Weighted $\mathcal{F}$-free Edge Editing

Bachelor Thesis of

Jonas Spinner

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. Peter Sanders
Advisors:	Michael Hamann, M.Sc.
	Lars Gottesbüren, M.Sc.

Time Period: 1st June 2019 – 29th November 2019

Selbstständigkeitserklärung

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, 29. November 2019

Abstract

An $\mathcal{F}$-free graph does not contain an induced subgraph from a set of forbidden subgraphs $\mathcal{F}$. Edges in a graph can be edited, i.e. removed or inserted, to reach a graph which is $\mathcal{F}$-free. Finding a minimal set of such edits is the goal of $\mathcal{F}$-free Edge Editing. In this thesis, we cover a generalization, Weighted $\mathcal{F}$-free Edge Editing, which allows non-unit costs for each editing operation.

We focus on a fixed-parameter tractable (FPT) search tree algorithm with the editing cost as the parameter and adapt existing speed-up techniques for unweighted editing to the weighted case. For instance, we investigate algorithms for calculating lower bounds and subgraph selection strategies for branching. Also, we discuss the problem of finding the optimal editing cost for the search tree algorithm and propose two novel search strategies. Additionally, we cover an integer linear program (ILP) algorithm and methods for restricting constraint generation.

Moreover, we evaluate the FPT and ILP algorithms and their speed-up techniques on protein-protein interaction networks for $\mathcal{F} = \{C_4, P_4\}$. The FPT algorithm benefits the most from the greedy lower bound and the “most adjacent” subgraph selection rule for branching. Also, we notice that the local search algorithm suffers from local maxima in the weighted case. Furthermore, we find that the restrictions on the generation of constraints lead to significant running time improvements for the ILP algorithm. Finally, we compare both solving algorithms and find that the ILP algorithm consistently manages to outperform the FPT approach.

Deutsche Zusammenfassung

Ein $\mathcal{F}$-freier Graph besitzt keinen induzierten Teilgraphen aus einer Menge von verbotenen Teilgraphen $\mathcal{F}$. Man kann Kanten in einem Graphen editieren (einfügen oder entfernen) um einen Graphen zu erreichen, der $\mathcal{F}$-frei ist. Das Ziel von $\mathcal{F}$-free Edge Editing ist es, eine minimale Menge an Editierungsoperation zu finden, die zu einem $\mathcal{F}$-freien Graphen führen. Wir betrachten eine Generalisierung, Weighted $\mathcal{F}$-free Edge Editing, die beliebige Kosten für die Editierungsoperationen erlaubt.

In dieser Arbeit fokussieren wir uns auf einen parametrisierten Suchbaumalgorithmus (FPT) mit den Editierungskosten als Parameter. Wir adaptieren bereits existierende Beschleunigungstechniken für das ungewichteten Editieren für den gewichteten Fall. Unter anderem betrachten wir Algorithmen zum Berechnen von unteren Schranken und Strategien für die Auswahl von Teilgraphen zum Verzweigen. Außerdem diskutieren wir das Problem, die optimalen Editierungskosten für den Suchbaumalgorithmus zu finden und präsentieren dafür zwei neue Suchstrategien. Zusätzlich behandeln wir einen Algorithmus basierend auf ganzahliger linearer Optimierung (ILP) und Methoden, die die Anzahl der generierten Bedingungen beschränken.

Des Weiteren evaluieren wir die FPT und ILP Algorithmen und ihre Beschleunigungstechniken auf Protein-Protein Interaktionsgraphen für $\mathcal{F} = \{C_4, P_4\}$. Wir stellen fest, dass der FPT Algorithmus am meisten von dem Greedy-Algorithmus für untere Schranken und der Auswahlstrategie “most adjacent” profitiert. Letztere präferiert Teilgraphen, die zu vielen anderen verbotenen Teilgraphen adjazent sind. Auch fanden wir heraus, dass der Algorithmus für die untere Schranken, der auf lokaler Suche basiert, im gewichteten Fall größere Probleme mit lokalen Maxima hat. Weiterhin bemerken wir, dass das Beschränken der Anzahl der generierten Bedingungen den ILP Algorithmus signifikant schneller werden lässt. Schlussendlich haben wir beide Lösungsalgorithmen verglichen und kamen zum Schluss, dass der ILP Algorithmus konsistent besser ist als der FPT Algorithmus.

Contents

1. Introduction

Graph modification problems cover a wide variety of classical problems in computer science. The task for edge modification problems is to transform a graph G by inserting and deleting edges into a graph $G' \in \mathcal{G}$, for some graph class $\mathcal{G}$. Each modification has an associated cost and the challenge is to find a set of edits such that $G' \in \mathcal{G}$ and the total cost is minimal. Some graph classes can be characterized by *forbidden subgraphs* [dRo18].[1] For example, forests are exactly the graphs which do not have a cycle as an induced subgraph, and chordal graphs are the graphs which do not have cycles with 4 or more vertices as induced subgraphs. A graph is called $\mathcal{F}$-free if no induced subgraph is isomorphic to a graph from a (potentially infinite) set of forbidden subgraphs $\mathcal{F}$. In this thesis, we only consider finite sets of forbidden subgraphs.

$\mathcal{F}$-free Edge Editing can be used for *graph clustering*. Graph clustering, or community detection, is the task of finding a community structure in the data, where objects in the same group are similar, or the objects within the group interact with each other more often than compared to the remaining objects. Vertices in a graph can be grouped into clusters, with a high edge density within a cluster and a low edge density between clusters. The $\mathcal{F}$-free Edge Editing approach for clustering defines the clusters by the connected components of “nearest” $\mathcal{F}$-free graph. One notion for a “good” clustering can be *cluster graphs*, which are a disjoint union of *cliques*. Vertices within a clique are all adjacent to each other and no edges between cliques exist. Cluster graphs can be characterized by a forbidden induced subgraph, the path of length three. They are also called P_3-free graphs. Figure 1.1a depicts a P_3. Furthermore, Figure 1.2 is an example of a graph that can be edited into a cluster graph with two edits. Cluster Editing is one of the most studied special cases of $\mathcal{F}$-free edge editing and early on has been used for clustering of animals, workers and companies [GW89].

Another perspective on this problem is to assume that a graph conforms to a given model defined by its forbidden subgraphs but noise has corrupted the data. The task is to remove the noise from the input data, such that the editing cost is minimal. This has been used for biological data sets, such as protein-protein interaction networks. Proteins that take part in the same processes often do interact with each other. The functions of a given unknown protein can then be predicted by observing its interaction with another protein with known functions [LCH$^+$07]. Cluster Editing has been used for protein-protein

[1] A list of graph classes characterized by forbidden subgraphs: `http://graphclasses.org/classes.cgi?search=-free`

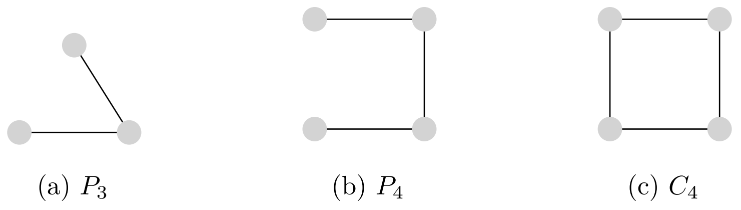
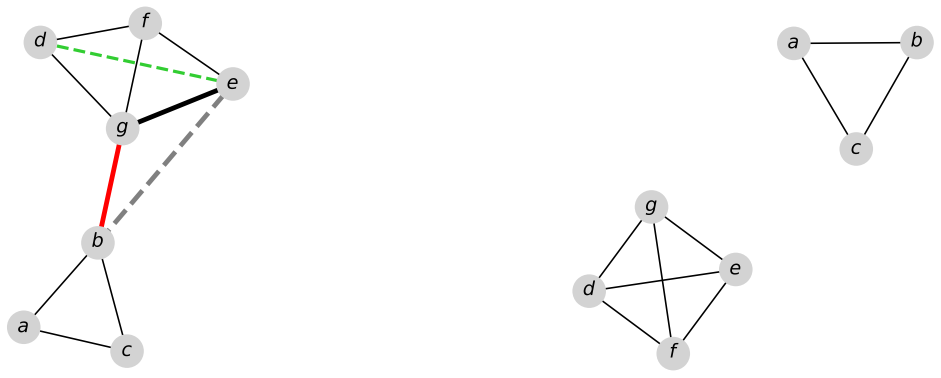


Figure 1.1: Forbidden subgraphs for CLUSTER EDITING (P_3) and QUASI-THRESHOLD EDITING (P_4 and C_4).



(a) The graph G. Vertices b, e are explicitly marked as non-adjacent. Green edges will be inserted in the graph. The edge bg is in multiple forbidden subgraphs. The induced path (b, g, e) is highlighted.

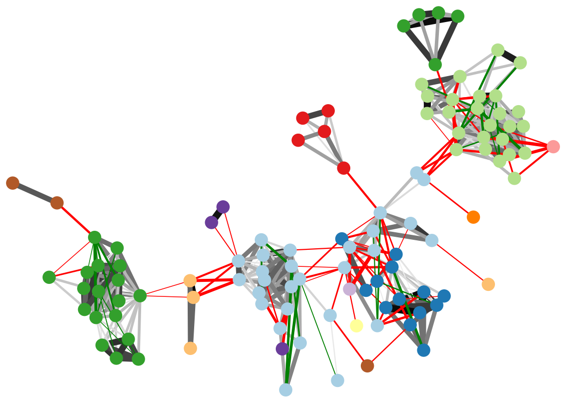
(b) The graph G' obtained by editing $\{bg, de\}$, i.e. $G' = G \triangle \{de, bg\}$. G' consists of two disjunct cliques.

Figure 1.2: Example for CLUSTER EDITING. Graph G is not a cluster graph. The vertices b, e and g induce a P_3. By deleting bg and inserting de, G can be transformed into the cluster graph G'.

interaction networks, but other models are also possible. The *core-periphery model* assumes that some proteins form a dense core and other proteins only interact with the core. *Split-cluster graphs* have been used to model such a structure and also can be characterized by forbidden subgraphs [BHK15]. Figure 1.3 depicts a connected component of a protein-protein interaction network and a closest $\{C_4, P_4\}$-free graph, meaning it has the minimum editing cost of all $\{C_4, P_4\}$-free graphs. C_4 and P_4 are the cycle and path of size four respectively.

Another application is the clustering of social networks. *Familial groups* are defined to be the connected components of a $\{C_4, P_4\}$-free graph with minimum edit distance to the original graph. They have been introduced as a method for community detection [NG13]. $\{C_4, P_4\}$-free graphs are also called *Quasi-Threshold graphs* or *trivially perfect graphs*. These graphs are useful for social networks, because they encode a hierarchy structure. The graphs are the transitive closure of rooted forests [Wol65]. Other models like $\{C_5, P_5\}$-free graphs [Sch15] and P_5-free graphs [Boh15] have also been used for clustering.

$\mathcal{F}$-free graphs have been used for several applications, but often only unweighted instances are considered. For some applications, we can define a function that assigns a similarity score $s(uv)$ to vertices u and v. A positive value represents similarity and a negative value dissimilarity. The magnitude of the (dis-)similarity represents how strong that relationship is. For unweighted instances, all similarity scores are either 1 or -1 and the cost of editing is equal to the number of edits. When considering WEIGHTED EDGE EDITING, not


(a) Graph G from instance no. 1010 from the bio dataset.

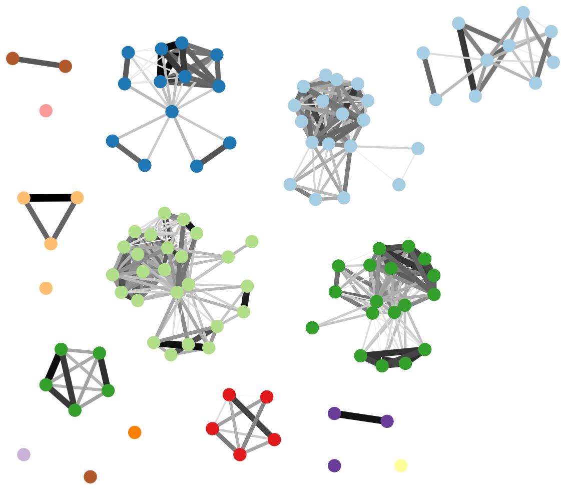

(b) A $\{C_4, P_4\}$-free graph with minimum editing cost to G.

Figure 1.3: A graph before and after editing. Editing costs are visualized by edge width and brightness. Non-edges and their costs are not considered in this visualization. Edge insertions/deletions are marked as green/red.

the number of edits, but the total cost of edits is minimized. For example, Weighted Cluster Editing, also called Correlation Clustering, is the weighted variant of P_3-free Edge Editing. In this thesis we discuss the Weighted $\mathcal{F}$-free Edge Editing Problem, generalizing both editing costs and the set of forbidden subgraphs, for finite sets $\mathcal{F}$.

Unfortunately, $\mathcal{F}$-free Edge Editing is $\mathcal{NP}$-hard for many useful forbidden subgraphs [KM86, ASS16, Boh15]. For finite $\mathcal{F}$, the problem is fixed-parameter tractable in the editing cost k [Cai96]. The problem can be solved by a search tree algorithm in $O(p^{2k} \cdot n^p)$ time, where k is the maximum editing cost, n is the number of vertices, p is the size of the largest graph in $\mathcal{F}$ and the editing costs are at least one for each vertex pair. Another method for solving the problem in practice is an *integer linear program* formulation (ILP) for Weighted $\mathcal{F}$-free Edge Editing [GW89].

The goal of this thesis is to adapt an existing search tree algorithm, originally developed for $\mathcal{F}$-free Edge Editing, to allow for weighted input instances [Zü17, GHS$^+$20]. We use lower bounds and selection rules as speed-up techniques and compare it against the ILP formulation on protein-protein interaction networks. A detailed overview of this bachelor thesis is given in Section 1.2.

1.1 Related Work

Whether the $\mathcal{F}$-FREE EDGE EDITING PROBLEM is $\mathcal{NP}$-hard, depends on the set of forbidden subgraphs $\mathcal{F}$. For example, P_2-FREE EDGE EDITING can trivially be done in linear time by deleting all edges. Furthermore, even more "complex" classes like *split graphs* ($\{2K_2, C_4, C_5\}$-free graphs) have a linear time editing algorithm.

In the case that $\mathcal{F}$ contains only a single graph H, the complexity has been completely determined for all graphs H: H-FREE EDGE EDITING is $\mathcal{NP}$-complete if and only if H is a graph with at least three vertices [ASS16]. For a single path and cycles of smaller sizes of at least 4, the following has been proven: $\{P_l, C_{l_1}, ...C_{l_i}\}$-FREE EDGE EDITING is $\mathcal{NP}$-hard for $l \geq 4$, $i \in \mathbb{N}_0$ and $4 \leq l_1 < ... < l_i \leq l$ [Sch15].

The most important result used by this thesis is the fixed-parameter tractability of $\mathcal{F}$-FREE EDGE EDITING for finite sets $\mathcal{F}$ [Cai96]. A problem is fixed-parameter tractable if there exists an algorithm with time complexity $O(f(k) \cdot |I|^c)$ where $f(k)$ is a computable function, k is a parameter of the problem instance, $|I|$ is the size of the instance and c is some fixed number. For WEIGHTED $\mathcal{F}$-FREE EDGE EDITING, the parameter k is the editing cost. The result in [Cai96] is based on a search tree with bounded branching factor and height. Each edit is symbolized by an edge in the search tree and the length of a path in the tree corresponds to the number of edits on the path. The branching factor is bounded by $\binom{p}{2}$, where p is the size of the largest forbidden subgraph. This stems from the observation that each forbidden subgraph needs at least one edit to be destroyed. Thus, it suffices to branch on the possible edits of a single forbidden subgraph. The size of the search tree is thus bounded by $\binom{p}{2}^k$, where k is the maximum number of edits allowed. An FPT algorithm traverses this search tree. For a run time discussion of the weighted case see Section 3.1.2.

The special case of P_3-FREE EDGE EDITING is called CLUSTER EDITING. The unweighted CLUSTER EDITING PROBLEM was proven to be $\mathcal{NP}$-complete multiple times [KM86]. Also, the problem can be formulated as an integer linear problem and was used to solve several real-world clustering tasks [GW89]. Search tree algorithms based on [Cai96] have been extensively used to solve the problem. Most theoretical advances of the running time came from elaborate branching rules. An automated search led to an algorithm with running time $O(1.92^k + n^3)$ and 137 initial cases [GGHN04]. The asymptotic running time of the best theoretical algorithm continued to improve. In [Bö12] it was proven that CLUSTER EDITING can be solved in $O(1.62^k + m + n)$. A linear problem kernel with at most $4k$ vertices has been developed for the unweighted case [Guo09]. Later $2k$ problem kernels for unweighted CLUSTER EDITING [CM12] and INTEGER-WEIGHTED CLUSTER EDITING [CC12] have been found.

QUASI-THRESHOLD EDITING or $\{C_4, P_4\}$-FREE EDGE EDITING is another $\mathcal{NP}$-hard unweighted editing problem [NG13]. Several heuristic solutions have been introduced in [NG13]. The authors of [BHSW15] introduced the Quasi-Threshold Mover as a scalable QUASI-THRESHOLD EDITING heuristic. In [Zü17] and [GHS+20] an exact FPT algorithm was engineered for unweighted $\mathcal{F}$-FREE EDGE EDITING, and evaluated for $\mathcal{F} = \{C_4, P_4\}$. A polynomial kernel of size $O(k^7)$ was presented in [DP18].

Several methods have been used for speeding up search tree based FPT algorithms for $\mathcal{F}$-FREE EDGE EDITING. Vertex disjoint packing, vertex pair disjoint packing, and a hitting set approximation have been investigated for calculating lower bounds [Boh15]. Another way to calculate lower bounds is to build and an instance of the HITTING SET PROBLEM and use an approximation algorithm to calculate lower bounds [Boh15]. The authors of [HH15] combine several approaches for CLUSTER EDITING. They use upper and lower bounds and extensive data reduction rules. A relaxation of the ILP formulation of the problem has been used as one of the lower bounds. Also, a "small-degree heuristic" for computing maximum-size independent set on a derived conflict graph has been used for lower bound computation. Vertex pair disjoint sets of forbidden subgraphs have been used multiple times to compute lower bounds for Cluster Editing [BBK11]. The authors of [GHS$^+$20] use an local search based lower bound algorithm, adapting a Maximum Independent Set heuristic [ARW12] for unweighted F-free Edge Editing.

1.2 Overview and Contributions

In this thesis, we discuss an FPT algorithm and an ILP formulation for exactly solving the Weighted $\mathcal{F}$-free Edge Editing Problem, present speed-up techniques and evaluate them for $\mathcal{F} = \{C_4, P_4\}$ on data from protein-protein interaction networks.

In Chapter 2 we start by presenting algorithms for listing forbidden induced subgraphs, concentrating on cycles and paths. We cover these algorithms because they are a core component for the editing algorithms in Chapter 3. The listing algorithms are based on different approaches. Later, in Chapter 4, we evaluate and compare the different algorithms with our benchmark dataset. Additionally, we describe symmetry breaking rules that guarantee that each induced subgraph is listed exactly once.

The algorithms for exactly solving the Weighted $\mathcal{F}$-free Edge Editing Problem are covered in Chapter 3. The FPT algorithm that we adapt from [Zü17] is the main focus of this thesis. We present the algorithm and motivate the speed-up methods in Section 3.1. We give a short proof of its theoretical running time and justify that every inclusion-minimal solution with editing cost less than k is listed exactly once. Also, we explain the rules for marking vertex pairs introduced by [Zü17] in Section 3.1.1.

