Scalable Graph Algorithms using Practically Efficient Data Reductions

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

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

genehmigte

Dissertation

von

Sebastian Emanuel Lamm

Tag der mündlichen Prüfung: 18. Juli 2022

1. Referent: Prof. Dr. Peter Sanders
Karlsruher Institut für Technologie
Deutschland

2. Referent: Prof. Dr. Jin-Kao Hao
Université d'Angers
Frankreich

To my parents, Franz and Claudia Lamm

Abstract

This dissertation presents both heuristic and exact approaches for the NP-hard optimization problems of finding maximum cardinality and weight independent sets, as well as finding maximum cardinality cuts. An independent set of a graph is a subset of vertices such that no pair of vertices in this set is adjacent. A maximum cardinality independent set is an independent set of maximal cardinality among all possible independent sets. If one is additionally given a vertex weighting function for this graph, a maximum weight independent set of this graph is an independent set of maximal weight. Finally, a maximum cardinality cut in a graph is a bipartition of the vertices such that the number of edges running across the partitions is maximal. All three of these problems are important for a variety of real-world applications. For example, maximum or high-quality cardinality and weight independent sets are used in map labeling [Klu+19; GNN13], modeling protein-protein interactions [GWA00], or vehicle routing [Don+22]. Examples for the usage of maximum or large cuts include social network modeling [Har59], statistical physics [Bar82], or VLSI design [Bar+88; Chi+07].

In this work, we discuss and further the usage of reduction rules for all these problems. Reduction rules are graph transformations that are able to generally reduce the size of a given input while also maintaining optimality, i.e., an optimal solution of the reduced instance can be extended to an optimal solution of the original input. In this dissertation, we also present inexact reduction rules that remove vertices that are likely to be a part of (or excluded from) a solution. We show that these types of reduction rules can drastically improve the performance of heuristic algorithms while still leading to high-quality solutions. Finally, we present graph transformations that maintain optimality but also temporarily increase the graph size. Counterintuitively, this can lead to new, easier to reduce structures and subsequently an overall reduction in size in the long run.

We propose multiple algorithms that incorporate these concepts into a wide spectrum of techniques. Our work on the maximum cardinality independent set problem includes an evolutionary algorithm that uses a combination of exact and inexact reduction rules to gradually shrink the graph size. We also propose an advanced local search algorithm that improves an existing state-of-the-art algorithm with reduction rules to very quickly compute high-quality independent sets. Next, we present a portfolio algorithm that won the PACE Challenge 2019 by using multiple existing approaches for different closely related problems. We then develop and evaluate multiple advanced branching rules for a state-of-the-art branch-and-reduce algorithm. For maximum weight independent sets, we present multiple new reduction rules and graph transformations that we then use in our newly developed branch-and-reduce algorithm. Finally, for maximum cardinality cuts, we also

propose new reduction rules that are used to build an efficient preprocessing algorithm to boost the performance of state-of-the-art approaches, both heuristic and exact.

We evaluated all our algorithms on a large set of instances stemming from multiple domains and applications for the corresponding problems. In general, our experiments show that our algorithms are able to significantly increase both the scale and speed at which instances can be processed in practice (by up to orders of magnitude). Furthermore, we show that our preprocessing algorithms and reductions can easily be integrated into other algorithms to improve their performance. Our contributions are either available as standalone libraries or as part of the libraries KaMIS, WeGotYouCovered (maximum cardinality and weight independent sets), and DMAX (maximum cardinality cuts).

Deutsche Zusammenfassung

Diese Dissertation behandelt sowohl heuristische als auch exakte Ansätze für die NP-schweren Optimierungsprobleme des Findens kardinalitätsmaximaler und gewichtsmaximaler unabhängiger Mengen, sowie des Findens kardinalitätsmaximaler Schnitte. Eine unabhängige Menge eines Graphen ist eine Teilmenge der Knoten, sodass kein Paar von Knoten in dieser Menge zueinander adjazent ist. Eine kardinalitätsmaximale unabhängige Menge ist eine unabhängige Menge maximaler Kardinalität aus der Menge aller möglichen unabhängigen Mengen. Besitzt der Graph zusätzlich eine Knotengewichtsfunktion, so ist eine gewichtsmaximale unabhängige Menge dieses Graphen eine unabhängige Menge mit maximalem Gewicht. Ein kardinalitätsmaximaler Schnitt in einem Graphen ist eine Bipartition der Knoten, sodass die Anzahl der Kanten, die zwischen den Partitionen verlaufen, maximal ist. Alle drei Probleme sind für eine Vielfalt von Anwendungen von Bedeutung. Beispielsweise werden unabhängige Mengen mit maximalem oder hochqualitativem Gewicht für das Beschriften von Karten [Klu+19; GNN13], der Modellierung von Proteininteraktionen [GWA00] oder Tourenplanung [Don+22] verwendet. Beispiele für die Verwendung von maximalen oder großen Schnitten beinhalten die Modellierung von sozialen Netzwerken [Har59], statistische Physik [Bar82] oder VLSI Design [Bar+88; Chi+07].

In dieser Arbeit wird die Verwendung von Reduktionsregeln für diese Probleme diskutiert und weiterentwickelt. Reduktionsregeln sind Graphtransformationen, die die Größe einer gegebenen Eingabe im Allgemeinen reduzieren und gleichzeitig die Optimalität beibehalten, d.h., eine optimale Lösung der reduzierten Instanz kann zu einer optimalen Lösung der ursprünglichen Eingabe erweitert werden. In dieser Dissertation stellen wir auch inexakte Reduktionsregeln vor, die Knoten entfernen, die wahrscheinlich Teil einer Lösung sind (oder von ihr ausgeschlossen werden können). Wir zeigen, dass diese Art von Reduktionsregeln zu drastischen Verbesserungen der Leistung heuristischer Algorithmen führen kann und trotzdem hochqualitative Lösungen erreicht werden. Zusätzlich stellen wir Graphentransformationen vor, die die Optimalität beibehalten, aber auch vorübergehend die Graphgröße erhöhen. Dieser kontraintuitive Ansatz kann zu neuen, leichter zu reduzierenden Strukturen und damit langfristig zu einer Verkleinerung des Graphen führen.

Unsere Arbeit für das Problem des Findens kardinalitätsmaximaler unabhängiger Mengen beinhaltet einen evolutionären Algorithmus, der eine Kombination aus exakten und inexakten Reduktionsregeln verwendet, um schrittweise den Graphen zu verkleinern. Wir präsentieren zusätzlich einen fortschrittlichen Algorithmus basierend auf lokaler Suche, der einen bestehenden State-of-the-Art Algorithmus mit Reduktionsregeln verbessert, um sehr schnell hochqualitative unabhängige Mengen zu berechnen. Als Nächstes stellen wir einen Portfolio-Algorithmus vor, der die PACE Challenge 2019 gewonnen hat und mehrere

bestehende Ansätze für eng verwandte Probleme miteinander kombiniert. Anschließend entwickeln und evaluieren wir verschiedene fortschrittlichen Verzweigungsregeln für einen State-of-the-Art Branch-and-Reduce Algorithmus. Für das Problem des Findens gewichtsmaximaler unabhängiger Mengen stellen wir mehrere neue Reduktionsregeln und Graphtransformationen vor, die wir in unserem neu entwickelten Branch-and-Reduce Algorithmus verwenden. Schließlich schlagen wir für das Problem des Findens kardinalitätsmaximaler Schnitte erneut neue Reduktionsregeln vor, die für einen effizienten Vorverarbeitungsalgorithmus verwendet werden, um die Leistung von sowohl heuristischen als auch exakten State-of-the-Art Algorithmen zu steigern.

Wir haben all unsere Algorithmen mit einer großen Anzahl von Instanzen aus verschiedenen Domänen und Anwendungen für die entsprechenden Probleme evaluiert. Im Allgemeinen zeigen unsere Experimente, dass unsere Algorithmen in der Lage sind, sowohl die Größe als auch die Geschwindigkeit, mit der Instanzen in der Praxis verarbeitet werden können, um bis zu mehreren Größenordnungen zu verbessern. Zusätzlich zeigen wir, dass unsere Vorverarbeitungsalgorithmus und Reduktionen leicht in andere Algorithmen integriert werden können, um deren Leistung zu steigern. Unsere Beiträge werden entweder als eigenständige Bibliotheken oder als Teil der Bibliotheken KaMIS, WeGotYouCovered (kardinalitätsmaximale und gewichtsmaximale unabhängige Mengen) und DMAX (kardinalitätsmaximale Schnitte) zur Verfügung gestellt.

Acknowledgements

The last years leading to this dissertation presented me with a plethora of opportunities that helped me grow, both academically and personally. Thus, I would like to briefly express my gratitude to all people directly or indirectly involved in this journey.

First and foremost, I want to give my sincerest thanks to my supervisor Peter Sanders for letting me be a part of his amazing research group, as well as for providing me with guidance and the freedom to pursue my research interests. I would also like to thank Jin-Kao Hao for being a part of my dissertation committee as a reviewer for this dissertation. Thanks go to Monika Henzinger for inviting me to her research group in Vienna and providing me with valuable feedback on my work.

I want to give special thanks to Christian Schulz and Darren Strash who were close colleagues and friends for the longest part of my academic journey. Without you two, I probably would not have ventured into academia and arrived at this point.

Furthermore, I would like to thank my co-authors Faisal Abu-Khzam, Michael Axtmann, Timo Bingmann, Ulrik Brandes, Carsten Dachsbacher, Jakob Dahlum, Damir Ferizovic, Daniel Funke, Alexander Gellner, Michael Hamann, Demian Hespe, Lorenz Hübschle-Schneider, Emanuel Jöbstl, Ulrich Meyer, Matthias Mnich, Huyen Chau Nguyen, Alexander Noe, Manuel Penschuck, Sebastian Schlag, Emanuel Schrade, Matthias Stumpp, Tobias Sturm, Ilya Safro, Peter Sanders, Christian Schorr, Christian Schulz, Darren Strash, Moritz von Looz, Renato Werneck, Robert Williger, Bogdan Zaválnij, and Huashuo Zhang.

Thanks goes to my former and current colleagues Yaroslav Akhremtsev, Michael Axtmann, Tomáš Balyo, Timo Bingmann, Daniel Funke, Simon Gog, Demian Hespe, Tobias Heuer, Lukas Hübner, Lorenz Hübschle-Schneider, Markus Iser, Florian Kurpicz, Moritz Laupichler, Hans-Peter Lehmann, Tobias Maier, Matthias Schimek, Sebastian Schlag, Dominik Schreiber, Daniel Seemaier, Jochen Speck, Tim Niklas Uhl, Marvin Williams, and Sascha Witt for the interesting and fun discussions that go above and beyond our day to day work. I want to thank Daniel Funke, Demian Hespe, Florian Kurpicz, Tobias Maier, Matthias Schimek, Dominik Schreiber, and Daniel Seemaier in particular, for helping me by proofreading this dissertation.

I also want to thank my bright and hard-working students Adrian Feilhauer, Damir Ferizovic, Alexander Gellner, Tom George, Christian Schorr, Tim Niklas Uhl, and Robert Williger.

I am forever grateful to my family for their love and support throughout my entire life. Also, I would like to thank my closest friends Maximilian Brückmann, Nicolas Claveau, Tobias Engelhardt, Johannes Kirschnick, Kai Klasen, Daniel Kramer, Kevin Lynott, Philip Matthias, Michael Remmel, Marvin Ruchay, Benedikt Schwende, Philipp Terzenbach, and Philip Winkler. I know many of you for more than half of my life and thoroughly enjoyed every minute. Finally, I would like to give thanks to the TMW community for providing me with motivation for my biggest pastime endeavor over the last two years.

Table of Contents

Chapter 1

1 Introduction

In this dissertation, we examine the use and benefits of reductions for NP-hard optimization problems, namely, finding maximum cardinality or weight independent sets and maximum cardinality cuts. Using reductions, we improve both the scale of instances and the speed at which these instances are processed. We thus cover a wide range of algorithms that make use of various forms of reductions. This includes both heuristic and exact approaches, ranging from local search to branch-and-reduce algorithms. Our extensive experimental evaluations indicate that our algorithms are able to rival and often outperform current state-of-the-art approaches both in terms of the size of feasible instances and the time required to compute solutions.

Before examining the individual problems and our algorithms in the following chapters, we now provide motivation for the relevancy of these problems and the usage of reductions. We also provide an overview of the main contributions of this dissertation.

Many of today's important optimization problems are NP-hard, i.e., the common assumption is that there exists no polynomial-time algorithm that is able to solve them. However, for many of these problems, we are able to solve them efficiently in theory (and practice) if specific problem parameters are small. These types of problems are called *fixed-parameter tractable* (FPT). A common example of this type of problem is the minimum vertex cover problem, which is NP-hard [Kar72]. However, by formulating this problem as a parameterized problem with the size of the vertex cover k as an additional parameter, it can be solved in $\mathcal{O}\left(kn + 1.2738^k\right)$ time [CKX06].

In the last decade, even though there have been significant advances on the theoretical side of FPT algorithms, little attention has been placed on their practical applications. However, in recent years this changed as multiple works have shown the practical efficiency of these approaches for various graph problems [AI16; Abu+20]. Most notably, the exhaustive application of reductions in order to compute a reduced instance has been able to show tremendous results. This allows algorithms to compute solutions for instances orders of magnitudes larger than previously possible [AI16; Abu+20]. Since then, reductions have been used for all sorts of problems and in both heuristic and exact approaches. Despite this success, there are still a lot of instances that remain infeasible, as many of these instances still have particularly large or "hard" reduced instances. This means that these reduced instances can not be solved by existing algorithms in a reasonable amount of time.

Achieving smaller reduced instances can be done by new reduction rules. Particular caution has to be paid to the practically efficient applicability of these reduction rules, as

otherwise, computing a reduced instance itself becomes infeasible in a reasonable amount of time. Solving "hard" reduced instances can be done by improving other algorithmic aspects of existing state-of-the-art approaches or developing entirely new ones. Examples include using evolutionary algorithms [LSS15a] or advanced local searches [Dah+16a] to compute high-quality solutions in a short amount of time or using alternative branching rules [HLS21a] for exact branch-and-reduce algorithms. Thorough experimental evaluations ensure that these algorithms are able to function well on a large spectrum of instances from various domains and applications.

One of the problems that have received a lot of attention in the context of reductions is the NP-hard maximum cardinality independent set problem (and its complementary problems minimum vertex cover and maximum clique) [FGK09; Xia+17; Abu+04; AI16; But+02]. An independent set is a set of vertices of a graph that are pairwise not adjacent. The maximum cardinality independent set problem, or in short maximum independent set problem, then asks for the largest possible independent set (in terms of its cardinality). Additionally, a maximum independent set is the complement of minimum vertex cover and a maximum clique in the complement graph. Together with its complementary problems, its spectrum of applications includes map labeling [Klu+19], route planning [Kie+10], social network analysis [Put+15], VLSI design [FHL19], or modeling protein-protein interactions [GWA00]. For example, high-quality independent sets are used for the optimization of the stampings on a wafer to maximize utilization in VLSI design [FHL19]. Another example is the use of maximum cliques to find maximally complementary sets of donor and acceptor pairs when modeling protein-protein interactions as a set of potential hydrogen bonds [GWA00].

Due to the success of reductions for the maximum independent set problem, there has been an increasing interest in achieving the same for the weighted generalization of this problem [Wan+19; HXC21; Zhe+20; Li+20]. To be more specific, the maximum weight independent set problem for a graph with a vertex weighting function asks for an independent set of the largest possible weight. The weight of an independent set is measured by the sum of the weights of its vertices. The applications of this problem (and its complementary problems minimum weight vertex cover and maximum weight clique) extend the spectrum of applications for the unweighted case by additional possibilities for computing map labelings [GNR16; Bar+16] or modeling protein-protein interactions [Mas+10]. Furthermore, entirely new opportunities that leverage weighted independent sets arise from disk scheduling [CKR11], coding theory [NKÖ97; Bro+90], combinatorial auctions [WH15b], and vehicle routing [Don+22]. For example, maximum weight independent sets can be used to maximize the number of non-overlapping labels in label conflict graphs used for map labeling [GNR16; Bar+16]. Maximum weight cliques can, for example, be used to solve the winner determination problem emerging in combinatorial auctions [WH15b].

