subsetneqq V$. Furthermore, we define $c(\{s\}, \{s\}) := \infty$ in order to also include singletons to the set of source communities, which is a usual approach in the context of cohesive subsets. The concept of source communities can be established between web communities and LS-sets, omitting the bad and combining the desired properties of both. One bad property of web communities as defined by Flake et al. [41] is, that they are not necessarily connected (see Fig. 11.3). On the other hand, however, they may be overlapping, nested or disjoint, and thus, suffice the notion of naturally inhered groups in networks. In contrast, LS-sets as defined by Borgatti et al. [15] extend the predominant connectivity from vertices to arbitrary subsets, and thus, satisfy a stricter condition

Table 11.1: Overview of different concepts of cohesive subsets. Web communities/α-sets and source communities are introduced in this section, LS-sets/extreme sets are introduced in Chapter 10, maximal components/λ-sets are introduced in Chapter 6.

A subgraph $S \subsetneq V$ is a			structure
LS-set [15] extm. set [110]	$\forall U \subsetneq S$	$c(U, S \setminus U) > c(U, V \setminus S)$	hierarchically nested
source com.	$\exists s \in S$: $\forall U \subsetneq S \setminus \{s\}$ and for s	$c(U, S \setminus U) > c(U, V \setminus S)$ $c(\{s\}, S) > c(\{s\}, V \setminus S)$	overlapping, nested, disjoint
α-set [15] web com. [41]	$\forall u \in S$	$c(\{u\}, S \setminus \{u\}) > c(\{u\}, V \setminus S)$	overlapping, nested, disjoint
λ-set [15] max. comp. [110]	$\forall u \in S,\ v \in V \setminus S$	$\min_{x,y \in S} \lambda_G(x,y) > \lambda_G(u,v)$	hierarchically nested

that on the one hand guarantees connectivity but on the other hand restricts the appearance of LS-sets in real data to only rare occasions. In particular, LS-sets are hierarchically nested but not overlapping. For a further discussion on advantages and disadvantages of LS-sets and other cohesive sets see also Borgatti et al. [15]. Unlike LS-sets, we will see that source communities may also overlap, as web communities, while in contrast to web communities, the connectivity of source communities follows directly from the definition, as it is the case also for LS-sets.

Table 11.1 systematically lists the different cohesive subsets considered in this work with their definitions and their interchangeable names used by different authors. As already mentioned before, Nagamochi [110] calls the LS-sets of Borgatti et al. extreme sets. On the other hand, Borgatti et al. introduced the web communities of Flake et al. already in 1990 as *α-sets*. While decomposing a graph into k web communities or α-sets is NP-hard [42], Nagamochi is able to enumerate all extreme sets or LS-sets in $O(nm + n^2 \log n)$ time with the help of maximum adjacency orderings (MA-orderings). The latter form a subset of the maximal components in a graph (recall Fig. 6.2(a)), and are thus hierarchically nested but not overlapping. Borgatti et al. use the term λ-sets to describe the maximal components of Nagamochi. Maximal components or λ-sets subsume vertices that are not separated by cuts cheaper than a certain lower bound and can be deduced from a Gomory-Hu tree, whose construction needs $n - 1$ maximum-flow computations [59]. They are used for example in the context of image segmentation by Wu and Leahy [141]. In social networks, we might be also interested in communities that surround a designated vertex, for instance a central person. Complying with this view, source communities describe vertex sets where each subset that does not contain a designated vertex (which is predominantly connected to the group) is predominantly connected to the remainder of the group and the designated vertex. The members of a source community can be interpreted as *followers* of the source in that sense that each subgroup feels more attracted by the source (and other group members) than by the vertices outside of the group. This predominant connectivity of source communities implements a close relation to minimum separating cuts. In fact, source communities are characterized as follows.

Lemma 11.1. *A set $S \subsetneq V$ is a source community with source $s \in S$ if and only if there exists a set $T \subseteq V \setminus S$ such that $(S, V \setminus S) = \mathfrak{uc}(s, T)$ and thus $S = \mathfrak{m}(s, T)$. That is, there is a one-to-one correspondence between source communities and generalized M-sets in any undirected, weighted graph $G = (V, E, c)$.*

Proof. Let $S \subsetneq V$ denote a source community with source s. If $S = \{s\}$, then $(\{s\}, V \setminus \{s\})$ is the only cut that separates $\{s\}$ and $V \setminus \{s\}$, and is thus the minimum s-T-cut with $T = V \setminus \{s\}$. Hence, it is $\{s\} = \mathfrak{m}(s,T)$. In any other case, $(S, V \setminus S)$ must be a minimum s-T-cut for $T = V \setminus S$, since a cheaper s-T-cut would split S into U and $S \setminus U \ni s$ with $c(U, S \setminus U) < c(U, V \setminus S)$, which contradicts the definition of source communities. The cut $(S, V \setminus S)$ further minimizes the number of vertices on the cut side of s, since a minimum s-T-cut with a smaller cut side of s would induce a set $U \subsetneq S$, $s \notin U$, with $c(U, S \setminus U) = c(U, V \setminus S)$, which is again a contradiction. Thus, it is $S = \mathfrak{m}(s,T)$ with $T = V \setminus S$.

Now consider a generalized M-set $\mathfrak{m}(s,T) =: S$ and the corresponding U-cut $\mathfrak{uc}(s,T) = (S, V \setminus S)$. We show that S is a source community with source s. If $S = \{s\}$, it is a source community by definition. In any other case, we observe the following. Since $(S, V \setminus S)$ is a minimum s-T-cut with $T \subseteq V \setminus S$, it is $c(U, S \setminus U) \geq c(U, V \setminus S)$ for all $U \subsetneq S \setminus \{s\}$. Otherwise, if there was a set U with $c(U, S \setminus U) < c(U, V \setminus S)$, then $(S \setminus U, V \setminus (S \setminus U))$, which also separates s and T, would be a cheaper s-T-cut. Since $(S, V \setminus S)$ further minimizes the number of vertices on the cut side of s, it is $c(U, S \setminus U) > c(U, V \setminus S)$ for all $U \subsetneq S \setminus \{s\}$. Otherwise, $(S \setminus U, V \setminus (S \setminus U))$, would be a minimum s-T-cut with a smaller side containing s. Hence, S is a source community with source s. □

Due to this correspondence between source communities and generalized M-sets, it follows directly from Lemma 7.6 that source communities may be overlapping, nested or disjoint. Furthermore, each source community can have several sources, since the corresponding M-set may have several representations resulting from generalized U-cuts with respect to different (generalized) cut pairs. We use the notation of M-sets also for source communities, that is, we denote a source community with source s and with respect to $T \subsetneq V$ by $\mathfrak{m}(s,T)$ and call $\mathfrak{m}(T,s)$ the opposite source community. We finally discuss some further properties of source communities and their interpretation in the context of network analysis. Suppose a source community $\mathfrak{m}(s,T)$ whose corresponding U-cut $\mathfrak{uc}(s,T)$ is not the only minimum s-T-cut in the underlying graph G. That is, the opposite U-cut $\mathfrak{uc}(T,s)$ differs from $\mathfrak{uc}(s,T)$ and $\mathfrak{m}(s,T) \neq \mathfrak{m}(T,s)$. Consequently, $X := V \setminus (\mathfrak{m}(s,T) \cup \mathfrak{m}(T,s)) \neq \emptyset$ and the vertices in X are neither predominantly connected within $\mathfrak{m}(s,T) \cup X$ nor within $\mathfrak{m}(T,s) \cup X$, that is, for all $U \subseteq X$ the set U might be as strongly connected to $V \setminus (\mathfrak{m}(s,T) \cup X)$ as to $\mathfrak{m}(s,T) \cup X$ (analogously for $\mathfrak{m}(T,s)$). In a social network this can be interpreted as follows. Whenever s and the group T become rivals, the network decomposes into followers of s (in $\mathfrak{m}(s,T)$), followers of T (in $\mathfrak{m}(T,s)$) and possibly some *indecisive* individuals in X. Figure 11.4 exemplarily shows two indecisive vertices in the (unweighted[1]) karate club network gathered by Zachary [142]. Another interesting behavior of members in source communities follows from the following observation. Let S_1 denote a source community with source s and $T_2 \subsetneq V$ an arbitrary set. If $T_2 \cap S_1 = \emptyset$, then $S_1 \subseteq \mathfrak{m}(s,T_2) =: S_2$. Now consider a source community $S_1 = \mathfrak{m}(s,T_1)$ and a further set of possible rivals $T_2 \subsetneq V$. If T_2 is outside of S_1 and further $S_1 \neq \mathfrak{m}(s,T_2)$, the observation states that S_1 is a proper subset of $S_2 = \mathfrak{m}(s,T_2)$. As described above, the followers

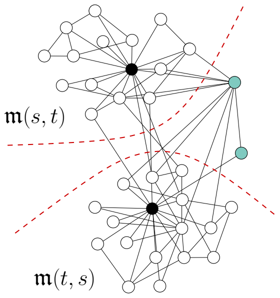


FIGURE 11.4: Indecisive vertices (green) with respect to rivals s and t (black) in the Zachary network.

---

[1] Zachary considers the weighted network and therein the minimum cut that separates the two central vertices of highest degree (black). In the weighted network this cut is unique.

of T_1 (in $\mathfrak{m}(T_1, s)$) thus break with the members in S_1 if s and T_1 become rivals. However, if there occurs a second rival group T_2 outside of S_1, then also vertices from outside of S_1 become followers of s, maybe even vertices of T_1.

11.1.3 Characterizing the Cut-Clustering Clusters

The following theorem provides a full characterization of the clusters that can be found by the cut-clustering algorithm. The clusters are special source communities, and thus, form a subclass of the source communities. We further give a counterexample that shows that this subclass is also a proper subclass. Let $G = (V, E, c)$ denote an undirected, weighted graph and let G_α denote the extended graph used by the cut-clustering algorithm, which in particular contains the artificial vertex t.

Theorem 11.2. *For a set $S \subseteq V$ there exists $s \in S$ and $\alpha \in \mathbb{R}_0^+$ such that $\mathfrak{m}(s, t) = S$ in G_α (that is, S can be found by the cut-clustering algorithm) if and only if S is a source community with source s in G with the following additional property: $c_2 < c_1$ where*
$c_1 := \min_{U \subsetneq S, s \in U} [c(S, V \setminus S) - c(U, V \setminus U)]/(|U| - |S|)$ *and*
$c_2 := \max\{c_3, c_4\}$ *with*
$c_3 := c(S, V \setminus S)/(|V| - |S|)$ *and*
$c_4 := \max_{S \subsetneq U \subsetneq V} [c(S, V \setminus S) - c(U, V \setminus U)]/(|U| - |S|)$

Note that this implies a correspondence between the representatives of the clusters and the sources of the corresponding source communities. Furthermore, we can immediately deduce two further bounds on the inter-cluster expansion $\overline{\psi}(C)$ of a cluster C from the correspondence to such a special source community S. These bound depend on the individual cluster, but not on α. First, observe that c_3 exactly represents $\overline{\psi}(C)$ and thus it holds $\overline{\psi}(C) < c_1$. The second bound results from the definition of source communities. Let C denote a cluster with representative r, which corresponds to a source community with source s. For C holds that $c(C \setminus \{r\}, \{r\}) > c(C \setminus \{r\}, V \setminus C)$ and $c(\{r\}, C) > c(\{r\}, V \setminus C)$. Adding up both inequalities, we see that $2c(\{r\}, C) > c(C, V \setminus C)$. This yields $\overline{\psi}(C) < 2c(\{r\}, C)/|V \setminus C|$.

Before we prove Theorem 11.2, we show the following lemma on the intersection behavior of two source communities S_1 and S_2 in a graph G with the same source s. Note that for such communities, we know that they are M-sets $\mathfrak{m}(s, T_1)$ and $\mathfrak{m}(s, T_2)$ in G, but we do not explicitly know the sets T_1 and T_2. Thus, we cannot directly apply Lemma 7.6.

Lemma 11.3. *Let S_1 and S_2 denote two source communities with the same source s. Then it holds $S_1 \subseteq S_2$ or $S_2 \subseteq S_1$.*

Proof. If at least one source community is a singleton, the lemma immediately holds. Thus, suppose both source communities consist of at least two vertices. We assume that S_1 and S_2 overlap without being nested and show that this leads to a contradiction. We distinguish two cases.

Case 1: $S_1 \cap S_2 = \{s\}$. Due to the definition of a source community, it holds
(i) $c(\{s\}, S_1) = c(\{s\}, S_1 \setminus S_2) > c(\{s\}, V \setminus S_1) \geq c(\{s\}, S_2 \setminus S_1)$, and analogously,
(ii) $c(\{s\}, S_2) = c(\{s\}, S_2 \setminus S_1) > c(\{s\}, V \setminus S_2) \geq c(\{s\}, S_1 \setminus S_2)$
This reveals a contradiction saying that $c(\{s\}, S_1 \setminus S_2) > c(\{s\}, S_2 \setminus S_1)$ and vice versa.

Case 2: $|S_1 \cap S_2| > 1$. Now we consider the set $U := (S_1 \cap S_2) \setminus \{s\}$. Due to the definition of a source community, it holds

(i) $c(U, S_1 \setminus U) > c(U, V \setminus S_1) \geq c(U, S_2 \setminus U)$, and analogously,
(ii) $c(U, S_2 \setminus U) > c(U, V \setminus S_2) \geq c(U, S_1 \setminus U)$
This reveals a contradiction saying that $c(U, S_1 \setminus U) > c(U, S_2 \setminus U)$ and vice versa. □

Proof of Theorem 11.2. First direction: We first consider the M-set $\mathfrak{m}(s,t) =: S$ in G_α (which is found as a cluster for this fixed α) and show that it is a source community with source s in G. In a second step, we show that $c_2 \leq \alpha < c_1$ holds. This finishes the first direction of the proof.

If $S = \{s\}$, it obviously is an M-set in G, namely $\mathfrak{m}(s, V \setminus \{s\})$. According to the correspondence of generalized M-sets and source communities (Lemma 11.1), it is also a source community with source s in G. Otherwise, $S = \mathfrak{m}(s,t)$ corresponds to a source community with source s in G_α, again by Lemma 11.1, and thus, due to the definition of a source community, it holds for each $U \subseteq S \setminus \{s\}$ that $c_\alpha(U, S \setminus U) > c_\alpha(U, V_\alpha \setminus S)$, and further, that $c_\alpha(\{s\}, S) > c_\alpha(\{s\}, V_\alpha \setminus S)$. We observe that $c_\alpha(U, S \setminus U) = c(U, S \setminus U)$ and $c_\alpha(\{s\}, S) = c(\{s\}, S)$ also in G, and $c_\alpha(U, V_\alpha \setminus S) = c(U, V_\alpha \setminus S) + |U|\alpha$ and $c_\alpha(\{s\}, V_\alpha \setminus S) = c(\{s\}, V \setminus S) + \alpha$. From this, it immediately follows that, according to the definition of source communities, S is also a source community with source s in G. Now we prove by contradiction that $c_2 \leq \alpha < c_1$.

Suppose $\alpha \geq c_1$. Then there would exist a proper subset U of S with $s \in U$ such that $(|U| - |S|)\alpha \leq c(S, V \setminus S) - c(U, V \setminus U)$, since $|U| - |S| < 0$. This yields $c(U, V \setminus U) + |U|\alpha \leq c(S, V \setminus S) + |S|\alpha$, which means $c_\alpha(U, V_\alpha \setminus U) \leq c_\alpha(S, V_\alpha \setminus S)$ contradicting the fact that $S = \mathfrak{m}(s,t)$, that is, S would not have been found by the algorithm.

Suppose $\alpha < c_2$. We prove this case for $c_4 \geq c_3$. The proof for $c_3 > c_4$ is analogous. So if $\alpha < c_4$, there would exist a set $U \subsetneq V$ with $S \subsetneq U$ such that $(|U| - |S|)\alpha < c(S, V \setminus S) - c(U, V \setminus U)$, since $|U| - |S| > 0$. This yields $c(U, V \setminus U) + |U|\alpha < c(S, V \setminus S) + |S|\alpha$, which means $c_\alpha(U, V_\alpha \setminus U) < c_\alpha(S, V_\alpha \setminus S)$ contradicting again the fact that $S = \mathfrak{m}(s,t)$.

Second direction: We consider a source community S with source s in G that has the additional property stated in Theorem 11.2. We show that S can be found by the cut-clustering algorithm, more precisely, that for $c_2 \leq \alpha < c_1$ it holds $S = \mathfrak{m}(s,t)$ in G_α.

In G_α, $(S, V_\alpha \setminus S)$ separates s and t, and is thus at least an s-t-cut. Now suppose, S is not the (unique) M-set $\mathfrak{m}(s,t)$ in G_α. Then, another set $U \subseteq V$ with $s \in U$ must be this M-set. From the first part proven for the first direction it follows that any candidate for such a set U is also a source community with source s in G. That is, S and the candidate U are both source communities in G with the same source s, and are thus nested, according to Lemma 11.3, that is, $U \subseteq S$ or $S \subseteq U$. Since we assume $U \neq S$, we even get $U \subsetneq S$ or $S \subsetneq U$. In the following, we distinguish both cases and show that if we choose in each case $c_2 \leq \alpha < c_1$, then S becomes the M-set $\mathfrak{m}(s,t)$ in G_α instead of U.

Case 1: $U \subsetneq S$. With $\alpha < c_1$, it holds in particular that $(|U| - |S|)\alpha > c(S, V \setminus S) - c(U, V \setminus U)$, since $|U| - |S| < 0$. This yields $c(U, V \setminus U) + |U|\alpha > c(S, V \setminus S) + |S|\alpha$, which means $c_\alpha(U, V_\alpha \setminus U) > c_\alpha(S, V_\alpha \setminus S)$. Thus, $(U, V_\alpha \setminus U)$ is no minimum s-t-cut in G_α.

Case 2: $S \subsetneq U$. Here we consider $\alpha \geq c_2 = \max\{c_3, c_4\}$. If $U = V$ it thus holds that $(|V| - |S|)\alpha \geq c(S, V \setminus S)$, since $|V| - |S| > 0$. This yields $|V|\alpha \geq c(S, V \setminus S) + |S|\alpha$, which means $c_\alpha(V, \{t\}) \geq c_\alpha(S, V_\alpha \setminus S)$. If $U \subsetneq V$, it holds that $(|U| - |S|)\alpha \geq c(S, V \setminus S) - c(U, V \setminus U)$, since $|U| - |S| > 0$. This yields $c(U, V \setminus U) + |U|\alpha \geq c(S, V \setminus S) + |S|\alpha$, which means $c_\alpha(U, V_\alpha \setminus U) \geq c_\alpha(S, V_\alpha \setminus S)$. Thus, in both cases $(U, V_\alpha \setminus U)$ might be indeed a minimum s-t-cut in G_α, but with a larger side of s.

In total we see that for $c_2 \leq \alpha < c_1$, there is no other cut in G_α than $(S, V_\alpha \setminus S)$ that could be the U-cut $\mathfrak{uc}(s,t)$, and thus, it is $S = \mathfrak{m}(s,t)$. □

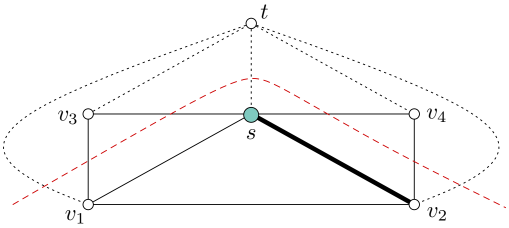


FIGURE 11.5: counterexample G (solid lines) proving that the clusters that are found by the cut-clustering algorithm form a proper subclass of the source communities. Artificial edges (dotted lines) in G_α are weighted with α, fat solid edge in G with 2, remaining edges with 1. Dashed red line indicates U-cut $\mathfrak{uc}(S, V \setminus S)$ with $S = \{v_1, v_2, s\}$ in G.

Counterexample. The following counterexample finally shows that the clusters found by the cut-clustering algorithm form a proper subclass of the source communities. More precisely, it shows that in the graph $G = (V, E, c)$ depicted by Fig. 11.5, the set $S = \{v_1, v_2, s\}$ is a source community that never becomes the M-set $\mathfrak{m}(s,t)$ in G_α.

Claim 1: The set S is a source community with source s in G. We check the definition of a source community. The subsets in S that do not contain s are

$$\begin{aligned} U_1 = \{v_1\} &\quad \text{with} \quad c(U_1, S \setminus U_1) = 2 > 1 = c(U_1, V \setminus S) \\ U_2 = \{v_2\} &\quad \text{with} \quad c(U_2, S \setminus U_2) = 3 > 1 = c(U_2, V \setminus S) \\ U_3 = \{v_1, v_2\} &\quad \text{with} \quad c(U_3, S \setminus U_3) = 3 > 2 = c(U_3, V \setminus S) \end{aligned}$$

Furthermore, it holds $c(\{s\}, S) = 3 > 2 = c(\{s\}, V \setminus S)$. Thus $S = \{v_1, v_2, s\}$ is a source community with source s in G.

Claim 2: For $\alpha < 2$ the cut $(S \cup \{v_3\}, V_\alpha \setminus (S \cup \{v_3\}))$ is a cheaper s-t-cut in G_α than the cut $(S, V_\alpha \setminus S)$. We compare the costs of both cuts. We denote $S \cup \{v_3\}$ by U.

$$\begin{aligned} c_\alpha(S, V_\alpha \setminus S) = c(S, V \setminus S) + |S|\alpha &= 4 + 3\alpha \\ c_\alpha(U, V_\alpha \setminus U) = c(U, V \setminus U) + |U|\alpha &= 2 + 4\alpha \end{aligned}$$

In total we see that $c_\alpha(S, V_\alpha \setminus S) - c_\alpha(U, V_\alpha \setminus U) = 2 - \alpha > 0$ for $\alpha < 2$. Thus, the cut $(U, V_\alpha \setminus U)$, which also separates s and t, is cheaper.

Claim 3: For $\alpha \geq 2$ the cut $(\{s\}, V_\alpha \setminus \{s\})$ is a cheaper s-t-cut in G_α than the cut $(S, V_\alpha \setminus S)$. We compare again the costs of both cuts.

$$\begin{aligned} c_\alpha(S, V_\alpha \setminus S) = c(S, V \setminus S) + |S|\alpha &= 4 + 3\alpha \\ c_\alpha(\{s\}, V_\alpha \setminus \{s\}) = c(\{s\}, V \setminus \{s\}) + \alpha &= 5 + \alpha \end{aligned}$$

In total we see that $c_\alpha(S, V_\alpha \setminus S) - c_\alpha(\{s\}, V_\alpha \setminus \{s\}) = -1 + 2\alpha > 0$ for $\alpha \geq 2$. Thus, the cut $(\{s\}, V_\alpha \setminus \{s\})$, which also separates s and t, is cheaper.

According to these claims, there exists no α such that the cut $(S, V_\alpha \setminus S)$ is a minimum s-t-cut in G_α. Hence, S never becomes the M-set $\mathfrak{m}(s,t)$ in G_α, and is thus, never found by the cut-clustering algorithm.

11.2 Complete Hierarchies of Cut Clusterings

As we have seen before, the cut-clustering algorithm of Flake et al. [42] admits to find hierarchies of at most n different cut clusterings by applying CutC iteratively with decreasing α, which is a nice and desirable property. The crucial point with the construction of such a hierarchy, however, is the choice of α. If we choose the next value too close to a previous one, we get a clustering we already know, which implies unnecessary effort. If we choose the next value too far from any previous value, we possibly miss a clustering. Flake et al. propose a binary search for the choice of α. However, this necessitates a discretization of the parameter range—an issue where we again need to deal with the trade-off of improving the running time by choosing wide steps and limiting the risk of missed clusterings by choosing small steps. In practice the choice of a good coarseness of the discretization requires previous knowledge on the graph structure, which we usually do not have. Thus, in this section, we introduce a simple parametric search approach that guarantees to find each cut clustering that can be obtained by CutC for any parameter value α. At the same time, this approach computes each level in the resulting hierarchy at most twice. In other words, our approach efficiently returns a complete hierarchy without the necessity of discretization and without requiring any previous knowledge.

11.2.1 Simple Parametric Search Approach

In order to find all different levels in a cut-clustering hierarchy, our approach constructs the breakpoints in the parameter range between consecutive levels. For two consecutive hierarchy levels $\Omega_i < \Omega_{i+1}$ (recall Fig. 11.2), we call α' the *breakpoint* if CutC returns Ω_i for α' and Ω_{i+1} for $\alpha' - \varepsilon$ with $\varepsilon \to 0$. That is, each clustering Ω_i is assigned to an interval $[\alpha', \alpha'')$, where CutC returns Ω_i. The breakpoint α'' marks the border between Ω_i and the previous clustering Ω_{i-1}. Hence, it is $\alpha'' > \alpha'$. Based on this interval, the quality guarantee given by the parameter can be extended to

$$\overline{\psi}(\Omega_i) \leq \alpha' < \alpha'' - \varepsilon \leq \psi(\Omega_i)$$

for each cut clustering Ω_i in the complete hierarchy. We call $[\alpha', \alpha'')$ the *guarantee interval* of Ω_i. The simple idea of our approach is to compute good candidates for breakpoints during a recursive search with the help of cut-cost functions of the clusters, such that each candidate that is no breakpoint yields a new clustering instead. In this way, we apply CutC at most twice per level in the final hierarchy. Beginning with the trivial clusterings $\Omega_0 < \Omega_{\max}$ ($\alpha_0 > \alpha_{\max}$), the following theorem directly implies an efficient algorithm that definitely finds all possible clusterings.

Theorem 11.4. *Let $\Omega_i < \Omega_j$ denote two different clusterings with parameter values $\alpha_i > \alpha_j$. In time $O(|\Omega_i|)$ a parameter value α_m with 1) $\alpha_j < \alpha_m \leq \alpha_i$ can be computed such that 2) $\Omega_i \leq \Omega_m < \Omega_j$, and 3) $\Omega_m = \Omega_i$ implies that α_m is the breakpoint between Ω_i and Ω_j.*

For the proof of Theorem 11.4, we use *cut-cost functions* that represent, depending on the parameter α, the cost $\omega_S(\alpha)$ of a cut $(S, V_\alpha \setminus S)$ in G_α based on the cost of the cut $(S, V \setminus S)$ in G and the size of S.

$$\begin{aligned} \omega_S : \mathbb{R}_0^+ \quad &\longrightarrow \quad [c(S, V \setminus S), \infty) \subset \mathbb{R}_0^+ \\ \omega_S(\alpha) \quad &:= \quad c(S, V \setminus S) + |S|\, \alpha \end{aligned}$$

The main idea is the following. Let $\Omega_i < \Omega_j$ denote two hierarchically nested clusterings. We call a cluster $C' \in \Omega_i$ that is nested in $C \in \Omega_j$ a *child* of C and C the *parent* of C'. If there exists another level Ω' between Ω_i and Ω_j, at least two clusters in Ω_i must be merged yielding a larger cluster in Ω'. The maximal parameter value where this happens is a value α^* where a child C' in Ω_i becomes more expensive than its parent C in Ω_j, and thus, is *dominated* by C in the sense that it will not become a cluster in any hierarchy level above α^* (that is, where $\alpha < \alpha^*$). For two nested clusters $C' \subseteq C$ this point is marked by the intersection point of the cut-cost functions $\omega_{C'}$ and ω_C (see Fig. 11.6). Note that the cut-cost functions do not necessarily intersect if the child C' does not contain a representative of the parent C. On the other hand, the functions of $C' \subseteq C$ definitely intersect if C' contains a representative of C. Since the slope of the cut-cost function is determined by the size of the particular cluster, in the latter case it holds $c(C', V \setminus C') \geq c(C, V \setminus C)$. Otherwise, the functions would not intersect. Since at least one of the representatives in C is contained in a child C', each parent C has at least one child C' such that the cut-cost functions intersect. This child dominates the parent for $\alpha > \alpha^*$ with respect to the representative it contains. Thus, the intersection point α^* is a good candidate for a breakpoint between Ω_i and Ω'.

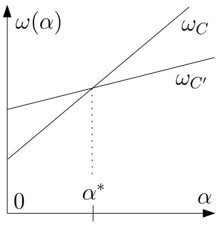


FIGURE 11.6: Intersecting cut-cost functions.

In the following, we show that if we choose the breakpoint candidate $\alpha_m := \min_{C \in \Omega_j} \lambda_C$ with $\lambda_C := \max_{C' \in \Omega_i : C' \subset C} \{\alpha \mid \omega_C(\alpha) = \omega_{C'}(\alpha)\}$, then Claim 1) to 3) as stated in Theorem 11.4 hold with this choice of α_m. The notation λ_C describes the maximum intersection point of a parent function ω_C (parent $C \in \Omega_j$) with the functions of all children in Ω_i. The minimum of these points among all parents in Ω_j then yields α_m. For the running time, observe that α_m is well-defined as each parent function intersects with at least one child function. In practice we construct α_m by iterating the list of representatives stored for children in Ω_i. Each of these representatives is assigned to a cluster in Ω_j, thus, matching children to their parents can be done in time $O(|\Omega_i|)$. The computation of the intersection points takes only constant time, given that the sizes and costs of the clusters are stored with the representatives by CUTC. In total, the time for computing α_m is thus in $O(|\Omega_i|)$.