Then we describe lower bound algorithms based on vertex pair disjoint subgraph packings in Section 3.2. We give a proof that they can be used for calculating lower bounds for Weighted $\mathcal{F}$-free Edge Editing and demonstrate a correspondence of the lower bound calculation with solving the Maximum Weight Independent Set Problem (MWIS). The lower bound algorithms are adapted heuristics for MWIS. We first introduce an algorithm used in [Zü17]. Then we adapt it to a greedy algorithm that considers editing costs. Furthermore, we adapt a local search algorithm for improving packings, that was used in [GHS$^+$20] for the unweighted case. The local search algorithm improves packings by removing and inserting subgraphs. We present two methods for such improvements and describe an efficient implementation. Next, we shortly describe subgraph selection strategies for branching used in [Zü17] and [GHS$^+$20] in Section 3.3.

A problem that can easily be solved for unweighted $\mathcal{F}$-free Edge Editing, but is non-trivial for the weighted case, is the search for the optimal editing cost. In Section 3.4, search strategies are covered. We present two possible adaptions of the unweighted case. Furthermore, we introduce two new search strategies and motivate them.

After we covered the FPT algorithm, we finally introduce an algorithm based on integer linear programming (ILP) in Section 3.5. We adapt an ILP formulation Cluster Editing from [GW89] for general sets of forbidden subgraphs. As the full set of constraints is too large, constraints are iteratively added to the linear program. One speed-up technique for ILP algorithms is restricting the number of constraints that are being added at once. We present two such methods for Weighted $\mathcal{F}$ Edge Editing.

Additionally, we evaluate all algorithms and speed-up techniques in Chapter 4. We start by describing the experimental setup and the protein-protein interaction dataset used in our evaluation. We compare the listing algorithms in Section 4.2. Next, we analyze the effects of the lower bound algorithms on the running time in Section 4.3. Also, we inspect the ability of the lower bounds to prune the search tree. Furthermore, we determine why the unweighted local search algorithm seems to be more successful for the unweighted case than for Weighted $\mathcal{F}$-free Edge Editing. In Section 4.4 we compare the effect of the subgraph selection strategies on the running and examine the effect of the strategies in combination with different lower bounds. Next, we analyze the search strategies and their behavior in respect to the search steps in Section 4.5. Finally, we evaluate the ILP algorithm and its variants with the FPT algorithm.

1.3 Preliminaries and Definitions

We use $\binom{A}{2} := \{\{a, b\} \mid a, b \in A \land a \neq b\}$ to denote the two element subsets of the set A.

A *graph* $G = (V, E)$ is a set of *vertices* $V(G) := V$ and a set of *edges* $E(G) := E$, $E \subseteq \binom{V}{2}$. An *edge* $e = \{u, v\}$ is a pair of vertices $u, v \in V$. We use $N(u) := \{v \in V \mid \{u, v\} \in E\}$ to denote the *neighbors* and $\overline{N}(u) := \{v \in V \mid u \neq v, \{u, v\} \notin E\}$ to denote the set of *non-neighbors* of a vertex $u \in V$. $d(u) := |N(u)|$ is the *degree* of vertex u. We use $\triangle$ to denote the maximum degree in the graph. If an edge is explicitly not in the graph, we call it a *non-edge*. We use the shorthand uv for the undirected pair of vertices $\{u, v\}$, $u \neq v$. In this thesis we consider simple undirected graphs.

A graph $H = (V_H, E_H)$ is called a *subgraph* of $G = (V, E)$ if $V_H \subseteq V$ and $E_H \subseteq E$. A subgraph $H = (V_H, E_H)$ is called an *induced subgraph* if H contains all possible edges, i.e. all pairs of vertices of V_H that are edges in G. An induced subgraph can be identified by its vertex set S. We use $G[S] := (S, \binom{S}{2} \cap E)$ to denote the induced subgraph with vertex set S. In this thesis we only consider induced subgraphs.

We use P_l to denote the graph with l vertices consisting of a simple path, $P_l = (\{v_1, \ldots, v_l\}, \{v_i v_{i+1} \mid i = 1 \ldots l{-}1\})$. Likewise, C_l is used to denote the graph with l vertices consisting of a single cycle, $C_l = (\{v_1, \ldots, v_l\}, \{v_i v_{i+1} \mid i = 1 \ldots l{-}1\} \cup \{v_l v_1\})$. Sometimes, a list of vertices $(v_1, \ldots, v_l) \in V^l$ is used to denote an induced subgraph $G[S]$ which is isomorphic to a P_l or C_l. The vertices are ordered, such that two vertices are only adjacent if they are next to each other in the list. We define the concatenation $a \cdot b := (a_1, \ldots, a_{l_1}, b_1, \ldots, b_{l_2})$ of two paths $a = (a_1, \ldots, a_{l_1})$, $b = (b_1, \ldots, b_{l_2})$, $l_1, l_2 \in \mathbb{N}$ if the result $a \cdot b$ is either a path or a cycle.

An *edit* on a graph G is either the insertion or deletion of an edge. We use $A \triangle B := (A \setminus B) \cup (B \setminus A)$ to denote the symmetric difference of sets A and B. For a graph $G = (V, E)$ and a set of vertex pairs L we use the notation $G \triangle L := (V, E \triangle L)$ for the graph we get by editing the vertex pairs from L. We say that an edit uv *destroys* a forbidden subgraph $G[S]$ if both vertices are in the subgraph. For the FPT algorithm, we use the concept of marked vertex pairs. For a graph $G = (V, E)$ and a set of marked vertex pairs $M \subseteq \binom{V}{2}$, we say that two induced subgraphs, $G[S_1]$ and $G[S_2]$, are *adjacent* if they share an unmarked vertex pair, i.e. $\mathrm{adj}_M(S_1, S_2) := \exists uv \in \binom{V}{2} : u, v \in S_1 \land u, v \in S_2 \land uv \notin M$. We consider the editing cost function $c : \binom{V}{2} \to \mathbb{R}_{\geq 0}$. For a set of vertex pairs L we define $c(L) := \sum_{e \in L} c(e)$ to be the total editing cost of the set L.

Two graphs are *isomorphic* if one can be obtained from the other by renaming the vertices. A graph G is called H-free for a graph H if no induced subgraph exists which is isomorphic to H. A graph G is called $\mathcal{F}$-free for a set of graphs $\mathcal{F}$ if G is H-free for all $H \in \mathcal{F}$. We use $\mathcal{C}_H(G) \subseteq 2^V$ to denote all vertex sets $S \subseteq V$ such that $G[S]$ is isomorphic to H. $\mathcal{C}_\mathcal{F}(G) := \bigcup_{H \in \mathcal{F}} \mathcal{C}_H(G)$ is used for multiple forbidden subgraphs. $\mathcal{C}_\mathcal{F}(G)$ is the *conflict set* of G. As the set $\mathcal{F}$ is considered to be constant throughout this thesis, we do not denote the set of forbidden subgraphs explicitly and use $\mathcal{C}(G) := \mathcal{C}_\mathcal{F}(G)$. The functions $p_l(G)$ and $c_l(G)$ denote the number of paths and cycles with size l in the graph G respectively.

We now formally define the central problem of this thesis.

Weighted $\mathcal{F}$-free Edge Editing (Search) Problem	
Input:	A graph $G = (V, E)$, a set of graphs $\mathcal{F}$ and a cost function $c : \binom{V}{2} \to \mathbb{R}_{\geq 0}$.
Question:	Find a set of edits $L \subseteq \binom{V}{2}$ such that $G \triangle L$ is $\mathcal{F}$-free and $\sum_{e \in L} c(e)$ is globally minimal.

We call a set of vertex pairs $L \subseteq \binom{V}{2}$ a *solution* for the graph G if $G \triangle L$ is $\mathcal{F}$-free. A solution L^* is *optimal* if its cost $c(L^*)$ is globally minimal. A solution is *inclusion-minimal* if no proper subset is a solution.

2. Listing Forbidden Subgraphs

In this chapter, we present different algorithms for listing forbidden subgraphs. A naive listing algorithm is not good enough, because it scales with the size of the graph and not with the number of forbidden induced subgraphs. An ideal algorithm has a running time only proportional to the size of its output. We present algorithms that aim to do the least amount of extra work. Listing algorithms are an important procedure for solving the Weighted $\mathcal{F}$-free Edge Editing Problem. The algorithms for solving the problem use these listing algorithms for finding subgraphs for branching, calculating lower bounds, or to generate constraints.

In this work, we focus on listing the subgraphs for Cluster- and Quasi-Threshold Editing, and cycles and paths. For specific $\mathcal{F}$, algorithms exist that can show that a graph is $\mathcal{F}$-free, or otherwise provide a counterexample in linear time. For example, there exists an algorithm to determine whether a graph is a split-cluster graph and if that is not the case, produce a forbidden subgraph in $O(n + m)$ time [BHK15]. We only consider listing all forbidden subgraphs, because this is predominantly needed for lower bounds in Section 3.2, subgraph selection in Section 3.3 and constraint generation in Section 3.5.

In this chapter, we use n and m to denote the number of vertices and edges respectively. Recall that $\triangle$ denotes the maximum degree in the graph. Furthermore, recall that $p_l(G)$ and $c_l(G)$ are the numbers of induced subgraphs of G isomorphic to P_l and C_l respectively.

For this chapter, we assume that the graph is represented with adjacency lists. The vertices in an adjacency list are sorted. The operations $N(u) \cap N(v)$ and $N(u) \setminus N(v)$ are possible in $O(\triangle)$ time. Some algorithms also need to check whether two vertices are adjacent. For that, we additionally assume an adjacency matrix. In our implementation, we use an adjacency matrix to represent graphs.

2.1 Listing P_3

Even for listing relatively simple subgraphs such as P_3s, several strategies exist. Apart from the naive approach, we investigate two algorithms.

Algorithm 2.1: Edge expanding. Listing P_3s.

```
1 foreach $u \in V, v \in N(u)$ do
2     foreach $w \in N(v) \setminus N(u), w \neq u$ do
3         output $(u, v, w)$
```

Algorithm 2.1 iterates over all adjacent vertices u and v and expands them with an extra vertex w. The running time of the algorithm is clearly in $O(m \cdot \triangle)$. Moreover, this is algorithms is a starting point for the algorithms discussed in the next sections.

Algorithm 2.2: Listing P_3s from outer vertices [HKSS13]

```
1 remove isolated vertices from $G$
2 foreach $u \in V, w \in \overline{N}(x)$ do
3     foreach $v \in N(u) \cap N(w)$ do
4         output $(u, v, w)$
```

Algorithm 2.2 takes another approach and start with the non-adjacent outer most vertices. Consequently, every iteration of the inner loop produces one P_3. It has a running time of $O(n + m + p_3(G) + co{-}p_3(G))$ [HKSS13].

2.2 Listing C_4s and P_4s

Before we talk about listing cycles and paths of any length, we present an algorithm for listing cycles and paths of length 4. These are the forbidden subgraphs for Quasi-Threshold Editing. The edge expansion strategy of Algorithm 2.1 can be adapted by also expanding with an edge in the other direction. This results in Algorithm 2.3. A very similar algorithm has been presented for listing P_4s [HKSS13].

Algorithm 2.3: Listing $\{C_4, P_4\}$

```
1 foreach $u \in V, v \in N(u)$ do
2     $A \leftarrow N(u) \setminus N(v)$
3     $B \leftarrow N(v) \setminus N(u)$
4     foreach $(a, b) \in A \times B$ do
5         if $ab \in E$ then
6             output $(a, u, v, b)$      // $C_4$
7         else
8             output $(a, u, v, b)$      // $P_4$
```

The algorithm is efficient in the sense that every iteration of the inner loop outputs either a C_4 or a P_4. Calculating the sets A and B can be done in $O(\triangle)$ time. This results in a total run time of $O(m \cdot \triangle + p_4(G) + c_4(G))$.

2.3 Listing Cycles and Paths

Destroying cycles and paths is a prominent special case for $\mathcal{F}$-free editing. Cluster graphs and quasi-threshold graphs do have P_3 and $\{C_4, P_4\}$ as forbidden subgraphs respectively. Editing to P_5-free and $\{C_5, P_5\}$-free graphs has been investigated as a method for graph clustering and community detection [Boh15, Sch15]. Algorithms for efficiently listing cycles and paths are needed for these problems.

The naive approach would be to enumerate all l-element subsets of vertices $\binom{V}{l}$ and check for each if it induces a path or cycle of length l. This takes $O(l^2 \cdot n^l)$ time and is unfeasible for larger graphs. We instead try to avoid unnecessary work and only calculate intermediate results if they may lead to a cycle or path of the desired length. The two algorithms we are going to present recursively expand smaller paths.

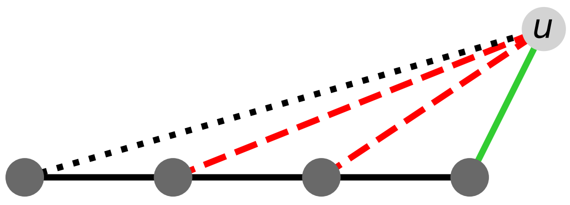


Figure 2.1: Endpoint search expanding a P_4. Candidate vertices v must have the pictured adjacency relations. Green relations must be an edge, red must be a non-edge and for dotted both are admissible. Whether the dotted vertex pair is an edge determines if the induced subgraph is a path or a cycle.

2.3.1 Endpoint Search

The endpoint search algorithm tries to expand an existing path of length $l-1$ by adding one vertex. In [Boh15] an endpoint algorithm for listing P_5s has been used. We present a generalization for cycles and paths of arbitrary size and give a theoretical running time analysis. Algorithm 2.4 iterates over all potential new endpoints $u \in V$, vertices which are adjacent to the current endpoint p_{l-1} and non-adjacent to all inner vertices of the path. The resulting subgraph $P \cdot (u)$ is either a cycle or a path. The algorithm recursively calls a variant of itself for $l-1$, that does not output cycles, and uses paths of size one, i.e. (u) for all $u \in V$, as its base case for $l = 1$.

Figure 2.1 depicts the expansion of a P_4 to a P_5 or C_5 with the new vertex u. The restrictions on the adjacency between u and the other vertices are depicted with colored edges. If the dotted relation between the first vertex of the path and the vertex u is an edge, the resulting induced subgraph is a cycle, otherwise, it is a path.

Algorithm 2.4: Endpoint search. Listing all $\{C_l, P_l\}$.

```
   Input: List of paths $L \subseteq V^{l-1}$.
   Output: Paths and cycles and of size $l$.
1  foreach $P := (p_1, p_2, \dots, p_{l-1}) \in L$ do
2      $U \leftarrow N(p_{l-1})$
3      foreach $p \in \{p_2, \dots, p_{l-2}\}$ do
4          $U \leftarrow U \setminus N(p)$
5      foreach $u \in U$ do
6          if $p_1 u \in E$ then
7              output $P \cdot (u)$      // $C_l$
8          else
9              output $P \cdot (u)$      // $P_l$
```

Theorem 2.1. *Algorithm 2.4 finds all C_l and P_l in $O(n \cdot \triangle^{l-1} \cdot l!)$ time.*

Proof. This can be proven by induction over l. Let $t(l)$ be the time that the algorithm takes to list all cycles and paths of length l. For the base case $l = 2$ the algorithm iterates over $\{(a, b) \mid a \in V, b \in N(a)\}$, therefore $t(2) \in O(n \cdot \triangle)$. Listing all paths of length $l-1$ costs $t(l-1)$ and $t(l-1)$ is also an upper bound for the number of P_{l-1}s. In the loop body $O(\triangle \cdot l)$ additional work is performed. The recurrence relation $t(l) = t(l-1) \cdot \triangle \cdot l$ results in $t(l) \in O(n \cdot \triangle^{l-1} \cdot l!)$. □

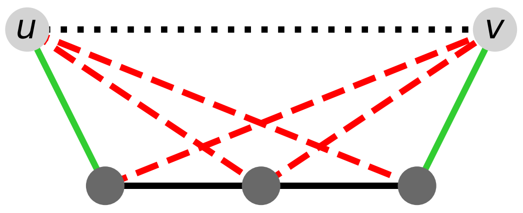


Figure 2.2: Midpoint search expanding a P_3. Candidate vertices for u and v must have the pictured adjacency relations. Green relations must be an edge, red must be a non-edge, and for the dotted relation both are admissible. Whether the dotted vertex pair is an edge determines if the induced subgraph is a path or a cycle.

2.3.2 Midpoint Search

Instead of expanding the path in only one direction, we can also start from the middle and expand the path simultaneously in both directions. The algorithm is discussed in [HKSS13]. Also, in [Boh15] a midpoint based algorithm has been used for listing all P_5s of a graph and finding a single P_5. In contrast to the authors of [HKSS13], they first expand one half the path in one direction from the middle vertex and then expand the rest of the path in the other direction.

Similar to Algorithm 2.4, the midpoint search algorithm is defined recursively. Algorithm 2.5 expands paths of length $l-2$ for $l \geq 4$ and Algorithm 2.6 expands paths of length $l-4$ for $l \geq 6$. The algorithm calls a variant of itself for $l-2$ or $l-4$ that only outputs paths. The base cases are paths of length 2 and 3. After an initial evaluation of P_3 listing algorithms as seen in Section 4.2, we chose Algorithm 2.2 as the P_3 base case.

An example for Algorithm 2.5 is given in Figure 2.2. A P_3 is expanded in both directions. If uv is a non-edge, the subgraph is an induced path, otherwise, it is an induced cycle.

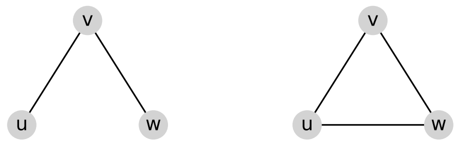
Algorithm 2.5: Midpoint search. Listing all $\{C_l, P_l\}$. Recursion on P_{l-2}. [HKSS13]

```
  Input: List of paths $L \subseteq V^{l-2}$, $l \geq 4$.
  Output: Paths and cycles and of size $l$.
1 foreach $P = (p_1, p_2, \ldots, p_{l-2}) \in L$ do
2     $A \leftarrow N(p_1) \setminus \bigcup_{p \in P \setminus \{p_1\}} N(p)$
3     $B \leftarrow N(p_{l-2}) \setminus \bigcup_{p \in P \setminus \{p_{l-2}\}} N(p)$
4     foreach $(a, b) \in A \times B$ do
5         if $ab \in E$ then
6             output $(a) \cdot P \cdot (b)$      // $C_l$
7         else
8             output $(a) \cdot P \cdot (b)$      // $P_l$
```

The sets of candidates A and B are chosen such that each iteration of the loop in Line 4 outputs either a path or a cycle. Only using Algorithm 2.5 in the recursion, the whole algorithm has a running time of $O(ln^{l-1} + p_l(G) + l \cdot c_l(G))$ ($l \geq 4$) [HKSS13]. This can be improved by starting with paths of size $l-4$ in Algorithm 2.6. This removes the factor of l and the algorithm has a total running time of $O(n^{l-1} + p_l(G) + l \cdot c_l(G))$ ($l \geq 4$) (Theorem 9 in [HKSS13]).