Finally, other NP-hard problems such as maximum cuts, vertex coloring, or cluster editing also benefit from the use of reductions [Fer+20; Abu+20]. The maximum cardinality cut problem, or in short maximum cut problem, in particular, will be covered in greater detail in this dissertation. Most notably, the benefits of reductions for practical algorithms have not been covered as thoroughly for this problem. The goal for the maximum cut problem is to find a bipartition of the graph such that the number of edges that run across partitions is maximized. There exist different generalizations of this problem, including weighted

and signed variants. Applications include social network modeling [Har59], statistical physics [Bar82], portfolio risk analysis [HLW02], and VLSI design [Bar+88; Chi+07].

1.1 Main Contributions

This dissertation is split into three different parts, each of which covers one of the three problems motivated in the previous section: maximum independent sets, maximum weight independent sets, and maximum cuts. Overall, in this dissertation, we present a wide range of contributions related to the field of practical reductions. These contributions range from theoretical aspects, such as new reduction rules, to algorithm design and engineering, such as an improved local search or novel branch-and-reduce algorithms. This dissertation also presents these ideas in a unified manner that goes beyond the individual publications on which they are based. In the following, we briefly cover the contributions made to each problem. Further references and attributions for each publication will be given in the corresponding chapters and sections.

1.1.1 Maximum Cardinality Independent Sets

For the maximum cardinality independent set problem, we provide both heuristic and exact algorithms. On the one hand, heuristic algorithms do not necessarily guarantee that a computed independent set is a maximum one but are often able to compute high-quality ones in a much shorter time frame than exact approaches. On the other hand, exact approaches can provide such a guarantee but often take much longer to compute their solution. To alleviate this, exact algorithms are also able to output inexact solutions that have been computed. In spite of their differences, both types of algorithms are able to benefit greatly from the use of reductions. This benefit often comes in the form of improved running times and a larger scale of instances that become feasible.

Inexact Iterative Reductions. We propose a very natural evolutionary algorithm to compute high-quality independent sets that makes use of reductions in two different ways. For this purpose, we make use of a large set of reductions proposed by Akiba and Iwata [AI16]. First, we use their reductions to compute a reduced instance that is used as the input for an existing evolutionary algorithm [LSS15b; LSS15a]. The main idea behind this algorithm are combine operations that use graph partitioning and local search to exchange large parts of independent sets while also ensuring that the resulting independent sets are valid. By operating on the (often smaller) reduced instance, we can boost the performance of this algorithm. Second, we use the evolutionary algorithm to select vertices that are likely to be in a large independent set. These vertices are then removed from the graph, which allows further reductions to be applicable. This process is repeated recursively until the graph is either completely reduced or a time limit is reached. Our algorithm is able to find independent sets up to four orders of magnitude faster than previous exact algorithms. Additionally, we are able to find high-quality independent sets on much larger graphs than previously reported in the literature. Finally, similar approaches have since been used in state-of-the-art heuristic algorithms [CLZ17; AC19].

On-the-fly Reductions. Our evolutionary algorithm allows us to compute high-quality independent sets at the cost of preprocessing time, e.g., computing a reduced instance and building the initial solutions. To reduce the time required to compute solutions of similar quality, we thus aim to improve the scale and speed at which local search algorithms can do so. To this end, we extend the iterated local search by Andrade et al. [ARW12] by reduction rules that can be applied very efficiently during the algorithm's execution. In particular, we propose two different approaches: (1) using reductions and applying local search on the reduced instance and (2) applying reductions on-the-fly during the local search, i.e., we apply reductions when vertices are moved in and out from the solution. To the best of our knowledge, these are the first practical algorithms for computing high-quality independent sets that extend local search with reduction rules. We also propose the use of inexact reductions by removing a small percentage of high-degree vertices, which pose a bottleneck for local search. Our experiments show that our algorithms are much faster (up to orders of magnitude) than previous algorithms while also producing solutions that are very close to the best-known ones. Whereas the first approach boosts the performance of local search on huge graphs at the cost of preprocessing time, the second approach is able to do so without significant overhead. Again, similar approaches have since been used in state-of-the-art heuristic algorithms [CLZ17; AC19].

Exact Portfolio Algorithm. To test the applicability and limits of algorithms that use reductions on a competitive benchmark, we participated in the Parameterized Algorithms and Computational Experiments (PACE) 2019 Challenge, which focused on the complementary minimum vertex cover problem. In this dissertation, we present our submitted state-of-the-art portfolio algorithm that won this competition. Our algorithm uses a portfolio of different approaches for finding vertex covers, independent sets, and cliques. These include the reduction rules by Akiba and Iwata [AI16], the iterated local search by Andrade et al. [ARW12], and the exact maximum clique algorithm by Li et al. [LJM17]. We integrate these approaches in an algorithm that cleverly uses increasing time intervals and switches between reduced and unreduced inputs. The particular choices are motivated by multiple important insights made during the participation in the challenge. Most notably, we observed that there is a benefit in using unreduced instances over their reduced counterparts. Finally, our algorithm remains competitive with current state-of-the-art approaches.

Target Branching Rules. Due to the extensive list of reductions available and the results achieved by making use of them, we then turn our attention to improving other (less explored) aspects of exact algorithms. Therefore, we present multiple novel strategies for selecting branching vertices for branch-and-bound or branch-and-reduce algorithms. Even though the branching strategy can have a large impact on the performance of exact algorithms [AI16], many of them select vertices based on their degree [FGK09; AI16; XN17]. We instead propose new branching strategies that can be classified as either decomposition-based or reduction-based: Decomposition-based strategies aim to select vertices that decompose the graph into multiple connected components, which can be solved independently. Doing so has been shown to boost the performance of branch-and-reduce [AC19]. For this purpose, our decomposition-based strategies make use of articulation points, edge cuts, or nested dissection. Reduction-based strategies select vertices that will lead to the application of

additional reduction rules and thus shrink the graph size. Our rules target a large set of reduction rules commonly used in practice. Our experiments show that by using our strategies, we are often able to find a solution faster than the commonly used degree-based branching strategy. In particular, our decomposition-based strategies are able to achieve an average speedup of 2.29 on sparse networks.

1.1.2 Maximum Weight Independent Sets

After presenting multiple successful approaches for unweighted independent sets, we then turn to algorithms that achieve similar results for the maximum weight independent set problem. Due to the lack of existing practical works on this problem that make use of reductions, our primary focus is on constructing algorithms for finding exact solutions. However, we also show that many of the techniques and preprocessing algorithms we present can be used to improve heuristic approaches. Again, we incorporate reductions in multiple ways, boosting the performance and scale of instances that can be handled by both heuristic and exact algorithms.

Generalized Reduction Rules. Our first contribution is the first practically efficient branch-and-reduce algorithm for the maximum weight independent set problem. Prior to our work, there were only a few known reduction rules for the maximum weight independent set problem [BT07; EHd84]. Additionally, these rules have only been tested on small synthetic instances. We present a more diverse set of new reduction rules. These reduction rules include generalizations of existing rules popularized by Akiba and Iwata [AI16] and entirely new ones. We also propose a set of so-called meta reductions that subsume many existing reduction rules and can be used as a theoretical framework for proving more concrete reductions. We incorporate these reduction rules as a reduction procedure in a branch-and-reduce algorithm that uses similar techniques to the one of Akiba and Iwata [AI16]. To the best of our knowledge, our algorithm is the first practical branch-and-reduce algorithm for the maximum weighted independent set problem. Our experimental evaluation shows that our algorithm and its set of reduction rules is able to handle real-world instances with up to millions of vertices and edges — a significant improvement to previous approaches that are only able to handle instances with hundreds of vertices. Furthermore, our algorithm is able to solve a large set of instances up to two orders of magnitude faster than even the best previously known heuristic approaches.

Increasing Transformations. Even though our newly proposed reductions are able to produce very successful results, they are often limited in their applicability, e.g., due to restrictive weight constraints. Therefore, we examine more general reduction rules that can be applied more liberally but whose practical impact on large graphs remained largely untested. To this end, we present a preprocessing algorithm that uses a set of practically efficient variants of the struction rule by Ebenegger et al. [EHd84]. The struction rule is a reduction rule that is able to decrease the weight of a maximum weight independent set of a given graph. However, it may also increase the graph size by introducing additional vertices and edges. We address this issue by introducing three new struction variants that are capable of limiting this size increase. We then engineer special cases of these variants that can be

efficiently applied in practice. These include cases that always maintain or reduce the graph size and can easily be integrated into existing algorithms. We also propose a preprocessing algorithm that makes use of the full potential of the struction rule by allowing a temporary increase in the graph size. This temporary increase can actually lead to further reductions and a smaller graph size in the long run. Our experiments indicate that our algorithm achieves significantly smaller graphs than previous approaches on all except one instance tested. Finally, using our preprocessing in combination with existing branch-and-reduce algorithms allows them to solve instances up to two orders of magnitude faster and also solve many previously infeasible instances.

1.1.3 Maximum Cuts

Lastly, we want to explore the impact of reductions for problems where they have received less attention when it comes to practically efficient algorithms. Therefore, we examine the use of reductions for building preprocessing algorithms for the maximum cardinality cut problem. Even though multiple works cover the usage of reductions for finding maximum (cardinality) cuts (and its signed and weighted generalizations) from a theoretical point-of-view [Cro+13; CJM15; EM18; MSZ18; Pri05; Far+17], to the best of our knowledge, previous practical state-of-the-art algorithms did not make use of them.

Practically Efficient Reductions. We propose and engineer a reduction algorithm for the maximum cardinality cut problem that uses a set of new and practically efficient reduction rules. Our reduction rules can often be applied much more easily (in practice) than existing rules while also subsuming most of them. In particular, our reductions do not rely on specific subgraphs such as clique-forests. We incorporate these reduction rules in a reduction algorithm that transforms a given instance between the unweighted, signed, and weighted variants of maximum cut to produce small reduced instances. We also reduce the number of checks for the applicability of reduction rules by using a timestamping mechanism. Our experimental evaluation indicates that our reduction algorithm can speed up exact state-of-the-art approaches by up to three orders of magnitude on real-world instances. As a result, we again increase the set of feasible instances, some of which can now be solved in less than two seconds. Finally, we can also drastically improve the performance of heuristic algorithms on large graphs with millions of vertices.

1.2 Outline

The rest of this dissertation is organized as follows. We provide important preliminaries in Chapter 2. This includes fundamentals and notation for the problems discussed in this dissertation, an overview of frequently used algorithmic components, and information on our research and experimental methodology. In Chapter 3, we then discuss our work on the maximum cardinality independent set problem. This is followed by our work on the maximum weight independent set problem presented in Chapter 4. Work on the maximum cardinality cut problem is given in Chapter 5. Finally, we conclude this dissertation in Chapter 6 with a brief summary and an outlook for future work.

Chapter 2

2 Preliminaries

We present the fundamentals required for this dissertation. This includes the notation and definitions for reductions, graphs, and the problems discussed in detail in later chapters. We then briefly cover important algorithmic components that are used throughout this dissertation. Finally, we discuss aspects related to our research and experimental methodology. In particular, we provide an overview of our research methodology, algorithm engineering, and present reoccurring elements used in our experimental evaluations, including problem instances, plot types, and the specific hardware used.

References. This chapter unifies multiple aspects including notation, definitions, and experimental methodology used by the publications that form the basis of this dissertation [Lam+16; Dah+16a; Lam+17; Lam+19; HLS21a; Gel+21; Fer+20]. Large parts are copied verbatim from these papers or the corresponding technical reports [Lam+15; Dah+16b; Hes+19; HLS21b; Lam+18; Gel+20; Fer+19].

2.1 Notation and Definitions

We now introduce reductions from a theoretical point-of-view and introduce the required notation for the following chapters. We also present the graph and problem definitions that are frequently used throughout this dissertation.

2.1.1 Kernelization and Reductions

We present definitions of kernelization and reductions based on Fomin et al. [Fom+19]. For this purpose, let Σ be a fixed, finite alphabet. A *parameterized problem* is a subset $L \subseteq \Sigma^* \times \mathbb{N}$. For an element $(x,k) \in \Sigma^* \times \mathbb{N}$ of this subset, k is called the *parameter*. A problem is called *fixed-parameter tractable* (FPT) if there exists an algorithm that solves the decision problem $(x,k) \in L$ in time that is exponential only in the parameter k. To be more specific, the running time of such an algorithm is bounded by $f(k) \cdot |x|^c$, where $f(k)$ is an arbitrary computable function of k (but independent of $|x|$), and c is a constant. We denote the running time of such an algorithm as $\mathcal{O}^*(f(k))$. An instance (x,k) is called a *yes-instance* if $(x,k) \in L$ and a *no-instance* otherwise.

Kernelization. For a given parameterized problem L, a *kernelization* algorithm transforms any given $(x,k) \in L$ into a *kernel* (x',k') such that

$$((x,k) \in L \Leftrightarrow (x',k') \in L) \text{ and } |x'|, k' \leq h(k) \tag{2.1}$$

where h is an arbitrary computable function and called the *size* of the kernel. Furthermore, the algorithm has to run in time polynomial in $|(x,k)|$. Finally, if h is polynomial, the kernel is also called polynomial.

Kernelization is closely related to the concept of FPT algorithms, as we will discuss in the following. First, we say that a problem L *admits* a kernel of size h if every instance of L has a kernel of size h. If this size is limited by some arbitrary computable function f, then the problem is FPT. To see why this is the case, recall that kernelization runs in time polynomial in $|(x,k)|$. Thus, we have an instance whose size is limited by f and can be computed in polynomial time. Afterwards, we can use an exponential time algorithm to decide if the reduced instance is a yes- or no-instance.

The reverse is also true, i.e., if a parameterized problem L is FPT, then it admits a kernelization. To show that this is the case, suppose that L is FPT, i.e., there is an algorithm that decides if $(x,k) \in L$ in time $f(k) \cdot |x|^c$. First, we cover the case $|x| \geq f(k)$. Then, depending on the output of the decision algorithm (either yes or no), the kernelization algorithm simply outputs a constant size yes- or no-instance. Second, when $|x| < f(k)$, then it suffices that the kernelization algorithm outputs x. Thus, kernelization can be seen as an alternative definition for FPT. Finally, we note that the term kernel is also often used to describe a reduced instance that does not necessarily conform to the definition above.

Reductions. Kernelization algorithms are usually composed of *reduction rules* or *reductions* in short. A reduction is an algorithm that transforms an instance (x,k) of a parameterized problem L into an *equivalent* instance (x',k') of L. Here "equivalent" means that

$$(x,k) \in L \Leftrightarrow (x',k') \in L \tag{2.2}$$

and is also called *safeness* or *soundness* of the reductions. One usually assumes that the reduction runs in time polynomial in $|x|$ and k and that $|x'| < |x|$ and $k' < k$.

In the context of this dissertation, we use reduction rules applied to combinatorial optimization problems (as opposed to decision problems). For this purpose, let I be a set of instances and $x \in I$ with a set of feasible solutions f_x. An *optimal solution* for x is a feasible solution $y \in f_x$ that minimizes (or maximizes) some cost function. We then define a reduction rule as an algorithm that transforms a given instance $x \in I$ into an equivalent instance $x' \in I$. Here "equivalent" means that we can compute the value of the cost function of an optimal solution of x from an optimal solution of x'. As before, we usually assume that reductions run in time polynomial in $|x|$ and that $|x'| < |x|$. However, in later chapters, we introduce reductions that do not follow these assumptions, i.e., they do not run in polynomial time and might not reduce the instance size but even increase it. To distinguish reductions that do not reduce the instance size from ones that do, we call them *transformations*.

2.1.2 Graphs

Let $G = (V,E)$ be an undirected graph with vertices $V = \{1,\dots,n\}$ and edges $E \subseteq \binom{V}{2}$. The number of vertices is denoted as $n = |V|$ and the number of edges as $m = |E|$. We assume that all graphs are *simple*, i.e., they contain no self-loops $\{v,v\}$ and no duplicate edges. In Chapter 4, we also make use of vertex-weighted graphs $G = (V,E,w)$ that have an additional

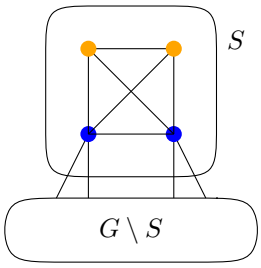


Figure 2.1: Example of external vertices (blue) and internal vertices (orange) for a subset of vertices $S \subseteq V$.

real-valued vertex weighting function $w : V \to \mathbb{R}^+$. Furthermore, in Chapter 5, we consider edge-weighted graphs $G = (V, E, w)$ with an additional real-valued edge weighting function $w : E \to \mathbb{R}^+$. For a subset S of vertices ($S \subseteq V$) or edges ($S \subseteq E$), we use $w(S) = \sum_{s \in S} w(s)$ and $|S|$ to denote the *weight* and *size* of S respectively.