For the following proofs of Claim 1) to 3), let C denote a parent in Ω_j with minimum value λ_C, that is, $\lambda_C = \alpha_m$. Let further $C' \subseteq C$ denote a child in Ω_i that contains a representative $r(C)$ of C, and let α' denote the intersection point of the corresponding cut-cost functions. Finally, we denote by $C^m \subseteq C$ a child in Ω_i whose function intersects the parent function at exactly α_m.

Proof of Claim 1). $\alpha_j < \alpha_m \leq \alpha_i$: We consider each inequality separately and prove them by contradiction. First assume $\alpha_j \geq \alpha_m$. Since $\lambda_C = \alpha_m$ denotes the maximum intersection point between C and its children in Ω_i, this implies $\alpha_j > \alpha'$. This however means, that C' dominates C (with respect to the representative $r(C)$) in G_{α_j} contradicting the fact that C is a cluster in Ω_j with representative $r(C)$. Now assume $\alpha_i < \alpha_m$. Then C immediately dominates C^m in G_{α_i} contradicting the fact that C^m is a cluster in Ω_i. ☐

After the computation of α_m, we apply CUTC with the newly obtained parameter value. The resulting clustering is denoted by Ω_m. According to Claim 1) and the hierarchical structure of the cut clusterings, it is $\Omega_i \leq \Omega_m \leq \Omega_j$. Claim 2) states, that Ω_m never equals Ω_j.

Proof of Claim 2). $\Omega_m \neq \Omega_j$: We show that $C \notin \Omega_m$. Recall that $\alpha' \leq \alpha_m$, since α_m is the maximum among all intersection points of C with its children. Thus, C' dominates C (with respect to $r(C)$) in G_{α_m}. Nevertheless, C might be a cluster in Ω_m with respect to another representative $u \neq r(C)$. However, the intersection point of C with any child containing another representative u is also smaller than α_m because of the maximality of α_m. Thus, C is dominated by all these children in G_{α_m}. Consequently, C is no cluster in Ω_m. □

If we find that the newly computed clustering Ω_m equals Ω_i, Claim 3) states that we have found a breakpoint. Hence, we can stop searching for further hierarchy levels between Ω_i and Ω_j. Note that due to the hierarchical structure, Ω_m and Ω_i can be easily compared by just counting clusters.

Proof of Claim 3). If $\Omega_m = \Omega_i$ *then* α_m *is the breakpoint between* Ω_i *and* Ω_j: We show first that α_m is the breakpoint between Ω_i and the next higher level in the complete hierarchy. In a second step we prove that the clustering on the next higher level equals Ω_j.

To see the first statement consider $\alpha_m - \varepsilon < \alpha_m$ for $\varepsilon \to 0$. This yields that C dominates C^m in $G_{\alpha_m - \varepsilon}$. Consequently, $\Omega_m = \Omega_i$ contains a cluster C^m that will never appear for $\alpha_m - \epsilon$, and thus, α_m is the breakpoint between Ω_i and the next higher level in the complete hierarchy.

In order to prove the second statement saying that the next higher level equals Ω_j, we show that each cluster in Ω_j corresponds to an M-set $\mathfrak{m}(s,t)$ in $G_{\alpha_m - \varepsilon}$, which is not obvious, since up to now, we just know that the clusters in Ω_j are M-sets with respect to α_j, that is, in G_{α_j}. The nesting property of these M-sets (Lemma 7.6(1),(2i)) together with the hierarchical structure then ensures that CutC returns Ω_j for $\alpha_m - \varepsilon$, which means that there exists no other clustering between Ω_i and Ω_j. For this final step we overload the notation of C and C^m as follows: Let $C \in \Omega_j$ denote an arbitrary cluster and let $C^m \in \Omega_i = \Omega_m$ denote a child of C whose cut-cost function intersects the parent function at λ_C, which now might be different from α_m, since C is now an arbitrary cluster. However, by definition, α_m is still the minimum value among all clusters in Ω_j, and thus it is $\lambda_C \geq \alpha_m$. Let further r denote the representative of C^m in Ω_m. Recall that $\Omega_m = \Omega_i$ does not imply the equivalence of the representative of C^m in Ω_i and the representative of C^m in Ω_m, as long as $\alpha_i \neq \alpha_m$. We show that (a) $\lambda_C = \alpha_m$, and based on this, that (b) C equals the M-set $\mathfrak{m}(r,t)$ in $G_{\alpha_m - \epsilon}$.

Claim (a). $\lambda_C = \alpha_m$: It already holds $\lambda_C \geq \alpha_m$. Now assume $\lambda_C > \alpha_m$. Since λ_C was supposed to be the intersection point of C and C^m, the parent C would then dominate the child C^m in G_{α_m}. This, however, contradicts the fact that C^m is a cluster in Ω_m.

Claim (b). C equals the M-set $\mathfrak{m}(r,t)$ *in* $G_{\alpha_m - \epsilon}$: Let $\widehat{C}$ denote the M-set $\mathfrak{m}(r,t)$ in $G_{\alpha_m - \epsilon}$. We show that $C = \widehat{C}$. The hierarchical structure implies $C^m \subseteq \widehat{C} \subseteq C$. It is further $\omega_{C^m}(\alpha_m) \leq \omega_{\widehat{C}}(\alpha_m)$, since otherwise $\widehat{C}$ would induce a smaller r-t-cut in G_{α_m} than the actual M-set C^m. On the other hand, it is $\omega_{\widehat{C}}(\alpha_m - \varepsilon) \leq \omega_C(\alpha_m - \varepsilon)$, by the same argument, that is, otherwise C would induce a smaller r-t-cut in $G_{\alpha_m - \varepsilon}$ than the actual M-set $\widehat{C}$. Finally, claim (a) tells us that the intersection point of C^m and C is $\lambda_C = \alpha_m$, that is, $\omega_{C^m}(\alpha_m) = \omega_C(\alpha_m)$. Thus, the cost-function of $\widehat{C}$ must lie above ω_C at α_m and below ω_C at $\alpha_m - \varepsilon$ (see Fig. 11.7). This implies that the slope of $\omega_{\widehat{C}}$ is at least the slope of ω_C, which means $|\widehat{C}| \geq |C|$. The hierarchical structure, however, requires that $\widehat{C} \subseteq C$, with implies $|\widehat{C}| \leq |C|$. Hence, it must hold $|\widehat{C}| = |C|$ and it follows $\widehat{C} = C$. □

11.2.2 Running Time

The parametric search approach calls CutC twice per level in the final hierarchy, once when computing a level the first time and again right before recognizing that the level already exists and a breakpoint is reached. The trivial levels $\Omega_{\max}$ and Ω_0 are calculated in advance without using CutC. Nevertheless, Ω_0 is recalculated once when the breakpoint to the lowest non-trivial level is found. This yields $2(h-2)+1$ applications of CutC, with h the number of levels. We denote the running time of CutC by $T(n)$ without further analysis. For a more detailed discussion on the running time of CutC see [42]. Since common maximum-flow algorithms run in $O(nm\log(n^2/m))$ time, a single minimum-cut computation already dominates the cost for determining α_m and further linear overhead. The running time of our simple parametric approach thus is in $O(2h\,T(n))$, where $h \leq n-1$. This clearly improves the running time of a binary search, which is in $O(h\,\log(d)\,T(n))$, with d the number of discretization steps—in particular since we may assume $d \gg n$ in order to minimize the risk of missing levels. We also tested the practicability of our simple approach by a brief experiment. The results confirm the improved theoretical running time.

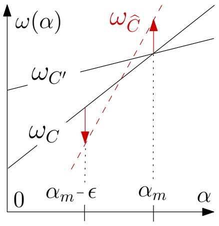


Figure 11.7: Intersecting cut-cost functions for C', C, and $\widehat{C}$.

A Brief Experimental Evaluation. For our experiments we used the collection of real world instances and generated instances listed in Table 11.2. Most instances are taken from the testbed of the 10th DIMACS Implementation Challenge, which provides benchmark instances for partitioning and clustering. The implementation was realized within the LEMON framework version 1.2.1. The instances and the LEMON framework are described in more detail in Section 1.4. We implemented CutC as described in Algorithm 7, extended by a heuristic that chooses the vertices in non-increasing order with respect to the weighted degree. Due to this heuristic, which was proposed by Flake et al., the number of minimum-cut computations in CutC becomes proportional to the number of clusters in the resulting clustering [42]. The implementation of a minimum-cut construction provided by LEMON runs in $O(n^2\sqrt{m})$. Note that we did not focus on a notably fast implementation. Instead, the implementation should be simple and practical using available routines for sophisticated parts like the minimum-cut computation.

In this context, it is however necessary to remark that most ready-to-use implementations of maximum-flow algorithms/minimum-cut algorithms require non-negative integers as edge capacities in the input graph. For many real-world instances this is no problem since their edge costs model email contacts, linked web pages or numbers of other indivisible entities, and are thus already in $\mathbb{N}_0$. However, in the extended graph G_α considered by the complete hierarchical cut-clustering approach, α is in $\mathbb{R}_0^+$. Thus, the costs in G_α cannot be scaled exactly to values in $\mathbb{N}_0$, and we possibly miss again some levels in the hierarchy. At this point, we stress that assuming α in $\mathbb{Q}_0^+$, instead of $\mathbb{R}_0^+$, is no restriction to the algorithm, if we assume integer costs in the original graph. It easy to see that then the intersection points of the cut-cost functions, and thus, the potential breakpoints are also in $\mathbb{Q}$. Since the proof of Theorem 11.4 does not care about the range of the intersection points, our parametric search approach also works for $\alpha \in \mathbb{Q}_0^+$, we do not miss any clustering in the hierarchy. Table 11.2 lists increasing CPU times determined on an AMD Opteron Processor 252 with 2.6 GHz and 16 GB RAM.

Table 11.2: Running times for the parametric search approach (ParS) and slowdown factors for the binary search approach with (BinS cont.) and without contraction (BinS). Instances are sorted by CPU times of ParS. Times longer than six days are marked by *.

graph	n	m	h	ParS [m:s]	BinS cont. [fac]	BinS [fac]
jazz	198	2742	3	0.062	7.726	7.871
celegans_metabolic	453	2025	8	0.300	7.620	8.380
celegansneural	297	2148	17	0.406	8.653	9.919
delaunay_n10	1024	3056	2	0.470	8.930	8.994
emailgraph550K_19	491	853	33	0.771	11.855	13.850
email	1133	5451	4	1.116	8.463	8.758
emailgraph550K_26	527	1046	38	1.231	10.434	11.957
delaunay_n11	2048	6127	2	1.792	9.256	8.893
netscience	1589	2742	38	4.310	4.030	11.952
bo_cluster	2114	2277	19	4.355	6.007	12.800
polblogs	1490	16715	7	4.493	11.560	12.097
delaunay_n12	4096	12264	2	7.226	10.914	9.870
data	2851	15093	4	11.506	8.021	9.620
dokuwiki_org	4416	12914	18	39.815	12.423	15.571
power	4941	6594	66	1:25.736	8.773	15.742
hep-th	8361	15751	56	6:26.213	7.373	18.183
PGPgiantcompo	10680	24316	94	13:25.121	6.463	18.575
as-22july06	22963	48436	33	39:54.495	12.419	20.583
cond-mat	16726	47594	80	44:15.317	14.917	27.425
astro-ph	16706	121251	60	98:25.791	21.843	24.825
rgg_n_2_15	32768	160240	46	245:25.644	32.748	22.573
cond-mat-2003	31163	120029	74	268:14.601	18.306	20.933
G_n_pin_pout	100000	501198	4	369:29.033	*	*
cond-mat-2005	40421	175691	82	652:32.163	*	21.446

For comparison, we further ran a binary search, as proposed by Flake et al., on the same instances, using the same CutC implementation in the same framework. Flake et al. [42] further observed that due to the hierarchical structure, it is also possible to contract the clusters of the lower hierarchy level in the current search interval in order to apply CutC to the resulting smaller graph instead of the original graph. Hence, we also extended the binary search by the possibility of contraction. In Table 11.2, the running times of both binary search variants are listed as factors saying how much longer the binary search ran compared to the parametric search. However, this is not meant to be a competitive running time experiment, since the running time of the binary search mainly depends on the discretization. We just want to demonstrate that being compelled to choose the discretization intuitively, without any knowledge on the final hierarchy, makes the binary search less practical. From a users point of view focusing on completeness, we defined the size of the discretization steps as $1/n^2$. The dependency on n is motivated by the fact that the potential number of levels increases with n, and by the hope that the breakpoints are distributed more or less equidistantly. For small graphs with $n < 1000$, one can even afford some more running time. Thus, we reduced the step size for those graphs to $1/(1000\ n)$ in further support of completeness. This yields 2^{10} to 2^{30} discretization steps depending on the length of the parameter range $[\alpha_{\max}, \alpha_0]$. With this discretization, the time for the binary search (with and without contraction) exceeds the time for the parametric search by a factor of four up to 32.

Moreover, as expected, the running time does not only depend on the input size but also on the number of different levels in the hierarchy. This can be observed for both approaches, parametric search and binary search, comparing the instances as-22july06 and cond-mat. Although the latter is smaller, it takes longer to compute 80 levels compared to only 33 levels in the former graph. Comparing the two binary search variants, we further observe that if the number of computed hierarchy levels is high enough relative to the graph size, the additional cost for contracting seems to pay off, see for example the instances netscience, bo_cluster, power, hep-th, PGPgiantcompo, as-22july06, and cond-mat. In contrast, finding only 46 hierarchy levels in the relative large instance rgg_n_2_15 was more time consuming at least with our implementation of the contraction approach than with the regular binary search approach.

11.3 A Case Study on Expansion and Modularity

Flake et al. [42] tested their algorithm on a citation network and a network of linked web pages with respect to the semantic meaning of the clusters. In this work we present an experimental analysis of the general behavior of cut clusterings on benchmark instances proclaimed within the 10th DIMACS Implementation Challenge on Graph Partitioning and Graph Clustering and some further instances constructed from email data and collected by ourselves. The instances are described in more detail in the next paragraph and in Section 1.4. Using the parametric search approach introduced in the previous section, we construct complete cut-clustering hierarchies and investigate the guaranteed quality of the cut clusterings in terms of expansion as well as the modularity quality reached by these cut clusterings. Since the intra-cluster expansion of a cluster and a clustering is hard to compute, we consider lower bounds in the analysis. Our study gives evidence that trivial bounds do not match up to the given guarantee. The analysis of a special non-trivial bound further indicates that the true intra-cluster expansion of the cut clusterings even surpasses the guarantee, and also the inter-cluster expansion turns out to be better than guaranteed.

Within the modularity analysis of the cut clusterings, we additionally consider reference clusterings obtained from a common modularity-optimizing heuristic [121]. Our study documents that for many of the tested graphs the cut clusterings reach modularity values quite close to the references. On the other hand the cut-clustering algorithm returns fine clusterings with small clusters and low modularity values if there are no other plausible clusterings supported by the graph structure. The reference clusterings found by the modularity-optimizing heuristic, however, often provide high modularity values even for those graphs. Thus, in contrast to clusterings of best possible modularity, the modularity values of cut clusterings admit to indicate how well a graph can be clustered.

Experimental Setting. The experiments in this work aim at three questions. The first question asks how much more information the given guarantee on intra-cluster expansion provides, compared to lower intra-cluster expansion bounds that are easy to compute. Additionally, we also consider the exact inter-cluster expansion, which can be efficiently computed. The second question focuses on the modularity values that can be reached by cut clusterings, and the plausibility of these values with respect to the graph structure. The third question finally asks whether the reference clusterings obtained by the modularity-optimizing heuristic outperform the cut clusterings in terms of intra-cluster expansion.

For our experiments, we use instances taken from the testbed of the 10th DIMACS Implementation Challenge, and additionally, the protein interaction network bo_cluster published by Jeong et al. [86] and a snapshot of the linked wiki pages at www.dokuwiki.org, which we gathered ourselves. The exact names of these instances are listed in the upper charts of the figures illustrating our experimental results. A more detailed description of the instances as well as overview on the sizes is given in Section 1.4. Moreover, we consider 275 snapshots of the email-communication network of the Department of Informatics at KIT also described in Section 1.4. The latter have around 200 up to 400 vertices.

Our analysis considers one cut clustering per instance, namely the cut clustering with the best modularity value of all clusterings in the complete hierarchy. The complete hierarchies of the instances are computed with the parametric search approach introduced in the previous section. We used the same implementation, based on the LEMON framework, as for the brief running time experiment in the previous section. For the computation of the reference clusterings, we used an implementation of Lisowski [101]. The underlying modularity-optimizing heuristic [121] is described in more detail in Section 11.3.2.

The results of our experiments are illustrated by three figures, one for each question we focus on. For the sake of a better readability, each figure consists of two parts. The first/upper part depicts the results for all instances except the snapshots of the email network. The second/lower part depicts the results for the latter instances. The instances, respectively their clusterings, are depicted on the x-axis, and all instances are decreasingly ordered by the amount of unclustered vertices in the cut clusterings, which corresponds to an increasing order by coarseness.

11.3.1 Intra- and Inter-Cluster-Expansion Analysis of Cut Clusterings

We consider the true inter-cluster expansion, which is easy to compute, and two lower and also an upper bound on intra-cluster expansion, since the true intra-cluster expansion is hard to compute. For a cluster C the first lower bound $B_\ell(C)$ and the upper bound $B_u(C)$ are trivially obtained from a global minimum cut $(M, C \setminus M)$ in C:

$$B_\ell(C) := \frac{c(M, C \setminus M)}{\lfloor |C|/2 \rfloor} \leq \psi(C) \leq \frac{c(M, C \setminus M)}{\min\{|M|, |C \setminus M|\}} =: B_u(C).$$

Note that $B_u(C)$ is simply the expansion of the global minimum cut. The corresponding bounds $B_\ell(\Omega)$ and $B_u(\Omega)$ for a whole clustering Ω are again given by the minimum among all clusters. Figure 11.8 considers for each instance the clustering of the complete hierarchy with the best modularity value. For these clusterings it shows how the bounds B_ℓ (solid black line) and B_u (dotted blue line) behave compared to the guarantee interval of the clustering given by the level breakpoints in the hierarchy (filled red area). For a better comparability, we normalized the upper interval boundary to 1. All further values are displayed proportionally. For the instances in the upper chart, Fig. 11.8 further shows the inter-cluster expansion $\overline{\psi}$. Comparing these values to the lower boundary of the guarantee interval proves that many clusterings have a better inter-cluster quality (that is, a lower value for $\overline{\psi}$) than guaranteed. See for example the instances lesmis, power and netscience. Moreover, this also holds for most of the snapshots of the email network depicted in the lower chart of Fig. 11.8. There, however, we omitted the presentation of the inter-cluster expansion for the sake of a better readability.

Regarding the intra-cluster quality, we observe that for most instances the trivial lower bound B_ℓ stays below the upper boundary of the guarantee interval. This reveals a true advantage from knowing the guarantee besides the trivial bound. The few exceptions, see for example

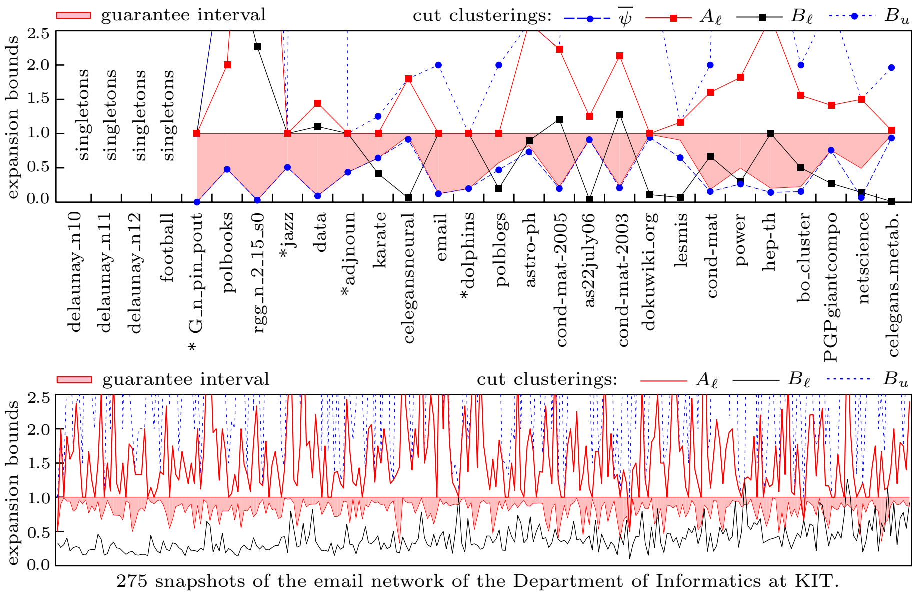


FIGURE 11.8: *Expansion Analysis of Cut Clusterings:* Inter-cluster expansion $\overline{\psi}$ and bounds on intra-cluster expansion: trivial lower bound B_ℓ and trivial upper bound B_u based on global minimum cut, alternative non-trivial lower bound A_ℓ. The upper boundary of the guarantee interval is normalized to 1, further values are displayed proportional. Instances where the guarantee meets B_u are marked by * in the upper chart. For the first four instances, the cut-clustering algorithm returns only singletons. For the sake of readability, $\overline{\psi}$ is omitted in the lower chart.

the instance polbooks, can be explained by the shape of the found cut clusterings. If the clustering contains only small clusters, the value of the global minimum cut in each cluster is only divided by a small number of vertices when computing the trivial lower bound B_ℓ. In unweighted graphs this often yields a value bigger than 1, that is, bigger than the maximum edge costs. The upper boundary of the guarantee interval, however, cannot exceed the maximum edge costs in the graph. For instance where the upper boundary of the guarantee interval meets the upper bound B_u, the guarantee equals the true intra-cluster expansion. In the upper chart of Fig. 11.8, these instances are marked by a star. For the snapshots of the email network, we counted 3.6% of the instances where the exact intra-cluster expansion is known. However, in most cases there is still a large gap between the guaranteed intra-cluster expansion and the upper bound B_u.

In order to explore this gap, we further consider an alternative non-trivial lower bound A_ℓ on intra-cluster expansion. This bound results from individually applying the hierarchical cut-clustering algorithm to the subgraphs induced by the clusters in a clustering. The algorithm returns a complete clustering hierarchy for each subgraph, thereby finding the breakpoint between the most upper hierarchy level, which consists of connected components, and the next lower level. If we assume that the considered subgraph is connected (otherwise the expansion is 0), this breakpoint α is the largest parameter value where CutC returns the whole subgraph as a cluster. According to the expansion guarantee of CutC the subgraph thus has at least intra-cluster expansion α, that is, the found breakpoint constitutes a non-trivial lower bound A_ℓ on the intra-cluster expansion of the considered cluster in the original clustering. This bound again expands to the whole clustering by taking the minimum value of all clusters. Since this method considers the clusters as independent instances ignoring the edges between the clusters, the resulting bound A_ℓ potentially lies above the guarantee interval, which is also confirmed by our experiment (see the solid red line in Fig. 11.8). That is, most of the cut clusterings are even better than guaranteed. Moreover, the lower bound A_ℓ increases the instances for which we know the intra-cluster expansion for sure to 20%, by reaching the upper bound B_u in some additional cases.

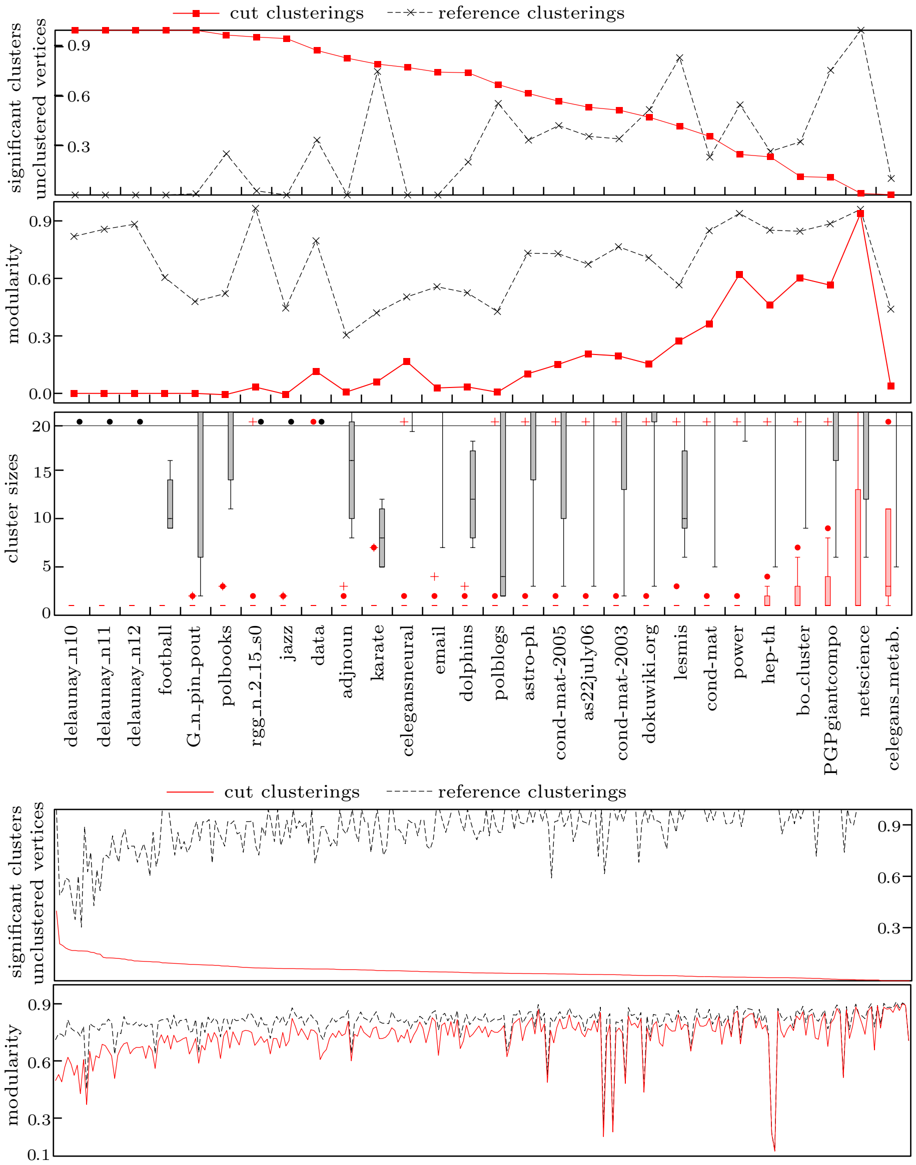
11.3.2 Modularity Analysis of Cut Clusterings

In the following we examine the modularity values of the best cut clusterings in the cut-clustering hierarchies. In order to justify whether a given modularity value is a good value for the particular instance, we also generate reference clusterings of high modularity, using a state-of-the-art modularity-optimizing greedy approach [121]. This multilevel greedy approach is widely used and has turned out to be reliable in many experiments. It starts with an initial clustering consisting of singleton clusters and moves vertices between clusters as long as this operation increases modularity. Then the found clusters are contracted and the algorithm continues on the next level. Finally the different levels are expanded top-down and the algorithm again allows single vertices to move in order to further increase modularity. In this way, a local maximum among all possible clusterings is found. Recall that computing a globally optimal clustering is NP-hard [20].

Since high modularity values are known to be misleading in some cases, we further establish a plausibility check by testing whether the clusters of the reference clusterings satisfy the following *significance property*. This property guarantees that the clusters are clearly indicated by the graph structure. A non-trivial cluster C in the reference clustering satisfies the significance property if there exists a vertex $r \in C$ such that $c(U, C \setminus U) > c(U, V \setminus C)$ for all $U \subseteq C \setminus \{r\}$. Note that this is exactly the first source-community condition (recall Table 11.1), and thus, the clusters of the cut clusterings satisfy this property due to their construction. Figure 11.9 shows the percentage amount of significant clusters, that is, clusters with significance property, in the reference clusterings (dashed black line in the upper chart of each part).

To get also a better idea of the structure of the cut clusterings, we further present the percentage amount of unclustered vertices in these clusterings (solid red line in the upper chart of each part). Unclustered vertices may occur due to the strict behavior of the cut-clustering algorithm, which is necessary in order to guarantee the significance property. We remark that in contrast none of the reference clusterings contains unclustered vertices. As an additional structural information on both types of clusterings, Fig. 11.9 depicts the cluster sizes in form of whisker-bars (lower chart of the first part, red bars for cut clusterings, black bars for reference clusterings, cluster sizes are omitted for snapshots of the email network in the second part). The results on modularity are depicted in the middle chart of the first part and the lower chart of the second part.

With this bunch of information at hand, we observe the following. In some cases, the modularity of the cut clustering is quite low, however, modularity increases with the amount of clustered vertices and the size of the clusters. It also reaches very high values, in particular for the snapshots of the email network and the instance `netscience`. The corresponding cut clustering is shown in Fig. 1.5(a). The latter is a rather unexpected behavior, since the cut-clustering algorithm is not designed to optimize modularity. We further observe a gap between the modularity values of many cut clusterings and the values of the corresponding reference clusterings. We conjecture that this is caused more by an implausibility of the modularity values of the reference clusterings than by an implausibility of the cut clusterings. Our conjecture is based on the observation that



275 snapshots of the email network of the Department of Informatics at KIT.