Algorithm 2.6: Midpoint search. Listing all $\{C_l, P_l\}$. Recursion on P_{l-4}. [HKSS13]

```
Input: List of paths $L \subseteq V^{l-4}$, $l \geq 6$.
Output: Paths and cycles of size $l$.
1  foreach $P = (p_1, p_2, \dots, p_{l-4}) \in L$ do
2      $A \leftarrow N(p_1) \setminus \bigcup_{p \in P \setminus \{p_1\}} N(p)$
3      $B \leftarrow N(p_{l-4}) \setminus \bigcup_{p \in P \setminus \{p_{l-4}\}} N(p)$
4      $C \leftarrow V \setminus \bigcup_{p \in P} N(p)$                // vertices non-adjacent to $P$
5      foreach $(a, b) \in A \times B$ do
6          if $ab \notin E$ then
7              $A' \leftarrow C \cap N(a) \setminus N(b)$
8              $B' \leftarrow C \cap N(b) \setminus N(a)$
9              foreach $(u, v) \in A' \times B'$ do
10                 if $uv \in E$ then
11                     output $(u, a) \cdot P \cdot (b, v)$      // $C_l$
12                 else
13                     output $(u, a) \cdot P \cdot (b, v)$      // $P_l$
```



(1, 2, 3), (2, 3, 1), (3, 1, 2),
(3, 2, 1), (1, 3, 2), (2, 1, 3)

Figure 2.3: Symmetries for three ordered elements. Cyclic symmetry (horizontal) and mirror symmetry (vertical). For example, the path (u, v, w) can either be the list $(1, 2, 3)$ or $(3, 2, 1)$).

2.4 Symmetry Breaking

The listing algorithms produce the same induced subgraph multiple times. A path is listed twice and a cycle of size l is listed $2l$ times. A path $(p_1, \dots, p_l)$ is also listed in reverse order $(p_l, \dots, p_1)$. A cycle $(c_1, \dots, c_l)$ is also listed in reverse order $(c_l, \dots, c_1)$ and with each vertex c_i as "first" vertex $(c_1, \dots), \dots, (c_l, \dots)$. For the editing algorithm we want to list each path and cycle *exactly* once.

We assume some arbitrary order on the vertices, e.g. $v_1 < v_2 < \dots < v_n$. The cyclic symmetry of cycles can be broken by only listing cycles $C = (c_1, c_2, \dots, c_l)$, where c_1 is the smallest vertex in C. The mirror symmetry is broken by additionally ensuring that $c_2 < c_l$ [HKSS13]. Algorithms for listing cycles can use this rules for only constructing representative cycles. For example, an algorithm for listing all C_ls in a graph in $O(n^{l-1} + p_l(G) + c_l(G))$ time is presented in [HKSS13]. The mirror symmetry of paths can be broken by only listing paths $P = (p_1, p_2, \dots, p_l)$ for which $p_1 < p_l$.

Figure 2.3 depicts both cyclic and mirror symmetries of $(1, 2, 3)$. Note that for a path (u, v, w) uw is a non-edge and uv and vw are edges. For a cycle (u, v, w) all uv, vw, wu are edges. Using the described rules, representative paths are $(1, 2, 3)$, $(1, 3, 2)$ and $(2, 1, 3)$. The representative cycle is $(1, 2, 3)$.

3. Algorithms

In this section, we present exact algorithms for the WEIGHTED $\mathcal{F}$-FREE EDGE EDITING PROBLEM. First, we describe a generic version of the FPT algorithm and introducing marked vertex pairs in Section 3.1. Next, we describe specific algorithms for calculating lower bounds in Section 3.2. We list different methods for selecting a subgraph for branching Section 3.3 and for strategies for finding the optimal editing cost in Section 3.4. Finally, we present an integer linear program (ILP) algorithm for solving the WEIGHTED $\mathcal{F}$-FREE EDGE EDITING PROBLEM in Section 3.5.

Throughout this work, we consider the set of forbidden subgraphs $\mathcal{F}$ and the edit cost function $c : \binom{V}{2} \to \mathbb{R}_{\geq 0}$ to be fixed.

3.1 FPT Algorithm

The FPT algorithm is a search tree algorithm based on [Cai96]. The parameter for WEIGHTED $\mathcal{F}$-FREE EDGE EDITING is the maximum allowed editing cost k. The algorithm only searches for solutions with editing costs at most k. The central idea of the algorithm given in [Cai96] is that, if the graph G has a forbidden induced subgraph $G[S]$, at least one edit $e \in \binom{S}{2}$ is needed to destroy it. If a forbidden induced subgraph exists, the algorithm branches for each possible edit. For each vertex pair uv the algorithm recursively solves the instance $(G \triangle \{uv\}, k - c(uv))$. If the remaining editing cost is negative, the instance is not solvable. When the graph is $\mathcal{F}$-free, a solution has been found.

This basic version of the algorithm has a running time in $O(\binom{p}{2}^k \cdot poly(n)) = O(p^{2k} \cdot poly(n))$. This uses the assumption that the minimum editing cost is at least one. The running time can be analyzed by looking at the branching factor and the maximum depth of the search tree. Each call to the algorithm is a node in the search tree. The initial instance (G, k) corresponds to the root of the search tree. For each call, either the algorithm finds a solution or it branches on the possible edits of a forbidden induced subgraph. Such a subgraph can be found in $O(n^p p^2)$ time. Because a subgraph with p vertices has $\binom{p}{2}$ vertex pairs, the search tree has a branching factor of $\binom{p}{2}$. The depth of the search tree is bounded by $\lfloor k \rfloor + 1$ if the minimum editing cost is at least 1.

The focus of this thesis is an improved FPT algorithm proposed in [Zü17] for unweighted $\mathcal{F}$-FREE EDGE EDITING. We adapt it for the weighted case. It is based on the same idea as the basic FPT algorithm but uses marked vertex pairs, lower bounds, and subgraph selection strategies. In the following, we will introduce these concepts and describe the improved FPT algorithm (Algorithm 3.1). For a correctness proof of Algorithm 3.1 we refer to [Zü17] where a proof for the unweighted case is given. This proof can be easily adapted to allow for weighted edits.

Algorithm 3.1: FPT Algorithm (adapted from [Zü17])

```
Input: A graph $G = (V, E)$, a maximum editing cost $k \in \mathbb{R}$, the set of marked
       vertex pairs $M \subseteq \binom{V}{2}$, and the set of currently edited vertex pairs $L$.
Output: Solutions to the Weighted $\mathcal{F}$-free Edge Editing Problem with
        total editing costs of at most $k$.
1  if $k <$ LowerBound$(G, k, M)$ then return
2  $S \leftarrow$ FindSubgraph$(G, M)$
3  if $S = \emptyset$ then
4      output $L$                                          // $G$ is $\mathcal{F}$-free
5      return
6  $m \leftarrow \emptyset$                                 // locally marked vertex pairs
7  forall $e \in \binom{S}{2}$ do
8      if $e \notin M$ then
9          $M \leftarrow M \cup \{e\}$, $m \leftarrow m \cup \{e\}$, $L \leftarrow L \cup \{e\}$
10         $G \leftarrow G \triangle \{e\}$
11         Edit$(G, k - c(e), M, L)$                        // recursive call
12         $G \leftarrow G \triangle \{e\}$
13         $L \leftarrow L \setminus \{e\}$
14 $M \leftarrow M \setminus m$                                // unmark vertex pairs
```

One problem of the basic algorithm is that it branches for every possible edit. As a result, a vertex pair can be edited twice or solutions are explored multiple times. In [Zü17] the authors introduce a rule for marking vertex pairs for some branches in the search tree. Algorithm 3.1 keeps a set of marked vertex pairs M and only edits unmarked vertex pairs. This rule is constructed in such a way that every set of edits is explored at most once and every inclusion-minimal solution output by the basic FPT algorithm is explored at least once. We describe the rules for marking vertex pairs in more detail in Section 3.1.1.

The basic FPT algorithm recursively calls itself until either a solution has been found or the remaining editing cost is negative. This can be improved by using lower bounds for the editing cost. If the remaining editing cost is smaller than the lower bound, we can guarantee that the problem is not solvable for the given instance and prune this branch, instead of fully exploring it. Good lower bounds can keep the size of the search tree small. In Section 3.2 we present algorithms for calculating lower bounds.

Another way to improve the basic algorithm is not using any forbidden induced subgraph for branching, but choosing one which most likely leads to a solution, keeps the search tree small or improves the calculation of lower bounds. We present strategies for selecting subgraphs for branching in Section 3.3.

Algorithm 3.1 does not search for an optimal solution to the Weighted $\mathcal{F}$-free Edge Editing Problem directly, but requires a maximum editing cost k as a parameter. To find an optimal solution, the algorithm has to be called for increasing values of k. This is non-trivial for the weighted case and we present different strategies for searching for the optimal editing cost in Section 3.4. Algorithm 3.1 also does not output *all* solutions with editing costs $\leq k$. For example, the algorithm does not add any additional edits to an already found solution. Inclusion-minimal solutions are guaranteed to be found, i.e. solutions for which no subset of edits is a solution. Nevertheless, non-inclusion-minimal

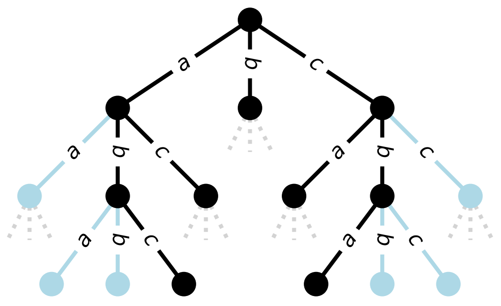


Figure 3.1: Visualization of Rule 1. The base is a three-ary tree with all tuples with up to three elements as nodes. Not all nodes are shown. Marked nodes are colored. Every tuple of distinct elements corresponds to one unmarked node.

solutions may also be output. We use a post-processing step on the set of solutions output by Algorithm 3.1 to ensure that only inclusion-minimal solutions remain.

We try to establish an intuition that all inclusion-minimal solutions with editing cost less than k are listed, but do not give a formal proof. When the algorithm explores a solution it does not recurse further, therefore not all non-inclusion-minimal solutions are guaranteed to be found. The basic algorithm recurses until either the current k is too small, or a solution is found. Inclusion-minimal solutions are only not outputted if their editing cost is larger than k. Now we argue that marking vertex pairs does not affect inclusion-minimal solutions by noting the following: a non-inclusion-minimal solution L can not block an inclusion-minimal solution L', with $L' \subsetneq L$, by marking a vertex pair. The marking rules, further described in Section 3.1.1, only affect child and sibling nodes in the search tree. The paths of the two solutions in the search tree only diverge at an edit which is not an element of the inclusion-minimal solution. Therefore the inclusion-minimal solution cannot be blocked by a non-inclusion-minimal solution.

Before we take a short look at the theoretical running time of Algorithm 3.1 in Section 3.1.2, we discuss marking vertex pairs.

3.1.1 Marking Vertex Pairs

To enumerate all solutions without visiting a solution twice, vertex pairs can be *marked* to be excluded from being edited further in the search tree [RWB$^+$07, Dam10, Zü17]. In [Dam10] marked vertex pairs have been used to enumerate all inclusion-minimal solutions for Cluster Editing. Marked vertex pairs can also be used to improve lower bound calculations (see Section 3.2). In this section, we describe the methods used in [Zü17] for $\mathcal{F}$-free Edge Editing. See Theorem 2.6 in [Zü17] for their proof of correctness. If a vertex pair is marked, it is not edited by Algorithm 3.1.

We observe that undoing an edit is useless because it is equivalent to not doing the edit at all. This results in the following rule, which ensures that an edit appears at most once in each path of the search tree.

Rule 1 (No Undo). *For branching options $e_1 < \cdots < e_l$, $l \in \mathbb{N}$, and the current search tree path P, for each edit e_i mark it for path $P \cdot (e_i)$.*

Rule 1 can be made stronger by using the fact that the order of edits in a solution does not matter. For a set of edits $\{a, b, c\}$, both paths (a, b, c) and (a, c, b) lead to the same set, $\{a, b, c\}$. All solutions with the edits $\{a, b\}$ can be reached from the path $(a, b, \dots)$,

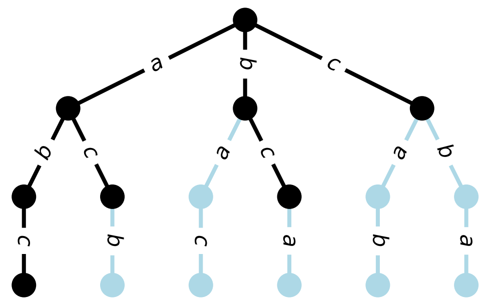


Figure 3.2: Visualization of Rule 2 adapted from [GHS+20]. The base is a tree conforming to Rule 1. Marked nodes are colored. Every subset of $\{a, b, c\}$ corresponds to one unmarked node.

therefore b can be marked for all paths $(a, x, \dots)$, $x \geq b$ and some fixed total ordering on the edits, e.g. $a < b < c$.

Rule 2 (No Redundancy). *For branching options $e_1 < \cdots < e_l$, $l \in \mathbb{N}$, and the current search tree path P, for each edit e_i mark it for paths $P \cdot (e_j)$, $j \geq i$.*

Figure 3.2 visualizes the rule for three elements, a, b and c. Each subset of $\{a, b, c\}$ is represented by exactly one unmarked node in the tree. The empty set $\emptyset$ is represented by the root. A search tree with n^n nodes is reduced by applying Rule 2 to a tree with 2^n nodes.

3.1.2 Running Time

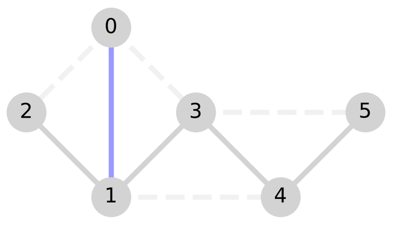
The asymptotic running time of Algorithm 3.1 is still $O(\binom{p}{2}^k \cdot poly(n)) = O(p^{2k} \cdot poly(n))$ for minimum cost of at least 1 and where p is the size of the largest forbidden subgraph. But with Rule 2 we establish another upper bound on the running time. As previously discussed, the search tree with marking vertex pairs with Rule 2 has at most $2^{\binom{n}{2}}$ nodes. Therefore the running time is bounded by $O(2^{\binom{n}{2}} \cdot poly(n, p))$, independently of the parameter k and only with a polynomial factor of p.

This running time analysis does not cover vertex pairs with zero editing costs. Whenever a zero cost pair is edited, the maximum allowed editing cost does not decrease. The algorithm still returns the correct result, but its running time is no longer fixed-parameter tractable in the parameter k. Let n_0 be the number of zero-cost edits in the instance. For any set of edits, there are at most 2^{n_0} additional potential solutions with the same set of non-zero-cost edits and every combination of edge/non-edge zero-cost edits Thanks to marking vertex pairs, the algorithm only outputs each of these solutions at most once. Therefore, Algorithm 3.1 is fixed-parameter tractable in the parameters editing costs k and number of zero-cost edits n_0.

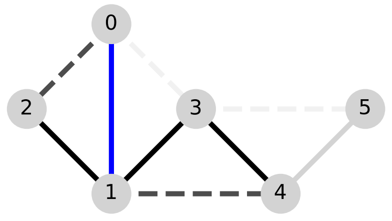
3.2 Lower Bounds

In this section, we discuss algorithms for calculating lower bounds for the Weighted $\mathcal{F}$-free Edge Editing Problem. A lower bound is a real number $k_{lb} \geq 0$ which bounds the actual optimal editing cost from below, i.e. $k_{lb} \leq k^*$ for optimal editing cost k^*.

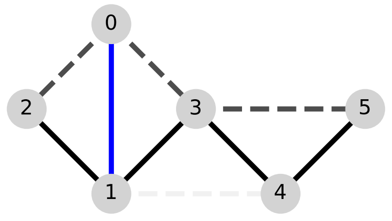
There are several approaches for calculating lower bounds. Algorithms that use subgraph packings try to find a set of forbidden induced subgraphs, for which destroying one subgraph



(a) The empty subgraph packing $\emptyset$.



(b) A subgraph packing $\{\{0,1,2\},\{1,3,4\}\}$. Both subgraphs do share a vertex but not a vertex pair.



(c) A subgraph packing $\{\{0,1,2\},\{0,1,3\},\{3,4,5\}\}$, using marked vertex pairs from Definition 3.1, allowing two subgraphs to share the marked edge $\{0,1\}$.

Figure 3.3: Three subgraph packings for the graph G, forbidden subgraph P_3 and marked vertex pairs $M = \{\{0,1\}\}$. Edges are solid lines and explicit non-edges are dashed lines. The marked edge is colored blue. The induced subgraphs in a subgraph packing are depicted with black vertex pairs. Editing costs are not considered.

in the packing does not affect any other subgraph in the packing. At least every forbidden induced subgraphs in such a set needs to be destroyed. The sum of the minimum editing cost needed for each one is a lower bound on the actual editing cost needed. In this thesis, we only use vertex pair disjoint packings. We adapt algorithms from [Zü17] and [GHS$^+$20] and generalize them to the weighted case. We now define the *vertex pair disjoint packing* formally. We additionally use the set of marked vertex pairs because they cannot be edited by Algorithm 3.1. Recall that $\mathcal{C}(G)$ is the set of forbidden induced subgraphs in G.

Definition 3.1. *A vertex pair disjoint packing of forbidden subgraphs of a graph G and a set of marked vertex pairs M is a set of forbidden subgraphs $B = \{S_1, S_2, \cdots\} \subseteq \mathcal{C}(G)$, such that no two $S_1, S_2 \in B$ share an unmarked vertex pair.*

The subgraph packing can be used to calculate lower bounds because two forbidden subgraphs that do not share an editable vertex pair do need at least two edits to be destroyed. A marked vertex pair can not be edited again and therefore not destroy incident subgraphs. Figure 3.3 depicts an example of a valid subgraph packing. We formally prove the lower bound with the following theorem.

Theorem 3.2. *For an instance of* Weighted $\mathcal{F}$-free Edge Editing (G, c)*, a subgraph packing $B \subseteq \mathcal{C}(G)$ and a set of marked vertex pairs $M \subseteq \binom{V}{2}$, a lower bound for the optimal editing cost k^* for solutions with no edits in M is*

$$c(B) = \sum_{S \in B} c_{\min}(S, M), \tag{3.1}$$

where

$$c_{\min}(S, M) = \begin{cases} \min\{c(e) \mid e \in \binom{S}{2} \setminus M\} & \text{if } \binom{S}{2} \setminus M \neq \emptyset \\ \infty & \text{otherwise} \end{cases} \tag{3.2}$$

is the minimal editing cost of the unmarked vertex pairs of the induced subgraph $G[S]$.

Proof. An optimal solution L^* must destroy each forbidden subgraph $G[S] \in \mathcal{C}(G)$ in G. A single edit $e \in L^*$ can destroy multiple subgraphs but at most one subgraph in B. To destroy a subgraph $G[S]$ at least one vertex pair needs to be edited. The cost to destroy a subgraph $G[S]$ is at least the cost it takes to edit the "cheapest" editable (unmarked) pair of vertices $\min_{e \in \binom{S}{2} \setminus M} c(e)$. Algorithm 3.1 does not edit marked vertex pairs. If all vertex pairs are marked, the instance is not solvable. □

A simple sanity check verifies that the trivial subgraph packing $B = \emptyset$ induces the lower bound $c(B) = 0$. This is a baseline for our evaluation of the other lower bound algorithms in Section 4.3. We now motivate the use of heuristics for the Maximum Weight Independent Set Problem (MWIS) as algorithms for calculating good lower bounds. The MWIS Problem is defined as follows.