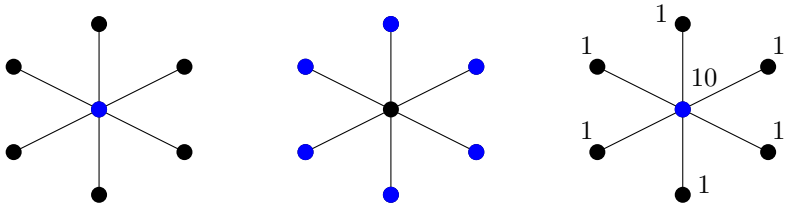
We say that vertices v and u are *adjacent* if $\{v, u\} \in E$. The *neighbors* of a vertex $v \in V$ are defined as $N(v) = \{u : \{v, u\} \in E\}$. We define the neighborhood of a set of vertices $U \subseteq V$ as $N(U) = \cup_{v \in U} N(v) \setminus U$. Analogously, we define the closed neighborhoods as $N[v] = N(v) \cup \{v\}$ and $N[U] = N(U) \cup U$. We also use $N^2(v)$ as a shorthand for $N(N(v))$ and use $\overline{N}(v)$ to denote the set of vertices that are not adjacent to v. The *degree* of a vertex $v \in V$ is denoted as $d(v) = |N(v)|$. The *maximum degree* of a graph is denoted as $\Delta = \max_{v \in V} d(v)$.

We say a graph is *dense* if $m \in \Omega(n^2)$. Likewise, a graph is *sparse* if $m \ll n^2$, e.g., if $m \in \mathcal{O}(n \log n)$. Finally, we say a graph is *scale-free* if the fraction of vertices with degree k is proportional to $k^{-\gamma}$ for some constant $\gamma > 1$.

A graph $H = (V_H, E_H)$ is a *subgraph* of $G = (V, E)$ if $V_H \subseteq V$ and $E_H \subseteq E$. We denote the set of vertices and edges of a (sub-)graph H by $V(H) = V_H$ and $E(H) = E_H$ respectively. For the sizes of these sets we use $n(H) = |V(H)|$ and $m(H) = |E(H)|$. H is an *induced subgraph* if $E_H = \{\{u, v\} \in E : u, v \in V_H\}$. For a set of vertices $U \subseteq V$, the subgraph induced by U is denoted as $G[U]$. We also use the notation $G \setminus U$ to refer to the subgraph $G[V \setminus U]$. We define the *cut* between two sets of vertices $V_1, V_2 \subseteq V$ as the set of edges $E(V_1, V_2) = \{\{u, v\} \in E \mid u \in V_1 \wedge v \in V_2\}$. Finally, the *complement* of a graph is defined as $\bar{G} = (V, \bar{E})$, where $\bar{E} = \binom{V}{2} \setminus E$.

The set of *external vertices* $C_{\text{ext}}(S) = \{v \in S \mid \exists w \in V \setminus S \wedge \{v, w\} \in E\}$ is the set of vertices in $S \subseteq V$ that have some neighbor in G outside S. Similarly, $C_{\text{int}}(S) = S \setminus C_{\text{ext}}(S)$ defines the set of *internal vertices*. An example of external and internal vertices is given in Figure 2.1.

A *path* P of length $l \in \mathbb{N}$ in a graph G is defined as a sequence $P = \langle v_1, \ldots, v_{l+1} \rangle$ such that $\{v_i, v_{i+1}\} \in E$ for $i = 1, \ldots, l$. A path with $v_1 = v_{l+1}$ is called a *cycle*. We say that a graph is *connected* if for every pair of vertices $v, w \in V$ there exists is a path $\langle v, \ldots, w \rangle$ in G from v to w. Otherwise, we call the graph *unconnected*. A *connected component* of G then is an



(a) Maximal independent set (b) Maximum independent set (c) Maximum weight independent set

Figure 2.2: Examples of maximal (a), maximum (b), and maximum weight (c) independent sets. The independent set vertices are highlighted in blue.

inclusion-maximal connected subgraph. A graph is called *biconnected* if it is both connected and the removal of any single vertex does not make the graph unconnected. A *biconnected component* of G then is an inclusion-maximal biconnected subgraph. Finally, a *bridge* in a connected graph is an edge $e \in E$ such that the removal of e *disconnects* the graph, i.e., removing it makes the graph unconnected.

A *k-way partition* of a graph is a division of V into k *blocks* of vertices $V_1, \dots, V_k$ such that $V_1 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \varnothing$ for $i \neq j$. The *balancing constraint* demands that $\forall i \in \{1, \dots, k\} : |V_i| \leq (1 + \epsilon) \lceil n/k \rceil$ for some imbalance parameter $\epsilon \geq 0$. The objective is to minimize the total *cut*, i.e., $\sum_{i<j} w(E(V_i, V_j))$. The set of cut edges is also called an *edge separator*. The *k-way vertex separator problem* asks to find k blocks $(V_1, V_2, \dots, V_k)$ and a separator $\mathcal{S} \subset V$ that partitions V such that there are no edges between the blocks, i.e., the removal of S disconnects the graph. Again, a balancing constraint demands $|V_i| \leq (1 + \epsilon) \lceil n/k \rceil$. The objective is to minimize the size $|\mathcal{S}|$ of the separator.

2.1.3 Problem Definitions

Independent Sets. An *independent set* of a graph is a subset of its vertices $\mathcal{I} \subseteq V$, such that all vertices in $\mathcal{I}$ are pairwise not adjacent, i.e., $\forall u, v \in \mathcal{I} : \{u, v\} \notin E$. An independent set is *maximal* if it is not a subset of any larger independent set. Furthermore, an independent set of the largest possible cardinality is called a *maximum (cardinality) independent set*. The *maximum independent set problem* (MIS) is that of finding such a maximum independent set. Note that there can be multiple maximum independent sets with the same size. Figure 2.2 shows examples of a maximal and maximum independent set. The size of a maximum independent set $\mathcal{I}$ of a graph G is denoted as $\alpha(G) = |\mathcal{I}|$ and called its *independence number*.

We say that an independent set $\mathcal{I}$ has *maximum weight* if there is no independent set with a larger weight, i.e., there exists no independent set $\mathcal{I}'$ such that $w(\mathcal{I}) < w(\mathcal{I}')$. The weight of a maximum independent set of G is denoted by $\alpha_w(G)$. The *maximum weight independent set problem* (MWIS) is that of finding an independent set of the largest weight among all possible independent sets. Again, maximum weight independent sets may not be unique. An example of a maximum weight independent set is given in Figure 2.2.

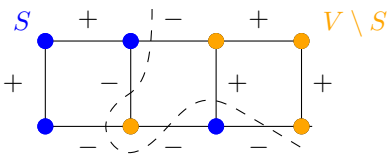


Figure 2.3: Example of a SignedMaxCut instance. A maximum cut with value $\beta_l(G) = 9$ in this graph is given by the vertex sets S (blue) and $V \setminus S$ (orange).

Complementary Problems. Both the *minimum vertex cover problem* (MVC) and the *maximum clique problem* (MC) are closely related to the maximum independent set problem. A *vertex cover* of a graph is a subset of its vertices $\mathcal{C} \subseteq V$, such that each edge $e \in E$ is adjacent to at least one vertex in $\mathcal{C}$, i.e., $\forall e = \{u, v\} \in E : u \in \mathcal{C} \vee v \in \mathcal{C}$. The minimum vertex cover problem asks for a vertex cover of minimum size, i.e., with the minimum number of vertices. Note that for a vertex cover $\mathcal{C}$, the complement $V \setminus \mathcal{C}$ is an independent set [Ski20]. Thus, the complement of a minimum vertex cover is a maximum independent set and vice versa.

A *clique* of a graph is a subset of its vertices $\mathcal{Q} \subseteq V$ such that all vertices in $\mathcal{Q}$ are pairwise adjacent, i.e., $\forall u \neq v \in V : \{u, v\} \in E$. The maximum clique problem asks for a clique of maximum size. A clique $\mathcal{Q}$ of a graph G is an independent set in the complement graph $\bar{G}$ [Ski20]. A maximum clique is a maximum independent set in the complement graph.

Finding maximum independent sets, as well as minimum vertex covers and maximum cliques, are well-studied NP-hard optimization problems [Kar72; GJ79; Abu+04; Abu+20; FGK09; Xia+17; WH15a]. Additionally, unless $P = NP$, there is no polynomial-time approximation for both the maximum independent set and maximum clique problem that can provide an $\mathcal{O}\left(n^{1-\epsilon}\right)$ guarantee for any constant $\epsilon > 0$ [Zuc07]. Both of these problems are also $W[1]$-hard when parameterized by the solution size k [DF99]. This means that it is unlikely that these problems are fixed-parameter tractable in k. However, it has been shown that finding minimum vertex covers is fixed-parameter tractable in k [DF99].

The *minimum weight vertex cover problem* (MWVC) and *maximum weight clique problem* (MWC) can be defined analogously. In particular, a minimum weight vertex cover $\mathcal{C}$ is a vertex cover of the smallest weight, i.e., there is no vertex cover $\mathcal{C}'$ such that $w(\mathcal{C}) > w(\mathcal{C}')$. A maximum weight clique is a clique $\mathcal{Q}$ of the largest weight, i.e., there exists no clique $\mathcal{Q}'$ such that $w(\mathcal{Q}) < w(\mathcal{Q}')$. The complement of a minimum weight vertex cover is a maximum weight independent set and a maximum weight clique in the complement graph is a maximum weight independent set.

Maximum Cuts. The *maximum cut problem* (MaxCut) is to find a vertex set $S \subseteq V$, such that the size of the cut $|E(S, V \setminus S)|$ is maximized. We denote the size of a maximum cut by $\beta(G) = |E(S, V \setminus S)|$.

The *weighted maximum cut problem* (WeightedMaxCut) is to find a vertex set S of a given graph G with an edge weighting function w such that $w(E(S, V \setminus S))$ is maximized. The weight of a maximum cut is then given by $\beta_w(G) = w(E(S, V \setminus S))$.

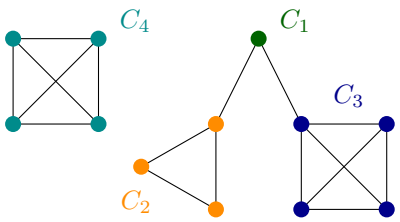


Figure 2.4: Example of a clique forest containing two clique trees: T_1 consisting of C_1,C_2, and C_3 and T_2 consisting of C_4.

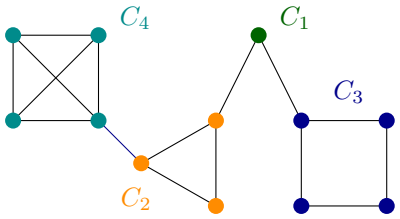


Figure 2.5: Example of a clique-cycle forest containing a single tree.

The *signed maximum cut problem* (SignedMaxCut) takes as input a graph G together with an edge labeling $l : E \to \{+,-\}$. Let $E_l^c(S, V \setminus S) = \{e \in E(S, V \setminus S) \mid l(e) = c\}$ be the set of edges of $E(S, V \setminus S)$ labeled with $c \in \{-,+\}$. Furthermore, for an induced subgraph $G[S]$, let $E^c(G[S], l) = \{e \in E \mid l(e) = c\}$ be the set of edges labeled with $c \in \{-,+\}$ in this subgraph. The goal of SignedMaxCut is to find a subset of vertices $S \subseteq V$, which maximizes the signed cut weight $\beta_l(G) = |E_l^-(S, V \setminus S)| + |E^+(G[S], l) \cup E^+(G[V \setminus S], l)|$, i.e., the sum of the number of "–"-edges with endpoints in different partitions and the number of "+"-edges with endpoints in the same partition. An example of a SignedMaxCut instance with an optimal cut is given in Figure 2.3. For the neighborhood of a vertex, we use the notation $N_l^c(v) = \{w \in V \mid \{v, w\} \in E^c(l)\}$ Respectively, for a set of vertices $X \subset V$, we use $N_l^c(X) = \bigcup_{v \in X} N_l^c(v) \setminus X$. We call a triangle ($u, v, w \in V$ such that $\{u, v\}, \{u, w\}, \{v, w\} \in E$) *positive* if its number of "–"-edges is even. Any MaxCut instance can be transformed into a SignedMaxCut instance by labeling all edges with "–". Note that maximum cuts do not have to be unique.

Closely related to maximum cut problems is the NP-hard *quadratic unconstrained binary optimization problem* (QUBO) [GJ79]. Let $Q \in \mathbb{R}^{n \times n}$ be symmetric $n \times n$ matrix and $x \in \{0, 1\}^n$. The goal of QUBO is to select x such that the value $x^* = x^\top Q x$ is minimized. A reduction from QUBO to WeightedMaxCut is given by Barahona et al. [BJR89].

Miscellaneous. A complete graph is a graph in which each vertex is connected to all other vertices, i.e., $E = \binom{V}{2}$. We denote a complete graph with n vertices as K_n. A *near-clique* of a graph is a clique that has one of its edges removed. A *clique tree* is a connected graph

whose biconnected components are cliques and a *clique forest* is a graph whose connected components are clique trees. The class of *clique-cycle forests* is defined as follows. A clique is a clique-cycle forest and so is a cycle. The disjoint union of two clique-cycle forests is a clique-cycle forest. In addition, a graph formed from a clique-cycle forest by identifying two vertices, each from a different component, is also a clique-cycle forest. Examples of a clique forest and clique-cycle forest are given in Figures 2.4 and 2.5, respectively.

2.2 Algorithmic Components

In the following, we briefly cover the main algorithmic components used throughout this dissertation. This includes local search, evolutionary algorithms, branch-and-reduce, as well as algorithm portfolios.

2.2.1 Local Search

Local search is a technique for tackling optimization problems based on moving between neighboring feasible solutions [MS08]. This is realized by iteratively transforming an initial solution by some simple local *moves*, e.g., by replacing a single vertex that is part of the solution. This is done in such a way that the cost function is gradually improved by each move. Finally, this process stops if there is no better neighboring solution with respect to the cost function, i.e., when a *local optimum* is reached. In general, no guarantee on the quality of the local optimum is provided by the local search.

There are multiple approaches for escaping local optima, including multistart local search, simulated annealing [vLA87], or tabu search [Glo89]. In particular, *tabu search* makes use of a set of previous solutions to guide the search by prohibiting or penalizing moves that would result in neighboring solutions of these previous solutions.

Of particular interest for this dissertation is the *iterated local search* metaheuristic [SR18]. Algorithms based on iterated local search start with an initial solution, which is then optimized using local search. They then generate new solutions by switching between applying a series of moves and local search. The series of moves used here can be augmented with different techniques for diversification, e.g., randomization or tabu mechanisms. A new solution is only accepted if it passes an acceptance criterion, e.g., if it is better than the current best solution. Finally, a stopping criterion is used to terminate the algorithm. A high-level overview of iterated local search adapted from Stützle and Ruiz [SR18] is given in Algorithm 2.1.

2.2.2 Evolutionary Algorithms

Inspired by natural selection, *evolutionary algorithms* are another metaheuristic used for solving optimization problems in practice [Gol89]. These types of algorithms make use of a *population* of solutions (that might be infeasible) called *individuals*. Additionally, they use a *fitness function* that assigns a fitness to each individual, which indicates the quality of the

Algorithm 2.1 : High-level overview of iterated local search.

```
1 S_0 ← create initial solution
2 S* ← apply local search to S_0
3 while stopping criterion not fulfilled do
4 |   S_t ← perturb S*                    // Apply diversification techniques
5 |   S_t' ← apply local search to S_t
6 |   if acceptance criterion fulfilled then
7 |   |   S* ← S_t'
8 return S*
```

corresponding solution. For example, one can use a population of independent sets and use the size (and feasibility) of the independent set as its fitness.