FIGURE 11.9: *Modularity Analysis of Cut Clusterings:* Results for the best cut clusterings and the reference clusterings. The upper charts in both parts show the ratio of (nontrivial) significant clusters in the reference clusterings and the ratio of unclustered vertices in the cut clusterings. The lower chart in the first part shows the cluster sizes for both types of clusterings in the form of whisker-bars with maximum (+) and minimum (•) of the outliers. Values greater than 20 are placed at the edge of the displayed range. Due to the high number of email snapshots, we omitted whisker-bars there. The results on modularity are depicted in the middle chart of the first part and the lower chart of the second part.

more significant clusters in the reference clustering lead to a modularity value closer to the value of the cut clustering, suggesting that the cut clusterings are more reliable.

Furthermore, comparing the results for the Delaunay triangulations and the snapshots of the email network also vividly reveals the meaningfulness and plausibility of the cut clusterings. The latter consider emails that were sent at most 72 hours ago. In contrast to other email networks, which consider a longer period of time, this makes the snapshots very sparse and stresses recent communication links, which yields clear clusters of people that recently work together, see Fig. 1.6(a) for an exemplary email snapshots. Thus, we would expect a reliable clustering approach to return meaningful non-trivial clusters. This is exactly what the cut-clustering algorithm does. In contrast, the Delaunay triangulations generated from random points in the plane are quite uniform structures (see Fig. 1.5(b)), and by intuition, significant clusters are rare therein. The cut-clustering algorithm confirms our intuition by leaving all vertices unclustered. This explains the low modularity values of these clusterings and indicates that the underlying graph can not be clustered well. The reference clusterings, however, which consist of larger clusters, contradict the intuition.

11.3.3 Intra-Cluster-Expansion Analysis of Reference Clusterings

Finally, we also examine whether the reference clusterings outperform the cut clusterings in terms of intra-cluster expansion. To this end, we study the same lower and upper bounds for as in Section 11.3.1. Figure 11.10 compares the guarantee interval and the non-trivial lower bound A_ℓ for the cut clusterings (solid red line, already seen in Section 11.3.1) to the bounds A_ℓ (solid black line), B_ℓ (dashed black line) and B_u (dotted blue line) for the reference clusterings (A_ℓ for cut clusterings only in the upper, B_ℓ for reference clusterings only in the lower chart).

We observe that the trivial lower bound B_ℓ for the reference clusterings stays clearly below the guaranteed intra-cluster expansion for the cut clusterings, and compared to the trivial lower bound B_ℓ for the cut clusterings in Section 11.3.1 (Fig. 11.8), this behavior is even more evident. In contrast, the non-trivial lower bound A_ℓ for the reference clusterings often exceeds the guarantee interval, particularly for the email snapshots. Nevertheless, it does rarely reach the corresponding bound for the cut clusterings. We counted 85% of the instances where it rather stays below the best lower bound for the cut clusterings. Thus, with respect to the lower bounds, there is no evidence that the intra-cluster expansion of the reference clusterings surpasses that of the cut clusterings. The upper bound B_u, which drops below the best lower bound for the cut clusterings in 23% of the cases, even proves a lower intra-cluster expansion for the reference clusterings. The corresponding instances are marked by a star in the upper chart of Fig. 11.10.

Conclusion. The experiments in this study document that the given guarantee on intra-cluster expansion provides a deeper insight compared to a trivial bound that is easy to compute. The true intra-cluster expansion and inter-cluster expansion often turned out to be even better than guaranteed. An analog analysis of the expansion of the reference clusterings could further give no evidence that modularity-optimizing clusterings surpass cut clusterings in terms of intra-cluster expansion. On the contrary, around one fourth of the considered reference clusterings could be proven to be worse than the cut clusterings.

Within the modularity analysis, we could reveal that, although it is not designed to optimize modularity, the hierarchical cut-clustering algorithm reliably finds clusterings of good modularity if such a clustering is structurally indicated. Otherwise, if no good clustering is indicated by

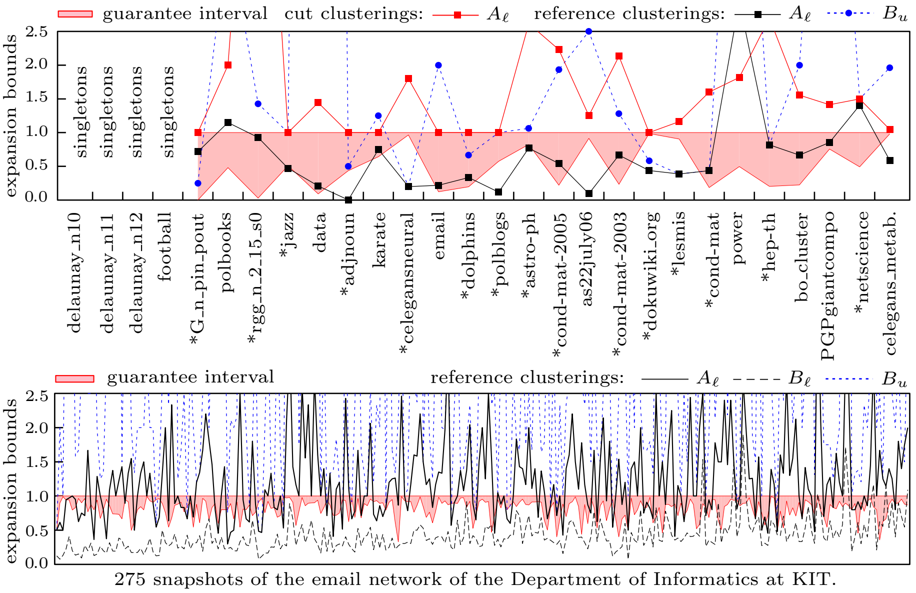


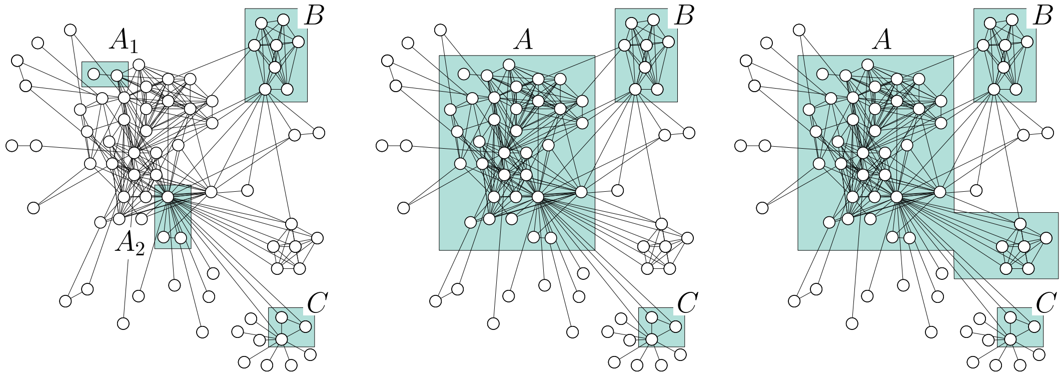
FIGURE 11.10: *Expansion Analysis of Reference Clusterings:* Guarantee interval and non-trivial lower bound A_ℓ for cut clusterings (only in upper chart), and bounds on intra-cluster expansion for the reference clusterings: trivial lower bound B_ℓ (only in lower chart) and trivial upper bound B_u based on global minimum cut, alternative non-trivial lower bound A_ℓ. The upper boundary of the guarantee interval is normalized to 1, further values are displayed proportional. Instances where the upper bound B_u for the reference clustering drops below the lower bound A_ℓ for the cut clustering are marked by * in the upper chart. For the first four instances, the cut-clustering algorithm returns only singletons.

the graph structure, the cut-clustering algorithm returns clusterings of low modularity. This confirms a high reliability of the cut-clustering approach and justifies modularity applied to cut clusterings as a feasible measure for how well a graph can be clustered.

CHAPTER 12

Maximum Source-Community Clusterings

In the previous chapter, we have seen that computing all existing cut clusterings for a given undirected, weighted graph is possible with an appropriate strategy for choosing the parameter values α (Section 11.2). However, depending on the graph structure, it might happen also in a complete hierarchy that no appropriate clustering with respect to a certain application is found. Consider for example the co-appearance network of the characters in the novel Les Miserables [94] (called lesmis) presented in Fig. 12.1. The complete cut-clustering hierarchy for this weighted network consists of 11 levels, three of which are depicted in the figure. Intuitively, the first clustering Ω_3 (Fig. 12.1(a)), which consists of only four small, non-trivial clusters and many unclustered vertices, is not desirable for most applications. On the other hand, the next higher clustering Ω_4 (Fig. 12.1(b)) contains already a very large cluster that covers a big part of the graph, while the remaining clusters are rather small or even singletons. In the third clustering Ω_5 (Fig. 12.1(c)) this large cluster even expands covering six further vertices. Hence, one might ask if there is no other, somehow nicer clustering indicated by the graph structure, which just is not found by the cut-clustering approach; a clustering that, for example, contains less unclustered vertices than Ω_3 or is more balanced than Ω_4 and Ω_5. Of course, such a clustering cannot be a cut clustering anymore, since we already know all existing cut clusterings, but there might be still a nicer clustering consisting of source communities, since the source communities that can



(a) Cut clustering Ω_3. (b) Next higher level Ω_4. (c) Next higher level Ω_5.

FIGURE 12.1: Three consecutive cut clusterings Ω_3, Ω_4, Ω_5 in the complete hierarchy for the weighted instance lesmis. Note that the edge costs are not depicted here. The clusters A_1 and A_2 in Ω_3 are covered by a larger cluster A in Ω_4, while in Ω_5 the cluster A even expands. The clusters B and C appear in all three clusterings.

be detected by the cut-clustering algorithm form a proper subclass of the class of all existing source communities (recall Section 11.1.3).

In this chapter, we present a possibility to investigate the source community structure of a given graph beyond the cut clusterings found by the cut-clustering algorithm. To this end, we efficiently construct maximum clusterings that are close or equal to the highest level of any clustering hierarchy that consists of source communities and contains a set of designated clusters. More precisely, we show that the unique-cut tree presented in Part II (Section 7.2) admits to answer the following queries in linear time.

(1) For a given source community S, return a clustering $\Omega(S)$ that consists of source communities, contains S, and is maximum in the sense that each clustering that also consists of S and further source communities is hierarchically nested in $\Omega(S)$.

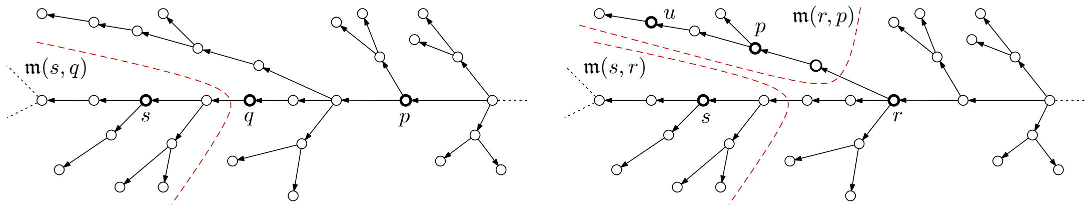
(2) For k disjoint source communities $S_1, \dots, S_k$, return an inclusion maximal clustering, denoted by $\Omega(S_1, \dots, S_k)$, that contains $S_1, \dots, S_k$, is hierarchically nested in each clustering $\Omega(S_1), \dots, \Omega(S_k)$ defined in (1), and is maximum in the sense that each clustering that consists of $S_1, \dots, S_k$ and further source communities is hierarchically nested in $\Omega(S_1, \dots, S_k)$.

If a clustering $\Omega(S)$, as required in (1), exists for a given source community S, it follows directly that $\Omega(S)$ is the unique inclusion-maximal clustering with respect to the required properties, since then each clustering that also consists of S and further source communities is already hierarchically nested in $\Omega(S)$. Thus, we call $\Omega(S)$ the *maximum SC-clustering* for S. In the following we will see that $\Omega(S)$ always exists and how it can be constructed. The clustering $\Omega(S_1, \dots, S_k)$, as required in (2), will also turn out to always exist and to be unique. We call $\Omega(S_1, \dots, S_k)$ the *overlay clustering* for $S_1, \dots, S_k$.

The linear query time of this approach admits an efficient computation of many such maximum clusterings with respect to different designated clusters, and thus, a broad analysis of the source-community structure of a given graph beyond cut clusterings. The designated clusters may result from the cut-clustering algorithm or other preliminary knowledge about the graph. The precomputation of the underlying unique-cut tree needs at most $2(n-1)$ maximum-flow computations (see Section 7.2). Recall further that source communities correspond to generalized M-sets. Thus, in the following, we identify source communities and generalized M-sets without further notice.

The Shape of Source Communities in Unique-Cut Trees. The key to the algorithms for constructing maximum SC-clusterings and overlay clusterings is the following lemma, which describes the shape of source communities with respect to the unique-cut tree of the underlying graph G. Recall the structure of the unique-cut tree $\mathcal{T}(G)$ for an undirected, weighted graph $G = (V, E, c)$. The unique-cut tree $\mathcal{T}(G)$ represents all regular M-set of G and consists of two parts. The first part is a rooted cut tree $T(G) = (V, E_T, c_T)$ with edges directed to the leaves such that each edge $(t, s) \in E_T$ represents the U-cut $\mathfrak{uc}(s, t)$, and for the opposing M-sets holds $|\mathfrak{m}(s, t)| \leq |\mathfrak{m}(t, s)|$. The second part is an $(n-1) \times n$ matrix where the rows store the opposite M-sets (the so-called matrix sets) assigned to the edges in $T(G)$. The following lemma now limits the shape of arbitrary source communities in G to subtrees in $\mathcal{T}(G)$, which admits an efficient enumeration of disjoint source communities by a depth-first search (DFS), as we will see in this chapter.

Lemma 12.1. *Let S denote an arbitrary source community in G. The subgraph $\mathcal{T}[S]$ induced by S in the unique-cut tree $\mathcal{T}(G)$ is connected.*



(a) p is predecessor of s.

(b) u is neither predecessor nor successor of s.

Figure 12.2: Different situations considered in the proof of Lemma 12.1.

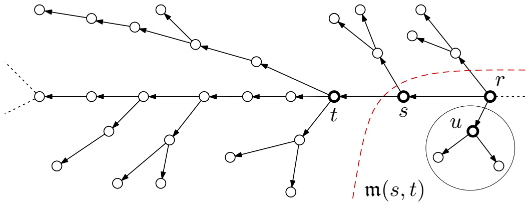


Figure 12.3: Situation considered in the proof Lemma 12.2. The subtree U rooted in u is marked by a circle.

Proof. If S is a (regular) M-set that belongs to a U-cut that is represented by an edge in $\mathcal{T}(G)$, the lemma obviously holds. Hence, assume S is a matrix set or another arbitrary source community, that is, a generalized M-set. In order to prove the connectivity of $\mathcal{T}[S]$, we first focus on the predecessors of the source s of S in the directed tree $\mathcal{T}(G)$. Let p denote a predecessor of s with $p \in \mathcal{T}[S]$ and q a successor of p on the path $\pi(p,s)$ between p and s (see Fig. 12.2(a)). We prove that $q \in \mathcal{T}[S]$. Assume $q \notin \mathcal{T}[S]$. Since s is a successor of q, s is in the M-set $\mathfrak{m}(s,q)$. With $q \notin \mathcal{T}[S]$, it follows from Lemma 7.6(2i) that $S \subseteq \mathfrak{m}(s,q)$. This, however, contradicts $p \in \mathcal{T}[S]$. Hence, the path $\pi(p,s)$ is completely contained in $\mathcal{T}[S]$.

In a second step, we consider the remaining vertices. Let u be a vertex that is no predecessor of s. Let r denote the nearest common predecessor of u and s (if u is a successor of s, it is $r = s$). We first show, that (i) if $u \in \mathcal{T}[S]$, then $r \in \mathcal{T}[S]$. Then we suppose there is also a predecessor $p \neq r$ of u on the path $\pi(r,u)$ between r and u (see Fig. 12.2(b) for the case where u is no successor of s) and prove (ii) that if $u \in \mathcal{T}[S]$, then $p \in \mathcal{T}[S]$. Together with the result on the predecessors of s, this ensures the connectivity of $\mathcal{T}[S]$.

Proof of (i): If $r = s$, we are done. Hence, assume $r \neq s$ and $r \notin \mathcal{T}[S]$. Since s is a successor of r, s is in the M-set $\mathfrak{m}(s,r)$, while $u \notin \mathfrak{m}(s,r)$. With $r \notin \mathcal{T}[S]$, it follows from Lemma 7.6(2i) that $S \subseteq \mathfrak{m}(s,r)$. This, however, contradicts the assumption that $u \in \mathcal{T}[S]$. Hence, it is $r \in \mathcal{T}[S]$ if $u \in \mathcal{T}[S]$.

Proof of (ii): From (i) we already know that $r \in \mathcal{T}[S]$. Assume $p \notin \mathcal{T}[S]$ and consider the M-set $\mathfrak{m}(r,p)$. It is $u \in \mathfrak{m}(r,p)$, but $s \notin (r,p)$. With p also not in $\mathcal{T}[S]$, it follows from Lemma 7.6(1) that $\mathfrak{m}(r,p)$ and $\mathcal{T}[S]$ are disjoint. This, however, contradicts the assumption that $u \in \mathcal{T}[S]$, since $u \in \mathfrak{m}(r,p)$. Hence, the path $\pi(r,u)$ is completely contained in $\mathcal{T}[S]$. □

If the source community S is a regular M-set, we can describe the shape of the induced subtree $\mathcal{T}[S]$ in $\mathcal{T}(G)$ in even more detail.

Lemma 12.2. *If S is a regular M-set and $u \in \mathcal{T}[S]$ is no predecessor of the source s of S, then the subtree rooted at u in $\mathcal{T}(G)$ is also contained in $\mathcal{T}[S]$.*

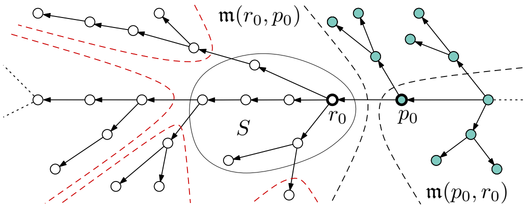


FIGURE 12.4: Initial situation when constructing a maximum SC-clustering. Clusters S is framed by a solid line, vertices in $\mathcal{T}_1$ are filled green. Dashed red lines indicate found source communities. Dashed black lines describe $\mathfrak{m}(r_0, p_0)$ and $\mathfrak{m}(p_0, r_0)$.

Proof. The lemma obviously holds if S belongs to a U-cut that is represented by an edge in $\mathcal{T}(G)$. Hence, assume S is a matrix set and let $(s,t) \in E_T$ denote the edge where S is assigned to (see Fig. 12.3). That is, $S = \mathfrak{m}(s,t)$. Let further $u \in \mathcal{T}[S]$ denote a vertex that is no predecessor of S, and let r denote the nearest common predecessor of u and s. Note that then u is in particular no successor of s, since all successors of s are outside of $\mathcal{T}[S]$. We denote the subtree rooted at u in $\mathcal{T}(G)$, by U. Obviously, $s, t \notin U$. However, the subtree U is also a regular M-set, and thus, a source community with source u. Since we assumed $u \in \mathcal{T}[S]$, it thus follows from Lemma 7.6(2i) that $U \subseteq \mathcal{T}[S]$. ☐

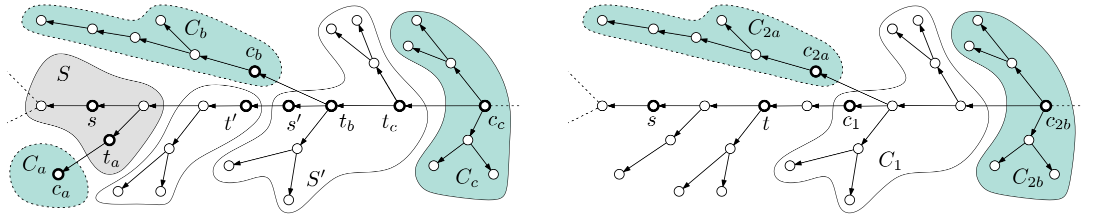
12.1 Constructing Maximum SC-Clusterings

Given an arbitrary source community S in an undirected, weighted graph $G = (V, E, c)$, we show how to construct the maximum SC-clustering for S, that is, a clustering $\Omega(S)$ of G that consists of source communities, contains S, and is maximum in the sense that each clustering that also consists of S and further source communities is hierarchically nested in $\Omega(S)$. This construction is based on the unique-cut tree of the underlying graph G.

Theorem 12.3. *Let S denote an arbitrary source community in G. The maximum SC-clustering for S can be determined in $O(n)$ time after preprocessing $\mathcal{T}(G)$.*

The maximum SC-clustering for $S =: S_0$ can be determined by the following construction, which proves Theorem 12.3 and directly implies a simple and efficient algorithm. Let r denote the root of $\mathcal{T}(G) =: \mathcal{T}_0$ and $\mathcal{T}[S_0]$ the subtree induced by S_0 in $\mathcal{T}_0$ (Lemma 12.1). Deleting $\mathcal{T}[S_0]$ decomposes $\mathcal{T}_0$ into connected components, each of which representing a source community, except the one that contains r if $r \notin S_0$ (see Fig. 12.4). If $r \in S_0$, we are done. Otherwise, let $\mathcal{T}_1$ denote the component containing r, and r_0 the root of $\mathcal{T}[S_0]$. Obviously, it is $p_0 \in \mathcal{T}_1$ for $(p_0, r_0) \in E_T$. The matrix set $\mathfrak{m}(p_0, r_0) =: S_1$ does not intersect with $\mathfrak{m}(r_0, p_0)$, which contains S_0 and all further components found so far besides $\mathcal{T}_1$. That is, $S_1 \subseteq \mathcal{T}_1$. According to Lemma 12.1, S_1 thus induces a subtree $\mathcal{T}[S_1]$ in $\mathcal{T}_1$. Hence, S_1 and $\mathcal{T}_1$ adopt the roles of S_0 and $\mathcal{T}_0$. Continuing in this way, we finally end up with a matrix set S_k containing r, such that deleting $\mathcal{T}[S_k]$ yields only source communities. The resulting clustering $\Omega(S)$ consists of the source communities $S = S_0, \ldots, S_k$, and the remaining source communities resulting from the decompositions of $\mathcal{T}_0, \ldots, \mathcal{T}_k$. We remark further that all source communities found by the construction correspond either to a U-cut that is represented by an edge in $\mathcal{T}(G)$ or to a matrix set. Hence, besides S, all clusters in $\Omega(S)$ are regular M-sets, by construction.

If we assume that we can crawl the edges in $\mathcal{T}(G)$ in both directions and that we can decide in constant time whether or not a vertex belongs to the currently considered source community,



(a) Three examples C_a, C_b, C_c of clusters (filled green) that can be proven to be M-sets with respect to s (by applying Claim 1 and 2); C_c is a matrix set, while C_a and C_b are not.

(b) Two examples C_{2a}, C_{2b} of clusters (filled green) that can be proven to be M-sets with respect to t (according to Claim 2); C_{2b} is a matrix set, while C_{2a} is not.

FIGURE 12.5: Situations considered in the proof of Lemma 12.4. Cluster S in (a) is filled gray, matrix sets are framed by solid lines, remaining M-sets by dotted lines.

this construction can be easily done in linear time by a DFS starting from the first vertex found in the current community. Recall that the matrix sets in $\mathcal{T}(G)$ can be accessed in constant time. Thus, it suffices to explicitly require a constant access time for the initial source community S. The remaining source communities appear in form of subtrees that contain all successors of their root in $\mathcal{T}(G)$. Thus, the vertices of these communities can be easily enumerated by the DFS, without checking the membership for each vertex. The maximality of $\Omega(S)$ finally follows from the following lemma.

Lemma 12.4. *Let Q denote an arbitrary source community of G that does not intersect with S. Then, Q is nested in a cluster of $\Omega(S)$.*

Proof. Let q denote a source of Q and s a source of S. Let further $C \in \Omega(S) \setminus S$ denote the cluster that contains q and recall that C is a regular M-set according to the construction. Moreover, C is associated to the edge (c, t) in $\mathcal{T}(G)$ if C is a matrix set, and otherwise, the U-cut of C is represented by the edge (t, c). That is, in both cases holds $C = \mathfrak{m}(c, t)$. In the following we show that C is also an M-set with respect to s, that is, $\mathfrak{m}(c, t) = \mathfrak{m}(c, s)$. Lemma 12.4 then follows directly by applying Lemma 7.6(2i) to Q and C with $q \in Q \cap C$, but $s \notin Q$.

In order to proof that $C = \mathfrak{m}(c, t) = \mathfrak{m}(c, s)$, we observe the following. For each cluster $C = \mathfrak{m}(c, t)$ in $\Omega(S)$, it holds by construction that t is either in S or in a cluster S' that is a matrix set $\mathfrak{m}(s', t')$ associated to the edge (s', t') in $\mathcal{T}(G)$, where (s', t') is on the (undirected) path $\pi(c, s)$ between the source c of C and the source s of S (see Fig. 12.5(a)). The matrix sets in $\Omega(S)$ in particular form a kind of a path. The idea is now to apply the following claims recursively along the clusters on this path.

Claim 1: Let C_1 and C_2 denote two clusters in $\Omega(S)$ with sources c_1 and c_2 and let further C_2 be the regular M-set $\mathfrak{m}(c_2, t)$ with $t \in C_1$. Then, C_2 is also an M-set with respect to c_2, that is, $C_2 = \mathfrak{m}(c_2, t) = \mathfrak{m}(c_2, c_1)$.

Claim 2: Let $\pi(c_1, s)$ denote the (directed) path in $\mathcal{T}(G)$ from a vertex c_1 to the source s of S. Let further C_1 and C_2 denote two clusters in $\Omega(S) \setminus S$ with $C_1 = \mathfrak{m}(c_1, t)$ a matrix set with $t \in \pi(c_1, s)$ and $C_2 = \mathfrak{m}(c_2, c_1)$ where the (undirected) path $\pi(c_2, c_1)$ shares exactly the vertex c_1 with $\pi(c_1, s)$ (see Fig. 12.5(b)). Then, C_2 is also an M-set with respect to t, that is, $C_2 = \mathfrak{m}(c_2, c_1) = \mathfrak{m}(c_2, t)$.

If the opponent t of $C = \mathfrak{m}(c, t)$ is in S, we are done according to the first claim, identifying S with C_1 and C with C_2. Otherwise, t is in a matrix set S', it holds again by Claim 1 that $C = \mathfrak{m}(c, s')$.

If now the opponent t' of S' is in S, according to the first claim, we get analogously that $S' = \mathfrak{m}(s', s)$. With $C = \mathfrak{m}(c, s')$, the second claim then ensures that $C = \mathfrak{m}(c, s)$. Otherwise, if $t' \notin S$, we apply Claim 1 and Claim 2 recursively to the path of matrix sets ending up with $S' = \mathfrak{m}(s', s)$ and again $C = \mathfrak{m}(c, s)$. This finishes the proof of Lemma 12.4.

Proof of the Claim 1: We first consider the nesting behavior of c_1 and the M-set $\mathfrak{m}(c_2, c_1)$. Since C_2 and C_1 do not intersect, it is $c_2 \notin C_1$. Furthermore, it is clearly $c_1 \notin \mathfrak{m}(c_2, c_1)$. Thus, according to Lemma 7.6(1), $\mathfrak{m}(c_2, c_1)$ and C_1 do also not intersect, that is, $C_1 \cap \mathfrak{m}(c_2, c_1) = \emptyset$. If now the opponent t of $C_2 = \mathfrak{m}(c_2, t)$ is in C_1, we get the situation of Lemma 7.6(2ii) for C_2 and $\mathfrak{m}(c_2, c_1)$ as follows. The vertex t is not in $\mathfrak{m}(c_2, c_1)$, since $\mathfrak{m}(c_2, c_1)$ and C_1 do not intersect, as we have just seen above. On the other hand, c_1 is not in C_2, since C_2 and C_1 do not intersect. Finally c_2 is in $C_2 \cap \mathfrak{m}(c_2, c_1)$. Hence, according to the lemma, it is $C_2 = \mathfrak{m}(c_2, c_1)$. This finishes the proof of Claim 1.

Proof of Claim 2: We consider the nesting behavior of $C_2 = \mathfrak{m}(c_2, c_1)$ and the M-set $\mathfrak{m}(c_2, t)$. If the opponent c_1 of C_2 is not in the M-set $\mathfrak{m}(c_2, t)$, we get the situation of Lemma 7.6(2ii), since t is also not in C_2 (by assumption) and clearly it is $c_2 \in C_2 \cap \mathfrak{m}(c_2, t)$. Hence, it holds $C_2 = \mathfrak{m}(c_2, t)$, and we are done. We show now by contradiction that the remaining case $c_1 \in \mathfrak{m}(c_2, t)$ does not occur.