Maximum Weight Independent Set Problem (MWIS)

Input: A graph $G = (V, E)$ and a weight function $w : V \to \mathbb{R}_{\geq 0}$.

Question: Find a set of vertices $I \subseteq V$ which maximizes $w(I) := \sum_{v \in I} w(v)$ and $uv \notin E$ holds for all $u, v \in I$.

Given this definition, we now give a correspondence between finding a subgraph packing and calculating an independent set.

Theorem 3.3. *Searching for a subgraph packing $B \subseteq \mathcal{C}(G)$ with maximum cost $c(B) = \sum_{S \in B} c_{\min}(S, M)$, given some set of marked vertex pairs M, is equivalent to searching a maximum weighted independent set for some MWIS instance.*

Proof. Given the set of forbidden subgraphs $\mathcal{C}(G)$ and a set of marked vertex pairs M, we construct an instance of the Maximum Weight Independent Set Problem (V', E', w). $G' = (V', E')$ is a graph and $w : V' \to \mathbb{R}_{\geq 0}$ is a weight function. We call two forbidden induced subgraphs *adjacent* if they share an unmarked vertex pair and use $\mathrm{adj}_M(S_1, S_2)$ to denote that. The instance (V', E', w) is defined as

$$V' := \mathcal{C}(G) \tag{3.3}$$
$$E' := \{\{S_1, S_2\} \mid S_1, S_2 \in V',\ \mathrm{adj}_M(S_1, S_2)\} \tag{3.4}$$
$$w(S) := c_{\min}(S, M). \tag{3.5}$$

We show that an independent set for the graph G' is a subgraph packing for the graph G. Let I be an independent set for G'. It holds that forbidden induced subgraphs in I are not adjacent, meaning that they do not share an unmarked vertex pair. This conforms to the definition of a subgraph packing. Therefore is I a subgraph packing.

From the definitions of w and c we can conclude that

$$w(I) = \sum_{S \in I} w(S) = \sum_{S \in I} c_{\min}(S, M) = c(I).$$

Therefore an independent set $I \subseteq V'$ is a subgraph packing for the graph G and its weight $w(I)$ and its cost $c(I)$ are the same. □

Given this correspondence, we now describe the lower bound algorithms used in this thesis. The algorithms can be viewed as heuristics for MWIS which are adapted for subgraph packing.

Simple Packing

A straight forward algorithm for calculating a valid subgraph packing is the following. The algorithm iterates over all forbidden subgraphs $\mathcal{C}(G)$ and inserts a subgraph into the subgraph packing if possible. This results in a valid subgraph packing by definition.

Algorithm 3.2: Simple packing lower bound

```
Input: A graph $G = (V, E)$, a cost function $c : \binom{V}{2} \to \mathbb{R}_{\geq 0}$, and a set of marked
       vertex pairs $M \subseteq \binom{V}{2}$.
Output: A lower bound on the parameter $k$.
1 $B \leftarrow \emptyset$                                   // subgraph packing
2 foreach $S \in \mathcal{C}(G)$ do
3     if $\forall S' \in B : \neg\, \mathrm{adj}_M(S, S')$ then
          // $S$ shares no vertex pair with any subgraph in $B$
4         $B \leftarrow B \cup \{S\}$
5 return $\sum_{S \in B} c_{\min}(S, M)$
```

This is a good algorithm for (unweighted) $\mathcal{F}$-free Edge Editing [Zü17]. For $\mathcal{F}$-free Edge Editing, we can incorporate the minimum editing costs of the forbidden induced subgraphs. This gives rise to the following algorithm.

Greedy Packing

For the previous lower bound, the editing costs of the forbidden subgraphs are completely ignored. For finding a Maximum Weight Independent Set one variant is to always choose the vertex with maximum weight, which is not adjacent to a vertex already in the independent set. Algorithm 3.3 uses this approach. For unweighted editing, Algorithm 3.2 and Algorithm 3.3 are equivalent.

Although both algorithms are very similar, they differ in their time and space complexity. The greedy lower bound needs additional space and time for storing all forbidden subgraphs and then sorting them by their editing cost.

Algorithm 3.3: Greedy packing lower bound

```
Input: A graph $G = (V, E)$, a cost function $c : \binom{V}{2} \to \mathbb{R}_{\geq 0}$, and a set of marked
       vertex pairs $M \subseteq \binom{V}{2}$.
Output: A lower bound on the parameter $k$.
1 $B \leftarrow \emptyset$                                   // subgraph packing
2 foreach $S \in \mathcal{C}(G)$ in descending order of $c_{\min}(S, M)$ do
3     if $\forall S' \in B : \neg\, \mathrm{adj}_M(S, S')$ then
          // $S$ shares no vertex pair with a subgraph in $B$
4         $B \leftarrow B \cup \{S\}$
5 return $\sum_{S \in B} c_{\min}(S, M)$
```

Local Search

In [GHS+20], a local search algorithm based on the (unweighted) Maximum Independent Set heuristic introduced in [ARW12] has been used for (unweighted) $\mathcal{F}$-free Edge Editing. The heuristic from [ARW12] has been adapted in [NPS18] for the Maximum Weight Independent Set Problem. We introduce a local search algorithm that adapts the algorithm from [NPS18] for Weighted $\mathcal{F}$-free Edge Editing. Algorithm 3.4 improves an existing subgraph packing B through local changes.

The algorithm uses a variable neighborhood descent method [NPS18]. We explore neighborhoods of the subgraph packing B by performing (i, j)-swaps, where i subgraphs are removed and j subgraphs are inserted into the packing. We switch between either performing $(1, 1)$- and $(1, 2)$-swaps or $(\omega, 1)$-swaps, described in Algorithm 3.5 and Algorithm 3.6 respectively. $\omega \in \mathbb{N}$ stands for an arbitrary number. We are now going to describe both algorithms further.

Algorithm 3.4: Local Search

```
   Input: A graph $G$, a cost function $c : \binom{V}{2} \to \mathbb{R}_{\geq 0}$, a set of marked vertex pairs $M$,
          a subgraph packing $B \subseteq \mathcal{C}(G)$, and the maximum number of rounds with
          no improvements $N$.
   Output: A subgraph packing $B' \subseteq \mathcal{C}(G)$ with $c(B') \geq c(B)$.
 1 $n \leftarrow 0$                              // number of rounds with no improvements
 2 repeat
 3     $B_{prev} \leftarrow B$
 4     foreach $S \in B$ do
 5         $B \leftarrow$ FindOneTwoImprovement$(G, c, M, B, S)$      // Algorithm 3.5
 6     if $c(B) = c(B_{prev})$ then
 7         $B \leftarrow$ FindOmegaImprovement$(G, c, M, B)$          // Algorithm 3.6
 8     if $c(B) = c(B_{prev})$ then $n \leftarrow n + 1$
 9     else $n \leftarrow 0$
10 until $B = B_{prev}$ or $n > N$
11 return $B$
```

The local search procedure executes multiple rounds of the optimization procedures and only resorts to Algorithm 3.6 if Algorithm 3.5 does not improve the lower bound. The algorithm terminates if the lower bound did not improve for more than N rounds or the subgraph packing did not change at all.

Algorithm 3.5 searches for $(1, 1)$- and $(1, 2)$-swaps, that improve the lower bound induced by the packing. It is called for each subgraph in the packing and checks whether the subgraph can be replaced. The set of candidates is the set of forbidden induced subgraphs, which are adjacent to the removed subgraph, but to no other subgraph in the packing. The algorithm finds the single subgraph or the pair of subgraphs which maximizes the minimum editing cost. It is not guaranteed that a subgraph with higher editing cost can be found. In that case, no improvement was made.

The major complexity of the algorithm is the iteration over adjacent subgraphs which are not adjacent to one already in the packing $N_{\bar{B}}(S) := \{S' \in \mathcal{C}(G) \mid \mathrm{adj}_M(S, S') \wedge \forall S'' \in B : \neg\, \mathrm{adj}_M(S', S'')\}$. To accomplish this efficiently, we modify the algorithms from Chapter 2. The algorithm only outputs forbidden subgraphs that are adjacent to $G[S]$ and not to a subgraph already in the packing. To check whether a vertex pair is in an induced subgraph in the packing, we update a set of vertex pairs for each change of the packing. This speeds up the local listing, as it can stop earlier.

Algorithm 3.6 iterates over all forbidden induced subgraphs, regardless if it is adjacent to some other forbidden induced subgraph, or not. The algorithm tries to improve the lower bound by inserting a single subgraph into the packing and removing the adjacent subgraphs already in the packing. This is a $(\omega, 1)$-swap, as potentially multiple subgraphs are removed and one is inserted. The swap improves the lower bound if the combined

Algorithm 3.5: Find (1, 2) or (1, 1) improvement

```
Input: A graph $G = (V, E)$, a cost function $c : \binom{V}{2} \to \mathbb{R}_{\geq 0}$, a set of marked vertex
       pairs $M$, a subgraph packing $B \subseteq \mathcal{C}(G)$ and an induced subgraph $S \in B$.
Output: A subgraph packing $B' \subseteq \mathcal{C}(G)$ with $c(B') \geq c(B)$.
1  $B \leftarrow B \setminus \{S\}$
2  $N_{\bar{B}}(S) \leftarrow \{S' \in \mathcal{C}(G) \mid \mathrm{adj}_M(S, S') \wedge \forall S'' \in B : \neg\, \mathrm{adj}_M(S', S'')\}$
3  $N_{\max} \leftarrow \{S\}$, $c_{\max} \leftarrow c_{\min}(S, M)$
4  foreach $S_1 \in N_{\bar{B}}(S)$ do
5      $c_1 \leftarrow c_{\min}(S_1, M)$
6      if $c_1 > c_{\max}$ then // inserting $S_1$ improves the bound
7          $N_{\max} \leftarrow \{S_1\}$, $c_{\max} \leftarrow c_1$
8      foreach $S_2 \in N_{\bar{B}}(S)$ do
9          $c_2 \leftarrow c_{\min}(S_2, M)$
10         if not $\mathrm{adj}_M(S_1, S_2)$ then
11             if $c_1 + c_2 > c_{\max}$ then // inserting $S_1$ and $S_2$ improves the
               bound
12                 $N_{\max} \leftarrow \{S_1, S_2\}$, $c_{\max} \leftarrow c_1 + c_2$
13 $B \leftarrow B \cup N_{max}$
14 return $B$
```

Algorithm 3.6: Find $(\omega, 1)$ improvements

```
Input: A graph $G$, a cost function $c : \binom{V}{2} \to \mathbb{R}_{\geq 0}$, a set of marked vertex pairs $M$,
       and a subgraph packing $B \subseteq \mathcal{C}(G)$.
Output: A subgraph packing $B' \subseteq \mathcal{C}(G)$ with $c(B') \geq c(B)$.
1  foreach $S \in \mathcal{C}(G)$ do
2      $N_B(S) \leftarrow \{S' \in B \mid \mathrm{adj}_M(S, S')\}$                // $|N_B(S)| \leq \binom{|S|}{2}$
3      if $c_{\min}(S, M) > \sum_{S' \in N_B(S)} c_{\min}(S', M)$ then
4          $B \leftarrow B \setminus N_B(S)$
5          $B \leftarrow B \cup \{S\}$
6  return $B$
```

minimum editing costs of the neighbors in the packing is less than the editing cost of the single subgraph. The algorithm iterates over all forbidden induced subgraphs once and performs a $(\omega, 1)$-swap whenever it improves the lower bound.

To accelerate the calculation of $N_B(S) := \{S' \in B \mid \mathrm{adj}_M(S, S')\}$, we keep track whether an unmarked vertex pair is in a subgraph of the packing and, if that is the case, which subgraph it belongs to. This is possible because each vertex pair can only be in at most one induced subgraph from the packing. We calculate that index at the start of Algorithm 3.6 and keep it updated whenever the packing changes.

In addition to calling Algorithm 3.4 while calculating a lower bound, the base subgraph packing is updated every time the graph is edited and/or a vertex pair is being marked. An edit may invalidate previously found forbidden induced subgraphs in the subgraph packing B or create new forbidden subgraphs, which potentially could be inserted.

3.3 Subgraph Selection for Branching

In this section we introduce methods for selecting forbidden subgraphs, i.e. variants of FindSubgraph called on Line 2 of Algorithm 3.1. The choice of subgraph and the order, in which Algorithm 3.1 iterates over the vertex pairs in Line 7, influences the performance of the algorithm. The running time is reduced when fewer branches are explored.

First

The simplest method for choosing a forbidden subgraph is taking the first one that is being found. We use the algorithms discussed in Chapter 2 and stop searching for other subgraphs after the first subgraph was found.

This method is highly dependent on the ordering of the vertices in the input instance. The algorithms we presented iterate over vertices, edges, and neighbors in a fixed order. A forbidden subgraph that contains a vertex from the beginning of the ordering is more likely to be the first one found.

Most Marked

Forbidden subgraphs where some vertex pairs are already marked result in fewer branches. Therefore, we search for forbidden subgraphs $G[S]$ with the most edited vertex pairs. This selection rule is used by [Zü17]. The vertex pairs are ordered by the number of incident forbidden subgraphs in decreasing order.

Most Adjacent Subgraphs

An edit that destroys many forbidden subgraphs is more likely to be in a solution. This also has the potential to improve lower bounds. This heuristic has already been proven to be successful for unweighted $\mathcal{F}$-free Edge Editing by the authors of [GHS+20].

Calculating the number of adjacent subgraphs $|N(S)| := |\{S' \in \mathcal{C}(G) \mid \mathrm{adj}_M(S, S')\} \setminus \{S\}|$ for each forbidden subgraph $S \in \mathcal{C}(G)$ is expensive. Instead of calculating $|N(S)|$ exactly, we use an upper bound to estimate it. An exact count on the number of forbidden subgraphs at a given vertex pair $|N(uv)| := |\{S' \in \mathcal{C}(G) \mid u, v \in S'\} \setminus \{S\}|$ can easily be updated with each edit operation. We use $\sum_{uv \in \binom{S}{2} \setminus M}(|N(uv)| - 1)$ as the estimate for $|N(S)|$. Subgraphs which share multiple unmarked vertex pairs with S will be counted more than once.

For each subgraph, we associate the vector of subgraph counts $|N(uv)|$ for each vertex pair uv in decreasing order. We want to find the subgraph with the lexicographically largest vector, i.e. the largest subgraph count matters, ties are broken by the second largest and so on. As the largest subgraph count matters the most, we only need to check subgraphs which are incident to vertex pairs for which $|N(uv)|$ is maximal.

3.4 Search Strategies

One major challenge for FPT algorithms for Weighted F-free Edge Editing is the choice of the parameter k. For (unweighted) F-free Edge Editing, the parameter value is incremented by one, i.e. exploring one more edit operation for each new search step. The corresponding search tree grows with the branching factor $\binom{p}{2}$, where p is the size of the largest forbidden subgraph. This has the nice property that the total runtime of the algorithm is dominated by the last step. For the weighted case, it is not obvious how much larger the search tree is for a new value $k' > k$. Incrementing by 1 in the unweighted case can be generalized in several different ways. In the following, we present several of them.

All search strategies have the problem, that a found solution is not guaranteed to have the optimal editing cost. As we normally want to find solutions with optimal editing cost, we need to keep searching until all branches are either explored or pruned. After we found the first solution, we can prune branches that exceed the currently optimal editing cost i.e. the value is an upper bound of the initial problem. This improves the search but is significantly slower than stopping after the first solution has been found.

Increment by Minimum Cost

As previously discussed, the proof of the theoretical running time of the FPT algorithm uses the assumption that the editing costs are at least one. With this assumption, the depth of the search tree is bounded by $\lfloor k \rfloor + 1$. This leads to a search strategy that increases the parameter k by the minimum editing cost for each search step. For each search step, the size of the longest path in the search tree can increase by at most one. We only consider the smallest non-zero cost of the instance.

We formulate the search strategies as rules for the next parameter value k'. When we increment by the minimum editing cost, the next value for the parameter k is

$$k' := k + \min\{c(e) \mid e \in \binom{V}{2} \land c(e) > 0\}.$$

Unfortunately, the running time with this strategy depends on the smallest editing cost in the input instance. Small absolute changes of the minimum editing cost can result in large relative changes of the step size.

Increment by 1

Another simple search strategy is directly adapting the strategy from unweighted $\mathcal{F}$-free Edge Editing [RWB+07]. The editing cost k is incremented by one until a solution is found.

$$k' := k + 1$$

This strategy is also dependent on the range of the cost function of the input instance. The costs cannot be scaled without affecting the effective step size. For example, consider the case where the editing costs sum to less than one. For each $k \geq 1$, the search tree will not be pruned.

Preventing Pruning

A method for making sure that every new call to the FPT algorithm results in progress, is to make sure that a significant amount of the branches will not be pruned again in the next step. Every time a branch is pruned in Line 1 of Algorithm 3.1 by a lower bound, we keep track of the difference $l - k$ between the lower bound l and the current editing cost k. We use D to denote the complete set of all such values. If the editing cost would be increased enough such that k is at least as large as l, the branch would have not been pruned. If the initial value of k is increased by the minimum value in D, then at least one branch would not have been pruned. Choosing the $q\%$ smallest values from D would prevent $q\%$ of the prunes. The next parameter value is

$$k' := k + \mathrm{quantile}_q(D)$$

where $\mathrm{quantile}_q(\cdot)$ returns the q-quantile.

In our implementation, we use the median value. The conclusion that the value k' prevents the pruning $q\%$ of all previously pruned branches is based on the assumption that the lower bound is calculated independently of current editing cost. This is not the case in our implementation, because we stop improving the lower bound as soon as it is larger than the remaining editing cost. Therefore it is not guaranteed to prevent the pruning of a branch. We try to compensate that with the chosen value for q.

Exponential Growth Estimation

We can observe, that the number of recursive calls is approximately an exponential function of the initial parameter k. The exact relation is dependent on the input instance and configuration. Figure 3.4 depicts number of calls for different values of k and different configurations of the FPT algorithm. Note that the trend is nearly linear in the log-domain with varying slopes. An example where this assumption does not hold is given with Figure 4.12 in Section 4.5.

We try to estimate this relationship for each instance and configuration of the FPT algorithm by a log-linear model. It assumes that the number of calls c grows exponentially in the parameter. This can be represented in the model

$$\log(c) \sim \alpha + \beta k.$$

For each execution of the algorithm we keep track of the parameter k_i and the number of recursive calls the algorithm makes c_i, $i = 1 \ldots t$. The model parameters are estimated with simple linear regression as seen in [Gol64], resulting in the estimated values $\hat{\alpha}$ and $\hat{\beta}$. Instead of using all data points, only the last few data points are used. In our implementation, we use the last three data points. The next value for k is then calculated by

$$k' := \frac{\log(r \cdot c_t) - \hat{\alpha}}{\hat{\beta}}, \tag{3.6}$$

where r is the desired growth rate, i.e. we want the next execution to have $r \cdot c_t$ recursive calls, where c_t is the last number of calls. In our implementation, we use $r = 2$.