New solutions are generated by using two (or more) individuals as *parents* that are combined using a *crossover* or *combine* operation. This operation usually exchanges whole parts of the solutions represented by the parents, resulting in two (or more) *offspring*. Most often, parents are selected based on their fitness, i.e., higher fitness results in a higher probability of being selected. In order to maintain or restrict the size of the population, one can evict individuals from the population when generating offspring or as time passes. Usually, individuals with low fitness or high similarity to new offspring are evicted. Finally, evolutionary algorithms use a *mutation* operation that is randomly applied to offspring in order to diversify the population, e.g., by using similar diversification techniques applied to local search algorithms.

In this dissertation, we make use of evolutionary algorithms that can further be classified as *memetic algorithms* [CM07]. These types of algorithms combine the benefits of the global population-based search performed by evolutionary algorithms with the locally optimal solutions obtained by local search. In particular, individuals of the initial population and all generated offspring are optimized using local search to improve the convergence behavior of the underlying evolutionary algorithm. A high-level overview of a memetic algorithm is given in Algorithm 2.2.

2.2.3 Branch-and-Reduce

The main approach we use for finding exact solutions for the optimization problems discussed in this dissertation is based on *branch-and-bound* [MS08]. In the following, we discuss branch-and-bound algorithms for maximization problems, but the same ideas can also be used for minimization problems. The general idea of branch-and-bound is to exhaustively explore the set of feasible solutions by recursively dividing it into smaller subsets (*branches*) and then backtracking once a branch has been fully explored. For example, this can be done by including or excluding a vertex v from an independent set. The resulting branches then correspond to the set of independent sets that contain v (*including branch*) or that do not contain v (*excluding branch*). To select a vertex for branching, algorithms often use

Algorithm 2.2 : High-level overview of a memetic algorithm.

```
P ← create initial population
P ← maximize P using local search
while stopping criterion not fulfilled do
    I_1, I_2 ← select parents from P based on their fitness
    O_1, O_2 ← combine I_1, I_2
    O_1*, O_2* ← maximize offspring O_1, O_2 using local search
    O_1', O_2' ← mutation on offspring O_1*, O_2*
    P ← insert offspring O_1', O_2' and evict individuals
return feasible individual with the highest fitness
```

heuristics, e.g., selecting a random vertex with the highest degree. Furthermore, one or more *bounding procedures* are used to place (tight) upper bounds on the values of the cost function that can be obtained from the set of solutions represented by the current branch. By doing so, branches can be *pruned* if the value of the upper bound is smaller than a lower bound, e.g., the value of the cost function of the best previously found solution or some precomputed value.

Branch-and-reduce is an extension of branch-and-bound that integrates reduction rules to further reduce the set of feasible solutions [XN17; AI16]. In particular, these algorithms use a set of reduction rules that is exhaustively applied before branching. Usually, this is achieved by applying each reduction rule in some fixed order until it is no longer applicable and then moving on to the next reduction rule. If this leads to an applicable reduction rule, the process starts over with the first reduction rule. Applying reductions can be repeated after each branching step, as branching itself often facilitates the application of additional reduction rules. We provide a high-level overview of a branch-and-reduce in Algorithm 2.3

2.2.4 Algorithm Portfolios

To win the PACE 2019 Challenge, we used an algorithm portfolio consisting of state-of-the-art approaches from multiple closely related problems (see Section 3.4). Inspired by economics, an *algorithm portfolio* for an optimization problem is a set of algorithms that usually vary in their associated performance and risk [HLH97]. The performance of an algorithm in our case describes the time required to compute a solution, whereas the risk describes the certainty with which a solution can be obtained. For example, an algorithm might require a long time to compute a solution, thus having a low performance, but given an appropriate time limit may always find a solution, thus also having a low risk. An algorithm portfolio then tries to leverage different algorithms in order to cover a (wide) range of performance and risk tradeoffs. This can be achieved by running the algorithms sequentially and assigning different time limits to each individual run. For example, one can start with short runs of algorithms that have both high performance and risk and then gradually increase the time limits while switching to algorithms that have lower performance and risk.

Algorithm 2.3 : High-level overview of a branch-and-reduce algorithm.

```
1  Solve (I, c, k)
2      (I', c) ← apply set of reduction rules to (I, c)      // Can increase solution cost c
3      u ← compute upper bound for cost of I'
4      if c + u < k then
5          return k
6      if I' completely reduced then
7          return c
8      (I_1, c_1), (I_2, c_2) ← branch on (I', c)
9      k ← Solve(I_1,c_1,k)
10     k ← Solve(I_2,c_2,k)
11     return k
12 k ← compute initial lower bound for cost      // For example using local search
13 c ← 0                                         // Starting solution cost
14 return Solve(I, 0, k)                         // Solve instance I
```

Alternatively, one can also run the algorithms in parallel if the underlying hardware or other restrictions allow it.

2.3 Methodology

We now present important concepts for the research and experimental methodology used throughout this dissertation. In particular, the results presented in this dissertation have been achieved using *algorithm engineering*, a research methodology that combines algorithm theory with experimental algorithmics [San09; San10; SW11; SW13]. The following explanation of algorithm engineering is based on Sanders and Wagner [SW11; SW13] and Schlag [Sch20a]. For a more thorough explanation, we refer to the original works.

2.3.1 Algorithm Engineering

Traditional algorithmic theory aims to provide universal and provable performance guarantees for algorithms that are most often designed with simple problems and machine models in mind. Even though this leads to timeless and often highly reliable algorithms, they are more and more removed from realistic hardware with its own set of idiosyncrasies like parallelism, pipelining, or memory hierarchies. While hardware, and the applications that run on it, are rapidly evolving and becoming increasingly complex, algorithm theory develops more and more intricate algorithms that pose a significant challenge when it comes to transferring them to practice. Furthermore, algorithm theory uses asymptotic analysis that ignores constant factors, which prove to be important for real-world applications. This leads to a growing gap between theory and practice that algorithm engineering aims to close.

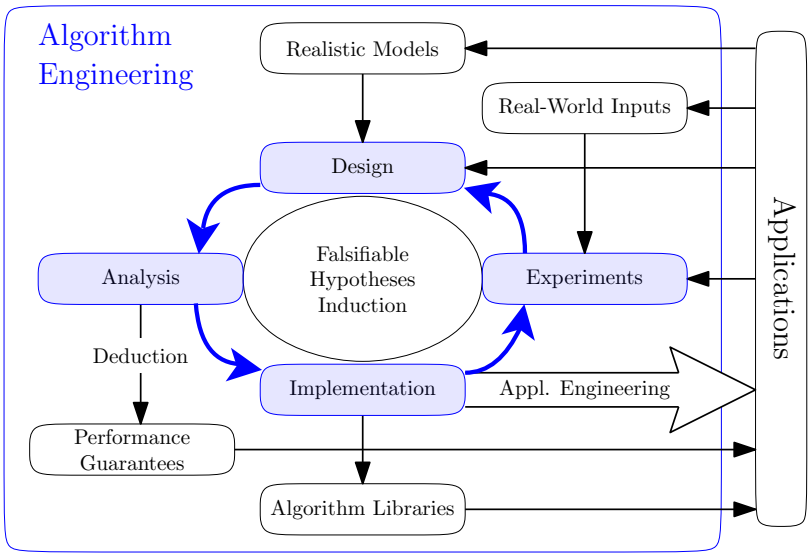


Figure 2.6: The cycle of algorithm engineering consisting of design, analysis, implementation, and experiments. Adapted from Schlag [Sch20a].

The main component of algorithm engineering is a feedback loop of design, analysis, implementation, and experiments given in Figure 2.6. Thus, algorithm engineering is different from experimental algorithmics, which focuses on the last two steps (implementation and experiments) of this loop. At the center of the algorithm engineering cycle lie falsifiable hypotheses that are supported and derived from experimental results through inductive reasoning. Each step of the cycle is also closely connected to real-world applications in different ways. For example, the design and evaluation of algorithms using the algorithm engineering methodology are based on realistic models for both the problems under consideration and for the underlying machine hardware. Additionally, the implementations and algorithm libraries are developed with the integration into applications in mind.

2.3.2 Experimental Methodology

We now present reoccurring elements used throughout the experimental evaluations in this dissertation. This includes graph instances and families that we use, descriptions of plot types, as well as the specific hardware used for our experiments.

2.3.2.a) Graph Instances

We give an overview of the instances used throughout this dissertation. Individual attributions for specific instances used during evaluations are provided in the corresponding sections. Note that our experimental evaluations often do not use all instances presented in

the following. This is due to certain instances being either too "easy" or "hard" for certain algorithms or the particular focus of the work. For example, an instance is too "hard" if none of the algorithms considered is able to solve it in a reasonable time. A complete overview of all instances with their corresponding numbers of vertices and edges is given in Appendix A.

Maximum Cardinality Independent Sets. We evaluate the algorithms for the maximum cardinality independent set problem on a diverse set of graph instances that have been used in multiple previous works [AI16; ARW12; GLP08]. Broadly, our set of instances can be divided into the following families: Sparse networks and matrices, road networks, complements of clique instances, meshes and networks from finite element computations, and PACE instances. Instances for this problem are readily available [LK14; RA15; Bad+18].

Sparse networks are graphs that, as their name implies, are sparse and, in our case, often have skewed degree distributions. These graphs can be exploited efficiently by even simple reduction rules [Str16], making them a good candidate for algorithms that make use of reductions. Additionally, finding large independent sets on these graphs is an important part of graph indexing methods [Che+12; FNS16]. For example, one can use independent sets to build a disk-based index for processing single-source shortest path and distance queries, which in turn can be used to compute different characteristics such as betweenness or closeness [Che+12]. Such an index can also be used for other problems that require similar queries, e.g., route planning. Our set of sparse networks includes social networks, citation networks, autonomous systems graphs, peer-to-peer networks, collaboration networks, communication networks, signed networks, Wikipedia networks, and web graphs taken from the 10th DIMACS Implementation Challenge [Bad+18], the Stanford Large Network Analysis Project (SNAP) [LK14], and the Network Data Repository [RA15]. Additional large web graphs are taken from the Laboratory of Web Algorithmics [BV04; Bol+11]. Graphs derived from sparse matrices have been taken from the Florida Sparse Matrix Collection [DH11]. We also use complements of dense *maximum clique instances* from the second DIMACS Implementation Challenge [JT96], which are often denser than the previously listed graphs and thus harder to reduce.

Road networks are another type of sparse graphs that are planar and often have a more uniform degree distribution than the aforementioned sparse networks. Again, these graphs can be handled well by reductions. Instances of this type are taken from Andrade et al. [ARW12] and the 9th DIMACS Implementation Challenge [DGJ09]. Additional large road networks are taken from [Del+09].

Meshes are graphs that have a highly uniform degree distribution and vertices often have low degrees. Due to their regular structure, they pose a challenge for reductions [Lam+17]. However, large or maximum independent sets on these graphs can be used for the efficient traversal of mesh edges in computer graphics [San+08]. Instances of this type are taken from Sander et al. [San+08] and are dual graphs of triangular meshes. Networks from *finite element computations* have been taken from Chris Walshaw's graph partitioning benchmark archive [SWC04].

Lastly, we make use of the instances from the *PACE 2019 Challenge* on the minimum vertex cover problem [DFH19]. This benchmark set consists of a variety of instances, including additional sparse and road networks, transit graphs, and a large set of SAT-based

instances. Furthermore, these instances (numbered 1 to 200) have been categorized and ordered by hardness using Gurobi [Gur21] with a time limit of six hours. Instances are categorized into "easy" (≤ 80), "medium" (between 81 and 160), and "hard" (≥ 161). "Easy", "medium", and "hard" instances were solved in the time intervals $[1, 300)$, $[300, 1500)$, and $[1500, 19000)$ seconds, respectively.

Maximum Weight Independent Sets. In contrast to the unweighted graphs presented previously, weighted instances for the maximum weight independent set problem are generally harder to obtain. The instances used in this dissertation can be divided into label conflict graphs, randomly weighted sparse networks, meshes, and complements of maximum weight clique instances. Real-world *label conflict graphs* obtained from OpenStreetMap [Fou22] (OSM) files of North America and generated according to the method described by Barth et al. [Bar+16] are one of the few sets of real-world graphs used in previous works [Cai+18]. To be more specific, these graphs are generated by constructing a vertex for each map label and assigning them a weight according to the label's importance. Edges are inserted between vertices that correspond to overlapping labels. Independent sets with large weight in these graphs can then be used for map labeling [GNR16; Bar+16]. In particular, computing a maximum weight independent set removes label conflicts and maximizes the importance of the labels that are displayed.

In addition to the OSM networks, we also use a set of *sparse networks* from the Stanford Large Network Dataset Repository [LK14]. However, these instances are unweighted, and comparable weighted instances are very scarce. Thus, we follow previous works [Cai+18; LCH17] and assign vertex weights uniformly at random from the fixed size interval $[1, 200]$.

We also consider randomly weighted dual graphs of triangular *meshes* from Sander et al. [San+08] and networks derived from simulations using the *finite element method* taken from Chris Walshaw's graph partitioning benchmark archive [SWC04].

Finally, we also tested the use of complements of instances for the *maximum weight clique problem* [McC+17]. However, since these instances are denser than many of the sparse networks that we target, they remain largely irreducible by our techniques.

Maximum Cuts. For evaluating maximum cut algorithms, we use a mix of synthetic and real-world graphs. *Synthetic instances* are generated using four different graph models included in the KaGen graph generator [Fun+19; SS16]. In particular, we used Erdős-Rényi graphs, random geometric graphs, random hyperbolic graphs, and Barabási-Albert graphs. These graphs are mainly used to evaluate certain aspects of our algorithm as they allow us to use controllable densities and degree distributions.

Our set of real-world graphs includes *sparse real-world instances* from the Network Data Repository [RA15]. We also include real-world instances from VLSI design and image segmentation applications taken from Dunning et al. [DGS18].

We also evaluated *denser instances* taken from the rudy category of the BiqMac Library [Wie18]. The rudy instances are random graphs with 100 vertices and different edge densities that are generated using the Rudy graph generator [Rin18]. However, due to their high density, these graphs are not susceptible to reductions. Note that the BiqMac library also contains the ising graph category stemming from statistical physics applications. However,

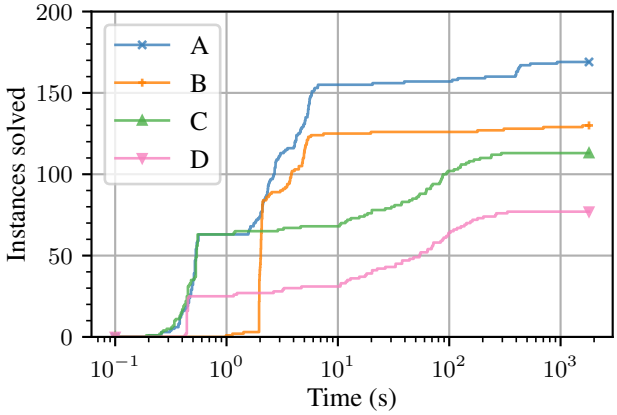


Figure 2.7: Example of a cactus plot comparing the number of solved instances over time for four algorithms.

in addition to their dense and uniform structure, these graphs also have large edge weights, making them more suited for finding maximum weight cuts.

2.3.2.b) Plot Types

Throughout this dissertation, we use cactus, convergence, and performance plots, which we now discuss in more detail.

Cactus Plots. A cactus plot shows the total number of solved instances over a given time period. More specifically, if an algorithm solves an instance at timestamp t, we increase the number of solved instances for this algorithm at time t by one. The plot then contains one line per algorithm that shows the number of instances solved by this algorithm over time. We only use this type of plot for the evaluation of exact algorithms. Note that a cactus plot does not provide information about how fast any specific instance is solved across different algorithms. Thus, it is hard to differentiate between "easy" and "hard" to solve instances. Nonetheless, it provides an overview of the overall performance of exact algorithms. An example of this type of plot is provided in Figure 2.7.

This plot shows four algorithms and the number of solved instances with a time limit of 30 minutes. Since all algorithms solve a large set of instances in less than ten seconds, a logarithmic scale for the x-axis is used to better emphasize the differences. Overall, algorithm A performs best as it has the largest number of solved instances for almost all time frames.