Assume $c_1 \in \mathfrak{m}(c_2, t)$. Since $t \notin C_2$, but clearly $c_2 \in C_2 \cap \mathfrak{m}(c_2, t)$, applying Lemma 7.6(2i) to C_2 and $\mathfrak{m}(c_2, t)$ yields $C_2 \subseteq \mathfrak{m}(c_2, t)$. Consequently, $(C_2, V \setminus C_2)$ also separates c_2 and t and it is $c(C_2, V \setminus C_2) = \lambda(c_2, c_1) > \lambda(c_2, t)$. Again by Lemma 7.6(2i), we also get $C_1 \subseteq \mathfrak{m}(c_2, t)$, since clearly $t \notin C_1$, but $c_1 \in C_1 \cap \mathfrak{m}(c_2, t)$. Consequently, the U-cut $\mathfrak{uc}(c_2, t)$ also separates c_1 and t and it is $c(\mathfrak{uc}(c_2, t)) = \lambda(c_2, t) \geq \lambda(c_1, t)$. Together, this yields $\lambda(c_2, c_1) > \lambda(c_1, t)$ and $C_1, C_2 \subseteq \mathfrak{m}(c_2, t)$.

Now recall that $\pi(c_2, c_1) \cap \pi(c_1, s) = \{c_1\}$ and $t \in \pi(c_1, s)$, and observe that, by construction, $\mathfrak{m}(c_2, t)$ is assigned to a cheapest edge e_1 on $\pi(c_2, t)$, C_1 to a cheapest edge e_2 on $\pi(c_1, t)$, and C_2 to a cheapest edge e_3 on $\pi(c_2, c_1)$ in $\mathcal{T}(G)$. From $C_1, C_2 \subseteq \mathfrak{m}(c_2, t)$ then follows that the edge e_1 must be the first of the edges e_1, e_2, e_3 when traversing $\mathcal{T}(G)$ from s to c_2. Then, however, the algorithm would have found the cluster $\mathfrak{m}(c_2, t)$ instead of the clusters C_1 and C_2, which is a contradiction. This finishes the proof of Claim 2. □

12.2 Constructing Overlay Clusterings

Given k arbitrary disjoint source communities $S_1, \dots, S_k$ in an undirected, weighted graph $G = (V, E, c)$, we show how to construct the overlay clustering for $S_1, \dots, S_k$, that is, an inclusion-maximal clustering $\Omega(S_1, \dots, S_k)$ of G that contains $S_1, \dots S_k$, is hierarchically nested in each maximum SC-clustering $\Omega(S_1), \dots, \Omega(S_k)$, and is maximum in the sense that each clustering that consists of $S_1, \dots, S_k$ and further source communities is hierarchically nested in $\Omega(S_1, \dots, S_k)$.

That such a clustering $\Omega(S_1, \dots, S_k)$ exists and is unique can be seen as follows. Consider two vertices as equivalent if they are in a common cluster in all maximum SC-clusterings $\Omega(S_1), \dots, \Omega(S_k)$ and let the resulting equivalence classes denote the clusters of a clustering Ω'. Since $S_1, \dots, S_k$ are pairwise disjoint, according to Lemma 12.4, each set $S_i \in \{S_1, \dots, S_k\}$ is nested in a cluster $C \in \Omega(S_j)$ for $j = 1, \dots k$. Hence, Ω' contains the clusters $S_1, \dots, S_k$ and is hierarchically nested in each maximum SC-clustering $\Omega(S_1), \dots, \Omega(S_k)$. Clearly, Ω' is also the unique inclusion-maximal clustering that contains $S_1, \dots, S_k$ and is hierarchically nested in $\Omega(S_1), \dots, \Omega(S_k)$. Furthermore, due to the maximality of $\Omega(S_1), \dots, \Omega(S_k)$, it holds that each source community S' that does not intersect with $S_1, \dots, S_k$ is nested in the intersection of the clusterings in $\Omega(S_1), \dots, \Omega(S_k)$ that contain S'. Hence, each clustering that consists of $S_1, \dots, S_k$ and further source communities is hierarchically nested in Ω'. Consequently, Ω', which exists and is unique, corresponds to the desired overlay clustering $\Omega(S_1, \dots, S_k)$. The construction of $\Omega(S_1, \dots, S_k)$ is possible in linear time as described below.

Theorem 12.5. *Let $S_1, \dots, S_k$ denote arbitrary disjoint source communities in G. The unique overlay clustering for $S_1, \dots, S_k$ can be determined in $O(kn)$ time after preprocessing $M \supset \mathcal{T}(G)$.*

The overlay clustering for $S_1, \dots, S_k$ can be determined by the following inductive construction, which directly implies a simple algorithm. We first compute the maximal SC-clustering $\Omega(S_1)$ and color the vertices in each cluster, using different colors for different clusters. Now consider the overlay clustering $\Omega(S_1, \dots, S_i)$ for the first i maximal SC-clusterings and color the vertices in S_{i+1}, which is nested in a cluster of $\Omega(S_1, \dots, S_i)$ (based on Lemma 12.4 and the maximality of $\Omega(S_1), \dots, \Omega(S_k)$), with a new color. Then we execute the algorithm for computing $\Omega(S_{i+1})$, and during this computation, we construct the intersections of each newly found cluster C with the clusters in $\Omega(S_1, \dots, S_i)$. We exploit that the intersection of two subtrees in a tree is again a subtree. Hence, the clusters in $\Omega(S_1, \dots, S_i, S_{i+1})$ will be subtrees in $\mathcal{T}(G)$, since the clusters in $\Omega(S_1), \dots, \Omega(S_i)$ and $\Omega(S_{i+1})$ are subtrees in $\mathcal{T}(G)$ by Lemma 12.1.

The intersection of a cluster C in $\Omega(S_{i+1})$ with each existing (that is, already colored) cluster in $\Omega(S_1, \dots, S_i)$ can be constructed as follows. Let r' denote the first vertex found in C during the computation of $\Omega(S_{i+1})$. We mark r' as root of a new cluster in $\Omega(S_1, \dots, S_i, S_{i+1})$ and choose a new color x for r', besides the color it already has in $\Omega(S_1, \dots, S_i)$. When constructing C (by applying a DFS), we assign the current color x to all vertices visited by the DFS as long as the underlying color in $\Omega(S_1, \dots, S_i)$ does not change. Whenever the DFS visits a vertex r'' (still in C) with a new underlying color, we chose a new color y for r'' and mark r'' as root of a subtree of a new cluster in $\Omega(S_1, \dots, S_i, S_{i+1})$. When the DFS passes r'' on the way back to the parent[1] p of r'', the color of p in $\Omega(S_1, \dots, S_i, S_{i+1})$ becomes the current color again. Continuing in this way yields a coloring that indicates the intersections of C with $\Omega(S_1, \dots, S_i)$. Repeating this procedure for all clusters in $\Omega(S_{i+1})$ finally yields $\Omega(S_1, \dots, S_{i+1})$. The running time is in $O(kn)$, since we just apply k computations of maximal SC-clusterings.

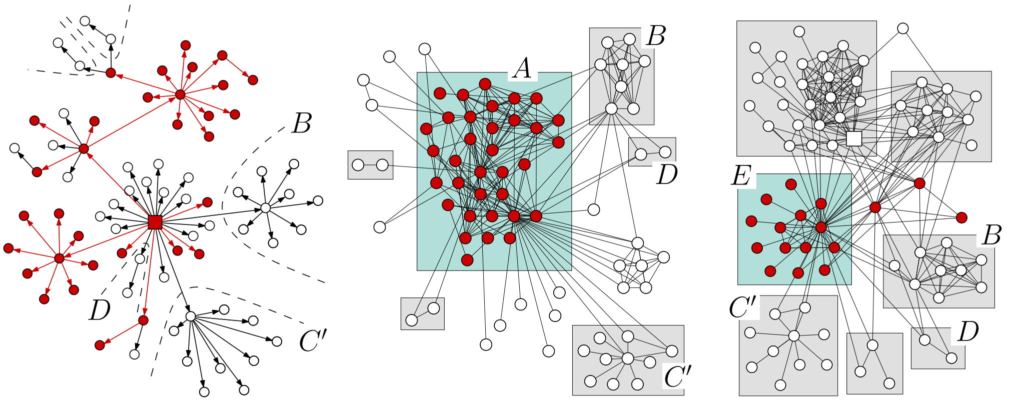
12.3 An Exemplary Analysis of the Lesmis Network

We exemplarily extract two of the many faces of the source-community structure of the weighted lesmis network that we have already seen in the introduction. Suppose, for example, we want to know if there exists a clustering of source communities that also contains the cluster A but less singletons than the cut clustering Ω_4 found by the cut-clustering algorithm (Fig. 12.1(b)). To answer this question, we construct the maximum SC-clustering $\Omega(A)$ for A, which is shown in Fig. 12.6(b). Besides A this clustering also contains the cluster B from Ω_4, a cluster C' that covers C from Ω_4 and three further non-singleton clusters.

Furthermore, we observe that the clustering Ω_5 (Fig. 12.1(c)) on the next higher level in the cut-clustering hierarchy cannot be nested in $\Omega(A)$ due to the size of the cluster A in Ω_5. Hence, the maximum SC-clustering $\Omega(A)$ does not appear above Ω_5 in cut-clustering hierarchy. Conversely, $\Omega(A)$ can neither be nested in Ω_5 nor in Ω_4 due to the cluster C' in $\Omega(A)$. That is,

---

[1] The predecessor adjacent to r'' in the rooted subtree induced by the DFS.



(a) Unique-cut tree $\mathcal{T}(G)$. (b) Maximum SC-clustering $\Omega(A)$. (c) Exemplary overlay clustering.

FIGURE 12.6: Exemplary clusterings of the lesmis network; clusters B, C', D appear in both clusterings and in the underlying unique-cut tree.

we found a completely new clustering of source communities that contains the given cluster A from Ω_4, but less unclustered vertices. Due to the maximality of $\Omega(A)$, we further know that any clustering of source communities that contains A contains at least as many singletons as $\Omega(A)$.

Figure 12.6(a) shows the underlying unique-cut tree $\mathcal{T}(\mathsf{lesmis})$. The root r of $\mathcal{T}(\mathsf{lesmis})$ is depicted as filled square. The subtree $\mathcal{T}[A]$ induced by A in $\mathcal{T}(\mathsf{lesmis})$ is indicated by filled vertices. Since the root r of $\mathcal{T}(\mathsf{lesmis})$ is in A, deleting $\mathcal{T}[A]$ immediately decomposes $\mathcal{T}(\mathsf{lesmis})$ into the unframed singletons and the round framed clusters of $\Omega(A)$ shown in Fig. 12.6(b).

The second example (Fig. 12.6(c)) considers the overlay clustering $\Omega(S_1, \dots, S_6, E)$ with the given source communities $S_1, \dots, S_6$ defined by the non-singleton subtrees of the root r in $\mathcal{T}(\mathsf{lesmis})$. The source community E (filled vertices in squared box) has been computed additionally. It equals the generalized M-set $\mathfrak{m}(r, T)$ with $T := \bigcup_{i=1}^{6} S_i$. If we consider the filled vertices in Fig. 12.6(c) as one cluster $F := V \setminus T$, then $S_1, \dots, S_6$ together with F represent the overlay clustering $\Omega(S_1, \dots, S_6)$. However, F is no source community, since for the two vertices $v_1, v_2 \in F \setminus E$ there exists a vertex $u \in T$ (unfilled square) such that $\mathfrak{m}(v_i, u) \subseteq F$ $(i = 1, 2)$ is a singleton. Hence, according to Lemma 7.6(2i), any source community nested in F, apart from $\{v_1\}$ and $\{v_2\}$, must be in E. This further shows that, in contrast to $\Omega(S_1, \dots, S_6)$, the overlay clustering $\Omega(S_1, \dots, S_6, E)$ indeed consists of source communities and any clustering that also consists of source communities and contains $S_1, \dots, S_6, E$ is nested in $\Omega(S_1, \dots, S_6, E)$. Compared to the other clusterings that we have seen so far, $\Omega(S_1, \dots, S_6, E)$ contains only two singletons and the clusters are more balanced. Hence, we conclude that the lesmis network admits several diverse decompositions into source communities, which are not all found by the cut-clustering algorithm. A further example of a maximum SC-clustering is shown in Fig. 1.4.

CHAPTER 13

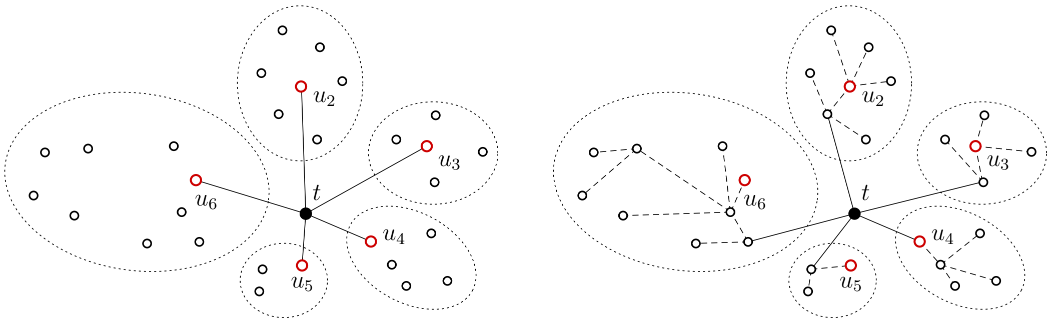
The Unrestricted Cut-Clustering Algorithm

As already mentioned in Section 11.1, Flake et al. [42] develop their final cut-clustering method step-by-step, each time simplifying a previous version. The first version, for which they already prove the quality guarantee, but do not consider a hierarchical approach, admits the use of arbitrary minimum separating cuts instead of U-cuts. Although arbitrary minimum separating cuts may cross, even if they all share a common vertex t in their cut pairs, their use for the cut-clustering technique is possible due to the close relation of the cut-clustering approach to Gomory-Hu trees. The cut-clustering algorithm described by CutC is basically a partial Cut Tree execution on the graph G_α with a special sequence of split cuts, namely U-cuts with respect to the artificial vertex t and the later representatives of the clusters (recall Fig. 11.1(a)). Hence, replacing a few lines in the Gomory-Hu tree construction Cut Tree would result in an algorithm that basically does the same as CutC. In this light it is also easy to formulate a cut-clustering approach that admits the use of arbitrary minimum separating cuts instead of U-cuts. We just replace line 2, line 3, and line 4, which define the choice of step pairs and split cuts in Cut Tree (see Algorithm 8). As input graph for the resulting *unrestricted* cut-clustering approach, called Unrestricted CutC, we use G_α, and to obtain the final clustering from the returned intermediate tree T_*, we simply delete the artificial vertex t, which is a singleton in T_* due to the condition of the while-loop in line 2. This decomposes T_* into connected components, which are interpreted as clusters. We call a clustering that can be found by Unrestricted CutC an *unrestricted cut clustering.*

In this way, any intermediate tree on G_α that contains t as a singleton node induces an unrestricted cut clustering. We call such a tree a *clustering tree.* We remark further that continuing the execution of Cut Tree on a clustering tree might change the edge structure of the subtrees incident to t, but does not change the set of vertices contained in each subtree. That is, the resulting Gomory-Hu tree would induce the same clustering as the clustering tree. Hence, for each clustering tree we may assume an underlying Gomory-Hu tree, that is, a fictional edge structure within the induced clusters (see also Fig. 13.1(b)). Vice versa, the clusters of an unrestricted cut clustering and their representatives induce a sequence of step pairs and split cuts, and thus, an Unrestricted CutC execution that returns a clustering tree that induces the clustering (Fig. 13.1(a)). Note that an unrestricted cut clustering can be induced by several clustering trees that differ in those edges that are not incident to t. Hence, as a convention, we identify an unrestricted cut clustering with the clustering tree that results from the Unrestricted CutC execution induced by the clustering. This clustering tree contains, besides t, exactly the clusters as nodes (see Fig. 13.1). This view on unrestricted cut clusterings

Algorithm 8: UNRESTRICTED CUTC

```
Input: Graph $G_\alpha = (V_\alpha, E_\alpha, c_\alpha)$
Output: Gomory-Hu tree of $G_\alpha$
Initialize tree $T_* = (V, E_t, E_f, c_f)$ with $E_t \leftarrow$ thin edges forming an arbitrary spanning tree of $V$, $E_f \leftarrow \emptyset$ and $c_f$ empty
while $t$ is no singleton node do                                          // unfold node of $t$
    $S \leftarrow$ subtree of thin edges that contains $t$                            // choose node
    $\{u, t\} \leftarrow$ step pair with $u$ arbitrary vertex in $S$                    // choose step pair
    $(U, V \setminus U) \leftarrow$ min-$u$-$t$-cut in $G_\alpha$, cost $\lambda_{G_\alpha}(u, t)$, $u \in U$       // choose split cut
    $E_t \leftarrow E_t \setminus S$
    forall the $x \in S \setminus \{u, t\}$ do                                    // split $S = S_u \dot\cup S_t$
        if $x \in U$ then $E_t \leftarrow E_t \cup \{\{u, x\}\}$;                       // reconnect $x$ to $u$
        if $x \in V \setminus U$ then $E_t \leftarrow E_t \cup \{\{v, x\}\}$;                 // reconnect $x$ to $t$
    $N \leftarrow$ all vertices $x$ that are linked by a fat edge to a vertex $v_x \in S$
    forall the $x \in N$ do                          // reconnect subtrees to $S_u$ and $S_t$
        if $x \in U$ then $E_t \leftarrow (E_t \setminus \{\{v_x, x\}\}) \cup \{\{u, x\}\}$;             // reconnect $x$ to $u$
        if $x \in (V \setminus U)$ then $E_t \leftarrow (E_t \setminus \{\{v_x, x\}\}) \cup \{\{v, x\}\}$;      // reconnect $x$ to $t$
    $E_f \leftarrow E_f \cup \{\{u, t\}\}$, $E_t \leftarrow E_t \setminus \{\{u, t\}\}$, $c_f(u, t) \leftarrow \lambda_{G_\alpha}(u, t)$       // draw $\{u, t\}$ fat
return $T_*$
```



(a) Clustering tree resulting from an UNRESTRICTED CUTC execution using the cuts induced by a given clustering.

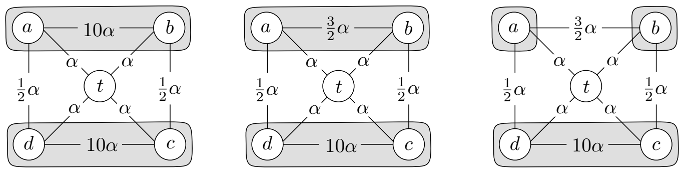
(b) Clustering tree with underlying Gomory-Hu tree, fictive edges as dashed lines.

FIGURE 13.1: Schematic illustration of the clustering tree induced by an unrestricted cut clustering via an UNRESTRICTED CUTC execution with step pairs consisting of t and the representatives $u_2, \ldots, u_6$ of the clusters and split cuts induced by the clusters. Note that, in contrast to the edges of the clustering tree (a), the edges incident to t in the underlying Gomory-Hu tree (b) may be also connected to vertices different from the representatives u_i.

will become in particular important in the next chapter, where we consider dynamic updates on unrestricted cut clusterings based on the results for dynamic Gomory-Hu trees.

The algorithm UNRESTRICTED CUTC is able to return also any cut clustering that can be found by CUTC. To this end, we simply restrict the used split cuts again to U-cuts. In the following we call an unrestricted cut clustering that can be also found by CUTC a *restricted cut clustering*. If we want to stress the difference between UNRESTRICTED CUTC and CUTC, we further call CUTC the restricted cut-clustering algorithm. We finally point out that, in contrast to restricted cut clusterings, unrestricted cut clusterings are not unique for a designated parameter value α.

Flake et al. already showed that unrestricted cut clusterings provide the same quality guarantee as restricted cut clusterings, but the authors did not investigate the nesting behavior of unrestricted cut clusterings for different values of α. Since unrestricted cut clusterings are not



(a) Initial unique cut clustering.

(b) New unrestricted cut clustering.

(c) New restricted cut clustering.

FIGURE 13.2: Example how unrestricted cut clusterings may support temporal smoothness. After a change of the cost of edge $\{a, b\}$ from 10α to $3/2\alpha$, the unrestricted cut-clustering approach allows to retain the old clustering and avoids singletons.

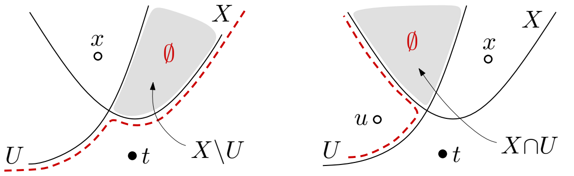
unique for fixed values of α, one might ask whether it is even possible to efficiently find clusterings that are hierarchically nested. In this chapter we show that arbitrary unrestricted cut clusterings with respect to different values of α are always hierarchically nested, independent from the chosen step pairs and split cuts in UNRESTRICTED CUTC. Hence, analogous to the hierarchical restricted cut-clustering approach based on CUTC, we get a hierarchical unrestricted cut-clustering approach based on UNRESTRICTED CUTC. In other words, we prove Theorem 13.1, which implies that applying UNRESTRICTED CUTC iteratively with decreasing α always yields a hierarchy of at most n different unrestricted cut clusterings, where the clustering quality on level i depends on α_i (recall the indexing of hierarchy levels and clusterings in Fig. 11.2).

Theorem 13.1. *Given a sequence $\alpha_1 > \cdots > \alpha_r$ of parameter values each set of unrestricted cut clusterings $\Omega_1(G), \ldots, \Omega_r(G)$ forms a hierarchy.*

In the next chapter we will further develop a dynamic unrestricted cut-clustering approach that applies for single unrestricted cut clusterings as well as for hierarchies of unrestricted cut clusterings. In dynamic scenarios, unrestricted cut clusterings reveal their hidden strength. Since, in contrast to the restricted approach, the unrestricted cut clustering algorithm is able to choose the most appropriate cut with respect to a given task, it is able to employ its additional degree of freedom for example to the benefit of temporal smoothness, thus compensating the fact that the found clusters are not guaranteed to be source communities. Figure 13.2 gives a brief example of a dynamic graph, where the initial unique cut clustering is restricted, while after an atomic change the resulting graph also admits an unrestricted cut clustering besides the unique restricted one. This unrestricted cut clustering, which still provides a quality with respect to the same bounds as the previous clustering, since the parameter value did not change, equals the previous clustering and is thus optimal in terms of temporal smoothness. The restricted cut clustering for the new graph however differs from the previous clustering and contains some singletons.

Proof of Theorem 13.1. The main ingredient of Theorem 13.1 is the nesting behavior of arbitrary minimum separating cuts with respect to different values of α. Lemma 13.2 characterizes the important aspects of this nesting behavior with respect to unrestricted cut clusterings.

Lemma 13.2. *Let $(U, V_{\alpha_j} \setminus U)$ denote a minimum u-t-cut in G_{α_j} ($u \in U$), and for $\alpha_i > \alpha_j$ let $(X, V_{\alpha_i} \setminus X)$ denote a minimum x-t-cut in G_{α_i} ($x \in X$). Then it holds (a) $X \subseteq U$ if $x \in U$, (b) $X \cap U = \emptyset$ if $x \notin U$, $u \notin X$, and (c) the case $x \notin U$, $u \in X$ does not occur if there exists an unrestricted cut clustering $\Omega_j(G)$ that contains U as a cluster.*



(a) If $x \in U$, then $X \subseteq U$. (b) If $x \notin U$ and $u \notin X$, then $X \cap U = \emptyset$.

FIGURE 13.3: Situation of Lemma 13.2. Filled areas are proven to not exist. Instead, U is shaped as indicated by the dashed red lines.

From Lemma 13.2 follows directly that two unrestricted cut clusterings $\Omega_i(G)$ and $\Omega_j(G)$ with respect to different values $\alpha_i > \alpha_j$ are hierarchically nested, that is, $\Omega_i(G) \leq \Omega_j(G)$. This immediately proves Theorem 13.1. Finally, the following corollary, which we will apply in Chapter 14 in the context of dynamic clustering hierarchies, can be deduced from Theorem 13.1.

Corollary 13.3. *A cluster $C \in \Omega_j(G)$ separates G into C and $V \setminus C$ such that both parts are clustered independently with respect to $\alpha_i > \alpha_j$, that is, no minimum s-t-cut in G_{α_i} with $s \in C$ separates any vertex in $V \setminus C$ from t, and vice versa.*

Otherwise there would exist a cut clustering $\Omega_i(G)$ that is not hierarchically nested in $\Omega_j(G)$ contradicting Theorem 13.1.

Proof of Lemma 13.2. Consider $\theta_i := (X, V_{\alpha_i} \setminus X)$ and $\theta_j := (U, V_{\alpha_j} \setminus U)$. We distinguish two cases depending on the shape of θ_j. Case (a) is characterized by $x \in U$, Case (b) by $x \in X \setminus U$ and $u \in U \setminus X$ (see also Fig. 13.3). Finally, we show that the situation where $x \notin U$ but $u \in X$ (Case (c)) does not occur if U is a cluster in an unrestricted cut clustering $\Omega_j(G)$.

Case (a): If $x \in U$, then $X \subseteq U$. We assume $X \setminus U \neq \emptyset$ and show that in this case $(U \cup X, V_{\alpha_j} \setminus (U \cup X))$ in G_{α_j} is cheaper than θ_j. This contradicts θ_j being a minimum u-t-cut in G_{α_j} and we conclude $X \setminus U = \emptyset$, that is, θ_j does not cut through X (dashed line in Fig. 13.3(a)). In the following, we compare different costs, which we express in terms of costs in G and an addend depending on α. For θ_i and $(U \cap X, V_{\alpha_i} \setminus (U \cap X))$ we get

$$\begin{aligned}
c_{\alpha_i}(\theta_i) = c_{\alpha_i}(X, V_{\alpha_i} \setminus X) &= c(X \setminus U, V \setminus (U \cup X)) &+ c(U \cap X, V \setminus (U \cup X))\\
&+ c(X \setminus U, U \setminus X) &+ c(U \cap X, U \setminus X)\\
&+ \alpha_i |X| &\\
c_{\alpha_i}(U \cap X, V_{\alpha_i} \setminus (U \cap X)) &= c(X \setminus U, U \cap X) &+ c(U \cap X, V \setminus (U \cup X))\\
&+ \alpha_i |U \cap X| &+ c(U \cap X, U \setminus X)
\end{aligned}$$

Since θ_i is a minimum x-t-cut in G_{α_i} it holds $c_{\alpha_i}(U \cap X, V_{\alpha_i} \setminus (U \cap X)) \geq c_{\alpha_i}(\theta_i)$, such that expressing these costs as shown above yields $c(X \setminus U, U \cap X) \geq c(X \setminus U, V \setminus (U \cup X)) + c(X \setminus U, U \setminus X) + \alpha_i |X \setminus U|$. With the trivial insight that $c(X \setminus U, U \setminus X) \geq 0$, it holds in particular

$$\textit{(i)} \quad c(X \setminus U, U \cap X) \; + \; c(X \setminus U, U \setminus X) \; \geq \; c(X \setminus U, V \setminus (U \cup X)) \; + \; \alpha_i |X \setminus U|$$

For θ_j and $(U \cup X, V_{\alpha_j} \setminus (U \cup X))$ we get

$$\begin{aligned}
c_{\alpha_j}(\theta_j) = c_{\alpha_j}(U, V_{\alpha_j} \setminus U) &= c(U \setminus X, V \setminus (U \cup X)) &&+ c(U \cap X, V \setminus (U \cup X)) \\
&+ c(X \setminus U, U \cap X) &&+ c(X \setminus U, U \setminus X) \\
&+ \alpha_j |U| \\
c_{\alpha_j}(U \cup X, V \setminus (U \cup X)) &= c(U \setminus X, V \setminus (U \cup X)) &&+ c(U \cap X, V \setminus (U \cup X)) \\
&+ c(X \setminus U, V \setminus (U \cup X)) &&+ \alpha_j |U \cup X|
\end{aligned}$$

and finally

$$\begin{aligned}
c_{\alpha_j}(U \cup X, V \setminus (U \cup X)) - c_{\alpha_j}(\theta_j) &= [c(X \setminus U, V \setminus (U \cup X)) + \alpha_j |X \setminus U|] \\
&- [c(X \setminus U, U \cap X) + c(X \setminus U, U \setminus X)] \\
&< 0
\end{aligned}$$

since assuming that $X \setminus U \neq \emptyset$ and replacing α_i in (i) by α_j, which is lower, yields $c(X \setminus U, V \setminus (U \cup X)) + \alpha_j |X \setminus U| < c(X \setminus U, U \cap X) + c(X \setminus U, U \setminus X)$.