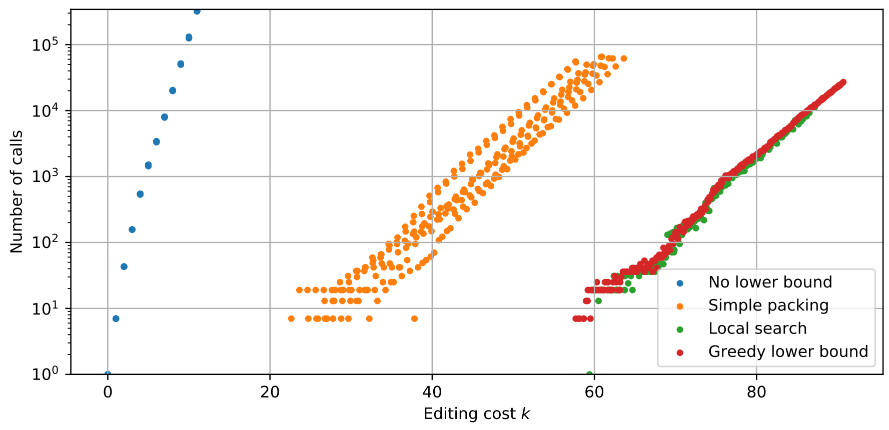


Figure 3.4: Example of a relationship between editing cost k and the number of calls from eight permutations of instance no. 1027 from the `bio` dataset for $\mathcal{F} = \{C_4, P_4\}$. The evaluation points are spaced with distance one after the initial lower bound. The relationship between editing cost k and the number of calls is approximately exponential for a given configuration of the algorithm. The approximation becomes better with a rising number of calls.

Unfortunately, using only the estimate from the log-linear model suffers from over- and underestimating the step size, especially for the first few data points. For example, if the first few executions of the algorithm result in the same number of calls, the search strategy fails. To mitigate this risk, we bound the estimate from above and below. For a lower bound on the new value, we use the previously discussed search strategy for preventing pruning of branches. We use $4 \cdot (k - k_{\text{init}}) + k_{\text{init}}$ as an upper bound for k', where k was the previous and k_{init} the first editing cost.

3.5 ILP Formulation

Grötschel and Wakabayashi formulated Cluster Editing as an integer linear program (ILP) and applied it successfully to real-world clustering tasks [GW89]. We use a similar formulation, generalized for any set of forbidden subgraphs $\mathcal{F}$. We use a binary vector $\boldsymbol{x} \in \mathbb{B}^{\binom{V}{2}}$ to decode whether a vertex pair uv is edited, i.e.

$$\boldsymbol{x}_{uv} = 1 \iff uv \in L$$

The ILP formulation of the Weighted $\mathcal{F}$-free Edge Editing Problem is the following:

$$\text{minimize} \sum_{u,v \in V} c(uv) \cdot \boldsymbol{x}_{uv} \tag{3.7}$$

$$\text{subject to} \sum_{uv \in L_{\neq}(G,F)} (1 - \boldsymbol{x}_{uv}) + \sum_{uv \in L_{=}(G,F)} \boldsymbol{x}_{uv} \geq 1 \qquad \forall F \in \mathcal{F}(V) \tag{3.8}$$

$$\boldsymbol{x}_{uv} \in \{0, 1\} \qquad \forall uv \in \binom{V}{2} \tag{3.9}$$

We use $L_{\neq}(G, F)$ ($L_{=}(G, F)$) to denote the vertex pairs in F which are edited (not edited) in respect to G. $\mathcal{F}(V)$ is the set of all graphs with vertices from V which are isomorphic to a graph in $\mathcal{F}$.

The constraints of Equation 3.8 range over all possible forbidden subgraphs on V. There are $\binom{|V|}{p}$ sets of p vertices and $p!$ permutations of these sets. Note that some of the forbidden subgraphs might be the same graph with differently ordered vertices. For example two P_3 on the set of vertices $\{a, b, c\}$. The paths (a, b, c) and (c, b, a) result in isomorphic subgraphs. For $\mathcal{F} = \{P_3\}$ there are $\frac{3!}{2} \cdot \binom{|V|}{3}$ constraints.

Even for the small forbidden subgraphs, the explicit generation of all constraints is not feasible. We instead resort to *row generation*.

3.5.1 Algorithm

Algorithm 3.7 calls the method MINIMIZE of an ILP solver. The solver minimizes the function defined in Equation 3.7. As already covered, constructing all constraints from Equation 3.8 is not feasible. The ILP solver calls Algorithm 3.8 as a callback for some heuristic solution $\boldsymbol{x} \in \mathbb{B}^{\binom{V}{2}}$. Constraints that are violated by $\boldsymbol{x}$ are inserted into the set of all constraints. We find violated constraints by constructing the graph corresponding to the edited vertex pairs $\{uv \in \binom{V}{2} \mid \boldsymbol{x}_{uv} = 1\}$. If $G \triangle \boldsymbol{x}$ is not $\mathcal{F}$-free, for each forbidden induced subgraph at least one additional edit must be made or an already applied edit must be undone. This is done until the solver found an optimal solution.

Algorithm 3.7: ILP Algorithm

```
    Input: A graph $G = (V, E)$ and a editing cost function $c : \binom{V}{2} \to \mathbb{R}$.
    Output: A set of vertex pairs $L \subseteq \binom{V}{2}$, for which $G \triangle L$ is $\mathcal{F}$-free and the costs
            are minimal.
1   $obj(\boldsymbol{x}) \leftarrow \sum_{uv \in \binom{V}{2}} c(uv) \cdot \boldsymbol{x}_{uv}$              // objective function to minimize
2   $constraints = \emptyset$                                   // initial constraints
3   $\boldsymbol{x} \leftarrow$ MINIMIZE($obj$, $constraints$, CALLBACK)          // the solver calls
      Algorithm 3.8 for each intermediate solution
4   return $\{uv \in \binom{V}{2} \mid \boldsymbol{x}_{uv} = 1\}$
```

Algorithm 3.8: ILP Callback

```
    Input: A graph $G = (V, E)$ and a heuristic solution $\boldsymbol{x} : \mathbb{B}^{\binom{V}{2}}$.
    Output: Constraints that are violated by $\boldsymbol{x}$.
1   $G' \leftarrow G \triangle \boldsymbol{x}$
2   foreach $S \in \mathcal{C}(G')$ do
        // at least one vertex pair needs to be edited from $G'$ to
           destroy $G'[S]$
3       $L_{\neq}(S) \leftarrow \{uv \in \binom{S}{2} \mid uv \in E(G) \neq uv \in E(G')\}$
4       $L_{=}(S) \leftarrow \{uv \in \binom{S}{2} \mid uv \in E(G) = uv \in E(G')\}$
5       $constraints \leftarrow constraints \cup \{(\sum_{uv \in L_{\neq}(S)} (1 - \boldsymbol{x}_{uv}) + \sum_{uv \in L_{=}(S)} \boldsymbol{x}_{uv} \geq 1)\}$
6   return $constraints$
```

3.5.2 Restricting Constraint Generation

A significant problem for ILP-solvers is the large number of constraints in the ILP formulation. The authors of [GW89] restrict the number of constraints added in each solving step by a fixed amount. They generate a multiple of the desired number of constraints and add the ones, that are the most violated. We use the following two methods for restricting the number of constraints added in each solving step.

Sparse Constraints

For keeping constraints sparse, we keep track which vertex pairs are already part of a constraint that was inserted in the current iteration. If all vertex pairs of a forbidden subgraph are already inserted in this step, we skip it. This bounds the number of constraints inserted in each step by $\binom{n}{2}$.

Single Constraints

A stricter approach is only allowing one additional constraint in each step.

4. Evaluation

In this chapter, we evaluate the algorithms described in Chapter 2 and in Chapter 3 on protein-protein interaction data. First, we describe the setup for our experiments and the dataset in Section 4.1. We evaluate the forbidden subgraph listing algorithms in Section 4.2. Next, we describe the effects of the lower bounds algorithms and subgraph selection strategies on the running time of the FPT algorithm in Section 4.3 and Section 4.4 respectively. The search strategies for the optimal editing costs are analyzed in Section 4.5. Finally, we evaluate the ILP algorithm and compare it to the FPT algorithm in Section 4.6.

4.1 Experimental Setup

In this section, we give a short overview of the hard- and software used for our experiments and describe some implementation details. We also define the instances we use for our experiments.

All experiments were executed on a single core of an Intel Xeon Gold 6144 dual CPU system clocked at 3.50 GHz. 16 experiments were executed simultaneously. A total of 192 GiB RAM is available. The server runs openSUSE Leap 15.1.

The algorithms have been implemented in C++14 and compiled with GCC version 8.2.1 and compiler flags `-O3 -march=native -DNDEBUG`. As most input instances are rather small, graphs are represented with adjacency matrices. We use the Boost `dynamic_bitset` library[1] [SAP+] for the rows of the adjacency matrix. Operations on vertex sets are implemented as bit operation, i.e. $N(u) \setminus N(v)$ is performed by `adj[u] - adj[v]` on the bit-vectors `adj[u]` and `adj[v]`. For the implementation of the ILP-algorithm, we use the Gurobi Solver version 8.1.1 [Gur]. For calculating graph properties and visualization, we used Python 3 and the Networkx package [HSS08].

To avoid floating-point errors, we internally transform the editing costs to integers in the implementation. We use a factor m and transform the costs by multiplying and rounding up. For the results, the costs are transformed back. While describing the algorithms and in the analysis, we talk about the unscaled costs. For a set of edits L, the absolute error introduced by the discretization can be bounded with $0 \leq c'(L) - c(L) \leq \frac{|L|}{m}$. After a preliminary study, we chose to use $m = 100$ for all experiments, resulting in a precision of 0.01.

The implementation can be found online[2].

[1] `http://boost.org/libs/dynamic_bitset`

[2] `https://github.com/jonasspinner/weighted-f-free-edge-editing`, November 28: `fa537279`

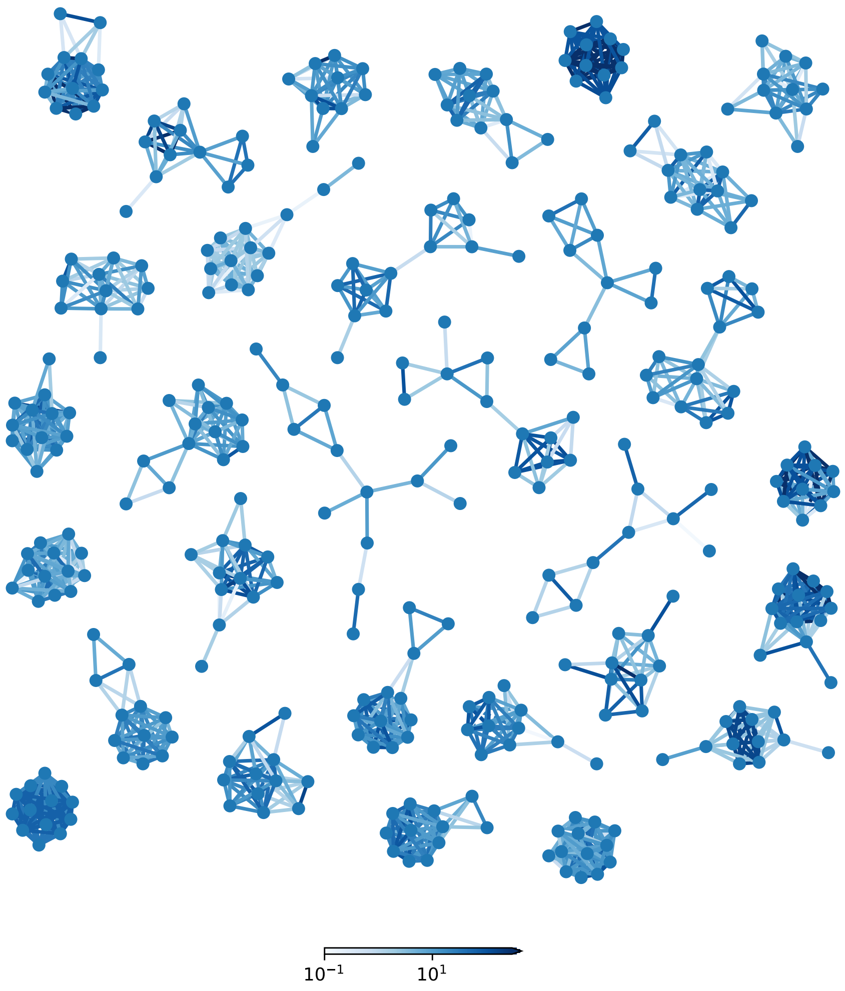


Figure 4.1: The first 30 instances of the `bio` dataset with less than 15 vertices. Higher editing costs are indicated by a darker color. Non-edges are ignored in this visualization.

4.1.1 Instances

The evaluation focuses on the `bio` dataset[3] from [RWB+07, BBBT07]. The instances are protein-protein interaction networks derived from 66 organisms of the COG dataset[4].

The dataset was created for evaluating Cluster Editing algorithms [RWB+07, BBBT07]. The initial data is a large matrix describing the similarity between proteins. Non-negative similarities are considered as edges. The graph consists of 50600 connected components, of which 42563 have one or two vertices and 4073 are already cliques. The remaining 3964 connected components are the instances of the `bio` dataset. We further reduce it by only considering graphs with less than 1000 vertices, ending with 3955 instances. Each instance is a matrix with similarity scores. Let $s(uv)$ be the score of proteins u and v. The instance (G, c) is a graph $G = (V, E)$ and a cost function c. An edge $uv \in E$ only exists if the interaction is non-negative. The editing cost $c(uv)$ of a vertex pair uv is the absolute magnitude of the similarity score, $c(uv) := |s(uv)|$.

[3] `https://bio.informatik.uni-jena.de/data/#cluster_editing_data`
[4] `http://www.ncbi.nlm.nih.gov/COG/`

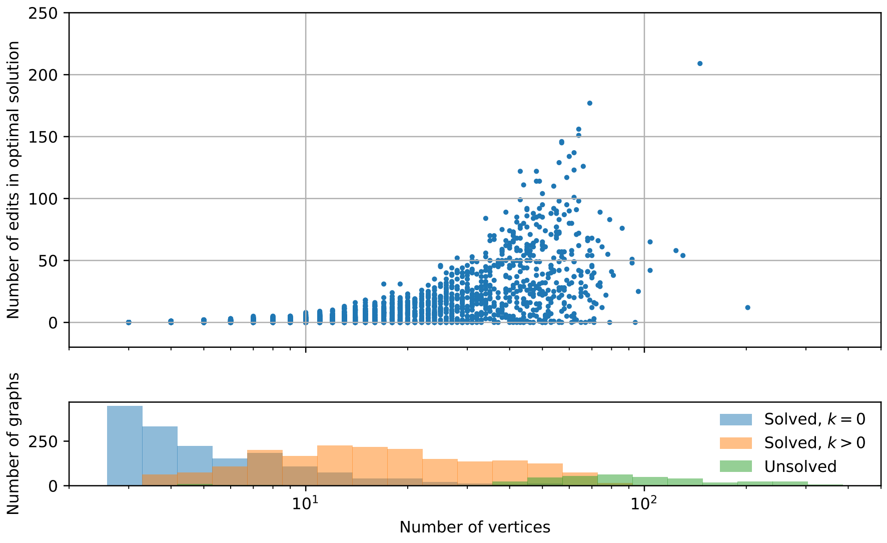


Figure 4.2: Solutions for bio instances for $\mathcal{F} = \{C_4, P_4\}$. The data is generated by the FPT algorithm with greedy lower bound, "most adjacent" subgraph selection and known optimal editing cost. and a time limit of 100 seconds. The instances with editing cost $k = 0$ do not necessarily have zero edits, as zero-cost vertex pairs are allowed. 1682 instances are solved with $k = 0$, 1916 with $k > 0$ and 357 remain unsolved.

The bio dataset has been used for evaluation of algorithms for unweighted $\mathcal{F}$-FREE EDITING [GHS+20] and WEIGHTED CLUSTER EDITING [RWB+07, BBBT07]. In Figure 4.1 30 smaller instances with less than 15 vertices are shown. It visualizes several properties of the bio dataset. The instances often consist of one or more groups which are nearly cliques. Sometimes smaller structures like single vertices, edges or triangles connect to the core. Not all edges in a cluster are equal. For example, the core of instance in the lower right corner has a visible subcluster with larger editing cost, as seen by the dark edges. Several instances in the middle of the figure have long induced paths of length up to 8. These paths can be destroyed by removing edges from the path.

Figure 4.2 depicts the number of edits in an optimal solution for each instance for $\mathcal{F} = \{C_4, P_4\}$. Larger instances do allow for "harder" instances with more edits. But even instances with a large number of vertices may have a solution with only a few edits. The lower figure visualizes the fact that many instances with 10 or fewer vertices can be solved with an editing cost parameter of zero. Most of the unsolved instances have between 40 and 300 vertices.

4.1.2 Hard Instances

Many of the bio instances can be solved with no or only a few edits. As we are mainly interested in the cases where constant startup costs cease to matter and scaling effects start to dominate, we mostly do not use the whole bio dataset for evaluation. We want only the "hard" instances and assume that the number of edits significantly influences the running time of the algorithms for finding solutions. We define the "hard" instances to be the ones, which need at least 10 edits for an optimal solution to the WEIGHTED $\{C_4, P_4\}$-FREE EDITING PROBLEM. This subset has 1058 instances. Figure 4.3 shows the running time of all instances of the bio dataset in regards to the editing cost of the optimal solution for variants of the FPT and the ILP algorithm. The instances that need at least 10 edits are colored orange. The variant of the ILP algorithm has a significant start-up overhead. Even

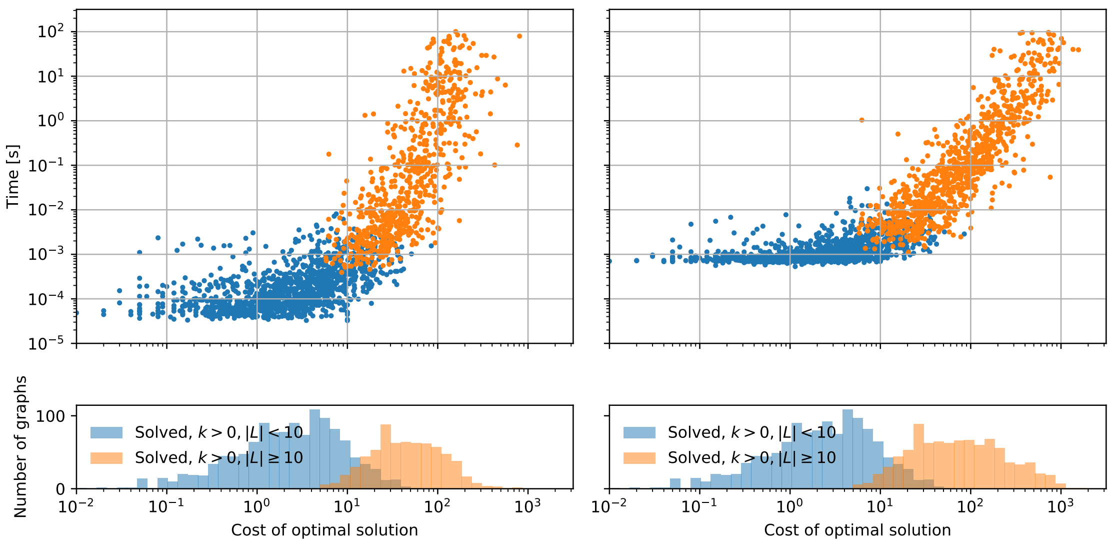


Figure 4.3: Running times for all instances of the `bio` dataset for $\{C_4, P_4\}$-free Editing for the FPT algorithm (left) and the ILP algorithm (right). The FPT algorithm was executed with known optimal cost, greedy lower bound calculation and branching on subgraphs with most adjacent subgraphs. The ILP algorithm used sparse constraints. For the FPT and ILP algorithm, 382 and 135 instances remained unsolved respectively. “Hard” instances that were solvable within the timelimit are colored orange. Note that both axes are scaled logarithmic.

the instances that need less than 10 edits need about 10^{-3} seconds. The FPT algorithm manages to solve them up to one order of magnitude faster. When we only consider the hard instances, the start-up costs are no longer the factor that dominates the instances in the dataset. This suggests that the chosen threshold of minimum 10 edits is a valid choice. The instances still have a wide range of running times, from 10^{-3} seconds up to the time limit of 10^2 seconds. All experiments use the hard instances and solve $\{C_4, P_4\}$-free Editing unless otherwise noted.