Convergence Plots. Convergence plots [SS12] show the solution quality over time for a given instance. More specifically, whenever an algorithm finds a new (and better) solution S at time t, it reports a pair $(t, f(S))$, where f is a function that quantifies the quality of the solution. Without loss of generality, we assume that a solution S_1 is better than a solution S_2 if $f(S_1) > f(S_2)$. The plot then contains one line per algorithm that shows the sequence

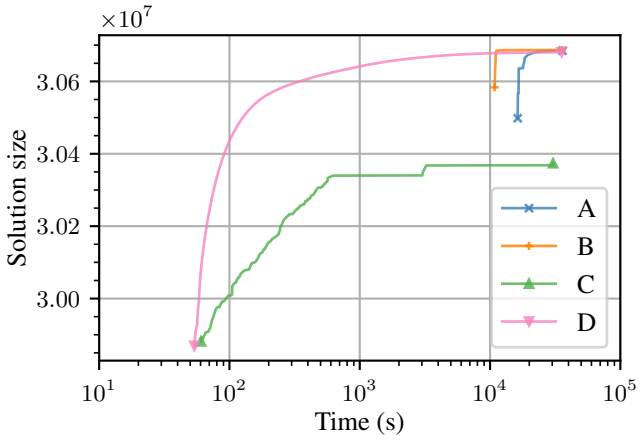


Figure 2.8: Example of a convergence plot comparing the solution size over time for four algorithms.

of these pairs. If not mentioned otherwise, convergence plots show event-based geometric average values $(t, \mathcal{G})$ over multiple runs of the same instance using the same algorithm (with different random seeds) as described by Sanders and Schulz [SS12]. Let s be the random seed used for a particular run. We merge the different sequences obtained by the multiple runs into one sequence T^{merged} containing triples $(t, f(S), s)$. This sequence is then sorted by the timestamps. We then iterate over the sorted sequence T^{merged} and for each triple $(t, f(S), s)$ we add a tuple $(t, \max_{t' \leq t} f(S, t'), s)$ to another sequence $T^{\text{merged}}_{\max}$, where $\max_{t' \leq t} f(S, t')$ is the best solution found until time t. Finally, we maintain a geometric mean $\mathcal{G}$ of the solution quality over all seeds until a timestamp t. We then iterate over $T^{\text{merged}}_{\max}$ and for each tuple $(t, f(S), s)$ first update $\mathcal{G}$ and then add $(t, \mathcal{G})$ to the final sequence.

Note that as a result of presenting average values, this type of convergence plot does not contain information about the best or worst solutions obtained by an algorithm. To get a clearer picture of the overall performance of specific algorithms, multiple of these plots are required. Alternatively, one can normalize the sequences and merge multiple instances together [SS12]. Convergence plots can be used for both heuristic and exact approaches if they output the current solution quality during execution. An example of this type of plot is provided in Figure 2.8.

This plot shows four different algorithms and their average solution size for three runs on an instance with a time limit of five hours. Due to the quick increase in solution size for algorithms C and D, a logarithmic scale for the x-axis is used. We can see that algorithm D reaches a large average solution size very quickly, and then the increase becomes gradually smaller for larger time frames. In contrast, algorithm B takes a very long time to compute its first solution but then very quickly reaches an average solution size that is greater than that of algorithm D.

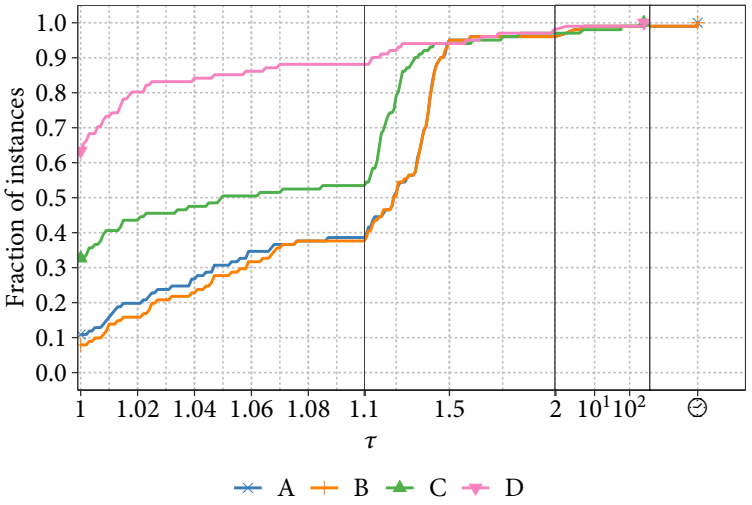


Figure 2.9: Example of a performance profile comparing the fraction of instances an algorithm performed best over varying τ for four algorithms.

Performance Profiles. Let $\mathcal{A}$ be the set of all algorithms that we want to compare. Furthermore, let $\mathcal{I}$ be a set of instances and $t_A(I)$ the running time of an algorithm $A \in \mathcal{A}$ on an instance $I \in \mathcal{I}$. Performance profiles [DM02] show the fraction of instances for which $t_A(I) \leq \tau \cdot \min_{A' \in \mathcal{A}} t_{A'}(I)$ over an increasing $\tau \geq 1$. In particular, for $\tau = 1$, the y-axis shows the fraction of instances for which an algorithm performs best. However, note that this fraction is only relative to the best solution, and thus, can not be used to rank algorithms [GS16]. Our plots also contain a value τ_{timeout} (displayed as 🕒) that is used if an algorithm does not finish computing a solution for an instance within a given time limit. Finally, the x-axis of our plots is divided into four segments to attribute for the skewness of the data, similar to Schlag [Sch20a]. The first (and largest) segment represents small values of $\tau \in [1, 1.1]$. The second segment then represents values of $\tau \in [1.1, 2]$ Both the first and second segments use a linear scale. The third segment represents values of $\tau \in [2, \tau_{\text{timeout}})$ and uses the approach of Tukey's Ladder of Powers [Tuk57; Tuk77] to fit the data to a normal distribution [McG12] as used by Schlag [Sch20a]. The fourth and final segment represents τ_{timeout}.

As mentioned before, for $\tau = 1$, performance profiles present performance in comparison to the best algorithm and thus it is hard to directly compare specific algorithms. Additionally, they have the same downside as cactus plots, in that it is hard to distinguish between "easy" and "hard" instances. Nonetheless, they are valuable for comparing the performance of algorithms as they are (1) relatively insensitive to changes in results on a few instances and (2) largely unaffected by small changes in the results over many instances [DM02]. An example of this type of plot is provided in Figure 2.9.

As described above, the x-axis of this plot is divided into four segments that use different scales. Since we are mainly interested in small differences in running times, smaller values of $\tau \in [1, 1.1]$ (and to a lesser extent $\tau \in [1.1, 2]$) are given the most space. Furthermore, these segments use a linear scale as changes are often not too pronounced for very small values within each segment. Segments containing larger values of $\tau > 2$ are mainly used to evaluate the overall fraction of instances for which a solution can be computed. For example, the first segment containing small values of $\tau \leq 1.1$ represents the fraction of instances for which an algorithm is at most 10% slower than the best one. Here we can see that algorithm A achieves the same running time as the best algorithm for a little over 60% of the instances ($\tau = 1$). We can also see that for close to 90% of the instances its running time is at most 10% worse than that of the best algorithm ($\tau = 1.1$). In contrast, algorithm B is only able to do so for a little over 50% of the instances. Afterwards, we can see that the fraction of instances that have a running time that is at most 50% worse than the best becomes roughly equal for all algorithms ($\tau = 1.5$). Finally, changes become very small for values of $\tau > 1.5$.

2.3.2.c) Hardware

Throughout this dissertation, we use the following three machines for our experiments.

Machine A has two Quad-core Intel Xeon X5355 processors running at 2667 MHz. It has 64 GB of main memory and each socket has 2×4 MB of L2-Cache.

Machine B has four Octa-core Intel Xeon E5-4640 processors running at 2400 MHz. It has 512 GB main memory and each socket has 8×256 KB L2-Cache and 20 MB of L3-Cache.

Machine C has four sixteen-core Intel Xeon Haswell-EX E7-8867 processors running at 2500 MHz. It has 1 TB of main memory and each socket has 16×256 KB of L2-Cache and 45 MB of L3-Cache.

Chapter 3

3 Maximum Cardinality Independent Sets

We present practically efficient heuristic and exact approaches for the maximum cardinality independent set problem that make use of reductions. We begin by examining an evolutionary algorithm that uses both exact and inexact reductions. Furthermore, we examine a local search algorithm that applies reductions on-the-fly when processing vertices. We then move on to exact algorithms and present a portfolio algorithm that combines state-of-the-art approaches from multiple related problems. Finally, we propose an exact branch-and-reduce algorithm that uses advanced branching strategies.

References. This chapter is based on the conference papers [Lam+16] (ALENEX 2016) published jointly with Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck, [Dah+16a] (SEA 2016) published jointly with Jakob Dahlum, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck, [Hes+20] (CSC 2020) published jointly with Demian Hespe, Christian Schulz, and Darren Strash, [HLS21a] (SEA 2021) published jointly with Demian Hespe and Christian Schorr, the survey paper [Abu+20] published jointly with Faisal Abu-Khzam, Matthias Mnich, Alexander Noe, Christian Schulz, and Darren Strash, and the journal paper [Lam+17] (J. Heuristics 23. 4) published jointly with Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. Large parts of this chapter were copied verbatim from these papers or the associated technical reports [Lam+15; Dah+16b; Hes+19; HLS21b].

We now outline the specific contributions the author of this dissertation made to each of these publications. For the conference paper [Lam+16] and the journal paper [Lam+17] the author, together with Christian Schulz and Darren Strash, was one of the main authors of the paper with editing done by Peter Sanders and Renato F. Werneck. The author made major contributions to the iterative reduction scheme presented in Section 3.2.3. In particular, contributions have been made to the vertex selection for the inexact reduction technique, as well as the additional acceleration techniques. Implementation of these techniques was done by the author. The reduction rules were implemented by Darren Strash. Evaluations were done by Christian Schulz. The evolutionary components (Section 3.2.2) are based on a previous publication [LSS15a] published jointly with Peter Sanders and Christian Schulz.

For the conference paper [Dah+16a] the author, together with Christian Schulz and Darren Strash, is one of the main authors of the paper with editing done by Peter Sanders and Renato F. Werneck. The author made major contributions to the on-the-fly application of exact reduction rules described in Section 3.3.1. Implementation and evaluations were done by the author. The idea of high-degree cutting (Section 3.3.1.b)) was implemented and

evaluated by Jakob Dahlum under the supervision of Christian Schulz and Darren Strash. The corresponding section is included to be self-contained.

Together with Demian Hespe, the author is one of the main authors of the conference paper [Hes+20] with editing done by Christian Schulz and Darren Strash. Additionally, the author made major contributions to the selection of algorithms used in the solver (Section 3.4.1), as well as the time-slicing approach (Section 3.4.2). In particular, the author tested multiple algorithms and analyzed graph characteristics that lead to the final solver. Implementation and evaluations were done in close cooperation with Demian Hespe.

Similarly, for the conference paper [HLS21a] the author, together with Demian Hespe, is one of the main authors of the paper with editing done by Christian Schorr. The author made major contributions to the separator based techniques (mainly articulation points and edge cuts) presented in Section 3.5.2. The ideas for the reduction-based techniques (Section 3.5.3) were developed in close cooperation with Demian Hespe. Implementation and evaluations were done by Christian Schorr under the supervision of Demian Hespe and the author.

Finally, for the survey paper [Abu+20] the author is the main author of the sections concerned with independent sets, vertex covers, and cliques.

Motivation. Finding a maximum independent set is a well-studied NP-hard problem [Kar72] with applications in many fields. These include map labeling [Klu+19], route planning [Kie+10], social network analysis [Put+15], rendering [San+08], modeling protein-protein interactions [GWA00], information retrieval [BY86] or computer vision [BY86].

For example, consider the fast spread of information to a large audience by users on social networks, which can have both positive and negative effects. An increasingly alarming example of the negative effects of this is the spread of misinformation. To enable social network operators to quickly counteract the spread of misinformation, it is beneficial to know a minimum set of vertices (e.g., users) that are able to cover the entire network [Put+15]. Such a minimum set of vertices corresponds to a minimum vertex cover or a maximum independent set.

Maximum independent sets can also be used for optimizing the traversal of a mesh in rendering which can, in turn, be used for different techniques such as motion blur or shadow volumes [San+08]. This is done by using the dual graph of a triangle mesh, i.e., the graph that contains a vertex for each triangle and vertices are connected if the corresponding triangles share an edge. By computing a maximum independent set in this graph, one can select a minimum number of triangles that cover all edges. The remaining triangles can then be assigned to ones in the maximum independent set by using bipartite matchings and the full set of triangles can be ordered to maximize cache reuse.

Many maximum independent set applications (including the two examples explained above) run on graphs with hundreds of thousands up to millions of vertices and edges. Processing these graphs using traditional heuristic or exact methods can be slow or even infeasible in realistic time frames [AI16]. Thus, we examine the use of reduction rules to improve the performance of these approaches.

Overview. In this chapter, we cover the usage of reduction rules for the maximum independent set problem from multiple angles. Therefore, we begin by presenting important

related work for the maximum independent set problem and its related problems with a focus on reductions in Section 3.1. Special attention is put on introducing the reductions that are frequently mentioned throughout this chapter, as well as presenting the algorithms by Akiba and Iwata [AI16] and Andrade et al. [ARW12].

The main contributions of this chapter are then presented throughout Sections 3.2–3.5. We first introduce two heuristic approaches that helped to popularize the usage of reductions for the maximum independent set problem due to the results they achieved. This includes an evolutionary algorithm (Section 3.2) that uses graph partitioning to combine multiple independent sets into new ones. This algorithm recursively applies reductions to shrink the graph. To make this possible, it exploits the population of the evolutionary algorithm to remove vertices that are likely to be in a large (or even maximum) independent set.

The second heuristic approach, which we introduce in Section 3.3, tries to bridge the gap between fast local search algorithms and more costly approaches like the previously mentioned evolutionary algorithm. The result is an algorithm that is able to output very high-quality independent sets in a very short amount of time. This is made possible by applying reduction rules on-the-fly during the execution of a state-of-the-art local search and the exclusion of high degree vertices that present performance bottlenecks. The techniques and algorithms presented in this section remain part of the state-of-the-art for the maximum independent set problem and laid important groundwork for current and future works.

We then turn to exact algorithms and cover the winner of the Parameterized Algorithms and Computational Experiments (PACE) 2019 Challenge in Section 3.4. This algorithm is a portfolio of multiple state-of-the-art approaches for the maximum independent set problem and its related problems. The individual algorithms are used in increasingly larger time slices to solve easy instances quickly while still being able to solve more instances in the long run than its competitors. This is motivated by the fact that many instances can be solved by either reductions alone or by additionally using an exact algorithm on the reduced instance.

Finally, we present a state-of-the-art branch-and-reduce algorithm that uses novel branching techniques in Section 3.5. These techniques either aim to decompose the graph or guarantee the application of additional reduction rules. Decomposing the graph is beneficial since previous works have shown that processing components individually can speed up algorithms. Guaranteeing the application of additional reduction rules allows us to exclude additional vertices and thus reduce search depth, speeding up the algorithm. Both of these techniques are beneficial depending on the structure of the input instance. We then conclude this chapter with an outlook for interesting open problems and future work in Section 3.6.

Overall, we thus present multiple approaches that investigate important aspects in this field of research. The algorithms we present allow previously infeasible instances to be solved efficiently, can handle larger instances than previously possible, and find solutions more quickly than previous approaches.

3.1 Related Work

We now cover related work for MIS with a focus on practically efficient reductions. This also includes works for the minimum vertex cover problem (MVC), since algorithms for these

two problems are easily interchangeable. Algorithms for the maximum clique problem (MC) and clique enumeration are covered separately, since there is non-negligible overhead when transforming an MC instance to an MIS instance, i.e., computing the complementary graph. Nonetheless, MC algorithms provide useful insights that can often be adapted for MIS.

3.1.1 Exact Approaches

In recent years, the gap between theoretically efficient algorithms and their practical applicability has been significantly reduced. Branch-and-reduce, i.e., branching algorithms that use a wide variety of reduction rules and only branch when no further reductions can be applied, has been (1) shown to achieve theoretical running times that are among the best for both MIS and MVC [FGK09; Xia+17] and (2) is able to solve large real-world networks in practice [AI16]. In particular, reduction rules and branch-and-reduce have been used to reduce the running time of the brute force $\mathcal{O}\left(n^2 2^n\right)$ algorithm to the $\mathcal{O}\left(2^{n/3}\right)$ time algorithm of Tarjan and Trojanowski [TT77], and to achieve the current best polynomial space algorithm with running time of $\mathcal{O}^*(1.1996^n)$ by Xiao and Nagamochi [XN17].