Case (b): If $x \notin U$ and $u \notin X$, then $X \cap U = \emptyset$. We assume $X \cap U \neq \emptyset$ and show that in this case $(U \setminus X, V_{\alpha_j} \setminus (U \setminus X))$ in G_{α_j} is cheaper than θ_j. This contradicts θ_j being a minimum u-t-cut in G_{α_j} and we conclude $X \cap U = \emptyset$, that is, θ_j does not cut through X (dashed line in Fig. 13.3(b)). In the following, we compare different costs, which we express in terms of costs in G and an addend depending on α. For θ_i and $(X \setminus U, V_{\alpha_i} \setminus (X \setminus U))$ we get

$$\begin{aligned}
c_{\alpha_i}(\theta_i) = c_{\alpha_i}(X, V_{\alpha_i} \setminus X) &= c(X \cap U, U \setminus X) &&+ c(X \setminus U, U \setminus X) \\
&+ c(X \cap U, V \setminus (U \cup X)) &&+ c(X \setminus U, V \setminus (U \cup X)) \\
&+ \alpha_i |X| \\
c_{\alpha_i}(X \setminus U, V_{\alpha_i} \setminus (X \setminus U)) &= c(X \cap U, X \setminus U) &&+ c(X \setminus U, U \setminus X) \\
&+ \alpha_i |X \setminus U| &&+ c(X \setminus U, V \setminus (U \cup X))
\end{aligned}$$

Since θ_i is a minimum x-t-cut in G_{α_i}, it holds $c_{\alpha_i}(X \setminus U, V_{\alpha_i} \setminus (X \setminus U)) \geq c_{\alpha_i}(\theta_i)$, and we get $c(X \cap U, X \setminus U) \geq c(X \cap U, U \setminus X) + c(X \cap U, V \setminus (U \cup X)) + \alpha_i |X \cap U|$ by expressing these costs as shown above. Replacing α_i by α_j, which is lower, and assuming that $X \cap U \neq \emptyset$ further yields $c(X \cap U, X \setminus U) > c(X \cap U, U \setminus X) + c(X \cap U, V \setminus (U \cup X)) + \alpha_j |X \cap U|$. With the trivial insight that $c(X \cap U, V \setminus (U \cup X)) + \alpha_j |X \cap U| \geq 0$, it holds in particular

$$\textit{(ii)} \quad c(X \cap U, X \setminus U) \; + \; c(X \cap U, V \setminus (U \cup X)) \; + \; \alpha_j |X \cap U| \; > \; c(X \cap U, U \setminus X)$$

For θ_j and $(U \setminus X, V_{\alpha_j} \setminus (U \setminus X))$ we get

$$\begin{aligned}
c_{\alpha_j}(\theta_j) = c_{\alpha_j}(U, V_{\alpha_j} \setminus U) &= c(U \setminus X, V \setminus (U \cup X)) &&+ c(X \cap U, V \setminus (U \cup X)) \\
&+ c(U \setminus X, X \setminus U) &&+ c(X \cap U, X \setminus U) \\
&+ \alpha_j |U| \\
c_{\alpha_j}(U \setminus X, V_{\alpha_j} \setminus (U \setminus X)) &= c(U \setminus X, V \setminus (U \cup X)) &&+ c(X \cap U, U \setminus X) \\
&+ c(U \setminus X, X \setminus U) &&+ \alpha_j |U \setminus X|
\end{aligned}$$

and finally, due to (ii),

$$\begin{aligned}
c_{\alpha_j}(U \setminus X, V_{\alpha_j} \setminus (U \setminus X)) - c_{\alpha_j}(\theta_j) &= c(X \cap U, U \setminus X) \\
&- c(X \cap U, V \setminus (U \cup X)) \\
&- c(X \cap U, X \setminus U) - \alpha_j |X \cap U| \\
&< 0
\end{aligned}$$

Case (c): The situation that $x \notin U$ *but* $u \in X$ *does not occur if* U *is a cluster in an unrestricted cut clustering* $\Omega_j(G)$*.* We prove that $x \notin U$ and $u \in X$ contradicts Case (a). Assume $x \in X \setminus U$ and let C denote the cluster in $\Omega_j(G)$ containing x. Then, by Lemma 7.6(2i), the source set $\mathfrak{m}(x, t)$ is a subset of $\mathfrak{m}(r(C), t)$ in G_{α_j}, and thus, also of C ($\mathfrak{m}(x, t) \subseteq C$). This however contradicts Case (a) (with C in the role of U), since $X \setminus \mathfrak{m}(x, t)$, and thus, $X \nsubseteq C$, if we additionally assume $u \in X$. □

CHAPTER 14

Fully-Dynamic Hierarchies of Unrestricted Cut Clusterings

In times of constantly increasing mobility accompanied by a technological progress that admits to interact with other individuals and most different devices from almost everywhere at almost any time, a huge amount of evolving data has emerged in the last few years and will emerge even more rapidly in the future. Hence, powerful tools to examine the structure of also evolving data become more and more important. In the context of graph clustering, there are meanwhile many activities in various scientific fields like physics, biology, sociology, and informatics that aim at developing new clustering approaches for evolving and dynamic networks or adapting existing static approaches to dynamic scenarios. For a survey on recent results on clustering evolving networks see, for example, [76] and the references therein.

In this chapter, we adapt the hierarchical, unrestricted cut-clustering approach, which is a further development of the elegant cut-clustering approach of Flake et al. [42] (see Chapter 13), to graphs that evolve due to atomic changes. An atomic change is either the increase or decrease of an edge cost, the insertion or deletion of an edge, or the insertion or deletion of a vertex, where vertex changes only occur for isolated vertices. The edge changes induce four different scenarios with respect to a cut clustering, namely the increase and decrease of the cost of an intra-cluster edge, and the increase and decrease of the cost of an inter-cluster edge. We note that, in contrast to dynamic Gomory-Hu trees (Chapter 8), edge insertion and deletion can be also considered as special cases of increasing or decreasing edge costs in the context of cut clustering. This is due to the fact that the extended graph G_α used by the cut-clustering approach is always connected and does not contain bridges. Furthermore, an isolated vertex can be easily added by just introducing a new singleton cluster and deleted without any further action, since an isolated vertex always forms a singleton cluster in a cut clustering. Hence, in this chapter we focus on the four non-trivial cases resulting from the edge changes. In a first step, we present an update procedure for each case that efficiently and dynamically maintains a single unrestricted cut clustering with a persisting quality and temporal smoothness, that is, consecutive clusterings are kept similar. There has already been an attempt to solve this task, by Saha and Mitra [123, 124], however, we found this attempt to be erroneous beyond straightforward correction. In a second step we extend these procedures to hierarchies of unrestricted cut clusterings. Due to the close relation of the unrestricted cut-clustering algorithm to the construction of a Gomory-Hu tree, we can build upon many insights that we have achieved in Part II in the context of dynamic Gomory-Hu trees. On the other hand, due to this close relation, it is also not surprising that our update procedures for unrestricted cut clusterings provide no guarantee on the asymptotic worst-case running time better than $n-1$ minimum-cut computations. For updates of Gomory-Hu trees, we

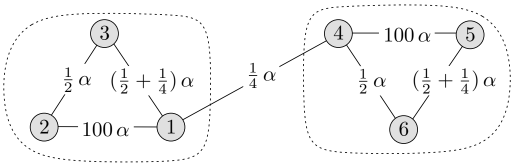
Algorithm 9: EARLIER INTER-EDGE INSERTION

```
Input: $G = (V, E, c)$, $\alpha$, $\Omega(G)$, $\{C_b, C_d\} \subseteq \Omega(G)$ with $b \in C_b$, $d \in C_d$
        // Recall that $\{b, d\}$ denotes the changing edge in a dynamic graph $G$
if inter-cluster quality of $C_d$ and $C_b$ is maintained with respect to $\alpha$ then
    return $\Omega$                                        // Case 1: do nothing
else if $\frac{2c(C_b,C_d)}{|V|} \geq \alpha$ then
    return $(\Omega \setminus \{C_b, C_d\}) \cup \{\{C_b \cup C_d\}\}$      // Case 2: merge $C_b$ and $C_d$
obtain $G_\alpha$ from $G$ according to CUTC                         // Case 3 (default)
in $G_\alpha$, contract all clusters in $\Omega(G) \setminus \{C_b, C_d\}$ to one node, dissolve $C_b$ and $C_d$
perform UNRESTRICTED CUTC (Algorithm 8) on resulting $G_\alpha$
return $(\Omega \setminus \{C_b, C_d\}) \cup \{$newly formed clusters from $C_b \cup C_d\}$
```

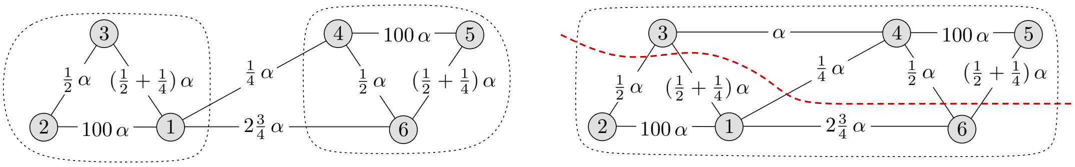
have already seen a presumably tight bound of the asymptotic worst-case running time of $n-1$ minimum-cut computations in Section 9.3. However, as already observed for dynamic Gomory-Hu trees, these worst cases occur very rarely. Moreover, we will see, by experimentally evaluating the performance of our procedures in comparison with the original unrestricted cut-clustering algorithm, that our update approach provides a large potential for saving cut computations. Note that, in contrast to the construction of a Gomory-Hu tree, the construction of an unrestricted cut clustering from scratch does not need a fixed number of cut computations. Instead, Flake et al. [42] observed that the number of computed minimum cuts is proportional to the number of found clusters, at least for the restricted cut-clustering approach.

14.1 The Dynamic Attempt of Saha and Mitra

Saha and Mitra [123, 124] published an algorithm that aims at the same goal as our work. The authors describe four procedures for updating a clustering and a data structure for the deletion and the insertion of intra-cluster and inter-cluster edges. Unfortunately, we discovered a methodical error in their work. Roughly speaking, it seems as if the authors implicitly (and erroneously) assume an equivalence between clusterings that just provide the quality guarantee and clusterings that can be indeed found by the unrestricted cut-clustering algorithm. A full description of issues is beyond the scope of this work, but we briefly point out errors in the authors' procedure that deals with the insertion of inter-cluster edges and give counterexamples in the following. These issues, alongside correct parts, are further scrutinized in-depth by Hartmann [75]. Algorithm 9 sketches the approach given in [124] for handling edge insertions between clusters. Summarizing, we found that Case 1 does maintain the quality guarantee but not the invariant that each updated clustering is supposed to be again an unrestricted cut clustering. Case 2 maintains both, the quality guarantee and the invariant, if and only if the input fulfills the invariant. However it can be shown that this case is of purely theoretical interest and extremely improbable. Finally, Case 3 neither maintains the quality guarantee nor the invariant. In the following we illustrate some of these shortcomings with examples. We further remark that these examples also disprove the correctness of the attempt of Saha and Mitra with respect to restricted cut clusterings.



(a) Graph G^0 with (restricted) cut clustering $\Omega(G^0) = \{C_b, C_d\}$ (clusters framed by dotted lines) found by the unrestricted cut-clustering algorithm.



(b) Adding inter-cluster edge $\{1, 6\}$ with cost $11/4\alpha$ yields G^1 with clustering $\Omega(G^1) = \Omega(G^0)$ resulting from Case 1 of Algorithm 9.

(c) Adding edge $\{3, 4\}$ with cost α yields G^2 with the trivial clustering $\Omega(G^2)$ resulting from Case 2 that violates the guarantee on intra-cluster quality (see cut indicated by dashed red line).

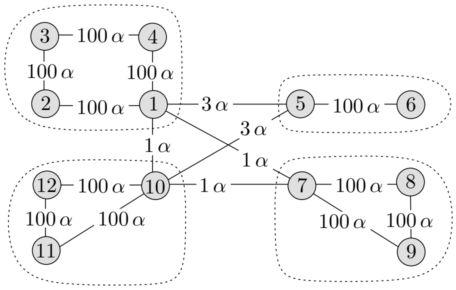
FIGURE 14.1: A dynamic instance violating the quality guarantee. Edge costs are parameterized by α corresponding to the parameter used in the cut-clustering algorithm. After two atomic changes in G^0, according to Case 1 and 2, Algorithm 9 returns one cluster which admits a cut (dashed red line) that violates the intra-cluster quality.

14.1.1 A Counterexample for Case 1 and Case 2

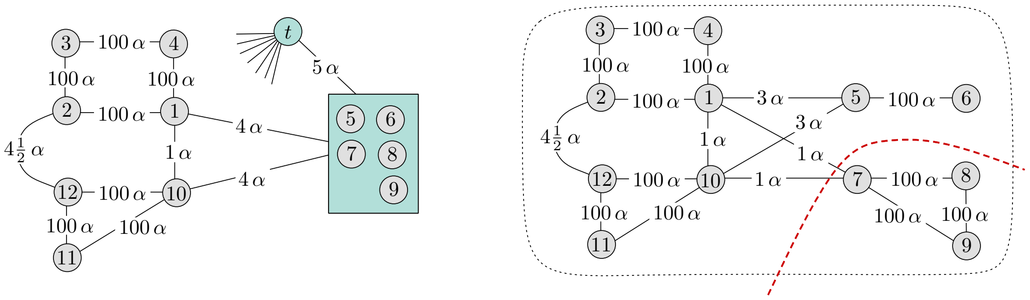
We give an example which Algorithm 9, taken from [124], fails to cluster correctly. Figure 14.1(a) shows the initial graph G^0 and the initial input clustering $\Omega(G^0)$ found by the unrestricted cut-clustering algorithm. This clustering consists of two clusters C_b and C_d of equal size and equal inter-cluster expansion $\overline{\psi}(C_b) = \overline{\psi}(C_d) = 1/12\alpha \leq \alpha$, which satisfies the guarantee on inter-cluster quality. Since $\Omega(G^0)$ was constructed by the (unrestricted) cut-clustering algorithm, it also satisfies the guarantee on intra-cluster quality (recall Equation (11.1) in Section 11.1.1). In Fig. 14.1(b), a first edge insertion then increases the inter-cluster expansion of both clusterings to α, which still satisfies the guarantee on inter-cluster quality, and thus triggers Case 1. Consequently, the clustering in Fig. 14.1(b) is kept unchanged. Obviously, the intra-cluster quality is also maintained, since the intra-cluster edges did not change. Then, in Fig. 14.1(c), a second edge with cost α is added between C_b and C_d. This edge insertion is now handled by Case 2, since the inter-cluster quality is violated ($\overline{\psi}(C_b) = \overline{\psi}(C_d) = 4/3\alpha > \alpha$) and the condition for Case 2 in line 3 of Algorithm 9 is fulfilled ($2 \cdot 4/6\alpha > \alpha$). Thus, C_b and C_d are merged, resulting in the cluster shown in Fig. 14.1(c). In this cluster the dashed line however indicates an intra-cluster cut with cost $11/4\alpha < 3\alpha$, which violates the guarantee on intra-cluster quality.

14.1.2 A Counterexample for Case 3

We also give an example which Algorithm 9, taken from [124], fails to cluster correctly due to shortcomings in Case 3. Figures 14.2(a) describes again the initial graph and the initial input clustering $\Omega(G^0)$ found by the unrestricted cut-clustering algorithm before edge $\{2, 12\}$ with cost $9/2\alpha$ is inserted. The initial clustering consists of four clusters that all satisfy the guarantee on inter-cluster quality. After the insertion of edge $\{b, d\} = \{2, 12\}$, the inter-cluster expansion of C_b and C_d in the resulting graph G^1 increases to $\overline{\psi}(C_b) = 19/16\alpha > \alpha$ and $\overline{\psi}(C_d) = 19/18\alpha > \alpha$, which violates the guarantee on inter-cluster quality. Thus, Case 1 does not occur.



(a) Graph G^0 with (restricted) cut clustering $\Omega(G^0)$ (clusters framed by dotted lines) found by the unrestricted cut-clustering algorithm. After inserting edge $\{2, 12\}$, G^0 becomes G^1 (Fig. (c)).



(b) Graph G' resulting from G^1_α by contracting the vertices in $\Omega(G^0) \setminus \{C_b, C_d\}$ and dissolving C_b and C_d, according to Case 3 of Algorithm 9.

(c) Applying UNRESTRICTED CUTC to G' (in Fig. (b)) yields only one cluster for G^1 that violates the guarantee on intra-cluster quality (see cut indicated by dashed red line).

FIGURE 14.2: A dynamic instance violating the quality guarantee. Edge costs are parameterized by α corresponding to the parameter used in the cut-clustering algorithm. After inserting edge $\{2, 12\}$ into G^0, according to Case 3, Algorithm 9 returns one cluster which admits a cut (dashed red line) that violates the intra-cluster quality.

Furthermore, the condition of Case 2 is also violated, since $2 \cdot 11/24\alpha = 11/12\alpha < \alpha$. Hence, Case 3 in line 5 is executed. Figure 14.2(b) depicts the instance G' resulting from contracting $\Omega(G^0) \setminus \{C_b, C_d\}$ in G^1_α and dissolving C_b and C_d (see line 6). Applying UNRESTRICTED CUTC to G' then results in a single cluster for G^1 (see Fig. 14.2(c)), since separating t from any other vertex in G' costs at least 12α and the only minimum s-t-cut with $s \in V$ and cost 12α in G' is the cut $(\{t\}, V \setminus \{t\})$. In this cluster the dashed line, however, indicates an intra-cluster cut with cost $2\alpha < 3\alpha$, which again violates the guarantee on intra-cluster quality. Furthermore, the resulting clustering for G^1 does not conform to what is attempted to be proven in [124], since a "newly formed cluster from $C_b \cup C_d$" as returned in line 8 does not exist. Instead the resulting cluster contains all vertices.

14.2 Correct Update Procedures for Single Clusterings

For the design of our update procedures for single unrestricted cut clusterings we exploit the results that we have already gained in the context of dynamic Gomory-Hu trees. To this end, we identify an unrestricted cut clustering $\Omega(G)$ of an undirected, weighted graph G with the clustering tree returned by UNRESTRICTED CUTC for the step pairs and split cuts induced by the clustering, as described in Chapter 13 (recall Fig. 13.1). We denote this clustering tree also by $\Omega(G)$.

By simply applying the very fundamental insight of Lemma 8.1, we can thus directly determine some reusable cuts in this clustering tree. Let $\Omega(G)$ denote an unrestricted cut clustering in an undirected, weighted graph $G = (V, E, c)$ and let the cost of edge $\{b, d\} \in E$ change by $\Delta > 0$. Then, the path $\pi(b, d)$ in an underlying Gomory-Hu tree of the clustering tree $\Omega(G)$ contains exactly two edges, namely $\{t, C_b\}$ and $\{t, C_d\}$, if $\{b, d\}$ is an inter-cluster edge, and no edge if $\{b, d\}$ is an intra-cluster edge. According to Lemma 8.1, we thus get the following.

(1) *Intra-cluster cost increase:* Path $\pi(b, d)$ contains no edge. Hence, each edge in $\Omega(G)$ remains a minimum separating cut in $G^{\oplus}_{\alpha}$ with the previous cost and with respect to the previous cut pairs.

(2) *Inter-cluster cost increase:* Path $\pi(b, d)$ contains $\{t, C_b\}$ and $\{t, C_d\}$. Hence, except for $\{t, C_b\}$ and $\{t, C_d\}$, each edge in $\Omega(G)$ remains a minimum separating cut in $G^{\oplus}_{\alpha}$ with the previous cost and with respect to the previous cut pairs.

(3) *Inter-cluster cost decrease:* Path $\pi(b, d)$ contains $\{t, C_b\}$ and $\{t, C_d\}$. Hence, the edges $\{t, C_b\}$ and $\{t, C_d\}$ remain minimum separating cuts in $G^{\ominus}_{\alpha}$ with the previous cost minus Δ and with respect to the previous cut pairs.

(4) *Intra-cluster cost decrease:* Path $\pi(b, d)$ contains no edge. Hence, applying Lemma 8.1 yields no reusable cuts.

Analogous to the approach for dynamic Gomory-Hu trees, the idea is now to construct a valid intermediate tree from $\Omega(G)$ that already contains the found reusable cuts as fat edges and the remaining old edges as thin edges, and can be further processed by a modified version of the static algorithm (which is Unrestricted CutC in the case of unrestricted cut clusterings). Figure 14.3 schematically illustrates the resulting intermediate trees for the four scenarios above.

14.2.1 Procedures for Increasing Costs

Intra-Cluster Cost Increase. In case of an intra-cluster cost increase (Fig. 14.3(a)), the resulting intermediate tree is already a cluster tree, since t is already a singleton. Hence, there is no need to further process this tree. It induces the previous clustering.

Inter-Cluster Cost Increase. In case of an inter-cluster cost increase (Fig. 14.3(b)), the two thin edges $\{t, C_b\}$ and $\{t, C_d\}$ represent old minimum separating cuts with respect to t and the representatives $r(C_b)$, $r(C_d)$. In this case we modify Unrestricted CutC such that it first considers U-cuts $\mathfrak{uc}(s, t)$ in $G^{\oplus}_{\alpha}$ with $s \in C_b$ (line 5) until all vertices in C_b are separated from t. Afterwards, we proceed in the same way with the vertices in C_d that are not yet separated from t. For each vertex $s \in C_b$ (analogously for $s \in C_d$), we claim that

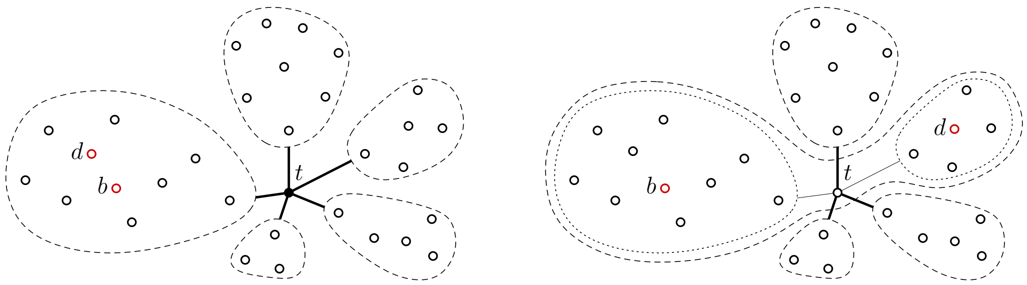
(i) we can either deduce a minimum s-t-cut from the M-set $\mathfrak{m}(s, t)$ in $G^{\oplus}_{\alpha}$ that equals the one induced by the old cluster C_b, or otherwise

(ii) the M-set $\mathfrak{m}(s, t)$ is either a proper subset of C_b in $G^{\oplus}_{\alpha}$, or

(iii) we can deduce a minimum s-t-cut $(U, V_\alpha \setminus U)$ from $\mathfrak{m}(s, t)$ in $G^{\oplus}_{\alpha}$ with $C_b \subsetneq U$, and there is no $s' \in C_b$ such that the cut induced by the old cluster C_b is a minimum s'-t-cut in $G^{\oplus}_{\alpha}$, or

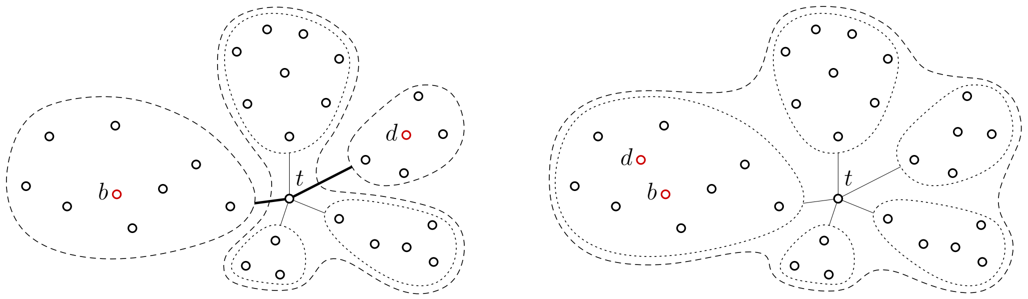
(iv) we can deduce no further cut from $\mathfrak{m}(s, t)$, but there is still no $s' \in C_b$ such that the cut induced by the old cluster C_b is a minimum s'-t-cut in $G^{\oplus}_{\alpha}$.

In each step, we use the resulting cut as split cut. Note that the old clusters in $\Omega(G) \setminus \{C_b, C_d\}$ are not split, since they already form nodes in the intermediate tree in Fig. 14.3(b).



(a) Intermediate tree for *intra*-cluster cost *increase*. All cuts are reusable.

(b) Intermediate tree for *inter*-cluster cost *increase*. All but two cuts are reusable.



(c) Intermediate tree for *inter*-cluster cost *decrease*. Only two cuts are reusable.

(d) Intermediate tree for *intra*-cluster cost *decrease*. No cuts are reusable.

FIGURE 14.3: Schematic illustration of intermediate trees obtained from cluster trees. Fat edges represent valid cuts, thin edges represent old cuts for which it is not yet known if they can be reused. Old nodes are indicated by dotted, new nodes by dashed lines. Black vertices correspond to singletons.

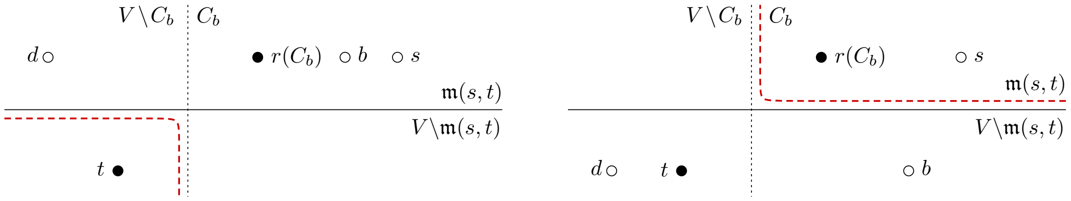
Proof of (i) to (iv). The proof of the above claims is based on Lemma 9.4, which admits to bend new cuts in $G_\alpha^\oplus$ along old minimum separating cuts. Form the lemma and the following proof, we can also see how to reshape the initially found U-cuts such that they satisfy the claimed conditions.

Let $\mathfrak{uc}(s,t)$ denote the newly found U-cut in $G_\alpha^\oplus$. We first distinguish possible costs of $\mathfrak{uc}(s,t)$. If $c_\alpha^\oplus(\mathfrak{uc}(s,t)) = c(C_b, V_\alpha \setminus C_b) + \Delta$, then C_b also induces a minimum s-t-cut in $G_\alpha^\oplus$, and we found a cut as claimed in (i).

Otherwise, $\mathfrak{uc}(s,t)$ is cheaper than the cut $(C_b, V_\alpha \setminus C_b)$ in $G_\alpha^\oplus$. Now we distinguish whether or not $\mathfrak{uc}(s,t)$ separates b and d. If $\mathfrak{uc}(s,t)$ separates b and d (as $(C_b, V_\alpha \setminus C_b)$ does), it is also cheaper in G_α. Hence, $\mathfrak{uc}(s,t)$ does not separate $r(C_b)$ and t, since this would contradict the fact that C_b induces a minimum $r(C_b)$-t-cut in G_α. Furthermore, $\mathfrak{m}(s,t)$ is also the M-set $\mathfrak{m}(s,t)$ in G_α. It thus follows that $\mathfrak{m}(s,t) \subsetneqq C_b$ in G_α (and $G_\alpha^\oplus$), as claimed in (ii), since otherwise the corresponding U-cut $\mathfrak{uc}(s,t)$ could be bent along $V_\alpha \setminus C_b$, deflected by t, according to the Non-Crossing Lemma (7.2), contradicting the fact that it induces an M-set. Note that, $\mathfrak{uc}(s,t)$ is also a minimum $r(C_b)$-s-cut in G_α in this situation.

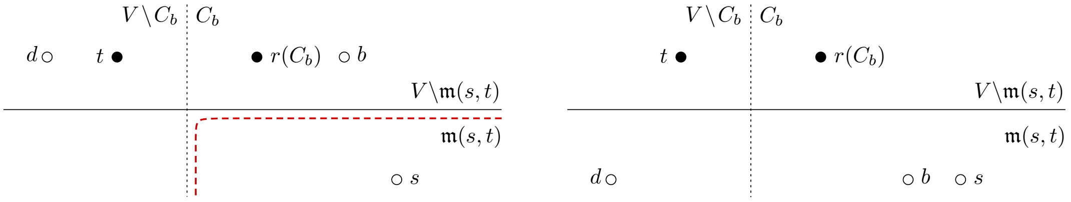
If $\mathfrak{uc}(s,t)$ does not separate b and d, we distinguish again four cases (see also Fig. 14.4). Recall that we still assume that $\mathfrak{uc}(s,t)$ is cheaper than the cut induced by C_b in $G_\alpha^\oplus$.

Case 1: $\mathfrak{uc}(s,t)$ separates t and $r(C_b)$ with b on the side of $r(C_b)$. In this case (Fig. 14.4(a)), Lemma 9.4(i) tells us how to reshape $\mathfrak{uc}(s,t)$ such that for the resulting cut $(U, V_\alpha \setminus U)$ holds $C_b \subseteq U$. Note that it even holds $C_b \subsetneqq U$, since $\mathfrak{uc}(s,t)$ is cheaper than the cut induced by C_b. If there further was a vertex $s' \in C_b$ such that $(C_b, V_\alpha \setminus C_b)$ was a minimum s'-t-cut in $G_\alpha^\oplus$, then the



(a) Case 1: It follows that $b, d, s, r(C_b)$ are separated from t. Deflected by $r(C_b)$, Lemma 9.4(i) bends $\mathfrak{uc}(s,t)$ downwards along C_b.

(b) Case 2: It follows that $r(C_b), s$ are separated from b, d, t. Deflected by t, Lemma 9.4(i) would bend $\mathfrak{uc}(s,t)$ upwards along C_b.



(c) Case 3: It follows that $b, d, r(C_b), t$ are separated from s. Deflected by t, Lemma 9.4(ii) would bend $\mathfrak{uc}(s,t)$ downwards along C_b.

(d) Case 4: It follows that $r(C_b), t$ are separated from b, d, s. Here, none of the cases of Lemma 9.4 apply.