4.2 Listing Forbidden Subgraphs

In this section we evaluate the algorithms for listing forbidden subgraphs from Chapter 2. We evaluate algorithms for listing P_3s, $\{C_4, P_4\}$s, and $\{C_5, P_5\}$s.

The experiments are performed on all 3955 instances of the `bio` dataset. The algorithms iterate over all forbidden subgraphs, incrementing a counter and discarding the actual subgraph data. This ensures that the effects of copying subgraphs do not influence the benchmarks, as some graphs have up to 10^8 forbidden subgraphs. The experiment was executed 10 times for each instance and 2 permutations of its vertex pairs. The running time only differed slightly between runs, so we only visualize the mean running time. For each instance, three permutations of the vertices are evaluated, resulting in 11865 total instances.

Listing P_3s

P_3 is the smallest forbidden subgraph we consider. We compare three different algorithms. The naive algorithm enumerates all three-tuples of vertices (a, b, c), checking if they form a P_3 and conform to the symmetry-breaking restrictions discussed in Section 2.4. The other two algorithms are described in Section 2.1. Algorithm 2.1 iterates over all edges and tries

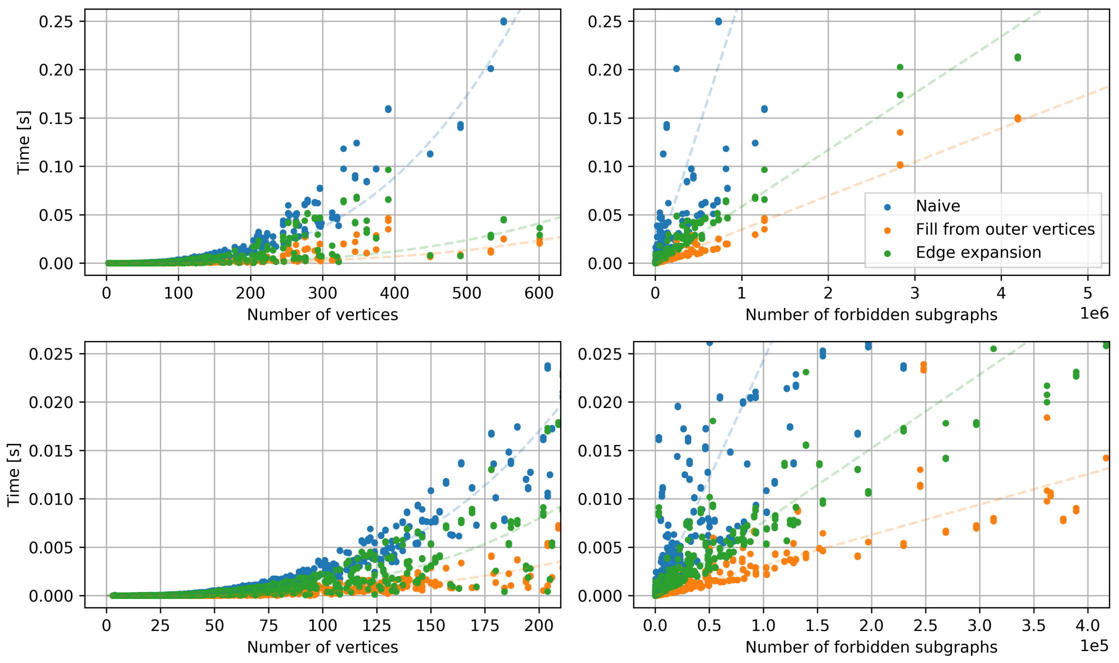


Figure 4.4: The average running time of P_3-listing algorithms plotted against the number of vertices (left) and the number of forbidden subgraphs (right). Dashed lines are the functions $a \cdot n^3$ (left) and $a \cdot p_3(G)$ (right) fitted with the least-squares method. The instances are from the complete `bio` dataset. The lower row is a zoomed-in version of the upper row.

to expand them into a P_3. Algorithm 2.2 completes the path by starting from the outer vertices.

The result of the experiments on the `bio` dataset can be seen in Figure 4.4. All figures are based on the same data. The left column visualizes the scaling behavior for the number of vertices in the graph and the right column the dependency of the running time with the number of forbidden subgraphs. The naive algorithm scales approximately with $O(n^3)$. In contrast, the running time of the other algorithms mainly scales with the number of forbidden subgraphs. All algorithms have nearly the same running time for graphs with less than 50 vertices. Although both non-naive algorithms have similar asymptotic behavior, filling from outer vertices has a consistently better running time.

Listing C_4s and P_4s

For listing C_4s and P_4s, we compare the naive algorithm, the endpoint algorithm (Algorithm 2.4) and the midpoint algorithm (Algorithm 2.6). Unsurprisingly the naive algorithm is not feasible for larger graphs because of the $O(n^4)$ running time. Both the end- and midpoint-based algorithms are more efficient and again have a linear running time in the number of forbidden subgraphs. The midpoint algorithm is consistently the fastest algorithm for listing $\{C_4, P_4\}$. Comparing the linear fits for $a \cdot (p_4(G) + c_4(G))$ suggests that the midpoint algorithm is about 1.75 times faster. Unlike the endpoint algorithm, the midpoint algorithm does not iterate over P_3s that are not part of a P_4 or C_4.

Listing C_5s and P_5s

Cycles and paths of length 5 are the largest forbidden subgraphs we consider in this evaluation. Again we compare the naive algorithm, the endpoint algorithm (Algorithm 2.4) and the midpoint algorithm (Algorithm 2.6).

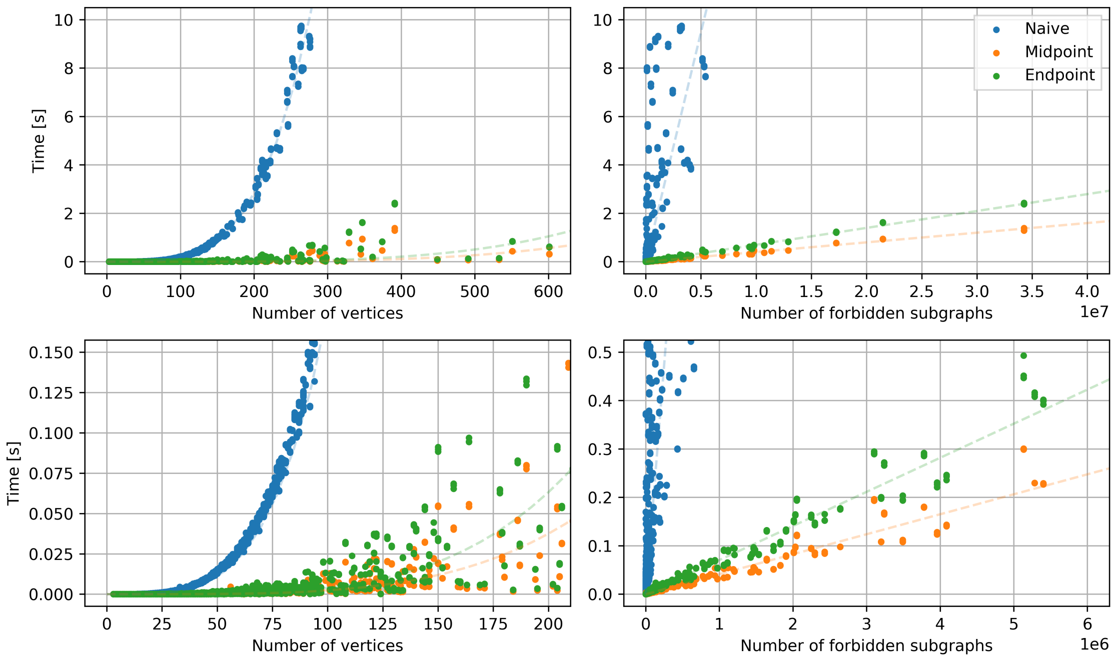


Figure 4.5: The average running time of $\{C_4, P_4\}$-listing algorithms plotted against the number of vertices (left) and the number of forbidden subgraphs (right). Dashed lines are the functions $a \cdot n^4$ (left) and $a \cdot (p_4(G) + c_4(G))$ (right) fitted with the least-squares method. The instances are from the complete `bio` dataset. The lower row is a zoomed-in version of the upper row. Note the different limits of the vertical scale in the lower row. Some instances did not finish in the 10 seconds time limit: 75, 3 and 3 for naive, midpoint and endpoint algorithms respectively.

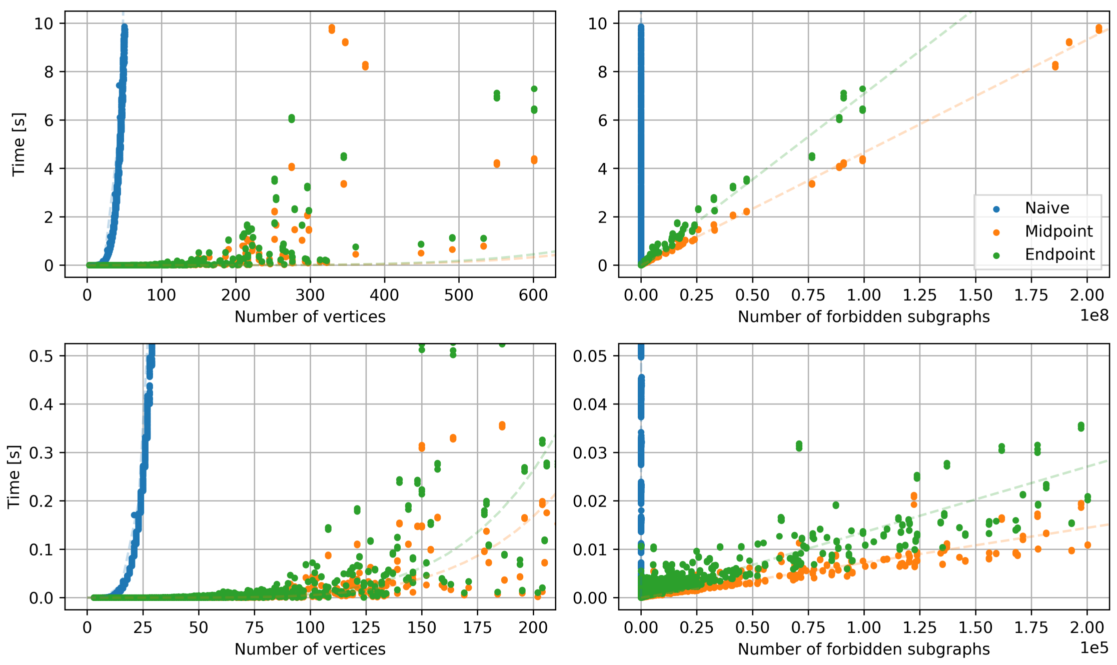


Figure 4.6: The average running time of $\{C_5, P_5\}$-listing algorithms plotted against the number of vertices (left) and the number of forbidden subgraphs (right). Dashed lines are the functions $a \cdot n^5$ (left) and $a \cdot (p_5(G) + c_5(G))$ (right) fitted with the least-squares method. The instances are from the complete `bio` dataset. The lower row is a zoomed-in version of the upper row. Some instances did not finish in the 10 seconds time limit: 1511, 9 and 18 for naive, midpoint and endpoint algorithms respectively.

The drawbacks of the naive algorithm become even more extreme for larger forbidden subgraphs. Both the midpoint and endpoint algorithms do scale linearly in the number of forbidden subgraphs. Comparing the linear fits for $a \cdot (p_5(G) + c_5(G))$ suggests that the midpoint algorithm is about 1.87 times faster

4.3 Lower Bounds

In this section, we compare the impact of the lower bound algorithms discussed in Section 3.2 on the FPT algorithm. Utilizing lower bounds is one of the most import factors for the running time of the FPT algorithm because the recursion can be terminated earlier, resulting in a smaller search tree. We compare the running times for having no lower bound, i.e. a lower bound of 0, the simple packing algorithm, the greedy algorithm, and the local-search-based algorithm.

Figure 4.7 depicts the running time of all `bio` instances for no lower bound, the greedy lower bound and the local search lower bound. The blue curve is the average running time for instances binned by their number of vertices and the band around it is the standard deviation. Some instances with less than 40 vertices are not solvable within the 100 seconds time limit when no lower bound is used. The variants that use a lower bound algorithm, solve all `bio` instances with up to about 40 vertices but are unable to solve some instances with more than 40 vertices. More than 50 % of the instances have at most 10 vertices and can be solved within 10^{-2} seconds.

In Figure 4.8a the lower bounds are compared by their ability to speed up the FPT algorithm. The instances are sorted by their running time and the figure shows how many

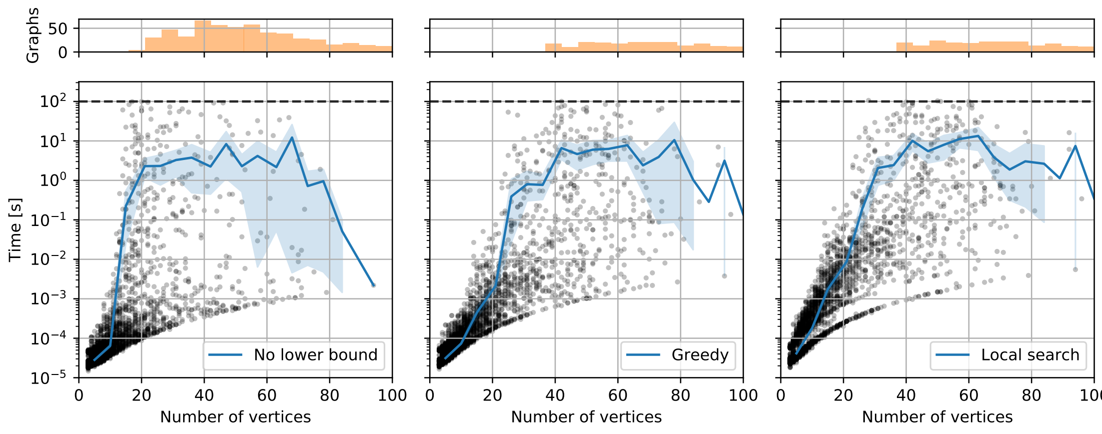
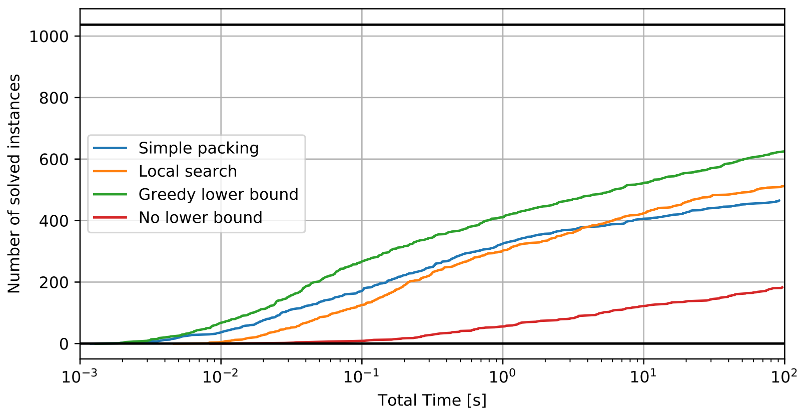


Figure 4.7: Running time of the FPT algorithm for three lower bounds. Instances are from the `bio` dataset and $\mathcal{F} = \{C_4, P_4\}$ is the set of forbidden subgraphs. The top row is a histogram of instances that did not finish within the 100 seconds time limit. The optimal editing cost was known beforehand and the "most adjacent" subgraph selection strategy was used for branching. The average running time of the solved instances, binned by their number of vertices, is shown as the blue line. The band is the size of one standard deviation.

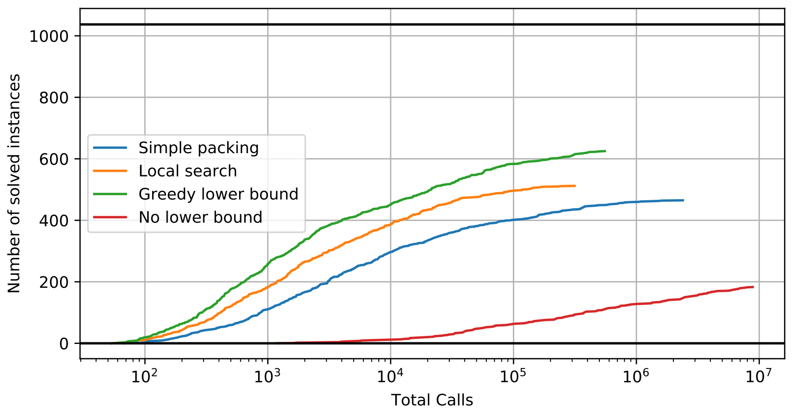
instances were solved within a given running time. Less than 200 instances can be solved when no lower bound algorithm is used. The greedy lower bound is consistently the best algorithm for all running times. The local search lower bound starts worse than the simple packing approach but can solve more instances if the maximum running time is more than 5 seconds.

Additionally, to analyzing the running time of the algorithms, we can also take a look at the number of total recursive calls of Algorithm 3.1. Every time the algorithm is called, either initially or in a recursion, the number of calls is incremented. In Figure 4.8b we sort the instances by the number of calls they needed and we can see how many instances are solvable with a given number of calls. We use this visualization to analyze the quality of the lower bounds, i.e. how accurate the lower bounds estimate the actual editing cost. If a lower bound is larger, more of the search tree is pruned and this results in fewer calls. Given a maximum number of calls, the local search lower bound now consistently outperforms the simple packing lower bound. However, it is still beaten by the greedy lower bound. The lower bounds of the local search algorithm are worse than the ones from the greedy algorithm. Recall that the local search algorithm updates the packing after each edit and the greedy algorithm builds the packing from scratch every time it is called. We suspect that updating the packing locally is not good enough to compensate for the cost of constructing a packing globally.

Furthermore, the local search algorithm seems to be keep getting stuck in a local maximum. If it did escape, the calculated lower bounds would be better or at least as good as the ones from the greedy lower bound. For unweighted $\mathcal{F}$-FREE EDGE EDITING, the local search approach proved to be successful [GHS+20]. The unweighted local search relies more heavily on plateau search. This seems not to be possible for the weighted case. It is more likely that the editing costs of two packings differ with at least some amount. This is a chance for improvement for future work. Strategies, like accepting slightly worse packings or forcing forbidden induced subgraphs into the packing, could help to escape local maxima.



(a) Number of solved instances for a certain running time.



(b) Number of solved instances for a certain number of calls.

Figure 4.8: Number of solved instances comparing different lower bound algorithms. For subgraph selection the most adjacent subgraphs are chosen. The exponential growth search strategy is used. The size of the dataset is marked with a horizontal line.