Butenko et al. [But+02] were the first to show that simple reductions could be used to compute a maximum independent set on graphs with several hundred vertices that are derived from error-correcting codes. Their algorithm works by first applying isolated clique reductions and then solving the remaining graph with a branch-and-bound algorithm. Later, Butenko and Trukhanov [BT07] introduced a reduction based on critical sets that can quickly solve graphs produced by the Sanchis graph generator.

Abu-Khzam et al. [Abu+04] introduced and analyzed the crown reduction rule (and the usage of reduction rules in this context in practice). Even though the crown rule is not as powerful as the linear programming (LP)-based rule [NJ75] when considering the worst-case size of the resulting reduced instance, they experimentally verified that it often performs as well as the LP-based rule and is significantly faster in many cases. Furthermore, they show that the LP-based rule is most useful for fairly sparse graphs and should be avoided for dense graphs as it yields little to no reduction in size.

Later, Akiba and Iwata [AI16] were the first to show the practicality of branch-and-reduce for MVC (and MIS) compared to other state-of-the-art approaches like branch-and-bound and branch-and-cut. Their algorithm uses a wide spectrum of reduction rules that form the foundation of many subsequent works [HSS19; SHG20; HLS21a; Hes+20]. This includes both conceptually simple reduction rules like degree-1 and degree-2 vertex folding [FGK09], as well as more complicated but practically significant rules like the unconfined reduction [XN13] and an LP-based rule [IOY14; NJ75]. Many of these reduction rules work by removing vertices that are part of some MIS. The authors show that their algorithm has superior performance to existing approaches for a large set of real-world sparse networks. Since the algorithm by Akiba and Iwata and its corresponding reduction rules are important for multiple works presented in this chapter, we discuss it in more detail in Section 3.1.5. A similar approach that uses a quantum annealer to solve instances once they are small enough was recently presented by Pelofske et al. [PHD19].

Even though Akiba and Iwata [AI16] use a sophisticated set of reduction rules, Strash [Str16] showed that many of the more complicated rules are not necessary to compute

a maximum independent set in many large complex networks. Additionally, the initial reductions applied to compute a reduced instance often have a bigger impact on performance, compared to further techniques used for branch-and-reduce algorithms [HSS19; PvdG21]. Recently, Stallmann et al. [SHG20] supported this idea by showing that networks with a small normalized average degree can be efficiently handled by simple reductions. Additionally, the authors make use of the so-called degree spread t/b, where t is the degree at the 95th percentile and b at the 5th percentile. Based on these characteristics, the authors devise thresholds that indicate (1) if reductions should be used at all and (2) if more complex rules provide a significant benefit. However, there is still a lot of work that needs to be done to determine which graph characteristics contribute to the success of specific reduction rules, both in theory and practice.

In order to quickly achieve smaller irreducible graphs than is possible with simple reduction rules, Hespe et al. [HSS19] provided the first shared-memory reduction algorithm based on the rules of Akiba and Iwata. For this purpose, they make use of both graph partitioning and parallel bipartite maximum matchings. The graph partitioning library KaHIP [SS13a] is used to compute a partition of that graph which allows parallel execution of reduction rules that only need to check highly localized subgraphs, where bipartite maximum matchings are used to enable the parallel execution of the LP-based reduction rule. The authors also present two speedup techniques for reduction algorithms: (1) dependency checking that prunes applicability checks for certain reductions and (2) reduction tracking that stops their algorithm once the application of reduction rules only decreases the graph size by a negligible amount. Extending this work to a distributed memory setting and the development of efficient distributed reductions still remain open problems.

Plachetta and van der Grinten [PvdG21] note that branch-and-bound algorithms often perform better on reduced instances than branch-and-reduce algorithms. They mainly attribute this to the fact that reduction rules often do not help in discarding branches. They then propose an algorithm that uses a SAT-based approach that is able to discard branches before reductions are applied. However, their experimental evaluation of the PACE instances only includes the individual algorithms used by the winning portfolio approach [Hes+20].

3.1.2 Heuristic Approaches

There is a wide range of heuristics and local search algorithms for MVC [Cai+13; Cai15; Fan+15; Ma+16] and MIS [ARW12; Pul09; WHG12; WH13; JH15; Dah+16a]. These algorithms typically maintain a single solution and try to improve it by performing vertex deletions, insertions, and swaps, as well as plateau search. Plateau search only accepts moves that do not change the cost function, which is commonly achieved through vertex swaps—replacing a vertex with one of its neighbors. Note that a vertex swap cannot directly improve the cost function. A very successful approach for the maximum clique problem was given by Grosso et al. [GLP08]. In addition to using plateau search, it applies diversification operations and restart rules.

For the MIS problem, Andrade et al. [ARW12] extended the notion of swaps to (j,k)-swaps, which remove j vertices from the current solution and insert k vertices. The authors present a fast linear-time implementation that, given a maximal solution, can find a $(1,2)$-

swap or prove that no $(1, 2)$-swap exists. Due to its importance for several works presented in this chapter, we discuss this algorithm in greater detail in Section 3.1.6. Not only is the ARW algorithm highly effective on small-to-medium-sized instances, but an external memory implementation exists, which allows it to be run on instances that do not fit into memory on a standard machine [Liu+15]. Most local search methods consider graphs with at most a few million vertices, which easily fit into memory [Cai15; Fan+15; Ma+16]. Fan et al. [Fan+15] and Dahlum et al. [Dah+16a] (Section 3.3) further show that combining reductions with local search significantly decreases the time to reach a solution of reasonably high quality. Note that ARW uses a form of *soft* tabu search since relatively old non-solution vertices are forced into the solution when no improvement is made. Further efficient local search algorithms use true tabu search [WHG12; WH13; JH15].

Reductions are also heavily used in many state-of-the-art heuristic approaches. Inexact reductions, i.e., excluding a subset of vertices that are likely to be part of a high-quality solution (as introduced by the work covered in Section 3.2), are explored by Gao et al. [Gao+17]. To select removable vertices they perform multiple runs of a state-of-the-art local search (either NuMVC [Cai+13] or FastVC [Cai15]). Vertices that are present in all resulting solutions are then added to the final solution and removed from the graph. Afterwards, a final run of the local search on the reduced graph is executed, and its solution is combined with the previously removed vertices.

Fan et al. [Fan+15] propose a local search algorithm for MVC that uses a set of three simple reduction rules including degree-1 and degree-2 removal for computing an initial solution. In particular, they alternate between exhaustively applying reduction rules and greedily selecting a vertex that is added to the solution. Thus, if the greedy selection is never used, they can guarantee that the computed solution is optimal. After computing an initial solution they then use a local search that removes vertices with a minimum loss, i.e., the number of covered edges that become uncovered, and then repeatedly adds the higher gain endpoint of a random uncovered edge until a new vertex cover is computed. To improve the performance of their algorithm, they also propose a new partitioning data structure that lowers the time complexity of multiple heuristics used. Finally, they show their algorithm outperforms the algorithm FastVC by Cai [Cai15] on a large set of real-world networks.

Chang et al. [CLZ17] also make use of the idea of combining simple reduction rules that can be applied in (near-)linear time with an inexact reduction rule that removes high degree vertices. For this purpose, they introduce the reducing-peeling framework that switches between the two types of reductions. Furthermore, they present a set of degree-2 path reductions that are special cases of the folding reduction. Combining these new rules with the degree-0, degree-1, dominance, and an LP-based reduction rule, they propose an efficient preprocessing algorithm that is then combined with the ARW local search. In light of this work, finding similar (near-)linear time special cases for more complex reduction rules remains an interesting open problem.

Similar reduction rules are also used in the preprocessing phase of the heuristic approaches by Cai et al. [CLL17]. In particular, they use degree-0, degree-1, and degree-2 removal in addition to the dominance rule. They split the application of these reductions into two phases: (1) degree-0, degree-1, and degree-2, and (2) dominance. This is motivated by the fact that even though dominance subsumes the reductions in the first phase

its application is more costly. Their preprocessing algorithm is then integrated into two improved local searches based on NuMVC [Cai+13] and FastVC [Cai15]. They show that the resulting algorithms are often able to produce higher quality solutions compared to variants that do not use preprocessing. The same technique has also been successfully used by Luo et al. [Luo+19] to boost the performance of their algorithm. Finally, Gu and Guo [GG21] examine the usage of a more general strong folding reduction to further reduce graph sizes.

Alsahafy and Chang [AC19] proposed an algorithm that combines the reducing-peeling framework [CLZ17] with the exact clique algorithm MoMC by Li et al. [LJM17]. Their algorithm splits reduction rules into two sets: ones that can be updated and applied incrementally (similar to Hespe et al. [HSS19]), and ones that can not. Additionally, they continuously compute and maintain the connected components of the graph, which are then reduced individually. If a reduced component is small enough, it is then transformed into its complement and solved by MoMC. To ensure that components continue to get smaller, they use the same inexact reduction rule as Chang et al. [CLZ17] and then continue recursively on the resulting components. Finally, the authors also present a new exact reduction rule called the pyramid reduction.

Chen and Hao [CH19] present a local search algorithm that alternates between two phases called thresholding search and conditional improving. During the thresholding search phase, their algorithm accepts both improving and non-improving solutions, whereas during the conditional improving phase only improving solutions are accepted. To be more specific, their thresholding search phase uses a vertex-based strategy that scans vertices in random order and applies a set of basic operations consisting of adding, removing, or swapping vertices. Additionally, they use a tabu mechanism to avoid cycling. The conditional improving phase uses hill-climbing to reach a local optimum and its output is conditionally accepted as the input of the next search phase based on a random choice. Their experimental evaluation on a large set of real-world graphs with up to millions of vertices indicates that their algorithm is able to compete well against the algorithms by Cai [Cai15] and Fan et al. [Fan+15].

3.1.3 Maximum Clique and Clique Enumeration

The most efficient exact maximum clique algorithms use branch-and-bound search with advanced vertex reordering strategies and pruning (typically using approximation algorithms for graph coloring [SRJ11; Tom+13] or MaxSAT [LFX13; LQ10]). The long-standing canonical algorithms for finding a maximum clique are the MCS algorithm by Tomita et al. [Tom+13] and the bit-parallel algorithms of San Segundo et al. [SRJ11; ST14]. However, recently Li et al. [LJM17] introduced the MoMC algorithm, which uses incremental MaxSAT logic to achieve speed-ups of up to 1000 over MCS. Experiments by Batsyn et al. [Bat+14] show that MCS can be sped up significantly by providing an initial solution found through local search. However, even with these state-of-the-art algorithms, some graphs with thousands of vertices remain intractable. For example, a difficult graph with 4000 vertices required 39 wall-clock hours in a highly-parallel MapReduce cluster and was estimated to require over a year of sequential computation [XGA13]. Recent exact approaches for sparse

graphs investigate applying simple reduction rules, using an initial clique given by some heuristic method [VBB15; SLP16; AC19]. However, these techniques rarely work on dense graphs, such as the complement graphs that we consider here. A more thorough discussion of many results in clique finding can be found in the survey of Wu and Hao [WH15a].

Eblen et al. [Ebl+12] present an exact maximum clique algorithm (MCF) that adapts some of the reduction rules that have already been shown to work well for MVC and MIS. In particular, their algorithm begins by greedily computing a large clique $\mathcal{Q}$ which is then used as a lower bound in order to remove vertices of degree less than $|\mathcal{Q}| - 1$ [APR98]. Next, they use an adaptation of the degree-0 reduction rule previously used in MVC algorithms, as well as a rule based on heuristic colorings [TK09] to remove additional vertices. The authors also investigated the use of other reduction rules including an adaptation of the degree-1 reduction rule used in MVC algorithms. Finally, they compared applying reduction rules as a preprocessing method for a branch-and-bound algorithm against running them in a branch-and-reduce algorithm. Their experiments indicate that the branch-and-reduce approach performs better on graphs stemming from real-world genome data.

Eblen et al. [Ebl+12] then used the previous MCF algorithm to develop several approaches for the maximum clique enumeration (MCE) problem based on the algorithm by Bron and Kerbosch [BK73]. In particular, they develop two reduction rules based on MCF: First, they propose a reduction rule that uses MCF to compute a maximum clique cover and removes vertices not adjacent to this cover. Second, they propose a data-driven preprocessing rule that computes so-called essential vertices, i.e., vertices that are present in every maximum clique. Vertices that are not adjacent to these vertices are subsequently removed from the graphs. Their experiments indicate that this rule works particularly well on large transcriptomic graphs, that often have a small set of essential vertices. However, performance degrades for networks that do not have a small set of essential vertices, e.g., for uniform random graphs.

Verma et al. [VBB15] propose another type of reduction rule based on k-communities. For this purpose, a k-community subgraph is defined as a subgraph $G' = (V', E')$ where each edge $\{u, v\} \in E'$ connects vertices that have at least k common neighbors in G'. A subset of vertices $V' \subseteq V$ is called a k-community if there is a k-community subgraph with vertex set V' in G. Note, that a clique of size k is a $(k - t)$-community for any $t \in \{2, \dots, k\}$. They then derive a reduction rule which computes a lower bound on the clique size based on maximum $(k - 2)$-communities and removes vertices with a smaller degree. They then combine this reduction rule with the k-core based approach of Abello et al. [APR98] and show that the resulting algorithm works well for handling large low-density graphs.

Chang et al. [AC19; Cha20] note that even though a lot of real-world networks are usually sparse, MC has been more extensively studied for dense instances. Thus, the authors propose a branch-and-reduce algorithm that leverages the existing work on MC for dense instances by transforming a sparse instance of MC to instances of k-clique finding (KCF) over dense subgraphs. For this purpose, they iteratively compute small and dense subgraphs (so-called ego networks) that are then handled by an exact KCF algorithm. In order to reduce the size of the subgraphs that are handled by this algorithm, their approach uses a combination of well-known upper bounds and lightweight reduction rules. In particular, they use five reduction rules for KCF, most of which are targeted toward removing vertices of high degree. The authors also present a heuristic algorithm for MC, as well as a two-stage approach

for MCE that makes use of their exact algorithm to compute the size of the largest clique. Furthermore, they show that the reduction rules used for MC can also be adapted for MCE.

Finally, there is a wide range of heuristics and local search algorithms for the maximum clique problem [BP01; HMU04; GLC04; KHN05; Pul06; GLP08]. A very successful approach has been presented by Grosso et al. [GLP08]. In addition to plateau search, it applies various diversification operations and restart rules.

3.1.4 Reduction Rules

We now present the reduction rules used in this chapter, roughly in order of increasing complexity. In particular, we cover the isolated clique reduction by Butenko et al. [But+02], the vertex fold reduction by Chen et al. [CKJ01], the dominance reduction by Fomin et al. [FGK09], the twin, alternative, funnel, desk, and unconfined reductions by Xiao and Nagamochi [XN13], the linear programming reduction by Nemhauser and Trotter [NJ75], as well as the packing reductions by Akiba and Iwata [AI16]. We limit ourselves to the definitions of these rules and refer the reader to the respective papers for their proofs.

Isolated Clique Reduction. We first cover the isolated clique reduction. Butenko et al. [But+02] have shown that this reduction is highly effective on small-sized instances with up to 500 vertices derived from error correcting codes. However, it also works well on large sparse instances [Str16]. The basic idea of this reduction is to find a clique $\mathcal{Q}$ that contains a vertex v, such that $d(v) = |\mathcal{Q}| - 1$. Such a vertex is called *isolated* or *simplicial*. These vertices (and their neighbors) can be removed from the graph since there is always a maximum independent set that contains them. An example application of this reduction rule is given in Figure 3.1.

Theorem 3.1 (Isolated Clique Reduction)
Let $G = (V, E)$ be an undirected graph and $\mathcal{Q} \subseteq V$ a clique containing a vertex $v \in \mathcal{Q}$, such that $d(v) = |\mathcal{Q}| - 1$. Let G' be the graph obtained by removing $N[v]$ from G. Then $\alpha(G) = \alpha(G') + 1$. For a maximum independent set $\mathcal{I}'$ of G', $\mathcal{I} = \mathcal{I}' \cup \{v\}$ is a maximum independent set of G.