FIGURE 14.4: Situations in the proof of Claim (i)-(iv) for the update procedure in case of inter-cluster cost increase. Lemma 9.4 is applied differently in three of four situations. Cluster C_b is depicted as dotted black line, original cut as solid black line, and reshaped cut as dashed red line.

deduced minimum s-t-cut $(U, V_\alpha \setminus U)$ is also a minimum s'-t-cut, and vice versa, since both cuts separate s and s' from t. This contradicts the assumption that $\mathfrak{uc}(s,t)$ is cheaper than the cut induced by C_b. Thus, the deduced cut satisfies (iii).

Case 2: $\mathfrak{uc}(s,t)$ separates t and $r(C_b)$ with d on the side of t. In this case (Fig. 14.4(b)), it is $\mathfrak{m}(s,t) \subsetneqq C_b$, as claimed in (ii). Otherwise, Lemma 9.4(i) would tell us how to reshape $\mathfrak{uc}(s,t)$ such that we get a smaller cut side for s, which contradicts the M-set $\mathfrak{m}(s,t)$.

Case 3: $\mathfrak{uc}(s,t)$ does not separate t and $r(C_b)$ with b on the side of $r(C_b)$. In this case (Fig. 14.4(c)), it is again $\mathfrak{m}(s,t) \subsetneqq C_b$, according to the same argument as in case 2. The only difference is, that here we apply Lemma 9.4(ii).

Case 4: $\mathfrak{uc}(s,t)$ does not separate t and $r(C_b)$ with b on the side of s. In this case (Fig. 14.4(d)), none of the previous arguments apply. If there was a vertex $s' \in C_b \cap \mathfrak{m}(s,t)$ such that $(C_b, V_\alpha \setminus C_b)$ was a minimum s'-t-cut in $G_\alpha^\oplus$, then the deduced minimum s-t-cut $(U, V_\alpha \setminus U)$ is also a minimum s'-t-cut, and vice versa, since both cuts separate s and s' from t. This contradicts the assumption that $\mathfrak{uc}(s,t)$ is cheaper than $(C_b, V_\alpha \setminus C_b)$.

If there was a vertex $s' \in C_b \setminus \mathfrak{m}(s,t)$ such that $(C_b, V_\alpha \setminus C_b)$ was a minimum s'-t-cut in $G_\alpha^\oplus$, then $\mathfrak{uc}(s,t)$ would also be a minimum s-s'-cut, since a cheaper s-s'-cut must separate t and s', which is a contradiction. Hence, according to the Non-Crossing Lemma (7.2), $\mathfrak{uc}(s,t)$ could be reshaped deflected by s' resulting in a cut with a smaller side for s, which contradicts the M-set $\mathfrak{m}(s,t)$. Hence, this case corresponds to (iv).

14.2.2 Procedures for Decreasing Costs

Inter-Cluster Cost Decrease. In case of an inter-cluster cost decrease (Fig. 14.3(c)), the thin edges $\{t, C_i\}$ represent old minimum separating cuts with respect to t and the representatives $r(C_i)$ of the clusters C_i in $\Omega(G) \setminus \{C_b, C_d\}$. We modify UNRESTRICTED CUTC such that it computes in each step a U-cut $\mathfrak{uc}(r(C_i), t)$ in $G_\alpha^\ominus$ with respect to one of these cut pairs that is not yet separated (line 5). We claim that for each such cut pair $\{r(C_i), t\}$ with $C_i \in \Omega(G) \setminus \{C_b, C_d\}$

(i) we can either deduce a minimum $r(C_i)$-t-cut from $\mathfrak{uc}(r(C_i), t)$ in $G_\alpha^\ominus$ that equals the one induced by the old cluster C_i, or

(ii) we can deduce at least a minimum $r(C_i)$-t-cut from $\mathfrak{uc}(r(C_i), t)$ in $G_\alpha^\ominus$ that does not split any old cluster.

In each step, we use the deduced cut as split cut. In this way UNRESTRICTED CUTC finishes after at most $|\Omega(G) \setminus \{C_b, C_d\}|$ cut computations. Note that C_b and C_d are not split since they already form nodes in the intermediate tree in Fig. 14.3(c).

Proof of (i) and (ii). The proof of the above claims is based on Lemma 8.9, which admits to bend new cuts in $G_\alpha^\ominus$ along old minimum separating cuts. Form the theorem and the following proof, we can also see how to reshape the initially found U-cuts such that they satisfy the conditions claimed above.

Let $\mathfrak{uc}(r(C_i), t)$ denote the newly found U-cut in $G_\alpha^\ominus$. If $c_\alpha^\ominus(\mathfrak{uc}(r(C_i), t)) = c(C_i, V_\alpha \setminus C_i)$, then C_i also induces a minimum $r(C_i)$-t-cut in $G_\alpha^\ominus$, as claimed in (i). Otherwise, $\mathfrak{uc}(r(C_i), t)$ is cheaper than $(C_i, V_\alpha \setminus C_i)$, and thus, separates b and d. Hence, Lemma 8.9 tells us how to reshape $\mathfrak{uc}(r(C_i), t)$ such that it does not split any old cluster in $\Omega(G) \setminus \{C_b, C_d\}$ (including C_i). The resulting cut behaves as claimed in (ii).

Intra-Cluster Cost Decrease. In the case of intra-cluster cost decrease (Fig. 14.3(d)), the thin edges $\{t, C_i\}$ represent old minimum separating cuts with respect to t and the representatives $r(C_i)$ of the clusters $C_i \in \Omega(G)$. We modify UNRESTRICTED CUTC such that it starts with separating the vertices in $C_{b/d}$ from t by computing U-cuts in $G_\alpha^\ominus$ (line 5)(line 5). For each vertex $s \in C_{b/d}$, we claim that

(iii) the M-set $\mathfrak{m}(s, t) \subseteq C_{b/d}$ in $G_\alpha^\ominus$.

If for the current U-cut holds that $c_\alpha^\ominus(\mathfrak{uc}(s, t)) = c(C_{b/d}, V_\alpha \setminus C_{b/d})$, then $C_{b/d}$ also induces a minimum s-t-cut in $G_\alpha^\ominus$ and the remaining thin edges are also reusable, according to Lemma 8.7. Then UNRESTRICTED CUTC returns $\Omega(G)$. Otherwise, we use the U-cut $\mathfrak{uc}(s, t)$ as split cut and continue. When all vertices in $C_{b/d}$ are separated from t, we proceed with computing in each step a U-cut $\mathfrak{uc}(r(C_i), t)$ in $G_\alpha^\ominus$ with respect to one of the remaining cut pairs of the thin edges that is not yet separated (line 5). We claim that for each such cut pair $\{r(C_i), t\}$ with $C_i \in \Omega(G) \setminus \{C_{b/d}\}$

(iv) we can either deduce a minimum $r(C_i)$-t-cut from $\mathfrak{uc}(r(C_i), t)$ in $G_\alpha^\ominus$ that equals the one induced by the old cluster C_i, or

(v) we can deduce at least a minimum $r(C_i)$-t-cut from $\mathfrak{uc}(r(C_i), t)$ in $G_\alpha^\ominus$ that does not split any old cluster.

In each step, we use the deduced cut as split cut. When all cut pairs $\{r(C_i), t\}$ with $C_i \in \Omega(G) \setminus \{C_{b/d}\}$ are separated, we are done.

Proof of (iii), (iv) and (v). The proof of the above claims is again based on Lemma 8.9, which admits to bend new cuts in $G_\alpha^\ominus$ along old minimum separating cuts. Form the theorem and the following proof, we can also see how to reshape the initially found U-cuts such that they satisfy the conditions claimed above.

We start with the proof of (iii). Let $\mathfrak{uc}(s, t)$ denote the newly found U-cut in $G_\alpha^\ominus$. If $c_\alpha^\ominus(\mathfrak{uc}(s, t)) = c(C_{b/d}, V_\alpha \setminus C_{b/d})$, then $C_{b/d}$ also induces a minimum s-t-cut in $G_\alpha^\ominus$, and thus, it is $\mathfrak{m}(s, t) \subseteq C_{b/d}$

in $G_\alpha^\ominus$, as claimed in (iii). Otherwise, $\mathfrak{uc}(s,t)$ is cheaper than the old cut $(C_{b/d}, V_\alpha \setminus C_{b/d})$. If $\mathfrak{uc}(s,t)$ does not separate b and d, it has the same cost also in G_α. Hence, $\mathfrak{uc}(s,t)$ does not separate $r(C_{b/d})$ and t, since this would contradict the fact that $C_{b/d}$ induces a minimum $r(C_{b/d})$-t-cut in G_α. Furthermore, $\mathfrak{m}(s,t)$ is also the M-set $\mathfrak{m}(s,t)$ in G_α. It thus follows that $\mathfrak{m}(s,t) \subseteq \mathfrak{m}(r(C_{b/d}),t)$ in G_α, according to Lemma 7.6(2i), and hence, $\mathfrak{m}(s,t) \subseteq C_{b/d}$ in G_α (and $G_\alpha^\oplus$), as claimed in (iii)

If $\mathfrak{uc}(s,t)$ (in $G_\alpha^\ominus$) separates b and d, we can apply Lemma 8.9. Since $\{t\} \cup (\Omega(G) \setminus \{C_{b/d}\})$ is sheltered by the old cut $(C_{b/d}, V_\alpha \setminus C_{b/d})$, Lemma 8.9 tells us that any minimum s-t-cut in $G_\alpha^\ominus$ can be reshaped such that the cut side containing s is nested in $C_{b/d}$. That is in particular $\mathfrak{m}(s,t) \subseteq C_{b/d}$.

The proof of (iv) and (v) also exploits Lemma 8.9. Let $\mathfrak{uc}(r(C_i),t)$ denote the newly found U-cut in $G_\alpha^\ominus$. If $c_\alpha^\ominus(\mathfrak{uc}(r(C_i),t)) = c(C_i, V_\alpha \setminus C_i)$, then C_i also induces a minimum $r(C_i)$-t-cut in $G_\alpha^\ominus$, as claimed in (iv). Otherwise, $\mathfrak{uc}(r(C_i),t)$ is cheaper than $(C_i, V_\alpha \setminus C_i)$, and thus, must separate b and d. Hence, Lemma 8.9 tells us how to reshape $\mathfrak{uc}(r(C_i),t)$ such that it does not split any old cluster in $\Omega(G) \setminus \{C_{b/d}\}$ (including C_i). Hence, the resulting cut behaves as claimed in (v).

14.3 Optimal Temporal Smoothness for Single Clusterings

In general, comparing clusterings is a difficult task. In the literature, many different distance measures are used to evaluate the similarity of clusterings, and each such measure induces an individual way to quantify temporal smoothness, that is, to evaluate how similar consecutive clusterings in a dynamic scenario are. For information on distance measures of clusterings see for example the articles of Fortunato [46] and Wagner et al. [137]. As a first step in the direction of temporal smoothness, we observe that our update procedures guarantee that each cluster that neither contains b nor d is nested in a cluster of the next time step. However, the structure of cut clusterings, which originates from the way they are constructed, admits an even stronger statement. More precisely, if we measure the similarity of two clusterings on the same vertex set by the number of clusters that occur in both clusterings, our procedures obtain optimal temporal smoothness. That is, in each time step, there exists no unrestricted cut clustering that contains more clusters of the previous clustering than the one found by our update approach. This in particular implies the *stability* of our approach, saying that a clustering that can be preserved indeed is preserved, that is, each procedure returns the previous clustering whenever the previous clustering is also an unrestricted cut clustering for the modified graph in the next time step.

Theorem 14.1. *The update procedures for unrestricted cut clusterings guarantee optimal temporal smoothness.*

Proof. Let $\Omega(G)$ and $\Omega(G^{\copyright})$ denote two consecutive unrestricted cut clusterings resulting from our update approach. We prove Theorem 14.1 in two steps. The first step considers clusters that neither contain b nor d and shows that if such a cluster is not in $\Omega(G^{\copyright})$ then it is also in no other unrestricted cut clustering for $G^{\copyright}$. The second step shows the same statement for the clusters that contain a vertex in $\{b,d\}$. These two statements already prove optimal temporal smoothness as follows. Assume an unrestricted cut clustering $\Omega'(G^{\copyright})$ that contains more old clusters than $\Omega(G^{\copyright})$. Then, $\Omega'(G^{\copyright})$ contains at least one old cluster that is not in $\Omega(G^{\copyright})$, which is already a contradiction.

First step: Let $C \in \Omega(G)$ denote a cluster that neither contains b nor d. We have already observed that C is always nested in a cluster in $\Omega(G^{\textcircled{U}})$. That is, if $C \notin \Omega(G^{\textcircled{U}})$, it is a proper subset of a cluster in $\Omega(G^{\textcircled{U}})$. Hence, let C' denote the cluster in $\Omega(G^{\textcircled{U}})$ that contains C. In our update procedure for increasing costs, C' results from a computation of a U-cut with respect to a vertex $s' \notin C$, while in the procedures for decreasing costs, C' results from a computation of a U-cut with respect to either $r(C)$ or a vertex $s' \notin C$. Note that in all cases C' is not necessarily an M-set due to possible reshaping. But due to the rules for reshaping, we know for all procedures that $r(C) \in \mathfrak{m}(s', t)$ if C' results from a U-cut with respect to a vertex $s' \notin C$.

Now assume an unrestricted cut clustering $\Omega'(G^{\textcircled{U}})$ that contains C. In case that C' results from a U-cut with respect to a vertex $s' \notin C$, let $\widehat{C}$ denote the cluster in $\Omega'(G^{\textcircled{U}})$ that contains s'. Then, according to Lemma 7.6(2i), it holds $\mathfrak{m}(s', t) \subseteq \mathfrak{m}(r(\widehat{C}), t) \subseteq \widehat{C}$. But on the other hand, it is $C \cap \widehat{C} = \emptyset$. This contradicts the fact that $r(C) \in \mathfrak{m}(s', t)$. Hence, there exists no unrestricted cut clustering for $G^{\textcircled{U}}$ that contains C. If C' results from a U-cut with respect to $r(C)$, we are in the situation of a cost decrease. Since C is not in $\Omega(G^{\ominus})$, we know from the structure of the procedures, that the U-cut $\mathfrak{uc}(r(C), t)$ is cheaper than the cut induced by C in $G^{\ominus}_{\alpha}$. On the other hand, since C is in $\Omega'(G^{\ominus})$, there is vertex s' in C such that C induces a minimum s'-t-cut in $G^{\ominus}_{\alpha}$. However, the cheaper split cut deduced from $\mathfrak{uc}(r(C), t)$ that induces C' in $\Omega(G^{\ominus})$ also separates s' and t, and thus yields a contradiction proving that there is no unrestricted cut clustering for $G^{\ominus}$ that contains C.

Second step: We first consider the cluster $C_{b/d}$ that contains both, b and d. In case of intra-cluster cost insertion, $C_{b/d}$ is always contained in $\Omega(G^{\oplus})$, since in this case it is $\Omega(G) = \Omega(G^{\oplus})$. Hence, in the following we consider $C_{b/d}$ in case of intra-cluster cost decrease. We observe that if $C_{b/d}$ is not in $\Omega(G^{\ominus})$, none of the computed U-cuts $\mathfrak{uc}(s, t)$ with $s \in C_{b/d}$ has the same cost in $G^{\ominus}_{\alpha}$ than the cut induced by $C_{b/d}$.

Now assume an unrestricted cut clustering $\Omega'(G^{\ominus})$ that contains $C_{b/d}$ and note that the cluster $C_{b/d}$ in $\Omega'(G^{\ominus})$ induces a minimum s'-t-cut in $G^{\ominus}_{\alpha}$ with $s' \in C_{b/d}$. Since s' was not considered by the update procedure, it must have been separated from t by a M-set $\mathfrak{m}(s, t)$ with $s \neq s'$. This, however means that $\lambda_{G^{\ominus}_{\alpha}}(s', t) = \lambda_{G^{\ominus}_{\alpha}}(s, t) = c^{\ominus}_{\alpha}(C_{b/d}, V_{\alpha} \setminus C_{b/d})$, and the procedure would have found $\mathfrak{m}(s, t)$ to have the same cost as the cut induced by $C_{b/d}$, which is a contradiction. Hence, there is no unrestricted cut clustering that contains $C_{b/d}$ if $C_{b/d}$ is no cluster in $\Omega(G^{\ominus})$.

Now we consider the cluster C_b in case of inter-cluster cost increase. The case for C_d is symmetric. If C_b is not in $\Omega(G^{\oplus})$ we distinguish three cases.

Case 1: C_b is a proper subset of another cluster. Due to the structure of the procedure this only happens if we find a U-cut for C_b as claimed in (iii) or if a cut deduced from a U-cut found for C_d separates C_b from t. In the first case, it follows directly from (iii) that there is no other clustering for $G^{\oplus}$ that contains C_b. In the second case, we observe that the found U-cut for C_d has already separated C_b from t or has at least split C_b. Since the corresponding M-set must be nested in a cluster in any unrestricted cut clustering for $G^{\oplus}$, C_b cannot occur in such a clustering.

Case 2: C_b is the union of at least two clusters. Due to the structure of the procedure this only happens if we find only U-cuts for C_b as claimed in (ii). If C_b was a cluster in another unrestricted cut clustering for $G^{\oplus}$, it follows with the same arguments as in the first step that we would have found the minimum separating cut induced by C_b, and thus, C_b would have been inn$\Omega(G^{\oplus})$, which is a contradiction.

Case 3: At least two vertices of C_b are in different clusters and at least one of these clusters *contains some more vertices not resulting from C_b.* This only occurs if C_b is split by either a U-cut according to (iv) or a cut deduced for C_d. In the first case it follows directly from (iv), that C_b does not occur in another unrestricted cut clustering for $G^\oplus$. In the second case, we can use the same arguments as in Case 1 in order to prove that C_b does not occur in another clustering. Hence, there is no unrestricted cut clustering for $G^\oplus$ that contains C_b if C_b is no cluster in $\Omega(G^\oplus)$.

We still need to consider the cluster C_b in case of inter-cluster cost decrease. Again the case for C_d is symmetric. If C_b is not in $\Omega(G^\ominus)$, then it is a proper subset of a cluster C' in $\Omega(G^\ominus)$. Recall, that C_b (and also C_d) are already nodes in the intermediate tree in Fig. 14.3(c) and are thus never split. Due to the structure of the procedure, C' results from a U-cut with respect to a vertex $s' \notin C_b$. Hence, from this point on, the same arguments as presented for the first step apply, proving that there is no unrestricted cut clustering that contains C_b if C_b is no cluster in $\Omega(G^\ominus)$. □

14.4 Running Times for Single Clusterings

As in the analysis of dynamic Gomory-Hu trees, we express running times of our procedures in terms of the number of necessary minimum-cut computations, leaving open how these are done. A summary of tight bounds is given in Table 14.1. The columns *lower bound/upper bound* denote bounds for the—possibly rather common—case that the old clustering is still valid after some graph update.

Table 14.1: Bounds on the number of minimum-cut computations.

	best case	worst case	old clustering still valid	
			lower bound	upper bound
InterDecrease	$\lvert\Omega(G^\ominus)\rvert - 2$	$\lvert\Omega(G)\rvert - 2$	$\lvert\Omega(G)\rvert - 2$	$\lvert\Omega(G)\rvert - 2$
IntraDecrease	1	$\lvert\Omega(G)\rvert + \lvert C_{b/d}\rvert - 1$	1	$\lvert\Omega(G)\rvert + \lvert C_{b/d}\rvert - 1$
InterIncrease	2	$\lvert C_b\rvert + \lvert C_d\rvert$	2	$\lvert C_b\rvert + \lvert C_d\rvert$
IntraIncrease	0	0	0	0

After an inter-cluster cost decrease (first line in Table 14.1), in the best case we only calculate as many cuts as new clusters arise while separating t from all vertices in V, except $r(C_b)$ and $r(C_d)$ (compare to Figure 14.3(c)). In the worst case, the procedure considers each representative $r(C_i)$ with $C_i \in \Omega(G) \setminus \{C_b, C_d\}$, and thus, requires $|\Omega(G)| - 2$ cuts. If the previous clustering is not preserved, we then calculate more cuts than finally needed to define the new clustering. Since $|\Omega(G^\ominus)| = |\Omega(G)|$ in case the old clustering remains valid, the remaining bounds are also correct, and since the procedure guarantees stability, we will find the old clustering. After an intra-cluster cost decrease, in the worst case, our approach needs to examine all clusters in $\Omega(G) \setminus \{C_{b/d}\}$, and all vertices in $C_{b/d}$, even if the previous clustering is retained. We obviously attain the best case, if the first considered U-cut already yields that $C_{b/d}$, and thus, the whole clustering remains valid. After an inter-cluster cost increase, we potentially end up separating every single vertex in $C_b \cup C_d$ from t, one by one, even if the previous clustering is valid, which yields the worst case bound and the upper bound in the last column. In case the previous clustering is still valid, however, we might get away with simply cutting off $r(C_b)$ and $r(C_d)$, alongside their

Table 14.2: Total number of atomic changes decomposed into different scenarios.

	total	InterDecrease	IntraDecrease	InterIcrease	IntraIncrease
atomic changes	61870	3742	26179	10010	21939
%	100	6.0482	42.3129	16.1791	35.4598
advantage static	40	0	40	0	0
of total changes %	0.0647	0.0000	0.0647	0.0000	0.0000
advantage dynamic	61830	3742	26139	10010	21939
of total changes %	99.9353	6.0482	42.2483	16.1791	35.4598

Table 14.3: Total number of clusters, cuts and savings for different scenarios.

	total	InterDecr	IntraDecr	InterIncr	IntraIncr
static clusters	3186155	314979	1090890	748442	1031844
%	100	9.8859	34.2384	23.4904	32.3852
dynamic clusters	3185398	314923	1090414	748287	1031774
%	100	9.8865	34.2316	23.4912	32.3907
static cuts	3300413	324098	1131538	773730	1071047
%	100	9.8199	34.2847	23.4434	32.4519
dynamic cuts	736826	308904	403499	24423	0
of total static cuts %	22.3253	9.3596	12.2257	0.7400	0.0000
amortized static costs	1.0359	1.0290	1.0373	1.0338	1.0380
amortized dynamic costs	0.2313	0.9809	0.3700	0.0326	0.0000
cut savings	2563587	15194	728039	749307	1071047
of total static cuts %	77.6747	0.4604	22.0590	22.7034	32.4519
average cut savings	41.4351	4.0604	27.8100	74.8558	48.8193

former clusters, which yields the best case bound. and the lower bound in the third column. The row for the case of intra-cluster cost increase is obvious.

Note that a computation from scratch (with the static algorithm Unrestricted CutC) entails a tight upper bound of $|V| - 1$ minimum-cut computations for all four cases, in the worst case; although, in practice, the heuristic recommended by Flake et al. usually finds a new clustering in time proportional to the total number of new clusters. In the best case it needs as many cut computations as new clusters arise. Comparing this to the bound for updating an inter-cluster cost decrease in the best case, lets us expect only little effort saving for this case; while the case of intra-cluster cost increase promises the biggest effect of effort saving.

Experiments. In this brief section, we very roughly describe some experiments we made with an implementation of the update procedures described above, just for a first proof of concept. The instance we use is a network of email communications within the Department of Informatics at KIT, obtained from the email data described in Section 1.4. Vertices represent members and edges correspond to email contacts, weighted by the number of emails sent between two individuals during the last 72 hours. This means, each email has a fixed time to live. After that time the contribution of the email to the weight of the edge expires and the weight of the edge decreases. We process a queue of 69 739 atomic changes, 61 870 of which are actual changes of edge costs, on an initial graph with $|V| = 247$ and $|E| = 307$. This queue represents

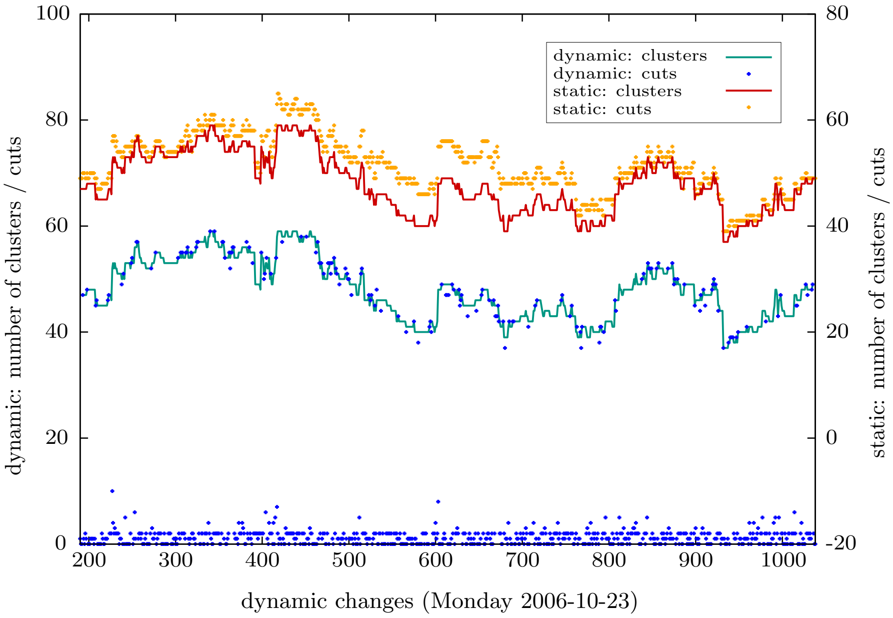


FIGURE 14.5: Numbers of clusters and cuts regarding consecutive clusterings (the two y-axes have a different offset for better readability).

about three months, starting on Sunday (2006-10-22). The number of vertices varies between 172 and 557, the number of edges varies between 165 and 1190. We delete zero-weight edges and isolated nodes. Following the recommendations of Flake et al. [42], we choose $\alpha = 0.15$ for the initial graph, yielding 73 clusters. We compare the static algorithm UNRESTRICTED CUTC (see Chapter 13) and our dynamic approach in terms of the number of minimum-cut computations necessary to maintain a clustering. Forty times out of the 61 870 total operations, the static computation needed less minimum cuts than the dynamic update. In all remaining cases (99.93%) the update procedures were at an advantage (see Table 14.2).

The first two rows of Table 14.3 show the numbers of clusters found by the static and dynamic approach over the whole experiment. As both algorithms range at similar levels we can be sure the observed savings are not induced by trivial clusterings. Thus, comparing dynamic and static cut computations is justified: For the 61 870 steps induced by edge changes, static computation needed 3 300 413 minimum cuts, and our dynamic update needed 736 826, saving more than 77% minimum cuts, such that one dynamic cluster on average costs 0.23 cut computations. The amortized costs of 1.03 cuts for a static cluster affirm the running time to be proportional to the total number of new clusters, as stated by Flake et al. This running time is also visible in Figure 14.5, which shows the consecutive development of the graph structure over one day (Monday, 2006-10-23). Obviously, the static and dynamic clusterings (upper red and lower green line) behave similarly. Note that the scale for static clusterings and cuts is offset by about 20 clusters/cuts for readability. However, the dynamic flows (blue dots) cavort around the clusters or, even better, near the ground, which means there are only few flow computations needed. In contrast, most of the static flow amounts (orange dots) are still proportional but clearly higher than the number of clusters in the associated static clustering.

Regarding the total number of atomic edge changes the savings finally average out at 41.4 cuts (Table 14.3), while inter-cluster cost decreases save the most effort per change. This is, the case of inter-cluster cost decrease surprisingly outperforms the trivial intra-cluster cost increases.

14.5 Update Procedures for Hierarchies

The second part of this chapter addresses a dynamic version of the hierarchical unrestricted cut-clustering approach. We present a method that employs the new degree of freedom (which we gain compared to the restricted cut-clustering algorithm) for consecutively updating cut-clustering hierarchies with respect to a given sequence of parameter values. According to the results of Chapter 13 (Theorem 13.1), where we have seen that arbitrary unrestricted cut clusterings with respect to different parameter values are already hierarchically nested, this can be naively done by simply updating each level independently using the update procedures for single unrestricted cut clusterings presented in the previous sections. Hence, we also call the update procedures for single unrestricted cut clusterings level update procedures. In this way, we would already achieve optimal temporal smoothness, since the level update procedures find all reusable clusters (Theorem 14.1), and also some remarkable cost savings due to the nice performance of the level update procedures. However, in the following we present an algorithm that additionally exploits the hierarchical structure of the clusterings in order to find reusable parts of the hierarchy even more effective. The remaining parts are finally updated by the update approach for single clusterings. We say that an unrestricted cut clustering satisfies the *copy-property* with respect to an edge change if it remains valid after the change, and thus, can be copied to the new hierarchy.

Note that in the context of clustering hierarchies, atomic edge changes induce four different situations with respect to a single level (inter- and intra-cluster cost increase and decrease) but in the end, we get only two different update procedures for the hierarchy, one for increasing costs and one for decreasing costs. We observe further that isolated vertices form singletons in each unrestricted cut clustering, and thus, on each level in the hierarchy. Hence, inserting and deleting a vertex can be easily done by simply adding or deleting a singleton on each level.

14.5.1 Reusable Parts of the Hierarchy in a Dynamic Scenario

Given an atomic edge change, a cut-clustering hierarchy decomposes into two parts. Levels where the change induces an inter-cluster event form the lower part, levels where the change induces an intra-cluster event form the upper part. Recall Fig. 11.2 and the indexing of hierarchy levels and clusterings.