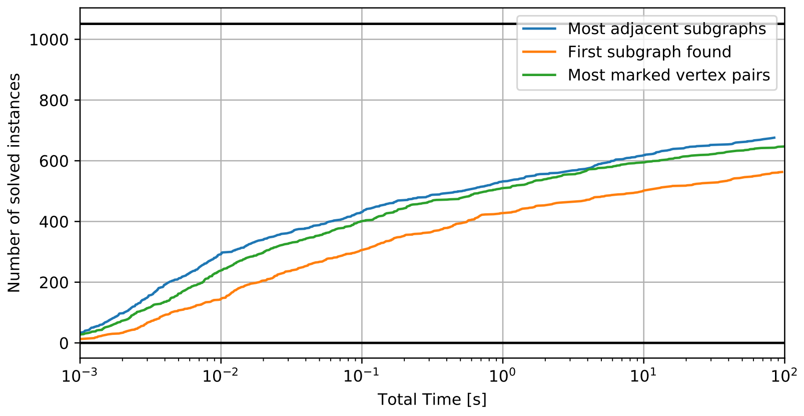


Figure 4.9: Number of solved instances for a certain running time comparing different subgraph selection strategies. The FPT algorithm uses the greedy lower bound and is executed with the known optimal cost.

	First found	Most marked	Most adjacent
No lower bound	331	**482**	337
Greedy	564	648	**677**
Local search	534	623	**658**

Table 4.1: Number of solved instances for a specific lower bound and selection strategy. The FPT algorithm is executed with known optimal editing cost and with a time limit of 100 seconds. The numbers can only be interpreted relatively because the “hardness” of instances does not scale uniformly. The maximum number of solvable instance is 1058. The best values for each lower bound is marked bold.

The effect of the lower bound is not independent of the other methods, like the subgraph selection strategies. In the next section, we cover subgraphs selection strategies and also the interaction with the lower bound algorithms.

4.4 Subgraph Selection

Selecting forbidden subgraphs for branching decides which subproblem is being solved first. Figure 4.9 depicts the influence of the subgraph selection strategies on the running time of the FPT algorithm. Selecting the most adjacent subgraphs results in a speedup of up to one order of magnitude for harder instances in comparison to just choosing the first forbidden induced subgraph that is found. Although the selection strategies manage to speed up the algorithm, the running time benefit is weaker than that of the lower bounds.

We already noted that the effect of the subgraph selection strategies and lower bound algorithms on the FPT algorithm are not independent. In Table 4.1 the number of solved instances for a specific lower bound and selection strategy is denoted. When no lower bound algorithm is used, the strategy of selecting the subgraph with most marked vertex pairs is significantly outperforming the others. We have to be careful with the interpretation of the absolute numbers. The absolute differences are not meaningful because we can make no statement about the “hardness” of each additionally solved instance. Nevertheless, we can compare their relative order. For the greedy and local search lower bounds, choosing the subgraph with the most adjacent other subgraphs is effective. In the following, we give a possible explanation.

Recall that we defined two forbidden induced subgraphs to be adjacent if they share an unmarked vertex pair. Consider a single vertex pair, that is shared by multiple subgraphs. If it is the only vertex pair shared by the subgraphs, marking it corresponds to making them no longer adjacent in the corresponding MWIS instance. This allows inserting more subgraphs into the packing and results in a better lower bound and therefore in a better running time. Although this observation is true for all marked vertex pairs, choosing the ones to edit (and mark), which have the most incident subgraphs, maximizes the number of adjacencies that are removed.

The "most marked" strategy helps to keep the search tree small. If more vertex pairs are marked in a forbidden subgraph, then the number of branches is reduced. This improvement can be seen when no lower bound is being used. Even with lower bounds, the algorithm is significantly more performant than just choosing any forbidden subgraph. Nevertheless, it fails to improve the subgraphs packing as much as the "most adjacent" strategy. The ability to improving the packings seems to be the dominating factor for the running time of the FPT algorithm.

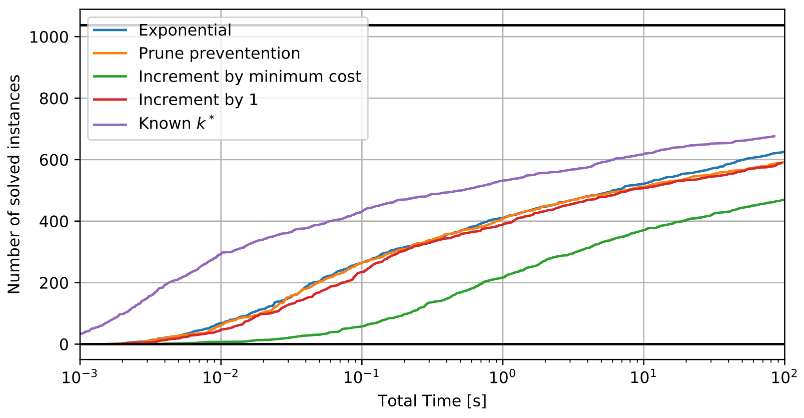
4.5 Search Strategies

In this section, we evaluate the search strategies discussed in Section 3.4. First, we take a look at the running time of the FPT algorithm. Next, we investigate the growth rate of the search strategies.

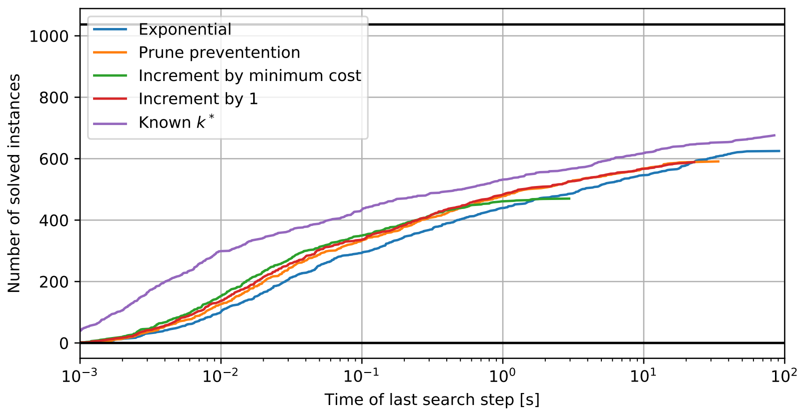
One major challenge for Weighted $\mathcal{F}$-free Edge Editing is "proving" that a given solution is optimal. For unweighted $\mathcal{F}$-free Edge Editing, the algorithm is executed for increasing editing costs $k = 0, 1$, etc. If a solution with editing cost k has been found and no solutions for $k-1$ have been found, we can deduce that the solution must be optimal and stop the search. This especially useful for subgraph selection strategies for branching which aim to prioritize branching options that most likely lead to a solution. When a solution for Weighted $\mathcal{F}$-free Edge Editing has been found, we can only use its editing cost as an upper bound. With this upper bound the search tree can be further pruned but the FPT algorithm still has to either prune or explore each branch in the search tree to make sure that no solution with lesser editing cost exists. Additionally to the search strategies for the editing cost from Section 3.4, we present a variant of the FPT algorithm that already knows the optimal editing cost k^*. Algorithm 3.1 is only executed once for (G, k^*) and stops as soon as one solution has been found. In Section 5 we describe possible variants of Weighted $\mathcal{F}$-free Edge Editing that produce a solution that solves the problem approximately or enumerate all inclusion-minimal solutions up to a given editing cost.

Knowing the optimal editing cost before the execution of the algorithm does not apply to actual data. We use it in our evaluation because it is better than any search strategy. It only executes the FPT algorithm once with the optimal editing cost and the first found solution is guaranteed to be optimal. Figure 4.10a depicts the influence of different search strategies on the running time of the FPT algorithm. Estimating the number of calls by an exponential model, preventing the pruning of branches and incrementing by 1 have nearly the same running time. Incrementing by the smallest editing cost is the worst strategy and fitting an exponential growth model is only slightly better than the other two strategies. Both "incrementing by the smallest editing cost" and "incrementing by 1" depend on the scaling of the cost function. A single (non-)edge with a very small editing cost or multiplying all costs by a large number could severely affect the running time of the algorithm. Most editing costs of instances from the `bio` dataset are between 0 and 10. The "natural" scale of one seems to be not a bad choice.

Ideally, the last search step dominates the total running time. We take a look at the running time of the last search step of each instance for the different search strategies.



(a) Number of solved instances for a certain total running time of all search steps.



(b) Number of solved instances for a certain running time of the last search step.

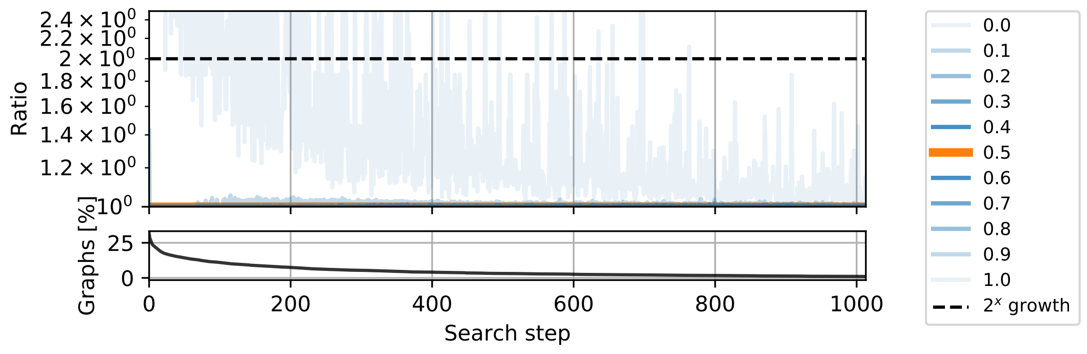
Figure 4.10: Number of solved instances for a certain running time comparing different search strategies. The FPT algorithm uses the greedy lower bound and subgraphs with most adjacent subgraphs are selected.

Figure 4.10b shows the number of solved instances with the running time of the last search step. When the optimal editing cost is known beforehand, the curve is the same as in Figure 4.10a because the algorithm is only executed once. The strategy of incrementing by the minimum editing cost solves the most instances for a given maximum running time of the last search step. This is probably the effect that small search steps lead to a final editing cost which is only slightly larger than the optimal editing cost. Furthermore, the "exponential" strategy is now performing slightly worse than the rest. This implies that the search strategy overestimates the optimal editing cost for the last search step. When we compare Figure 4.10a and Figure 4.10b, we see that the curve for the exponential strategy is not significantly different. This suggests that this search strategy is successful with having the last search step dominate the total running time. Nevertheless, when the FPT algorithm is executed with the optimal editing cost k^*, the algorithm is much faster. As previously discussed, the algorithm for known k^* can terminate after the first solution has been found. This explains the gap to the other algorithms.

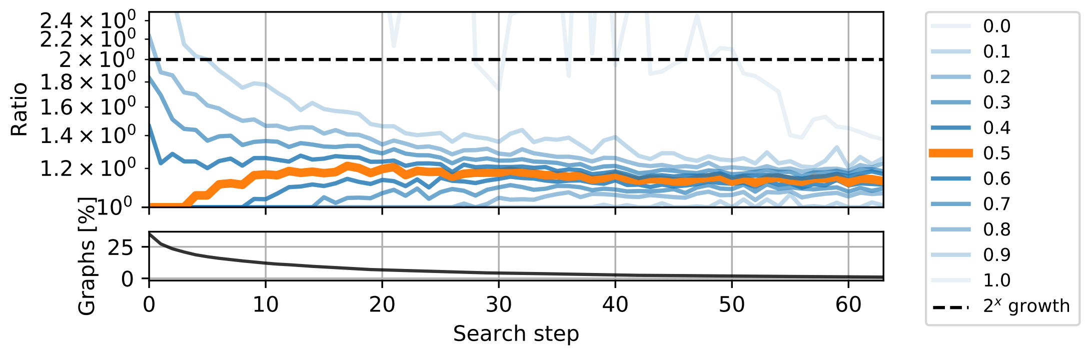
Although the total running time of the FPT algorithm only differs slightly for the different search strategies, we want to take a closer look at their behavior. An ideal algorithm would guess the optimal editing cost exactly and the FPT algorithm would be only called once with that parameter. As we cannot expect search strategies to succeed on the first try, we analyze the behavior of multiple search steps. The algorithm starts with an initial lower bound and solves the problem for increasing values of the parameter k. The size of the search tree grows with the parameter and with each editing step. We can compare the size of the search tree, i.e. the number of recursive calls, between editing steps by taking a look at the ratio of calls between the search steps. A good search strategy would make significant progress with each search step, but not "overshoot" the optimal editing cost too much. Another desired property is that the search strategy minimizes the number of "costly" search steps, near the optimal editing cost. As a result, the running time is dominated by only one (the last) search step. This can be achieved by for example doubling the number of calls for each search step, i.e. having a local growth behavior like the function 2^i for search step i.

The ratios for all instances of the `bio` dataset and all search strategies are visualized in Figure 4.11. The ratios are plotted against the search steps taken by each strategy. The data is summarized by their quantiles. The first thing to note is that the number of search steps needed differs widely. About 1% of the instances needed more than 1014 steps when the strategy is incrementing by the minimum editing cost. Meanwhile, about 90% of the instances need less than 16 steps when using the exponential search strategy. The strategies other than incrementing by the minimum cost, consistently make some progress for most of the instances, i.e. the median growth ratio is larger than 1. The exponential growth estimation strategy is configured to target a growth ratio of two and the quantiles do converge to a ratio of two after only a few steps. This is a trade-off between making enough progress, but not too much to overshoot the optimal editing cost.

Unfortunately the exponential growth search strategy "overshoots" for some search steps. There are even search steps with ratios larger than 100. This indicates that the maximum step size is not bounded strongly enough or the assumption of exponential growth does not hold locally. Figure 4.12 shows the relationship of the allowed editing cost k with the number of calls for eight permutations of one instance. We see a horizontal segment where the number of calls does not increase with k. This contradicts the exponential growth assumption. The assumption only starts to hold after the initial horizontal section. The update rule from Equation 3.6 cannot be applied when no slope is present. The strategy must resort to the upper bound constraint. The knowledge of such plateaus could be used to further improve the search strategy.

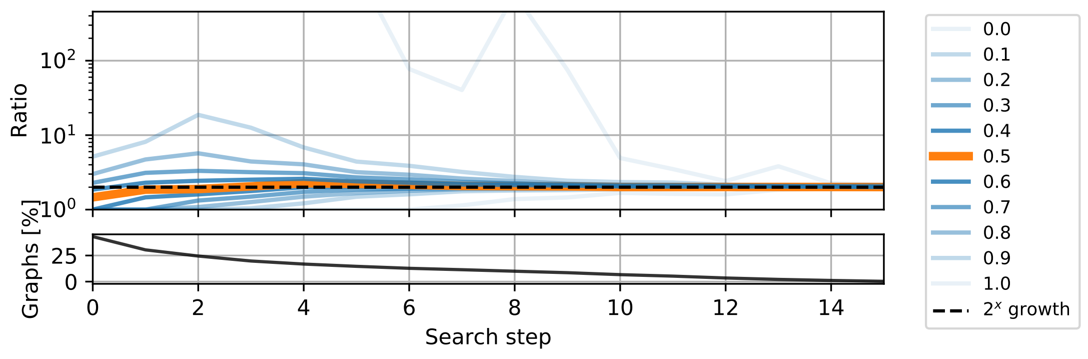


(a) Incrementing by minimum editing cost. At most 3536 steps were needed. More than 99% of the instances needed less than 1014 steps. The average growth rate is 1.

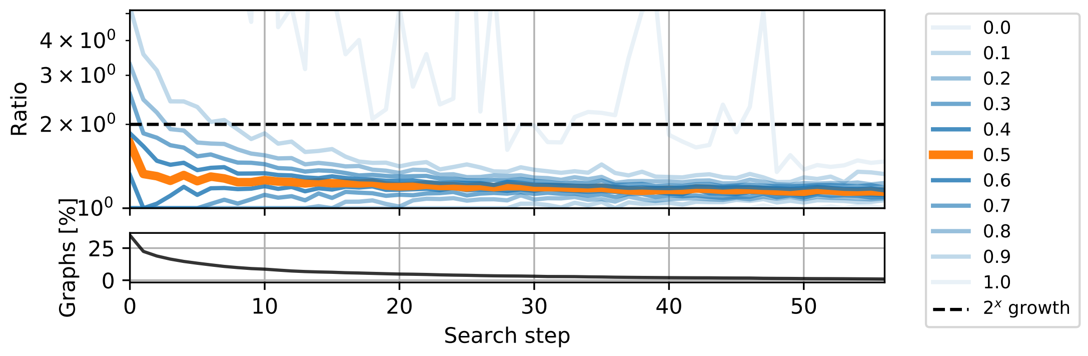


(b) Incrementing by 1. At most 720 steps were needed. More than 99% of the instances needed less than 64 steps. The average growth rate is 1.14.

Figure 4.11: Quantiles of the ratios between the number of calls for consecutive search steps. A desirable growth of 2^x is plotted as a horizontal line. Some graphs only need a few search steps. The percentage of graphs that needed up to a given search step, is visualized in the bottom figures. The x-axis is restricted such that the search steps that were being done by less than 1% are cut off. Data from `bio` instances, the "most adjacent" subgraph selection strategy and the greedy lower bound is used.



(c) "Exponential" search strategy with target growth of 2^x. At most 17 steps were needed. More than 99% of the instances needed less than 16 steps. The growth rate approaches the target after a few steps.



(d) "Prune prevention" search strategy that chooses the median. At most 94 steps were needed. More than 99% of the instances needed less than 57 steps. The average growth rate is 1.21.

Figure 4.11: The ratio between the number of calls for consecutive search steps (continued).

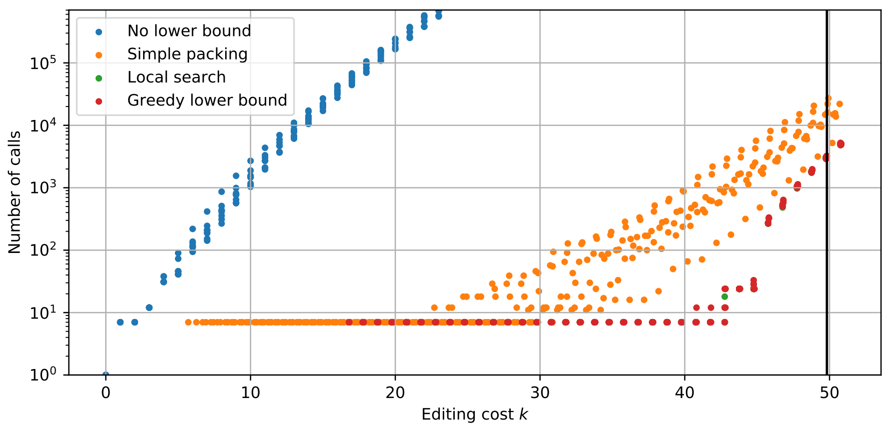


Figure 4.12: Example of a relationship between editing cost k and the number of calls from eight permutations of instance no. 161 from the `bio` dataset for $\mathcal{F} = \{C_4, P_4\}$. The evaluation points are evenly spaced with distance one after the initial lower bound. The greedy and local-search-based lower bounds have nearly the same number of calls. The horizontal segments up to the bends at $k = 30$ and 43 do not follow the exponential growth assumption.