Important special cases of this reduction are detecting cliques of size one, two, and three. These reductions are also often referred to as degree-0, degree-1, and degree-2 removal, respectively. Finding such cliques can be done efficiently in practice even for sparse graph representations. In particular, for cliques of size one and two it suffices to look for vertices of degree zero or one, respectively. For cliques of size three, one can store the neighbors of a vertex in increasing order and perform a single binary search to determine if the neighbors of a vertex v are adjacent. However, if the input does not contain sorted neighborhoods, sorting itself can be costly. Finally, if the neighbors of a vertex are not adjacent, one can instead perform the now following reduction.

Vertex Fold Reduction. Next, we cover the vertex fold reduction by Chen et al. [CKJ01]. This reduction looks for vertices v with degree two, whose neighbors u, w are not adjacent. In this case, there is always a maximum independent set that either contains v or both u

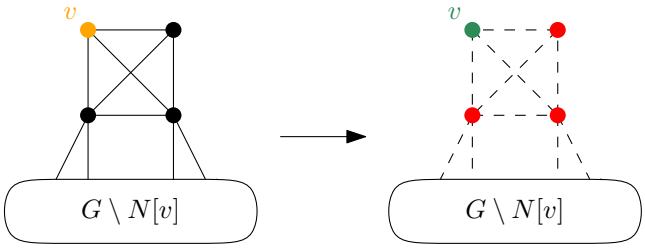


Figure 3.1: Example application of the isolated clique reduction to vertex v (orange). The neighborhood $N[v]$ can be removed from the graph. The vertex v is added to the independent set (and $N(v)$ is excluded).

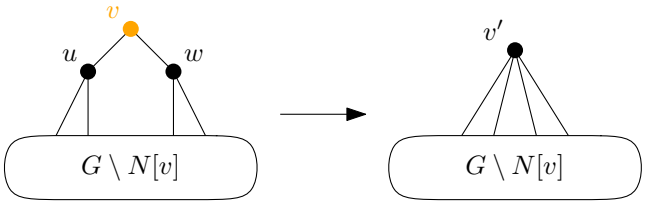


Figure 3.2: Example application of the vertex fold reduction to vertex v (orange). Vertices v, u and w are contracted to the new vertex v'.

and w. However, determining which of the two cases is correct can not be determined right away. Chen et al. [CKJ01] solve this by first contracting the three vertices v, u, and w and computing a maximum independent set for the resulting graph. A contraction is performed by removing the three vertices v, u, and w and their incident edges from the graph and inserting a new vertex v' whose neighborhood is the union of the neighborhoods of u and w. An example application of this reduction rule is given in Figure 3.2.

Theorem 3.2 (Vertex Fold Reduction)
Let $G = (V, E)$ be an undirected graph and $v \in V$ a vertex with $N(v) = \{u, w\}$, such that $\{u, w\} \notin E$. Let $G' = (V', E')$ be the graph resulting from contracting u, v, and w to a single vertex v', i.e., $V' = (V \setminus \{u, v, w\}) \cup \{v'\}$ and $E' = \{\{x, y\} \in E \mid x, y \in V'\} \cup \{\{v', x\} \mid x \in (N(u) \cup N(w)) \setminus \{v\}\}$. Then $\alpha(G) = \alpha(G') + 1$ and for a maximum independent set $\mathcal{I}'$ of G', if $v' \in \mathcal{I}'$, then $\mathcal{I} = (\mathcal{I}' \setminus \{v'\}) \cup \{u, w\}$ is a maximum independent set of G. Otherwise, if $v' \notin \mathcal{I}$, then $\mathcal{I} = \mathcal{I}' \cup \{v\}$ is a maximum independent set of G.

Dominance Reduction. The dominance reduction by Fomin et al. [FGK09] requires two vertices v, w, such that $N[w] \subseteq N[v]$. In this case, we say that v *dominates* w. Dominating vertices can be removed from the graph since there is always a maximum independent

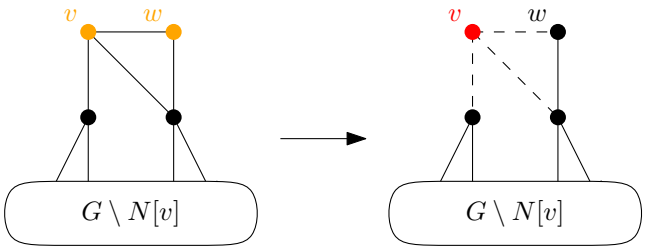


Figure 3.3: Example application of the dominance reduction to vertex v and w (orange). Vertex v can be removed from the graph and excluded from the independent set since it dominates vertex w.

set that does not include them. An example application of this reduction rule is given in Figure 3.3.

Theorem 3.3 (Dominance Reduction)

Let $G = (V, E)$ be an undirected graph and $v, w \in V$ vertices, such that $N[w] \subseteq N[v]$. Let G' be the graph resulting from removing v and its incident edges. Then $\alpha(G) = \alpha(G')$ and a maximum independent set $\mathcal{I}'$ of G' is also a maximum independent set of G.

Twin Reduction. The twin reduction by Xiao and Nagamochi [XN13] is an extension of the vertex fold reduction [CKJ01] and a special case of the crown reduction by Chor et al. [CFJ04]. This reduction looks for two nonadjacent vertices $u, v \in V$ of degree three, such that $N(u) = N(v)$. Depending on the induced subgraph $G[N(u)]$, the twin reduction is divided into two cases: (1) If $G[N(u)]$ contains at least one edge, then there is always a maximum independent set that contains both u and v. Thus, $N[u]$ and $N[v]$ can be removed from G. (2) If $G[N(u)]$ has no edges, replace $N[u] \cup N[v]$ with a new vertex v' that is connected to $N(N(u)) \setminus \{u, v\}$. Similar to the vertex fold reduction, depending on the inclusion of v' in a maximum independent set of the resulting graph G', either $N(u)$ or $\{u, v\}$ are included in the maximum independent set of G. An example application of cases (1) and (2) is given in Figure 3.4 and Figure 3.5.

Theorem 3.4 (Twin Reduction)

Let $G = (V, E)$ be an undirected graph and $u, v \in V$ nonadjacent vertices with $d(u) = d(v) = 3$ and $N(u) = N(v)$. If $G[N(u)]$ contains at least one edge, then let $G' = (V', E')$ be the graph resulting from removing $N[u]$ and $N[v]$, i.e., $V' = V \setminus (N[u] \cup N[v])$ and $E' = E \cap \binom{V'}{2}$. Then $\alpha(G) = \alpha(G) + 2$ and for a maximum independent set $\mathcal{I}'$ of G', $\mathcal{I} = \mathcal{I}' \cup \{u, v\}$ is a maximum independent set of G.

Otherwise, if $G[N(u)]$ has no edges, then let $G' = (V', E')$ be the graph resulting from contracting $N[u] \cup N[v]$ to a single vertex v', i.e., $V' = (V \setminus (N[u] \cup N[v])) \cup \{v'\}$ and $E' = \{\{x, y\} \in E \mid x, y \in V'\} \cup \{\{v', x\} \mid x \in N(N(u)) \setminus \{u, v\}\}$ Then $\alpha(G) = \alpha(G') + 2$ for

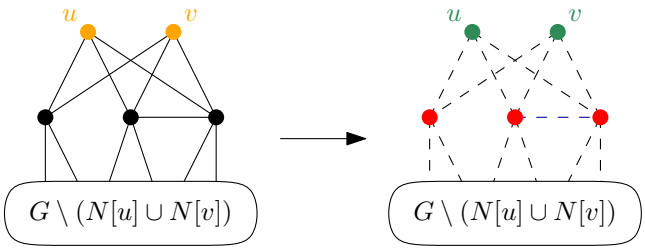


Figure 3.4: Example application of the first case of the twin reduction to vertex u and v (orange). $N[u] \cap N[v]$ can be removed from the graph and both u and v can be added to the independent set.

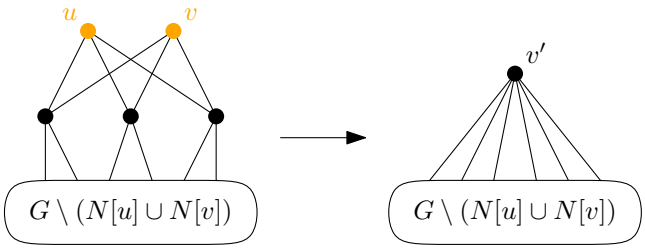


Figure 3.5: Example application of the second case of the twin reduction to vertex u and v (orange). $N[u] \cap N[v]$ is contracted to a single vertex v'.

a maximum independent set $\mathcal{I}'$ of G', if $v' \in \mathcal{I}'$, then $\mathcal{I} = (\mathcal{I}' \setminus \{v'\}) \cup N(u)$ is a maximum independent set of G. If $v' \notin \mathcal{I}'$, then $\mathcal{I} = \mathcal{I}' \cup \{u, v\}$ is a maximum independent set of G.

Alternative Reduction. Two subsets of vertices $A, B \subseteq V$ are called *alternatives*, if $|A| = |B| \geq 1$ and there exists a maximum independent set $\mathcal{I}$ such that $\mathcal{I} \cap (A \cup B)$ is either A or B. Xiao and Nagamochi [XN13] proposed the alternative reduction that removes alternatives in combination with parts of their neighborhoods from the graph. In particular, a reduced graph G' is obtained by removing $A \cup B$ and $C = N(A) \cap N(B)$ from G and inserting edges from each $a \in N(A) \setminus C$ to each $b \in N(B) \setminus C$. Then, if $(N(A) \setminus N[B]) \cap \mathcal{I}'$ is empty, we add A to $\mathcal{I}'$ to obtain a maximum independent set of G. Otherwise, we add B to $\mathcal{I}'$. It holds that $\alpha(G) = \alpha(G') + |A|$.

Theorem 3.5 (Alternative Reduction)

Let $G = (V, E)$ be an undirected graph and $A, B \subseteq V$ alternatives. Let $G' = (V', E')$ be the graph resulting from removing $A \cup B$ and $C = N(A) \cap N(B)$ from G, i.e., $V' = V \setminus (A \cup B \cup C)$ and $E' = (E \cap \binom{V'}{2}) \cup \{\{u, v\} \mid u \in N(A) \setminus C, v \in N(B) \setminus C\}$. Then $\alpha(G) = \alpha(G') + |A|$ and for a maximum independent set $\mathcal{I}'$ of G', if $(N(A) \setminus N[B]) \cap \mathcal{I}' = \emptyset$, then $\mathcal{I} = \mathcal{I}' \cup A$ is a

maximum independent set of G. Otherwise, if $(N(A) \setminus N[B]) \cap \mathcal{I}' \neq \emptyset$*, then* $\mathcal{I} = \mathcal{I}' \cup B$ *is a maximum independent set of G.*

An important property of this reduction is that it might add new edges between existing vertices. This increases the density of the graph which might not be beneficial.

Funnel and Desk Reduction. We now cover two special cases of the alternative reduction called the funnel and desk reductions [XN13]. First, vertices $u, v \in V$ are called *funnels* if $G[N(v) \setminus \{u\}]$ is a complete graph, i.e., if $N(v) \setminus \{u\}$ is a clique. If this is the case, then $\{u\}$ and $\{v\}$ are alternatives.

Second, for vertices $a_1, a_2, b_1, b_2 \in V$ we define the *chordless 4-cycle* $a_1b_1a_2b_2$ as a path $\langle a_1, b_1, a_2, b_2, a_1\rangle$ such that the vertices are not connected by edges other than the ones contained in the path. Now, let $a_1b_1a_2b_2$ be a chordless 4-cycle where each vertex has at least degree three. Such a cycle is called a *desk* if $A = \{a_1, a_2\}$ and $B = \{b_1, b_2\}$ have no common neighbor, i.e., $N(A) \cap N(B) = \emptyset$, and they have at most two neighbors outside the cycle, i.e., $|N(A) \setminus B| \leq 2$ and $|N(B) \setminus A| \leq 2$. If $a_1b_1a_2b_2$ is a desk, then A and B (as defined above) are alternatives.

Linear Programming Reduction. Nemhauser and Trotter [NJ75] proposed a linear programming (LP) based reduction rule using the following LP relaxation:

$$\text{minimize} \sum_{v \in V} x_v \text{ such that} \tag{3.1}$$

$$x_u + x_v \geq 1 \text{ for } \{u, v\} \in E, \tag{3.2}$$

$$x_v \geq 0 \text{ for } v \in V. \tag{3.3}$$

Nemhauser and Trotter [NJ75] showed that this LP relaxation (1) has an optimal half-integral solution, i.e., each variable is either 0, 1/2, or 1, and (2) if a variable takes an integer value (0 or 1) then there always exists an optimal integer solution where the variable has the same value. Finally, they showed that a half-integral solution can be computed using a maximum bipartite matching for the graph $G' = (L_V \cup R_V, E')$ with

$$L_V = \{l_v \mid v \in V\}, \tag{3.4}$$

$$R_V = \{r_v \mid v \in V\}, \tag{3.5}$$

$$E' = \{\{l_u, r_v\} \mid \{u, v\} \in E\}. \tag{3.6}$$

To compute a minimum vertex cover $\mathcal{C}'$ of G', Akiba and Iwata [AI16] then use the Hopcroft-Karp algorithm [HK73]. The resulting half-integral solution to the LP relaxation then is

$$x_v^* = \begin{cases} 0 & \text{if } l_v, r_v \notin \mathcal{C}' \\ 1 & \text{if } l_v, r_v \in \mathcal{C}' \\ \frac{1}{2} & \text{otherwise.} \end{cases} \tag{3.7}$$

Afterwards, vertices with a value of 1 can be added to the maximum independent set of G and are therefore removed along with their neighbors. Iwata et al. [IOY14] present an improved version of this approach that computes a solution to the LP relaxation whose half-integral part is minimal. By using this approach, the authors note that this reduction rule subsumes the crown reduction rule by Abu-Khzam et al. [Abu+04].

Unconfined Reduction. The unconfined reduction by Xiao and Nagamochi [XN13; XN17] is a generalization of the dominance reduction [FGK09] and satellite reduction by Kneis et al. [KLR09]. A vertex is called *unconfined* if Algorithm 3.1 returns true.

Algorithm 3.1 : Algorithm for finding unconfined vertices [XN13; XN17].

Data : $G = (V, E), v \in V$
Result : True if v is unconfined, False otherwise

```
1  S ← {v}
2  while True do
3      u ← u ∈ N(S) such that |N(u) ∩ S| = 1 and |N(u) \ N[S]| is minimized
4      if u = None then
5          return False
6      if N(u) \ N[S] = ∅ then
7          return True
8      if |N(u) \ N[S]| = 1 then
9          S ← S ∪ (N(u) \ N[S])
10     else
11         return False
```

Unconfined vertices are removable, i.e., they can be removed from G without affecting the independence number, since there always is a maximum independent set that does not include them. Thus, if G' is the graph resulting from removing an unconfined vertex v from G, then $\alpha(G) = \alpha(G')$. Finally, Hespe et al. [HSS19] note that the implementation of Akiba and Iwata also targets the so-called diamond reduction which is not explicitly mentioned in the original work [AI16].

Packing Reduction. The packing reduction by Akiba and Iwata [AI16] is based on the packing branching used in the branch-and-reduce algorithm by the same authors. This branching strategy creates so-called *packing constraints* that are updated incrementally. In particular, assume that $v \in V$ is the vertex that is branched on. In the branch that includes v in the solution (including branch), one can determine whether a maximum independent set containing v exists. Otherwise, in the branch that excludes v from the solution (excluding branch), one can be sure that there is no maximum independent set that contains v. Based on this observation, constraints for the remaining vertices can be derived. For example, if no maximum independent set includes v then at least two of v's neighbors have to be in a maximum independent set. Otherwise, one could exchange the neighbor of v that is part of the maximum independent set with v and obtain a maximum independent set of the same size. Thus, if x_u is a binary variable that indicates if $u \in V$ is part of the current solution computed by the branch-and-bound algorithm, it holds that $\sum_{u \in N(v)} x_u \geq 2$.

If $C \subseteq V$ is a set of candidate vertices, constraints of the form $\sum_{v \in C} x_v \geq k$ can be utilized in the following two reductions. (1) If k is equal to $|C|$, all vertices from C have to be

included in the current solution. Thus, if there exists an edge between the vertices in C, no valid solution exists and the corresponding branch can be pruned. (2) If there is a vertex $v \in V$ such that $|C| - |N(v) \cap C| < k$, then v has to be excluded from the current solution. Furthermore, if $k > |C|$, the constraint can not be fulfilled and the corresponding branch can be pruned.