The first theorem admits to efficiently detect reusable levels in the upper part of the hierarchy, based on the following lemma.

Lemma 14.2. *In case of a cost decrease, let α_k denote the largest value such that an unrestricted cut clustering $\Omega_k(G^\ominus)$ exists with b and d in a common cluster (if no such $\alpha_k \geq \alpha_{\max}$ exists, we define $\alpha_k := 0$). Then, each $\Omega_j(G)$ with $j \leq k$ satisfies the copy-property.*

Proof. In case of an intra-cluster cost decrease, let $\Omega(G)$ denote an unrestricted cut clustering before, and $\Omega(G^\ominus)$ an unrestricted cut clustering after the change. Based on the level update procedure for this scenario, we show that $\Omega(G) = \Omega(G^\ominus)$ if and only if b and d are in a common cluster in $\Omega(G^\ominus)$. Then, it follows directly that in the situation of Lemma 14.2 the clustering $\Omega_k(G)$ as well as each clustering $\Omega_j(G)$ with $j < k$ satisfies the copy-property.

If the consecutive clusterings $\Omega(G)$ and $\Omega(G^\ominus)$ are identical, b and d are clearly in a common cluster in $\Omega(G^\ominus)$, since we consider an intra-cluster event in $\Omega(G)$.

If b and d are in a common cluster in $\Omega(G^\ominus)$, we can see by the structure of the update procedure for intra-cluster cost decrease, that b and d must be in an M-set $\mathfrak{m}(s,t)$ in $G^\ominus_\alpha$ with $s \in C_{b/d}$.

This holds since the remaining cuts considered by the procedure with respect to the representatives of the clusters in $\Omega(G) \setminus \{C_{b/d}\}$ either equal the previous cluster of the representative or, if they are cheaper than the cut induced by the old cluster, separate b and d. For the M-set $\mathfrak{m}(s,t)$ we know that it is a subset of $C_{b/d}$. Now consider the representative $r(C_{b/d})$ of the cluster $C_{b/d}$ in $\Omega(G)$. If $r(C_{b/d})$ is in $\mathfrak{m}(s,t)$, the corresponding U-cut $\mathfrak{uc}(s,t)$ (which does not separate b and d, but $r(C_{b/d})$ and t) has the same cost as the cut induced by $C_{b/d}$ in G_α and $G_\alpha^\ominus$ (since $\mathfrak{m}(s,t) \subseteq C_{b/d}$). Thus, the update procedure returns the old clustering $\Omega(G)$. Otherwise, if $r(C_{b/d})$ is not in $\mathfrak{m}(s,t)$, there exists a minimum $r(C_{b/d})$-t-cut in $G_\alpha^\ominus$ that does not separate b and d (since b and d are sheltered by $\mathfrak{m}(s,t)$). Then, however, it is $\lambda_{G_\alpha}(r(C_{b/d}),t) = \lambda_{G_\alpha^\ominus}(r(C_{b/d}),t)$ and the update procedure returns $\Omega(G)$. □

While Lemma 14.2 tells us how to find reusable levels in the case of decreasing costs, we obtain a similar statement for the case of increasing costs directly from the level update procedure for intra-cluster cost increase. Since for an intra-cluster cost increase each clustering satisfies the copy property, in a hierarchy clearly the whole upper part can be reused in case of increasing costs.

Theorem 14.3. *In a given hierarchy, let k denote the lowest intra-event level that fulfills the copy-property. Then all levels $j \leq k$ also satisfy the copy-property, and thus, can be reused as part of a new hierarchy.*

The second theorem admits to efficiently detect reusable subtrees in the lower part of the hierarchy, based on Lemma 14.4. A *subtree* of a cluster C on level i consists of C and all clusters on lower levels (that is, with higher indices) that are nested in C.

Lemma 14.4. *Let C denote a cluster in $\Omega_j(G)$ that neither contains b nor d, and let further $C' \subseteq C$ denote a cluster in $\Omega_i(G)$, $i > j$. If C induces a minimum separating cut with respect to t and a further vertex in V after the change (in $G_{\alpha_j}^{\textcircled{U}}$), then C' also induces a minimum separating cut with respect to t and another vertex in V after the change (in $G_{\alpha_i}^{\textcircled{U}}$).*

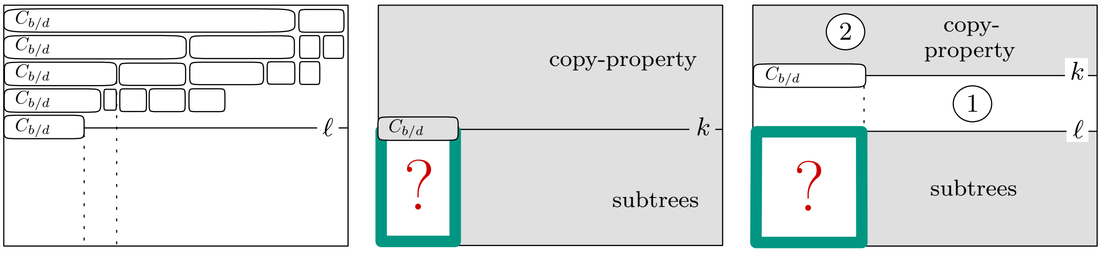
Proof. The cluster $C' \subseteq C$ is a minimum $r(C')$-t-cut before the change (in G_{α_i}) and obviously it is $r(C') \in C$. Since C is a minimum s-t-cut ($s \in C$) after the change (in $G_{\alpha_j}^{\textcircled{U}}$), by Lemma 7.6(2i), the M-set $\mathfrak{m}(r(C'),t) =: S_j$ in $G_{\alpha_j}^{\textcircled{U}}$ is a subset of $\mathfrak{m}(r(C),t)$ in $G_{\alpha_j}^{\textcircled{U}}$, and thus, of C. Furthermore, by Lemma 13.2(a), the M-set $\mathfrak{m}(r(C'),t) =: S_i$ in $G_{\alpha_i}^{\textcircled{U}}$ is a subset of S_j. Thus, S_i does not separate b and d. Since the cut induced by C' before the change (in G_{α_i}) also does not separate b and d, it is $\lambda_{G_{\alpha_i}^{\textcircled{U}}}(r(C'),t) = \lambda_{G_{\alpha_i}}(r(C'),t)$ and C' also induces a minimum $r(C')$-t-cut after the change (in $G_{\alpha_i}^{\textcircled{U}}$). □

If C in Lemma 14.4 induces not only a minimum separating cut (and thus, might be nested in a final clustering for the new graph), but occurs as a cluster in a new unrestricted cut clustering $\Omega_j(G^{\textcircled{U}})$, according to Corollary 13.3 the following holds:

Theorem 14.5. *In a hierarchy of unrestricted cut clusterings, let C denote a cluster in $\Omega_j(G)$ that neither contains b nor d and is also a cluster in a cut clustering $\Omega_j(G^{\textcircled{U}})$. Then the whole subtree of C can be used as part of a new hierarchy.*

14.5.2 Update Procedures for Increasing and Decreasing Costs

Our update approach consists of two phases. The first phase copies reusable parts of the old hierarchy by employing Theorem 14.3 and Theorem 14.5, the second phase updates the remaining



(a) Old hierarchy. (b) Cost increase. (c) Cost decrease.

FIGURE 14.6: Sketch of the first phase of the update approach for hierarchies of unrestricted cut clusterings. Shaded areas represent parts that are simply copied.

part by the help of level update procedures. We estimate the running times in terms of costs of level update procedures. To this end we denote the time for updating consecutive levels i to j (bottom-up, with $i > j$) for a graph $G^{\oplus}$ by $\mathcal{T}([i, j], G^{\oplus})$. For one level i, the time $\mathcal{T}([i, i], G^{\oplus})$ corresponds to the running time of the applied level update procedure, that is, it depends on the type of edge change (cost increase or decrease) and on the type of the event the change induces on the level (intra-cluster or inter-cluster event). The procedure for intra-cluster cost increase runs in constant time.

We start with the application of Theorem 14.3. In case of increasing cost, the theorem tells us that we can simply copy each level of the upper part (that is, each intra-cluster event level) of the old hierarchy to the new hierarchy without further cost (see upper shaded area in Fig. 14.6(b)). In case of decreasing cost, we search for the lowest intra-cluster event level k in the old hierarchy that satisfies the copy-property. To this end we start from the lowest over-all intra-cluster event level ℓ iteratively updating consecutive levels bottom-up by the help of the level update procedure for intra-cluster cost decrease. Recall that for level k, which satisfies the copy property, this procedure stops as soon as it has found a vertex s in $C_{b/d}$ such that $C_{b/d}$ induces a minimum s-t-cut in $G^{\ominus}_{\alpha_k}$. The time for finding level k is $\mathcal{T}([\ell, k], G^{\ominus})$. After we have found level k, we simply copy each level j above level k ($j < k$) to the new hierarchy (see shaded area (2) in Fig. 14.6(c)), which already contains level ℓ to k (see white area (1) in Fig. 14.6(c)).

In both cases, cost increase and cost decrease, we can further apply Theorem 14.5 in order to maintain subtrees in the lower part of the hierarchy without further cost. In fact, we can reuse the subtree of each cluster on level ℓ that is not nested in the cluster $C_{b/d}$ on level k. This clearly holds for the case of increasing cost, since in this case it is $\ell = k$ (see lower shaded area in Fig. 14.6(b)). For the case of decreasing cost, we observe that, according to the theorem, we can reuse the subtrees of the clusters different from $C_{b/d}$ on level k, which contain all clusters on ℓ that are not nested in $C_{b/d}$ on level k. Note that the upper parts of these subtrees between level ℓ and k are already part of the new hierarchy, due to the stability of the level update procedure, which we used to update these levels (see lower shaded area in Fig. 14.6(c)). By updating the intra-cluster event levels and the corresponding subtrees in this way, we reduce the problem of updating a hierarchy of r levels for the underlying graph G to the problem of updating a hierarchy of $\ell - 1$ levels for an underlying graph G' just as big as the cluster $C_{b/d}$ on level k (see boxed question marks in Fig. 14.6). We remark, that G' is no vertex induced subgraph of G, since the edges incident to vertices outside of $C_{b/d}$ still count for the computation of minimum separating cuts. But due to Corollary 13.3, it is however feasible to contract the vertices outside of $C_{b/d}$ in G. This remaining part is now handled by the second phase of our update approach.

The second phase updates the remaining levels $\ell - 1$ to 0 top-down using the level update procedure for inter-cluster cost increase or inter-cluster cost decrease. Recall that the procedure for inter-cluster cost increase considers only U-cuts with respect to vertices in $C_b \cup C_d$. Due to Corollary 13.3 and the stability of the level update procedure, neither the U-cuts nor the cuts resulting from reshaping touch the clusters we have already found by the reuse of subtrees. Hence, in terms of minimum-cut computations the running time of this procedure is the same independent from whether or not we already know some of the clusters. However, we can possibly speed up the single minimum-cut computations by contracting the already known clusters on the current level in G. In terms of running time of the level update procedure, the time for finishing the hierarchy after a cost increase is thus $\mathcal{T}([0, \ell-1], G^{\oplus})$. In contrast, the procedure for inter-cluster cost decrease considers U-cuts with respect to the representatives of the clusters different from C_b and C_d. So when applying this procedure to a level on which we already now some clusters form the reuse of subtrees, we simply skip the consideration of the representatives of these clusters in the procedure. This is feasible and does not change the result of the procedure due to Corollary 13.3 and the stability of the procedure. In particular, the found U-cuts and the cuts resulting from reshaping leave the already known clusters untouched. Additionally contracting the known clusters in G may again speed up the single cut computations. In both cases, increasing and decreasing cost, we can additionally reuse further subtrees whenever a cluster that neither contains b nor d persists on the current level. In terms of running time of the level update procedure, the time for finishing the hierarchy after a cost decrease is thus $\mathcal{T}([0, \ell-1], G^{\ominus}) - \sum_{i=0}^{\ell-1} |S_i^*|$, with S_i^* the set of clusters on level i in the new hierarchy that are already known at the time level i is processed by the level update procedure.

14.5.3 Performance

We first observe that our update approach inherits optimal temporal smoothness from the level update procedures.

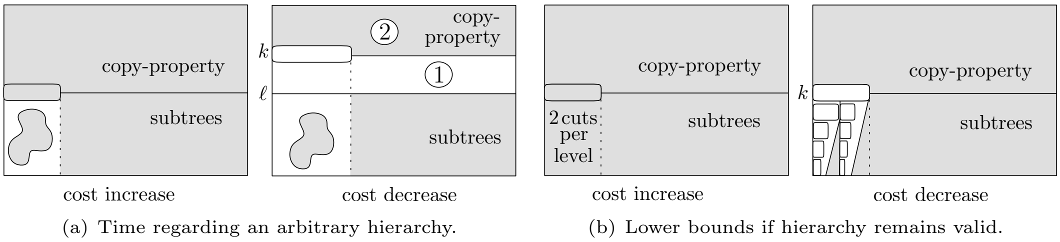
Theorem 14.6. *The update approach for hierarchies of unrestricted cut clusterings guarantees optimal temporal smoothness.*

Summing up the total running times of our update approach (in terms of time of the level update procedures), we get $\mathcal{T}([0, \ell-1], G^{\oplus})$ for increasing cost, and $\mathcal{T}([\ell, k], G^{\ominus}) + \mathcal{T}([0, \ell-1], G^{\ominus}) - \sum_{i=0}^{\ell-1} |S_i^*|$ for decreasing cost. Note that we assume a fixed height r of the old hierarchy and a fixed level ℓ where the lowest intra-cluster event occurs.

Table 14.4: Bounds on the number of minimum-cut computations.

	best case	worst case	old hierarchy still valid	
			lower bound	upper bound
Inc	$2(\ell - 1)$	$\sum_{i=0}^{\ell-1} \lvert C_b^i\rvert + \lvert C_d^i\rvert$	$2(\ell - 1)$	$\sum_{i=0}^{\ell-1} \lvert C_b^i\rvert + \lvert C_d^i\rvert$
Dec	$\sum_{i=\ell}^{k}(\lvert\Omega_i(G^{\ominus})\rvert - 2)+$ $\sum_{i=0}^{\ell-1} \lvert(\Omega_i(G^{\ominus})\rvert - 2)$ $- \sum_{i=0}^{\ell-1} \lvert S_i^*\rvert$	$\sum_{i=\ell}^{k}(\lvert\Omega_i(G)\rvert - 2)+$ $\sum_{i=0}^{\ell-1} \lvert(\Omega_i(G)\rvert - 2)$ $- \sum_{i=0}^{\ell-1} \lvert S_i^*\rvert$	$\sum_{i=0}^{\ell}(\lvert\Omega_i(G)\rvert - 2)$ $- \sum_{i=0}^{\ell-1} \lvert S_i\rvert$	$\sum_{i=0}^{\ell}(\lvert\Omega_i(G)\rvert - 2)$ $- \sum_{i=0}^{\ell-1} \lvert S_i\rvert$

Table 14.4 further gives tight bounds of the running time in terms of minimum-cut computations. The bounds are closely related to the bounds given for the level update procedures



(a) Time regarding an arbitrary hierarchy. (b) Lower bounds if hierarchy remains valid.

FIGURE 14.7: Sketch of the running times of the update approach for hierarchies of unrestricted cut clusterings. Shaded areas represent parts that are simply copied, and thus, save time compared to a computation from scratch.

in Table 14.1. We denote the set of clusters on level i that are nested in clusters that neither contain b nor d on the next higher level $i-1$ in the old hierarchy by S_i. These clusters are all maintained by the reuse of subtrees if the hierarchy remains valid. The clusterings and clusters on different levels are indexed by their level number. Figure 14.7 additionally sketches the running times for arbitrary hierarchies and the lower bounds in case the hierarchy remains valid.

As for the level update procedure for inter-cluster cost increase, the best and worst-case running times for increasing cost occur if the old hierarchy remains valid. Both running times result from only updating the remaining part in the second phase. The running times for decreasing cost consist of the time for searching levels k in the first phase and the time for updating the remaining part in the second phase. The best case occurs if the procedure needs only one cut computation per final cluster in the new hierarchy (apart from those clusters that are simply copied).

Conclusion of Part III

We examined and further improved the hierarchical cut-clustering algorithm introduced by Flake et al. [42]. This algorithm returns a set of cut clusterings at different levels of granularity forming a clustering hierarchy. The striking feature of the clusterings computed by this method is that they provide a guaranteed *expansion*—an NP-hard bottleneck measure—within and between clusters, tunable by an input parameter α.

Experimental and Theoretical Analysis. We comprehensively investigated the cut-clustering approach from an experimental and theoretical point of view, thereby bringing it together with the concept of cohesive subsets, which is popular in social network analysis.

In the experimental study in Section 11.3, we examined the behavior of the hierarchical cut-clustering algorithm in the light of expansion and modularity. Our experiments document that the given guarantee on intra-cluster expansion provides a deeper insight compared to a trivial bound that is easy to compute. The true intra-cluster expansion and inter-cluster expansion turned out to be even better than guaranteed. An analog analysis of the expansion of modularity-based clusterings could further give no evidence that modularity-based clusterings surpass cut clusterings in terms of intra-cluster expansion. On the contrary, around one fourth of the considered modularity-based clusterings could be proven to be worse than the cut clusterings.

Within the modularity analysis we could reveal that, although it is not designed to optimize modularity, the hierarchical cut-clustering algorithm fairly reliably finds clusterings of good modularity if those clusterings are structurally indicated. Otherwise, if no good clustering is clearly indicated, the cut-clustering algorithm returns only clusterings of low modularity. This confirms a high trustability of the cut-clustering algorithm and justifies the use of modularity applied to cut clusterings as a feasible measure for how well a graph can be clustered.

Our theoretical considerations in Section 11.1 revealed that, besides the quality guarantee on expansion, the clusters returned by the cut-clustering algorithm also provide nice cohesion properties, and thus, form a proper subclass of the class of source communities. The latter is a class of cohesive subsets squeezed in between the existing concepts of LS-sets/extreme sets and α-sets/web communities. We fully characterized the class of source communities as the set of regular M-sets and further gave a full characterization of the special subclass defined by the clusters in cut clusterings.

Improvements in Static Scenarios. Besides the detailed analysis, we further improved the cut-clustering approach in two directions. The first direction focuses on the cohesion properties of the clusters and aims at a better understanding of the source-community structure of a network. In this context we gave a parametric search approach in Section 11.2 that admits to efficiently compute complete hierarchies of cut clusterings where the clusterings on each level provide the guaranteed quality in terms of expansion as well as the cohesion properties. We note that this improved complete hierarchical approach was used for the experimental study in Section 11.3. Furthermore, in Chapter 12, we exploited the structure of unique-cut trees (Section 7.2) in order to develop a framework that admits the efficient construction of maximal SC-clusterings and overlay clusterings for given source communities, after precomputing at most $2(n-1)$ maximum flows. Moreover, precomputing only around $n-1$ maximum flows often suffices, since the cases that cause the additional flow computations (when the matrix set becomes invalid during the construction of the unique-cut tree) are rare in practice. For the lesmis network in the example in Section 12.3 we needed only $n+3$ maximum-flow computations, where $n = 77$ denotes the number of vertices in the network. We point out that a single maximal SC-clustering for a source community S can be also constructed directly by iteratively computing regular M-sets with respect to vertices not in S and the source of S. However, in the worst case, this needs $|V \setminus S|$ flow computations, namely if the M-sets are singletons or if they are considered in an order that causes many unnecessary computations of nested M-sets. In contrast, due to its short query times, our framework efficiently supports the detailed analysis of a network's source-community structure based on many different maximal SC-clusterings and overlay clusterings.

The second direction skips the focus on any cohesion properties to the benefit of more flexibility in choosing clusters, in particular, in the context of dynamically maintaining cut clusterings in evolving graphs.

Improvements Related to Dynamic Scenarios. In Chapter 13 we generalized the cut-clustering method, which so far was restricted to U-cuts, to an unrestricted method that may choose arbitrary minimum separating cuts for the construction of clustering hierarchies. The resulting clusters on the different hierarchy levels still provide a guaranteed expansion. The new degree of freedom makes the method more powerful, since the algorithm may now use the most appropriate cut with respect to a given objective.

In Chapter 14 we made use of the new degree of freedom in order to achieve optimal temporal smoothness for our dynamic version of the unrestricted cut-clustering approach. We presented an algorithm that efficiently and fully-dynamically maintains an entire hierarchy of cut clusterings based on update procedures we previously developed for a single cut clustering. Both dynamic approaches benefit from insights we gained in the context of dynamic Gomory-Hu trees.

Open Questions. The dynamic approach developed for hierarchies of unrestricted cut clusterings skips the cohesion properties of the clusterings to the benefit of optimal temporal smoothness. Conversely, weakening the notion of temporal smoothness to the benefit of cohesion properties possibly admits a dynamic approach for hierarchies of also restricted cut clusterings. Such a dynamic restricted approach can be possibly further extended such that it also guarantees the completeness of the hierarchy found in each time step. Furthermore, we considered a dynamic scenario based on atomic changes in the evolving graph. An extension of our results to more general dynamic scenarios could be addressed in future work.

CHAPTER 15

Conclusion

Graph connectivity is a wide field with many faces regarding different graph classes and different notions of connectivity, as *edge* and *vertex* connectivity in a *global* and *local* view, and extended connectivity concepts like, for example, *expansion*. Many connectivity and cut related problems are solvable in polynomial time, like the all-pairs minimum cut problem or the 2-connected planar 3-regular augmentation problem with fixed embedding. Others are NP-hard, like determining the expansion of a graph, or deciding for a planar graph whether it admits a c-connected planar 4-regular augmentation. Interestingly, although representing analogous concepts, edge and vertex connectivity behave diversely in many aspects. For example, it is not possible to represent all-pairs minimum separating vertex cuts by a tree [12], as it is realized for all-pairs minimum separating edge cuts by Gomory-Hu trees. The extract of connectivity and cut problems considered in this thesis also demonstrated the variety of connectivity related problems and presented different techniques solutions may rely on.

In Part I, we considered a bunch of regular augmentation problems for simple, planar, unweighted graphs, that increase the *global vertex* connectivity. We found that generalized matchings [50] and the characterization of valid node assignments by the help of indicator sets are convenient instruments to design efficient algorithm for those problem variants that are not proven to be NP-hard.

In contrast, Part II addressed the *local edge* connectivity between vertices in undirected, weighted graphs. Although there exist $\binom{n}{2}$ vertex pairs in a graph, a minimum separating edge cut for each vertex pair can be represented by a Gomory-Hu tree, which can be constructed by the help of only $n-1$ maximum-flow computations. Furthermore, due to the special nesting behavior of U-cuts and M-sets, the whole set of U-cuts can be also represented by a tree-based data structure, which we called unique-cut tree. Our investigation of the nesting behavior of U-cuts and M-sets was based on the Non-Crossing-Lemma (7.2) stated for static graphs by Gomory and Hu [59], and Gusfield [66]. For dynamic graphs, which evolve due to atomic edge and vertex changes, we could prove similar *non-crossing lemmas*, which provided the necessary tools for designing efficient and optimally smooth update algorithms for Gomory-Hu trees.

In Part III, we finally faced the connectivity indicated by the *(sub)global expansion* of subgraphs and the *local expansion* of cuts separating subgraphs. Flake et al. [42] discovered an interesting relation between the expansion of a subgraph, which is NP-hard to compute, and minimum separating edge cuts in parametric graphs. Based on this relation they designed a cut-based clustering algorithm that guarantees a lower bound on the expansion of the clusters. This guarantee is due to the fact that the clusters correspond to cut sides of minimum separating edge cuts in a special parametric graph. Based on the knowledge how minimum s-t-cuts and maximum s-t-flows, respectively, behave in parametric graphs for varying parameter values [54], Flake et al. further extended their approach to clustering hierarchies.

We investigated the relation between the found cut clusters and cohesive subsets. The latter are characterized by a special predominant connectivity property and are popular in social network analysis. We further found that the M-sets represented by unique-cut trees also provide some cohesion properties and that they admit a fast construction of inclusion-maximal clusterings consisting of such cohesive subsets. Moreover, we also employed the nesting behavior of minimum s-t-cuts in parametric graphs in order to show that complete cut-clustering hierarchies can be quickly found by a parametric search approach, and clustering hierarchies with a guaranteed expansion also exist for slightly more general clusters than those proposed by Flake et al. Our fully-dynamic update algorithm for these more general (unrestricted) cut-clustering hierarchies in dynamic scenarios finally benefits from a close relation to dynamic Gomory-Hu trees.

Bibliography

[1] M. Abellanas, A. García, F. Hurtado, J. Tejel, and J. Urrutia. Augmenting the connectivity of geometric graphs. *Computational Geometry Theory and Applications*, 40(3):220–230, 2008. [see page 25].

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993. [see page 15].

[3] M. Al-Jubeh, M. Ishaque, K. Rédei, D. L. Souvaine, and C. D. Tóth. Tri-edge-connectivity augmentation for planar straight line graphs. In: *Proceedings of the 20th International Symposium on Algorithms and Computation (ISAAC'09)*, volume 5878 of *Lecture Notes in Computer Science*, pages 902–912. Springer, 2009. [see page 25].

[4] N. Alon, R. Yuster, and U. Zwick. Finding and Counting Given Length Cycles. *Algorithmica*, 17(3):209–223, 1997. [see page 12].

[5] P. Angelini, G. Di Battista, F. Frati, V. Jelínek, J. Kratochvíl, M. Patrignani, and I. Rutter. Testing planarity of partially embedded graphs. In: *Proceedings of the 21st ACM-SIAM Symposium on Discrete Algorithms (SODA'10)*, pages 202–221. SIAM, 2010. [see page 74].

[6] S. R. Arikati and K. Mehlhorn. A correctness certificate for the Stoer-Wagner min-cut algorithm. *Information Processing Letters*, 70(5):251–254, 1999. [see page 12].

[7] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009. [see page 19].

[8] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation - Combinatorial Optimization Problems and Their Approximability Properties*. Springer, 2002. [see page 178].

[9] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, editors. *Graph Partitioning and Graph Clustering: 10th DIMACS Implementation Challenge*, volume 588 of *DIMACS Book*. American Mathematical Society, 2013. [see page 15].

[10] D. W. Barnette. On Steinitz's theorem concerning convex 3-polytopes and on some properties of planar graphs. In: *The Many Facets of Graph Theory*, volume 110 of *Lecture Notes in Mathematics*, pages 27–40. Springer, 1969. [see page 34].

[11] D. Barth, P. Berthomé, M. Diallo, and A. Ferreira. Revisiting parametric multi-terminal problems: Maximum flows, minimum cuts and cut-tree computations. *Discrete Optimization*, 3(3):195–205, 2006. [see pages viii, 5, 103, 129, 131, 173].

[12] A. A. Benczúr. Counterexamples for Directed and Node Capacitated Cut-Trees. *SIAM Journal on Computing*, 24(3):505–510, 1995. [see pages vii, 98, 239].

[13] A. Bhalgat, R. Hariharan, T. Kavitha, and D. Panigrahi. An $\tilde{O}(mn)$ Gomory-Hu Tree Construction Algorithm for Unweighted Graphs. In: *Proceedings of the 39th Annual ACM Symposium on the Theory of Computing (STOC'07)*, pages 605–614. ACM Press, 2007. [see pages vii, 98].

[14] V. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10), 2008. [see page 183].

[15] S. P. Borgatti, M. G. Everett, and P. R. Shirey. LS sets, lambda sets and other cohesive subsets. *Social Networks*, 12(4):337–357, 1990. [see pages ix, 178, 180, 187, 188].

[16] G. Borradaile and P. N. Klein. An O(n log n) algorithm for maximum st-flow in a directed planar graph. *Journal of the ACM*, 56(2):9:1–9:30, 2009. [see page 14].

[17] G. Borradaile, P. Sankowski, and C. Wulff-Nilsen. Min st-cut Oracle for Planar Graphs with Near-Linear Preprocessing Time. In: *Proceedings of the 51st Annual IEEE Symposium on Foundations of Computer Science (FOCS'10)*, pages 601–610. IEEE Computer Society, 2010. [see pages vii, 98].

[18] J. M. Boyer and W. J. Myrvold. On the Cutting Edge: Simplified O(n) Planarity by Edge Addition. *Journal of Graph Algorithms and Applications*, 8(3):241–273, 2004. [see page 27].

[19] U. Brandes. Eager *st*-Ordering. In: *Proceedings of the 10th Annual European Symposium on Algorithms (ESA'02)*, volume 2461 of *Lecture Notes in Computer Science*, pages 183–197. Springer, 2002. [see page 13].

[20] U. Brandes, D. Delling, M. Gaertler, R. Görke, M. Höfer, Z. Nikoloski, and D. Wagner. On Modularity Clustering. *IEEE Transactions on Knowledge and Data Engineering*, 20(2):172–188, 2008. [see pages 183, 201].

[21] U. Brandes and T. Erlebach, editors. *Network Analysis: Methodological Foundations*, volume 3418 of *Lecture Notes in Computer Science*. Springer, 2005. [see page 178].

[22] C. Bron and J. A. G. M. Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973. [see page 179].