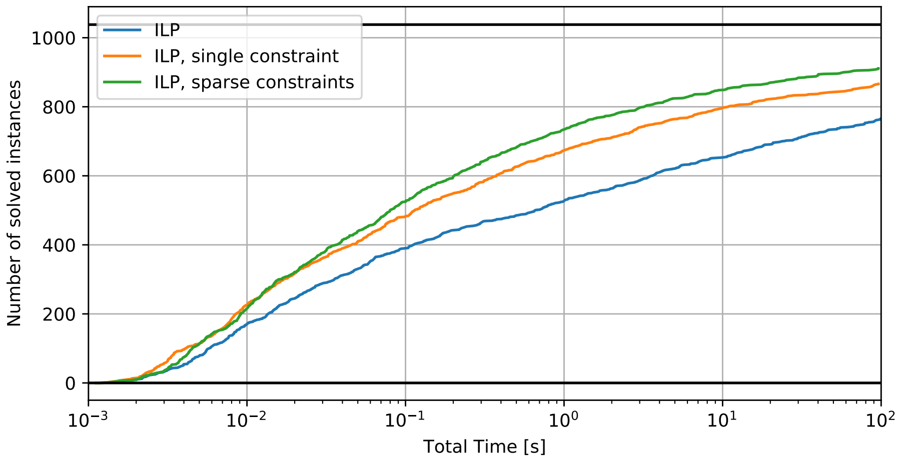


Figure 4.13: The number of solved instances for a certain total running time, comparing the constraint restrictions.

4.6 The ILP Algorithm and Comparison to the FPT Algorithm

First, we take a look at the ILP algorithm. Then we finally compare the two basic algorithms for WEIGHTED $\mathcal{F}$-FREE EDGE EDITING discussed in this thesis: the search-tree based FPT algorithm and the ILP algorithm based on the formulation of the problem as a linear program. Looking back at Figure 4.3, we can see that the ILP solver has some significant start-up cost. The FPT algorithm is faster for easy instances, but this effect ceases to matter for harder instances.

First, we take a look at variants of the ILP algorithm. We restricted the number of constraints generated in each callback with the methods discussed in Section 3.5.2. The comparison is shown in Figure 4.13. The basic ILP algorithm does not restrict any constraint generation, the “single” variant only generates a single constraint and the “sparse” variant generates at most $\binom{n}{2}$ constraints. Using either restriction method results in more instances that are solvable within a given time limit. The “sparse” variant can solve 935 of the 1058 instances within a 100 seconds time limit. Consequently, the sparse row generation restriction is the best performing variant of the ILP algorithm.

Not restricting constraint generation at all, seems to generate too many constraints. Only generating one constraint improves on that and restricting the generation with the “sparse” method and potentially adding multiple constraints is even better. However, we do not know whether the “sparse” variant with at most $\binom{n}{2}$ constraints is too restrictive or generates too many constraints. In one of the earliest publications, the generated constrains are restricted by a constant amount: either 400 or 500 constraints were added [GW89]. Other methods for restricting constraint generation are also possible and an opportunity for future work.

In Figure 4.14 variants of the FPT and ILP algorithms are compared. In contrast to Figure 4.3, only the hard instances are used for comparison. In the figure, the ILP algorithm is considered without restrictions and with sparse constraints. The FPT algorithm is considered once with the “base” variant (no lower bound, selecting the first subgraph that is found and incrementing by one) and twice with the “best” configuration (greedy lower bound, “most adjacent” subgraph selection). One of the “best” FPT variants is executed with the known optimal editing cost and one used the “exponential” search strategy. Although the FPT algorithm benefits significantly from the speed-up techniques, the ILP

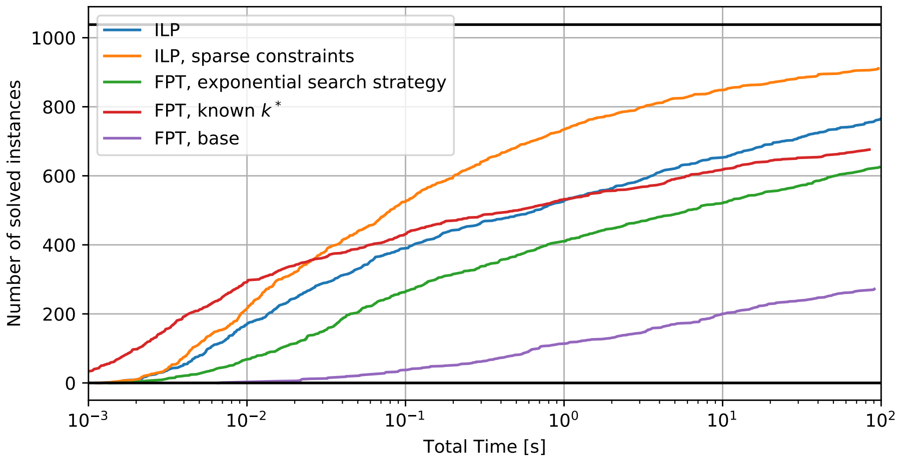
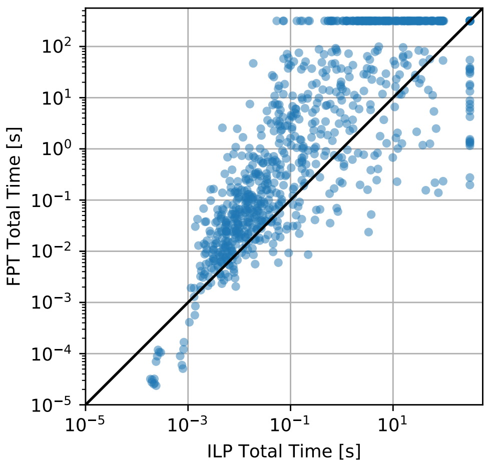


Figure 4.14: The number of solved instances for a certain total running time, comparing the ILP and FPT algorithm. The basic FPT algorithm uses no lower bound, branches on the first subgraph found and increments the editing cost by one in each search step. The other two FPT algorithms use the greedy lower bound and the “most adjacent” subgraph selection strategy.

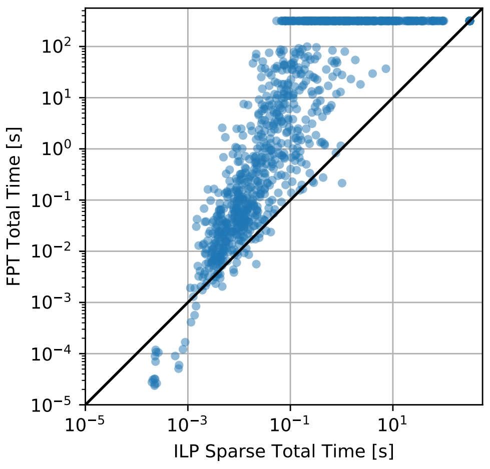
algorithm is still better. Only for small time limits of up to 10^{-2}, the FPT algorithm with known optimal editing cost manages to solve the most instances.

We conclude, that the ILP algorithm outperforms the FPT algorithm in our evaluation.

Furthermore, we investigate whether both the ILP and FPT algorithms struggle with the same instances. Figure 4.15 compares the running times of two versions of the ILP algorithm against one version of the FPT algorithm by plotting the total time they took for each instance. Points above the line are solved faster by the ILP algorithm than the FPT algorithm. The “best” FPT algorithm without known optimal editing cost (greedy lower bound, “most adjacent” subgraph selection, “exponential” search strategy) is used and the ILP algorithm without restricted constraint generation is used in Figure 4.15a and the ILP algorithm with sparse constraints is used in Figure 4.15b. In Figure 4.15a, we can see in that some instances are solved by the FPT algorithm, but not by the ILP algorithm. There exist a few instances which are solved within one second by the FPT algorithm, but fail to finish within the time limit of 100 seconds with the ILP algorithm. Nevertheless, most instances are solved faster by the basic ILP algorithm. Looking at Figure 4.15b, this small advantage of the FPT algorithm for some instances ceases to exist when the improved ILP algorithm with sparse constraints is used. All instances are either solved by both algorithms, by the ILP algorithm or not solved at all. While they achieve similar running times for easy instances which are solved within 10^{-2} seconds, the ILP is up to 3 orders of magnitude faster for harder instances.



(a) The basic ILP vs. the FPT algorithm. Some instances are only solved by the the FPT algorithm.



(b) The ILP algorithm with sparse constraints vs. the FPT algorithm.

Figure 4.15: Running times per instance for ILP and FPT algorithm. The FPT algorithm uses the greedy lower bound, the "most adjacent" selection strategy and the "exponential" search strategy. Instances which did not finish within 100 seconds are displayed at 300 seconds.

5. Conclusion

We investigated algorithms for WEIGHTED $\mathcal{F}$-FREE EDGE EDITING, generalizing previous work for unweighted $\mathcal{F}$-FREE EDGE EDITING with methods applicable to any finite set of forbidden subgraphs $\mathcal{F}$. All algorithms were evaluated on the protein-protein interaction data from [RWB+07, BBBT07]. Also, we present and evaluate algorithms for listing forbidden induced subgraphs for cycles and paths.

We adapted lower bound methods based on vertex pair disjoint packings and find that a weight-aware adaption of a greedy packing lower bound algorithm results in the best running times. Although a local-search-based lower bound has been successfully used for unweighted $\mathcal{F}$-FREE EDGE EDITING by the authors of [GHS+20], it did not outperform the greedy algorithm. We assume that is a problem with local maxima. Also performing plateau search seems to more efficient for unweighted editing. Letting the FPT algorithm branch on forbidden induced subgraphs with the most other adjacent subgraphs, improved the lower bounds.

Finding the optimal editing cost for the FPT algorithm is non-trivial for WEIGHTED $\mathcal{F}$-FREE EDGE EDITING. We discussed two previously used methods and introduced two new methods. We evaluated these search strategies and analyze the behavior of their search steps. The “exponential” search strategy proved to be marginally better than the other strategies and is independent of the scaling of the cost function.

Also, we investigated an ILP algorithm and methods to restrict the generation of constraints. Our evaluation shows that the restriction methods significantly speed up the ILP algorithm and the sparse restriction work the best. In a comparison of the solving algorithms, the ILP algorithm manages to outperform the variants of the FPT algorithm.

Future Work

Other lower bound methods adapted from algorithms for the MAXIMUM WEIGHTED INDEPENDENT SET PROBLEM are also possible. Some MWIS heuristics do not only consider the weight, but also the degree of vertices. These could be adapted for packing-based lower bound algorithms.

The ILP algorithm can probably be further improved by adding constraints for fractional solutions. Another technique for keeping the linear program small enough to be solved efficiently by an ILP solver is *row elimination*, where constraints that are no longer active are removed from the model.

Instead of further investigating algorithms for Weighted $\mathcal{F}$-free Edge Editing, one could also investigate different variants of the problem itself. The FPT algorithm can enumerate all inclusion-minimal solutions, which is not easily possible with the ILP algorithm. This gives rise to several variants of Weighted $\mathcal{F}$-free Edge Editing, like finding all near optimal inclusion-minimal solutions, i.e. such that $k \leq (1+\varepsilon) \cdot k^*$, for the optimal editing cost k^* and some $\varepsilon > 0$. The FPT algorithm can also be used to search for a single solution that is at most some given amount worse than the optimal solution.

Bibliography

[ARW12] Diogo V. Andrade, Mauricio G. C. Resende, and Renato F. Werneck. Fast local search for the maximum independent set problem. *Journal of Heuristics*, 18(4):525–547, August 2012. doi:10.1007/s10732-012-9196-4.

[ASS16] N. R. Aravind, R. B. Sandeep, and Naveen Sivadasan. Parameterized Lower Bounds and Dichotomy Results for the NP-completeness of H-free Edge Modification Problems. In *LATIN 2016: Theoretical Informatics*, Lecture Notes in Computer Science, pages 82–95, Berlin, Heidelberg, 2016. Springer. doi:10.1007/978-3-662-49529-2_7.

[BBBT07] Sebastian Böcker, Sebastian Briesemeister, Quang Bao Anh Bui, and Anke Trub. A fixed-parameter approach for Weighted Cluster Editing. In *Proceedings of the 6th Asia-Pacific Bioinformatics Conference*, pages 211–220, Kyoto, Japan, December 2007. doi:10.1142/9781848161092_0023.

[BBK11] Sebastian Böcker, Sebastian Briesemeister, and Gunnar W. Klau. Exact Algorithms for Cluster Editing: Evaluation and Experiments. *Algorithmica*, 60(2):316–334, June 2011. doi:10.1007/s00453-009-9339-7.

[BHK15] Sharon Bruckner, Falk Hüffner, and Christian Komusiewicz. A graph modification approach for finding core–periphery structures in protein interaction networks. *Algorithms for Molecular Biology*, 10(1):16, December 2015. doi:10.1186/s13015-015-0043-7.

[BHSW15] Ulrik Brandes, Michael Hamann, Ben Strasser, and Dorothea Wagner. Fast Quasi-Threshold Editing. In *Algorithms - ESA 2015*, Lecture Notes in Computer Science, pages 251–262, Berlin, Heidelberg, 2015. Springer. doi:10.1007/978-3-662-48350-3_22.

[Boh15] Felix Bohlmann. *Graphclustern durch Zerstören langer induzierter Pfade*. Bachelor thesis, Technische Universität Berlin, 2015. URL: http://fpt.akt.tu-berlin.de/publications/theses/BA-felix-bohlmann.pdf.

[Bö12] Sebastian Böcker. A golden ratio parameterized algorithm for Cluster Editing. *Journal of Discrete Algorithms*, 16:79–89, October 2012. doi:10.1016/j.jda.2012.04.005.

[Cai96] Leizhen Cai. Fixed-parameter tractability of graph modification problems for hereditary properties. *Information Processing Letters*, 58(4):171–176, May 1996. doi:10.1016/0020-0190(96)00050-6.

[CC12] Yixin Cao and Jianer Chen. Cluster Editing: Kernelization Based on Edge Cuts. *Algorithmica*, 64(1):152–169, September 2012. doi:10.1007/s00453-011-9595-1.

[CM12] Jianer Chen and Jie Meng. A 2k kernel for the cluster editing problem. *Journal of Computer and System Sciences*, 78(1):211–220, January 2012. doi:10.1016/j.jcss.2011.04.001.

[Dam10] Peter Damaschke. Fixed-Parameter Enumerability of Cluster Editing and Related Problems. *Theory of Computing Systems*, 46(2):261–283, February 2010. `doi:10.1007/s00224-008-9130-1`.

[DP18] Pål Grønås Drange and Michał Pilipczuk. A Polynomial Kernel for Trivially Perfect Editing. *Algorithmica*, 80(12):3481–3524, December 2018. `doi:10.1007/s00453-017-0401-6`.

[dRo18] H.N. de Ridder and others. Information System on Graph Classes and their Inclusions (ISGCI), December 2018. URL: `http://graphclasses.org`.

[GGHN04] Jens Gramm, Jiong Guo, Falk Hüffner, and Rolf Niedermeier. Automated Generation of Search Tree Algorithms for Hard Graph Modification Problems. *Algorithmica*, 39(4):321–347, August 2004. `doi:10.1007/s00453-004-1090-5`.

[GHS$^+$20] Lars Gottesbüren, Michael Hamann, Philipp Schoch, Ben Strasser, Dorothea Wagner, and Sven Zühlsdorf. Engineering Exact Quasi-Threshold Editing. *Proceedings of the 18th International Symposium on Experimental Algorithms (SEA 2020)*, 160:7:1–7:14, 2020. Accepted for publication, preliminary version available. URL: `https://arxiv.org/abs/2003.14317`.

[Gol64] Arthur Stanley Goldberger. *Econometric Theory*. Probability and Statistics: Applied Probability and Statistics. John Wiley & Sons, Incorporated, 1964.

[Guo09] Jiong Guo. A more effective linear kernelization for cluster editing. *Theoretical Computer Science*, 410(8-10):718–726, March 2009. `doi:10.1016/j.tcs.2008.10.021`.

[Gur] Gurobi Optimization, LLC. Gurobi Optimizer Version 8.1.1. URL: `https://www.gurobi.com`.

[GW89] M. Grötschel and Y. Wakabayashi. A cutting plane algorithm for a clustering problem. *Mathematical Programming*, 45(1-3):59–96, August 1989. `doi:10.1007/BF01589097`.

[HH15] Sepp Hartung and Holger H. Hoos. Programming by Optimisation Meets Parameterised Algorithmics: A Case Study for Cluster Editing. In *Learning and Intelligent Optimization*, Lecture Notes in Computer Science, pages 43–58, Cham, 2015. Springer International Publishing. `doi:10.1007/978-3-319-19084-6_5`.

[HKSS13] Chính T. Hoàng, Marcin Kamiński, Joe Sawada, and R. Sritharan. Finding and listing induced paths and cycles. *Discrete Applied Mathematics*, 161(4):633–641, March 2013. `doi:10.1016/j.dam.2012.01.024`.

[HSS08] Aric A Hagberg, Daniel A Schult, and Pieter J Swart. Exploring Network Structure, Dynamics, and Function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy 2008)*, pages 11–15, 2008. URL: `http://conference.scipy.org/proceedings/SciPy2008/paper_2/full_text.pdf`.

[KM86] Mirko Křivánek and Jaroslav Morávek. NP-hard problems in hierarchical-tree clustering. *Acta Informatica*, 23(3):311–323, June 1986. `doi:10.1007/BF00289116`.

[LCH$^+$07] Chuan Lin, Young-rae Cho, Woo-chang Hwang, Pengjun Pei, and Aidong Zhang. Clustering Methods in a Protein–Protein Interaction Network. In Xiaohua Hu and Yi Pan, editors, *Knowledge Discovery in Bioinformatics: Techniques, Methods and Application*, Wiley Series in Bioinformatics, pages 319–355. John Wiley & Sons, July 2007. `doi:10.1002/9780470124642.ch16`.

[NG13] James Nastos and Yong Gao. Familial groups in social networks. *Social Networks*, 35(3):439–450, July 2013. `doi:10.1016/j.socnet.2013.05.001`.

[NPS18] Bruno Nogueira, Rian G. S. Pinheiro, and Anand Subramanian. A hybrid iterated local search heuristic for the maximum weight independent set problem. *Optimization Letters*, 12(3):567–583, May 2018. `doi:10.1007/s11590-017-1128-7`.

[RWB$^+$07] Sven Rahmann, Tobias Wittkop, Jan Baumbach, Marcel Martin, Anke Truß, and Sebastian Böcker. Exact and Heuristic Algorithms for Weighted Cluster Editing. In *Computational Systems Bioinformatics*, pages 391–401, University of California, San Diego, USA, September 2007. `doi:10.1142/9781860948732_0040`.

[SAP$^+$] Jeremy Siek, Chuck Allison, Gennaro Prota, Ahmed Charles, Glen Fernandes, and Riccardo Marcangelo. Boost dynamic_bitset module. URL: `http://boost.org/libs/dynamic_bitset`.

[Sch15] Philipp Schoch. *Editing to (P5, C5)-free Graphs - a Model for Community Detection?* Bachelor thesis, Karlsruhe Institute of Technology, 2015. URL: `https://i11www.iti.kit.edu/_media/teaching/theses/ba-schoch-15.pdf`.

[Wol65] E. S. Wolk. A Note on "The Comparability Graph of a Tree". *Proceedings of the American Mathematical Society*, 16(1):17–20, 1965. `doi:10.2307/2033992`.

[Zü17] Sven Zühlsdorf. Engineering FPT-based Edge Editing Algorithms. Master's thesis, Karlsruhe Institute of Technology, 2017. URL: `https://i11www.iti.kit.edu/_media/teaching/theses/ma-zuehlsdorf-17.pdf`.