[23] S. Chaudhuri, K. V. Subrahmanyam, F. Wagner, and C. Zaroliagis. Computing Mimicking Networks. *Algorithmica*, 26(1):31–49, 2000. [see page 101].

[24] C. Cheng and T. Hu. Maximum Concurrent Flows and Minimum Cuts. *Algorithmica*, 8(1-6):233–249, 1992. [see page 101].

[25] A. Clauset, M. E. J. Newman, and C. Moore. Finding community structure in very large networks. *Physical Review E*, 70(066111), 2004. [see page 184].

[26] S. A. Cook. The complexity of theorem-proving procedures. In: *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing (STOC'71)*, pages 151–158. ACM Press, 1971. [see page 20].

[27] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001. [see pages 1, 11, 108].

[28] G. B. Dantzig and D. R. Fulkerson. On the max-flow min-cut theorem of networks. In: *Linear Inequalities and Related Systems*, volume 38 of *Annals of Mathematics Studies*, pages 215–221. Princeton University Press, 1956. [see page 11].

[29] M. de Berg and A. Khosravi. Optimal binary space partitions for segments in the plane. *International Journal on Computational Geometry & Applications*, 22(3):187–206, 2012. [see page 74].

[30] D. Delling, R. Görke, C. Schulz, and D. Wagner. ORCA Reduction and ContrAction Graph Clustering. In: *Proceedings of the 5th International Conference on Algorithmic Aspects in Information and Management (AAIM'09)*, volume 5564 of *Lecture Notes in Computer Science*, pages 152–165. Springer, 2009. [see page 179].

[31] I. Derényi, G. Palla, and T. Vicsek. Clique Percolation in Random Networks. *Physical Review Letters*, 94:160202, 2005. [see page 179].

[32] Y. Dinitz, A. V. Karzanov, and M. V. Lomonosov. On the structure of the system of minimum edge cuts in a graph. In: *In Studies in Discrete Optimization*, pages 290–306. Nauka, 1976. [see pages vii, 5, 101].

[33] G. A. Dirac. Some theorems on abstract graphs. *Proceedings of the London Mathematical Society*, 3(2):69–81, 1952. [see page 40].

[34] C. Doll, T. Hartmann, and D. Wagner. Fully-Dynamic Hierarchical Graph Clustering Using Cut Trees. In: *Proceedings of the 12th International Symposium on Algorithms and Data Structures (WADS'11)*, volume 6844 of *Lecture Notes in Computer Science*, pages 338–349. Springer, 2011. [see page 182].

[35] J. Duch and A. Arenas. Community Detection in Complex Networks using Extremal Optimization. *Physical Review E*, 72(027104):1–4, 2005. [see page 184].

[36] S. E. Elmaghraby. Sensivity Analysis of Multiterminal Flow Networks. *Operations Research*, 12(5):680–688, 1964. [see pages viii, 5, 129].

[37] J. Erickson. Maximum flows and parametric shortest paths in planar graphs. In: *Proceedings of the 21st Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'10)*, pages 794–804. SIAM, 2010. [see page 14].

[38] K. Eswaran and R. E. Tarjan. Augmentation Problems. *SIAM Journal on Computing*, 5(4):653–665, 1976. [see page 25].

[39] S. Even, A. Itai, and A. Shamir. On the complexity of time table and multi-commodity flow problems. In: *Proceedings of the 16th Annual IEEE Symposium on Foundations of Computer Science (FOCS'75)*, pages 184–193. IEEE Computer Soc., 1975. [see page 15].

[40] M. G. Everett and S. P. Borgatti. Analyzing clique overlap. *Connections*, 21(1):49–61, 1998. [see page 179].

[41] G. W. Flake, S. Lawrence, C. L. Giles, and F. M. Coetzee. Self-Organization and Identification of Web Communities. *IEEE Computer*, 35(3):66–71, 2002. [see pages 187, 188].

[42] G. W. Flake, R. E. Tarjan, and K. Tsioutsiouliklis. Graph Clustering and Minimum Cut Trees. *Internet Mathematics*, 1(4):385–408, 2004. [see pages viii, 4, 5, 6, 178, 179, 180, 181, 184, 185, 188, 193, 196, 197, 198, 213, 219, 220, 231, 237, 239].

[43] L. Fleischer. Building Chain and Cactus Representations of All Minimum Cuts from Hao-Orlin in the Same Asymptotic Run Time. *Journal of Algorithms*, 33(1):51–72, 1999. [see page 101].

[44] M. A. Fonoberova and D. D. Lozovanu. The maximum flow in dynamic networks. *Computer Science Journal of Moldova*, 12(3):378–396, 2004. [see page 110].

[45] L. R. Ford, Jr. and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956. [see pages v, 11].

[46] S. Fortunato. Community detection in graphs. *Physics Reports*, 486(3–5):75–174, 2010. [see pages 178, 227].

[47] S. Fortunato and M. Barthélemy. Resolution limit in community detection. *Proceedings of the National Academy of Science of the United States of America*, 104(1):36–41, 2007. [see page 184].

[48] A. Frank. Connectivity augmentation problems in network design. In: *Mathematical Programming: State of the Art*, pages 34–63. The University of Michigan, 1994. [see pages vi, 25].

[49] G. N. Frederickson and J. Ja'Ja'. Approximation Algorithms for Several Graph Augmentation Problems. *SIAM Journal on Computing*, 10(2):270283, 1981. [see page 25].

[50] H. N. Gabow. An efficient reduction technique for degree-constrained subgraph and bidirected network flow problems. In: *Proceedings of the 15th Annual ACM Symposium on the Theory of Computing (STOC'83)*, pages 448–456. ACM Press 1983. [see pages 27, 34, 54, 61, 66, 67, 73, 239].

[51] H. N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. *Journal of Computer and System Sciences*, 50(2):259–273, 1995. [see page 12].

[52] H. N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74(3-4):107–114, 2000. [see page 13].

[53] H. N. Gabow. Using expander graphs to find vertex connectivity. *Journal of the ACM*, 53(5):800–844, 2006. [see page 13].

[54] G. Gallo, M. D. Grigoriadis, and R. E. Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM Journal on Computing*, 18(1):30–55, 1989. [see pages 129, 180, 186, 240].

[55] M. R. Garey and D. S. Johnson. *Computers and Intractability. A Guide to the Theory of $\mathcal{NP}$-Completeness.* W. H. Freeman and Company, 1979. [see page 178].

[56] A. V. Goldberg and R. Tarjan. Finding minimum-cost circulations by canceling negative cycles. *Journal of the ACM*, 36(4):873–886, 1989. [see page 14].

[57] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988. [see pages v, 14, 108].

[58] A. V. Goldberg and K. Tsioutsiouliklis. Cut Tree Algorithms. In: *Proceedings of the 10th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'99)*, pages 376–385. SIAM, 1999. [see page 99].

[59] R. E. Gomory and T. Hu. Multi-terminal network flows. *SIAM Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961. [see pages vii, 5, 6, 7, 98, 100, 102, 115, 129, 173, 186, 188, 239].

[60] R. Görke. *An Algorithmic Walk from Static to Dynamic Graph Clustering.* PhD thesis, Fakultät für Informatik, 2010. [see pages 178, 182].

[61] R. Görke, T. Hartmann, and D. Wagner. Dynamic Graph Clustering Using Minimum-Cut Trees. In: *Proceedings of the 11th International Symposium on Algorithms and Data Structures (WADS'09)*, volume 5664 of *Lecture Notes in Computer Science*, pages 339–350. Springer, 2009. [see page 182].

[62] R. Görke, T. Hartmann, and D. Wagner. Dynamic Graph Clustering Using Minimum-Cut Trees. *Journal of Graph Algorithms and Applications*, 16(2):411–446, 2012. [see page 182].

[63] J. L. Gross and J. Yellen. *Handbook of Graph Theory.* CRC Press, 2003. [see page 2].

[64] R. Guimerà and L. A. N. Amaral. Functional Cartography of Complex Metabolic Networks. *Nature*, 433:895–900, 2005. [see page 184].

[65] R. P. Gupta. Two Theorems on Pseudosymmetric Graphs. *SIAM Journal on Applied Mathematics*, 15(1):168–171, 1967. [see page 98].

[66] D. Gusfield. Very simple methods for all pairs network flow analysis. *SIAM Journal on Computing*, 19(1):143–155, 1990. [see pages 7, 100, 102, 103, 114, 115, 161, 162, 173, 239].

[67] D. Gusfield and D. Naor. Efficient algorithms for generalized cut trees. In: *Proceedings of the 1st Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'90)*, pages 422–433. SIAM, 1990. [see page 98].

[68] D. Gusfield and D. Naor. Efficient algorithms for generalized cut-trees. *Networks*, 21(5):505–520, 1991. [see page 98].

[69] D. Gusfield and D. Naor. Extracting Maximal Information About Sets of Minimum Cuts. *Algorithmica*, 10(1):64–89, 1993. [see pages 100, 108, 110].

[70] C. Gutwenger and P. Mutzel. A Linear Time Implementation of SPQR-Trees. In: *Proceedings of the 8th International Symposium on Graph Drawing (GD'00)*, volume 1984 of *Lecture Notes in Computer Science*, pages 70–90. Springer, 2001. [see page 13].

[71] T. Hagerup, J. Katajainen, N. Nishimura, and P. Ragde. Characterizing Multiterminal Flow Networks and Computing Flows in Networks of Small Treewidth. *Journal of Computer and System Sciences*, 57(3):366–375, 1998. [see pages 14, 99, 101].

[72] M. Hamann, T. Hartmann, and D. Wagner. Complete Hierarchical Cut-Clustering: A Case Study on Expansion and Modularity. In: *Graph Partitioning and Graph Clustering: 10th DIMACS Implementation Challenge*, volume 588 of *DIMACS Book*, pages 157–170. American Mathematical Society, 2013. [see page 181].

[73] M. Hamann, T. Hartmann, and D. Wagner. Hierarchies of Predominantly Connected Communities. In: *Proceedings of the 13th International Symposium on Algorithms and Data Structures (WADS'13)*, volume 8037 of *Lecture Notes in Computer Science*, pages 365–377. Springer, 2013. [see pages 102, 181].

[74] J. Hao and J. B. Orlin. A Faster Algorithm for Finding the Minimum Cut in a Directed Graph. *Journal of Algorithms*, 17(3):424–446, 1994. [see pages 12, 102].

[75] T. Hartmann. Clustering Dynamic Graphs with Guaranteed Quality. Master's thesis, Department of Informatics, 2008. [see page 220].

[76] T. Hartmann, A. Kappes, and D. Wagner. Clustering Evolving Networks. To appear in: *Algorithm Engineering*, special issue of *Lecture Notes in Computer Science*, edited by P. Sanders and L. Kliemann. Springer. http://arxiv.org/abs/1401.3516 [see page 219].

[77] T. Hartmann, J. Rollin, and I. Rutter. Cubic Augmentation of Planar Graphs. In: *Proceedings of the 23rd International Symposium on Algorithms and Computation (ISAAC'12)*, volume 7676 of *Lecture Notes in Computer Science*, pages 402–412. Springer, 2012. [see page 26].

[78] T. Hartmann, J. Rollin, and I. Rutter. Regular Augmentation of Planar Graphs. Accepted under minor revision at: *Algorithmica*. [see page 26].

[79] T. Hartmann and D. Wagner. Fast and Simple Fully-Dynamic Cut Tree Construction. In: *Proceedings of the 23rd International Symposium on Algorithms and Computation (ISAAC'12)*, volume 7676 of *Lecture Notes in Computer Science*, pages 95–104. Springer, 2012. [see page 103].

[80] D. Hartvigsen. Characterizing the flow equivalent trees of a network. *Discrete Applied Mathematics*, 128(2-3):387 –394, 2003. [see page 100].

[81] D. Haugland, M. Eleyat, and M. L. Hetland. The Maximum Flow Problem with Minimum Lot Sizes. In: *Proceedings of the 2nd International Conference on Computational Logistics*, volume 6971 of *Lecture Notes in Computer Science*, pages 170–182. Springer, 2011. [see page 15].

[82] M. R. Henzinger, S. Rao, and H. N. Gabow. Computing vertex connectivity: new bounds from old techniques. *Journal of Algorithms*, 34(2):222–250, 2000. [see page 13].

[83] J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2(3):135–158, 1973. [see page 13].

[84] F. Hurtado and C. D. Tóth. Plane geometric graph augmentation: a generic perspective. In: *Thirty Essays on Geometric Graph Theory*, pages 327–354. Springer, 2013. [see page 25].

[85] G. F. Italiano, Y. Nussbaum, P. Sankowski, and C. Wulff-Nilsen. Improved algorithms for min cut and max flow in undirected planar graphs. In: *Proceedings of the 43rd Annual ACM Symposium on the Theory of Computing (STOC'11)*, pages 313–322. ACM Press, 2011. [see pages 12, 14].

[86] H. Jeong, S. P. Mason, A.-L. Barabási, and Z. N. Oltvai. Lethality and Centrality in Protein Networks. *Nature*, 411:41–42, 2001. [see pages 18, 199].

[87] R. Kannan, S. Vempala, and A. Vetta. On Clusterings: Good, Bad, Spectral. *Journal of the ACM*, 51(3):497–515, 2004. [see pages viii, 4, 178, 184, 185].

[88] G. Kant and H. L. Bodlaender. Planar Graph Augmentation Problems. In: *Proceedings of the 2nd International Workshop on Algorithms and Data Structures (WADS'91)*, volume 519 of *Lecture Notes in Computer Science*, pages 286–298. Springer, 1991. [see pages vi, 25].

[89] G. Karakostas. Faster approximation schemes for fractional multicommodity flow problems. *ACM Transactions on Algorithms*, 4(1):13:1–13:17, 2008. [see page 15].

[90] D. R. Karger. Minimum cuts in near-linear time. *Journal of the ACM*, 47(1):46–76, 2000. [see page 12].

[91] D. R. Karger and M. S. Levine. Finding Maximum Flows in Undirected Graphs Seems Easier than Bipartite Matching. In: *Proceedings of the 30th Annual ACM Symposium on the Theory of Computing (STOC'98)*, pages 69–78. ACM Press, 1998. [see page 14].

[92] D. R. Karger and D. Panigrahi. A Near-Linear Time Algorithm for Constructing a Cactus Representation of Minimum Cuts. In: *Proceedings of the 20st Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'09)*, pages 246–255. SIAM, 2009. [see page 102].

[93] R. M. Karp. Reducibility among Combinatorial Problems. In: *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972. [see page 21].

[94] D. E. Knuth. *The Stanford GraphBase : a platform for combinatorial computing.* Addison-Wesley, 1993. [see page 205].

[95] D. E. Knuth and A. Raghunathan. The problem of compatible representatives. *SIAM Journal on Discrete Mathematics*, 5(3):422–427, 1992. [see page 31].

[96] P. Kohli and P. H. Torr. Dynamic Graph Cuts for Efficient Inference in Markov Random Fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(12):2079–2088, 2007. [see page 110].

[97] R. Krauthgamer and I. Rika. Mimicking Networks and Succinct Representations of Terminal Cuts. In: *Proceedings of the 24th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'13)*, pages 1789–1799. SIAM, 2013. [see page 101].

[98] J. Łącki and P. Sankowski. Min-Cuts and Shortest Cycles in Planar Graphs in O(n loglogn) Time. In: *Proceedings of the 19th Annual European Symposium on Algorithms (ESA'11)*, volume 6942 of *Lecture Notes in Computer Science*, pages 155–166. Springer, 2011. [see page 12].

[99] D. Lichtenstein. Planar formulae and their uses. *SIAM Journal on Computing*, 11(2):329–343, 1982. [see page 31].

[100] S. C. Lin and E. Ma. Sensitivity analysis of 0-1 multiterminal network flows. *Networks*, 21(7):713–745, 1991. [see pages viii, 129].

[101] D. Lisowski. Modularity-basiertes Clustern von dynamischen Graphen im Offline-Fall. Master's thesis, Department of Informatics, Karlsruhe Institute of Technology (KIT), 2011. [see page 199].

[102] L. Lovász. Computing ears and branchings in parallel. In *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science (FOCS'85)*, pages 464–467. IEEE Computer Society, 1985. [see page 13].

[103] F. Luccio and M. Sami. On the decomposition of networks in minimally interconnected subnetworks. *IEEE Transactions on Circuit Theory*, CT-16:184–188, 1969. [see page 178].

[104] R. D. Luce and A. Perry. A method of matrix analysis of group structure. *Psychometrika*, 14:95–116, 1949. [see page 179].

[105] A. Madry. Fast Approximation Algorithms for Cut-Based Problems in Undirected Graphs. In *Proceedings of the 51st Annual IEEE Symposium on Foundations of Computer Science (FOCS'10)*, pages 245–254. IEEE Computer Society, 2010. [see page 21].

[106] W. Mayeda. Terminal and Branch Capacity Matrices of a Communication Net. *IRE Transactions on Circuits Theory*, CT-7:261–269, 1960. [see pages 15, 97].

[107] K. Menger. Zur allgemeinen Kurventheorie. *Fundamenta Mathematicae*, 10:96–115, 1927. [see page 12].

[108] N. Mishra, R. Schreiber, I. Stanton, and R. Tarjan. Finding Strongly Knit Clusters in Social Networks. *Internet Mathematics*, 5(1-2):153–172, 2008. [see page 180].

[109] R. J. Mokken. Cliques, clubs, and clans. *Quality and Quantity*, 13:161–173, 1979. [see page 180].

[110] H. Nagamochi. Graph Algorithms for Network Connectivity Problems. *Journal of the Operations Research Society of Japan*, 47(4):199–223, 2004. [see pages 97, 99, 100, 107, 178, 179, 188].

[111] H. Nagamochi and T. Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7:583–596, 1992. [see page 14].

[112] H. Nagamochi and T. Ibaraki. Computing Edge-Connectivity in Multigraphs and Capacitated Graphs Computing Edge-Connectivity in Multigraphs and Capacitated Graphs . *SIAM Journal on Discrete Mathematics*, 5(1):54–66, 1992. [see page 12].

[113] H. Nagamochi and T. Ibaraki. Graph connectivity and its augmentation: applications of ma orderings. *Discrete Applied Mathematics*, 123(1–3):447–472, 2002. [see pages vi, 25].

[114] H. Nagamochi and T. Kameda. Constructing Cactus Representation for all Minimum Cuts in an Undirected Network. *Journal of the Operations Research Society of Japan*, 39(2):135–158, 1996. [see page 101].

[115] M. E. J. Newman. Modularity and Community Structure in Networks. *Proceedings of the National Academy of Science of the United States of America*, 103(23):8577–8582, 2006. [see page 184].

[116] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(026113):1–16, 2004. [see pages 178, 183, 184].

[117] J. B. Orlin. Maximum-Throughput Dynamic Network Flows. *Mathematical Programming*, 27(2):214–231, 1983. [see page 110].

[118] J.-C. Picard and M. Queyranne. On the Structure of All Minimum Cuts in a Network and Applications. In: *Combinatorial Optimization II*, volume 13 of *Mathematical Programming Studies*, pages 8-16. Springer, 1980. [see pages 5, 100, 108, 109].

[119] A. Pilz. Augmentability to cubic graphs. In: *Proceedings of the 28th European Workshop on Computational Geometry (EuroCG'12)*, pages 29–32, 2012. [see pages vii, 26].

[120] J. Reichardt and S. Bornholdt. Statistical Mechanics of Community Detection. *Physical Review E*, 74(016110):1–16, 2006. [see page 184].

[121] R. Rotta and A. Noack. Multilevel local search algorithms for modularity clustering. *ACM Journal of Experimental Algorithmics*, 16:2.3:2.1–2.3:2.27, 2011. [see pages 198, 199, 201].

[122] I. Rutter and A. Wolff. Augmenting the Connectivity of Planar and Geometric Graphs. *Journal of Graph Algorithms and Applications*, 16(2):599–628, 2012. [see pages 25, 26, 31, 32].

[123] B. Saha and P. Mitra. Dynamic Algorithm for Graph Clustering Using Minimum Cut Tree. In: *Proceedings of the Sixth IEEE International Conference on Data Mining (Workshops)*, pages 667–671. IEEE Computer Society, 2006. [see pages 219, 220].

[124] B. Saha and P. Mitra. Dynamic Algorithm for Graph Clustering Using Minimum Cut Tree. In *Proceedings of the 2007 SIAM International Conference on Data Mining*, pages 581–586. SIAM, 2007. [see pages ix, 182, 219, 220, 221, 222].

[125] S. E. Schaeffer. Graph Clustering. *Computer Science Review*, 1(1):27–64, 2007. [see page 178].

[126] C. P. Schnorr. Multiterminal Network Flow and Connectivity in Unsymmetrical Networks. In: *Proceedings of a Workshop on Optimization and Operations Research*, volume 157 of *Lecture Notes in Economics and Mathematical Systems*, pages 241–254. Springer, 1978. [see page 98].

[127] C. P. Schnorr. Multiterminal network flow and connectivity in unsymmetrical networks. In: *Proceedings of the 5th International Colloquium on Automata, Languages and Programming (ICALP'98)*, volume 62 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 1978. [see page 98].

[128] C. P. Schnorr. Bottlenecks and edge connectivity in unsymmetrical networks. *SIAM Journal on Computing*, 8(2):265–274, 1979. [see page 98].

[129] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Springer, 2003. [see page 14].

[130] J. Schroeder and E. P. D. Jr.˙Fault-Tolerant Dynamic Routing Based on Maximum Flow Evaluation. In: *Proceedings of the 3rd Latin-American Symposium on Dependable Computing*, volume 4746 of *Lecture Notes in Computer Science*, pages 7–24. Springer, 2007. [see page 5].

[131] M. G. Scutellà. A note on the parametric maximum flow problem and some related reoptimization issues. *Annals of Operations Research*, 150(1):231–244, 2006. [see page 129].

[132] S. B. Seidman and B. L. Foster. A graph-theoretic generalization of the clique concept. *Journal of Mathematical Sociology*, 6:139–154, 1978. [see page 180].

[133] M. Stoer and F. Wagner. A Simple Min-Cut Algorithm. *Journal of the ACM*, 44(4):585–591, 1997. [see page 12].

[134] J. J. Sylvester. On an Application of the New Atomic Theory to the Graphical Representation of the Invariants and Covariants of Binary Quantics, With Three Appendices. *American Journal of Mathematics*, 1(1):64–104 and 1(2):105–125 1878. [see page 2].

[135] C. D. Tóth. Connectivity augmentation in plane straight line graphs. *European Journal of Combinatorics*, 33(3):408–425, 2012. [see page 25].

[136] R. Vahrenkamp. Multiterminal network flows and applications. *Zeitschrift für Operations Research*, 25(5):133–142, 1981. [see page 98].

[137] S. Wagner and D. Wagner. Comparing Clusterings – An Overview. Technical Report 2006-04, ITI Wagner, Department of Informatics, Universität Karlsruhe (TH), 2007. [see page 227].

[138] T. Watanabe and A. Nakamura. Edge-connectivity augmentation problems. *Journal of Computer and System Sciences*, 35(1):96–144, 1987. [see page 25].

[139] S. White and P. Smyth. A Spectral Clustering Approach to Finding Communities in Graphs. In: *Proceedings of the 5th SIAM International Conference on Data Mining*, pages 274–285. SIAM, 2005. [see page 184].

[140] H. Whitney. Congruent graphs and the connectivity of graphs. *American Journal of Mathematics*, 54:150–168, 1932. [see page 32].

[141] Z. Wu and R. Leahy. An Optimal Graph Theoretic Approach to Data Clustering: Theory and its Application to Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(11):1101–1113, 1993. [see page 188].

[142] W. W. Zachary. An Information Flow Model for Conflict and Fission in Small Groups. *Journal of Anthropological Research*, 33(4):452–473, 1977. [see page 189].

[143] K. A. Zweig. To Cluster or not to Cluster ? A Meta-Analytic Approach. In *Proceedings of the European Conference of Complex Systems (ECCS'08)*, pages 1–14. ECCS, 2008. [see page 177].

Appendices

List of Publications

Journal Articles

[1] **Identifikation von Clustern in Graphen**. *Informatik Spektrum*, 36(2):144–152, 2013. Joint work with Andrea Kappes and Dorothea Wagner.

[2] **Dynamic Graph Clustering Using Minimum-Cut Trees**. *Journal of Graph Algorithms and Applications*, 16(2):411–446, 2012. Joint work with Robert Görke and Dorothea Wagner.

Articles in Refereed Conference Proceedings

[1] **Hierarchies of Predominantly Connected Communities**. In: *Proceedings of the 13th International Symposium on Algorithms and Data Structures (WADS'13)*, volume 8037 of *Lecture Notes in Computer Science*, pages 365–377. Springer, 2013. Joint work with Michael Hamann and Dorothea Wagner.

[2] **Complete Hierarchical Cut-Clustering: A Case Study on Expansion and Modularity**. In: *Graph Partitioning and Graph Clustering: 10th DIMACS Implementation Challenge*, volume 588 of *DIMACS Book*, pages 157–170. American Mathematical Society, 2013. Joint work with Michael Hamann and Dorothea Wagner.

[3] **Fast and Simple Fully-Dynamic Cut Tree Construction**. In: *Proceedings of the 23rd International Symposium on Algorithms and Computation (ISAAC'12)*, volume 7676 of *Lecture Notes in Computer Science*, pages 95–104. Springer, 2012. Joint work with Dorothea Wagner.

[4] **Cubic Augmentation of Planar Graphs**. In: *Proceedings of the 23rd International Symposium on Algorithms and Computation (ISAAC'12)*, volume 7676 of *Lecture Notes in Computer Science*, pages 402–412. Springer, 2012. Joint work with Jonathan Rollin and Ignaz Rutter.

[5] **Fully-Dynamic Hierarchical Graph Clustering Using Cut Trees**. In: *Proceedings of the 12th International Symposium on Algorithms and Data Structures (WADS'11)*, volume 6844 of *Lecture Notes in Computer Science*, pages 338–349. Springer, 2011. Joint work with Christof Doll and Dorothea Wagner.

[6] **Dynamic Graph Clustering Using Minimum-Cut Trees**. In: *Proceedings of the 11th International Symposium on Algorithms and Data Structures (WADS'09)*, volume 5664 of *Lecture Notes in Computer Science*, pages 339–350. Springer, 2009. Joint work with Robert Görke and Dorothea Wagner.

Recently Accepted Articles

[1] **Regular Augmentation of Planar Graphs**. Accepted under minor revision at: *Algorithmica*. Joint work with Jonathan Rollin and Ignaz Rutter.

[2] **Clustering Evolving Networks**. To appear in: *Algorithm Engineering*, special issue of *Lecture Notes in Computer Science*, edited by Peter Sanders and Lasse Kliemann, Springer. Joint work with Andrea Kappes and Dorothea Wagner.

Master's Theses

[1] **Clustering Dynamic Graphs with Guaranteed Quality**. Master's thesis, Department of Theoretical Informatics, Fakultät für Informatik, Universität Karlsruhe (TH), October 2008.

[2] **Werbung zwischen Kunst und Manipulation – Denkanstöße zur persönlichen Auseinandersetzung mit dem polarisierenden Wesen der Werbung (Advertising between Art and Manipulation – The Divisive Character of Advertising)**. Master's thesis, Department of Electronic Media, Hochschule der Medien (HDM), February 2002.

Curriculum Vitæ

Name	Tanja Hartmann
Date of Birth	17 January 1978
Place of Birth	Pforzheim
Citizenship	German

08/1988 – 06/1997	Abitur (final secondary school examinations), Lise-Meitner-Gymnasium, Königsbach, Germany
10/1997 – 02/2002	Diploma in Advertising and Market Communication, Department of Electronic Media, Hochschule der Medien (HDM), Stuttgart, Germany
03/2002 – 09/2002	Traveling New Zealand, Australia and Southeast Asia
10/2002 – 07/2009	Diploma in Mathematics, Fakultät für Mathematik, Universität Karlsruhe (TH), Karlsruhe, Germany
04/2004 – 10/2008	Diploma in Informatics, Fakultät für Informatik, Universität Karlsruhe (TH), Karlsruhe, Germany
since 08/2009	Ph.D. student and research assistant in the Priority Programme 1307 *Algorithm Engineering* funded by the German Research Foundation (DFG), Department of Theoretical Informatics, Karlsruhe Institute of Technology (KIT). Advisor: Prof. Dr. Dorothea Wagner

10/1998 – 03/1999	Internship in the field of *desktop publishing and reproduction*, printing office Polyfoto Vogt KG, Stuttgart, Germany
08/1999 – 09/1999	Internship in the field of *customer consulting and contact*, advertising agency Mauch, Kirschner, Lelewel (MKL), Stuttgart, Germany
03/2000 – 08/2000	Internship in the field of *communication management*, Power Tools Devision, Robert Bosch GmbH, Leinfelden, Germany
09/2000 – 02/2001	Internship in the field of *marketing and communication*, International Sales Department, Atco-Qualcast, Stowmarket, England
04/2003 – 11/2006	Working student in the field of *analysis and valuations*, EnBW Trading GmbH, Karlsruhe, Germany

Ich versichere, diese Dissertation selbstständig angefertigt, alle benutzten Hilfsmittel vollständig angegeben, und kenntlich gemacht zu haben, was aus Arbeiten anderer und eigener Veröffentlichungen unverändert oder mit Änderungen entnommen wurde.

Tanja Hartmann