3.1.5 Branch-and-Reduce

We now present the branch-and-reduce algorithm by Akiba and Iwata [AI16] in greater detail. This algorithm (and its set of reduction rules) serves as a main building block for multiple approaches presented in this chapter. High-level pseudocode is given in Algorithm 3.2.

Algorithm 3.2 : Branch-and-reduce algorithm by Akiba and Iwata [AI16].

Data : $G = (V, E)$, packing constraints P, current solution size c, lower bound k
Result : Size of maximum independent set for G

```
1  Solve (G, P, c, k)
2      (G, P, c) ← Reduce(G, P, c)
3      if Unsatisfied(P) then
4          return k
5      if c + UpperBound(G) < k then
6          return k
7      if G is empty then
8          return c
9      if G is not connected then
10         forall (G_i, P_i) ∈ Components(G, P) do
11             c ← c + Solve(G_i, P_i, 0, k − c)
12         return max(k, c)
13     (G_1, P_1, c_1), (G_2, P_2, c_2) ← Branch(G, P, c)
14     k ← Solve(G_1, P_1, c_1, k)                    // Excluding branch
15     k ← Solve(G_2, P_2, c_2, k)                    // Including branch
16     return k
```

As outlined in Section 2.2.3, the main idea of branch-and-reduce algorithms is to alternate between reductions and branching. In particular, their algorithm begins by exhaustively applying a set of reduction rules in a predefined order. For each reduction rule $r_1, \ldots, r_j$ the algorithm iterates over all vertices and tries to apply each rule r_i. If this leads to the successful application of a reduction rule r_i, the algorithm loops back to the first reduction rule r_1. If this process terminates, and no further reduction rule can be applied to a vertex, a reduced instance has been computed. The full order of reduction rules is as follows: degree-1, dominance, unconfined, linear programming, packing, vertex fold, twin, funnel, desk.

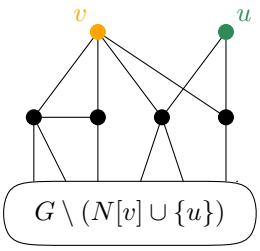


Figure 3.6: Example of a branching vertex v (orange) and a corresponding mirror vertex u (green). When excluding v from the solution, u can also be excluded.

Once the reduced instance has been computed, the algorithm continues by determining if the packing constraints (see Section 3.1.4) can be satisfied. Furthermore, it checks if the upper bound for the current branch is at least the size of the best solution so far. If this is not the case, the current branch can be pruned. Akiba and Iwata [AI16] implemented multiple upper bounds using clique covers, LP relaxation, and cycle covers. In their algorithm, they use the minimum of these as their final upper bound. If none of the above conditions prune the branch and the remaining graph is empty, the best solution can be updated. Otherwise, if the graph is not empty but contains multiple connected components, each of these is solved independently which speeds up the computation [AC19].

Finally, if there is only one component, a branching rule is applied resulting in two branches that can be solved recursively. For their branching rule, Akiba and Iwata [AI16] use the same strategy that is used in the algorithm by Fomin et al. [FGK09]. In particular, a vertex of maximum degree is selected. Tie-breaking is implemented by choosing the vertex $v \in V$ that minimizes the number of edges among its neighbors $N(v)$. The authors also compared this strategy to random and minimum degree selection. However, maximum degree branching performed significantly better in their experiments. The algorithm then performs recursion on the excluding and including branch (in this order). When branching, the algorithm also detects so-called mirrors and satellites. This allows them to avoid branching on certain vertices.

A *mirror* of a vertex $v \in V$ is a vertex $u \in N^2(v)$ such that $N(v) \setminus N(u)$ is a clique (that is possibly empty). If $M(v)$ is the set of mirrors of v, then, when excluding v from the solution during branching, one can also remove $M(v)$ without it impacting the independence number. In particular, $\alpha(G) = \max\{\alpha(G \setminus \{v\} \cup M(v)), 1 + \alpha(G \setminus N[v])\}$ [KLR09]. This prevents branching on mirrors individually and decreases the size of the remaining graph (and thus the depth of the search tree). An example of a mirror is shown in Figure 3.6.

According to Kneis et al. [KLR09] mirror branching only works well for cases where either $N^2(v)$ is large or $N^2(v)$ is small but contains some mirrors. To alleviate this, they propose satellite branching. A *satellite* of a vertex v is a vertex $u \in N^2(v)$ such that there exists a vertex $w \in N(v)$ and $N(w) \setminus N[v] = \{u\}$ [KLR09]. Let $S(v)$ be the set of satellites of v and $S[v] = S(v) \cup v$. Then, when including v into the solution during branching, one can also add $S(v)$ to the solution (and remove $S[v]$ from the graph). To be more specific,

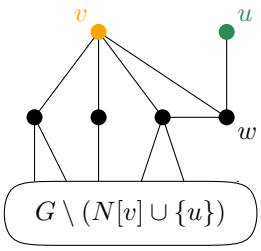


Figure 3.7: Example of a branching vertex v (orange) and a corresponding satellite vertex u (green). When including v into the solution, u can also be included.

$\alpha(G) = \max\{\alpha(G \setminus \{v\}), \alpha(G \setminus N[S[v]]) + |S(v)| + 1\}$ [KLR09]. This allows eliminating the satellites of a vertex from branching, again reducing the graph size and therefore search depth. An example of a satellite is shown in Figure 3.7. Finally, Akiba and Iwata [AI16] note that they do not explicitly use satellite branching since it is subsumed by the packing branching presented in Section 3.1.4.

3.1.6 The ARW Algorithm

We now review the algorithm by Andrade et al. [ARW12] (ARW) in more detail. Their algorithm follows the iterated local search metaheuristic as described in Section 2.2.1. We use this algorithm in several sections of this chapter. As mentioned in Section 3.1.2, the ARW local search extends the notion of vertex swaps to (j,k)-swaps, which remove j vertices from the current solution and insert k vertices instead. The authors present a linear-time implementation that, given a maximal solution, can find a $(1,2)$-swap or prove that no $(1,2)$-swap exists.

One iteration of the ARW algorithm consists of a perturbation and a local search step. During the local search step the algorithm iterates over all vertices of the graph and looks for a potential $(1,2)$-swap. By using a data structure that allows insertion and removal operations on vertices in time proportional to their degree, this procedure can find a valid $(1,2)$-swap in $\mathcal{O}(m)$ time, if it exists.

The perturbation step, used for diversification, forces vertices into the solution and removes neighboring vertices as necessary. In most cases a single vertex is forced into the solution; with a small probability, the number of forced vertices f is set to a higher value (f is set to $i+1$ with probability $1/2^i$). Vertices to be forced into a solution are picked from a set of random candidates, with priority given to those that have been outside the solution for the longest time.

An even faster incremental version of the algorithm (which we use in this chapter) maintains a list of *candidates*, which are vertices that may be involved in $(1,2)$-swaps. It ensures a vertex is not examined twice unless there is some change in its neighborhood. An external memory version of this algorithm by Liu et al. [Liu+15] runs on graphs that do not fit into memory on a standard machine. The ARW algorithm is efficient in practice,

finding maximum independent sets of optimal size orders of magnitude faster than exact algorithms on many benchmark graphs. However, it can not verify that these independent sets are optimal.

3.2 Inexact Iterative Reductions

Finding a maximum independent set in practice is commonly done using either branch-and-bound [Tom+13; SRJ11; ST14] or branch-and-reduce algorithms [AI16] and will be covered in greater detail in Sections 3.4 and 3.5. However, these algorithms often struggle to obtain solutions for huge sparse graphs with tens of millions of vertices or more. Furthermore, even the quality of existing heuristic-based solutions tends to degrade for massive inputs such as web graphs and road networks. Therefore, novel techniques are necessary to find high-quality independent sets for these graphs. For this purpose, we present a very natural evolutionary framework for the computation of large maximal independent sets. The core innovations of the evolutionary algorithm are new combine operations based on graph partitioning and local search algorithms. More precisely, we employ the state-of-the-art graph partitioner KaHIP [SS13a] to derive operations that enable us to quickly exchange *whole blocks* of given individuals. The newly computed offspring are then improved using a local search algorithm. In contrast to previous evolutionary algorithms [BK94; BZ03], each computed offspring is *valid*. Hence, we only allow valid solutions in our population and are able to use the cardinality of the independent set as a fitness function for individuals.

Additionally, we incorporate the advanced reduction rules used by Akiba and Iwata [AI16], which are known to be effective in exact algorithms. Our final algorithm may be viewed as performing two functions simultaneously: (1) reduction rules are used to boost the performance of the evolutionary algorithm *and* (2) the evolutionary algorithm opens up the opportunity for further reductions by selecting vertices that are likely to be in large independent sets. In short, we apply reduction rules to form a reduced instance, compute vertices to insert into the final solution, and remove their neighborhood (including the vertices themselves) from the graph so that further reductions can be applied. This process is repeated recursively. We show that this technique finds large independent sets much faster than previous local search algorithms, is competitive with state-of-the-art exact algorithms for smaller graphs, and allows us to compute large independent sets on huge sparse graphs, with billions of edges.

Organization. We begin this section by introducing important related work on evolutionary algorithms for MIS in Section 3.2.1. We then cover the core components of our evolutionary algorithm in Section 3.2.2, including the computation of initial solutions, as well as the combine and mutation operations. We explain how we use reductions in Section 3.2.3 and conclude with an experimental evaluation in Section 3.2.4.

3.2.1 Previous Work on Evolutionary Algorithms

We now discuss previous work on evolutionary algorithms for MIS. Both Bäck and Khuri [BK94] and Borisovsky and Zavolovskaya [BZ03] use fairly similar approaches. They encode

solutions as bitstrings such that the value at position i equals one if and only if vertex i is in the current solution. In both cases, a classic two-point crossover is used which randomly selects two crossover points p_1, p_2. Then all bits in between these positions are exchanged between both input individuals. Note that this likely results in invalid solutions even if both parents are valid solutions. To guide the search towards valid solutions a penalty approach is used. A major drawback of the work by Bäck and Khuri [BK94] is that the authors only test their algorithm on synthetic instances. Moreover, in both cases, experiments are only performed on very small instances—large instances are not considered.

Mehrabi et al. [MMM09] also use the same bitstring encoding and present a combine and repair operation using greedy heuristics. Their combine operation gathers the independent set vertices of two parents and sorts them by increasing degree. They then repeatedly select the lowest degree vertex from this combined set and add it to a single offspring if it does not contain any neighboring vertex. Mutation is performed by randomly flipping one bit in the bitstring of the offspring. Since this can result in infeasible solutions they use a repair operation that repeatedly selects the highest degree vertex and removes it from the solution until it becomes feasible. Again, their algorithm is only evaluated on a small set of graphs with up to 512 vertices.

3.2.2 Evolutionary Components

We now present our basic evolutionary algorithm, which we call EvoMIS. We begin by outlining the general structure of our evolutionary algorithm and then explain how we build the initial population. Finally, we present our new combine operations and how we handle mutation in our algorithm.

3.2.2.a) General Structure

As in previous work [BK94; BZ03] we use bitstrings as a natural way to represent individuals/solutions in our population. More precisely, an independent set $\mathcal{I}$ is represented as an array $s = \{0,1\}^n$ where $s[v] = 1$ if and only if $v \in \mathcal{I}$. The general structure of our evolutionary algorithm is very simple. We start with the creation of a population of individuals (in our case independent sets in the graph) and evolve the population into different populations over several rounds until a stopping criterion is reached.

In each round, our evolutionary algorithm uses a selection rule that is based on the fitness of the individuals (in our case the size of the independent set) of the population to select good individuals and combine them to obtain improved offspring. In contrast to previous work [BK94; BZ03], our combine and mutation operations always create valid independent sets. Hence, we use the size of the independent set as a fitness function, i.e., there is no need to use a penalty function to ensure that the final individuals generated by us are independent sets. As we will see later, when an offspring is generated it is possible that it is a non-maximal independent set. Thus, we apply one iteration of ARW local search without the perturbation step to ensure that each offspring is locally maximal and additionally apply a mutation operation. We use mutation operations since it is of major importance to keep

the diversity in the population high [Bäc96]. That is, the individuals should not become too similar, otherwise, the algorithm will converge prematurely.

We then use an eviction rule to select a member of the population and replace it with the new offspring. In general, one has to take into consideration both the fitness of an individual and the difference between individuals in the population [Bäc96]. We evict the solution that is *most similar* to the newly computed offspring among those individuals of the population that have a smaller or equal objective than the offspring itself. Once an individual has been accepted into the population, we further refine it using additional iterations of the ARW algorithm. The general structure of our evolutionary algorithm follows the memetic algorithm outline presented in Section 2.2.2. additional, we use a steady-state approach [Jon06] which generates only one offspring per generation.

3.2.2.b) Initial Solutions

We use two different approaches to create initial solutions. Each time we create an individual for the population, we pick one of the approaches uniformly at random. The first and most simplistic approach is to start from an empty independent set and add vertices at random until no further vertices can be added. To ensure that adding a vertex results in a valid independent set, we first check that the vertex has no neighbors that are already in the independent set. This method adds a decent amount of diversity during the construction phase, which over an extended period of time can lead to high-quality solutions.

Second, we use a greedy approach similar to Andrade et al. [ARW12]. Starting from an empty solution, we add the vertex with the least residual degree, which is the number of neighbors in the residual graph. After a vertex is added to the solution, we remove all its neighbors from the graph and update the residual degree of their neighbors. We repeat the procedure until the residual graph is empty. The implementation uses a simple bucket priority queue that groups vertices into buckets based on their residual degree. This allows us to pick a random vertex whenever multiple vertices share the same residual degree, thus creating diversity.

3.2.2.c) Combine Operations

To produce new offspring we perform different kinds of combine operations which are all based on graph partitioning. The main idea of our operations is to use a partition of the graph to exchange whole blocks of solution vertices. In general, our combination operations may generate new independent sets that are not maximal. We then perform a maximization step that adds as many vertices as possible. Afterwards, we apply a single iteration of the ARW local search algorithm to ensure that our solution is locally maximal. Depending on the type of operation, we use a vertex separator or an edge separator of the graph that has been computed by the graph partitioning framework KaHIP [SS13a]. As a side note, small edge or vertex separators are vital for our combine operations to work well. Otherwise, large separators in the combine operations yield offspring that are far from being maximal. Hence, the maximization step performs lots of vertex insertions and the computed offspring might have lower quality. This is supported by experiments presented in Section 3.2.4.a).

The first and second operations need precisely two input solutions while our third operation is a multi-point combine operation—it can take multiple input solutions. In the first case, we use a simple tournament selection rule [MG95] to determine the inputs (i.e., I_1 is the fittest out of k random individuals $r_1, \ldots, r_k$ from the population). The same is done to select I_2. Note that, since our algorithms are randomized, performing the same combine operation twice on the same parents can yield different offspring.

Vertex Separator Combination. In its simplest form, the operation starts by computing a vertex separator $\mathcal{S}$ of the input graph, such that V is separated into sets $V = V_1 \cup V_2 \cup \mathcal{S}$. We then use $\mathcal{S}$ as a crossover point for our operation. The operation generates two offspring, $O_1 = (V_1 \cap I_1) \cup (V_2 \cap I_2)$ and $O_2 = (V_1 \cap I_2) \cup (V_2 \cap I_1)$. In other words, we exchange whole parts of independent sets from the blocks V_1 and V_2 of the vertex separator. Note that the exchange can be implemented in time linear in the number of vertices. Recall that the definition of a vertex separator implies that no edges are running between V_1 and V_2. Hence, the computed offspring are independent sets, but may not be maximal since separator vertices have been ignored and potentially some of them can be added to the solution. We maximize these offspring by using the greedy independent set algorithm from Section 3.2.2.b). The operation finishes with one iteration of the ARW algorithm to ensure that we reached a local optimum and to add some diversification. An example illustrating the combine operation is shown in Figure 3.8.

Edge Separator Combination. This operation computes offspring by using complementary vertex covers. It starts by computing a bipartition $V = V_1 \cup V_2$ of the graph. Let $\mathcal{C}_i$ be the vertex cover $V \backslash \mathcal{I}_i$. We define two temporary vertex cover offspring similar to before: $D_1 = (\mathcal{C}_1 \cap V_1) \cup (\mathcal{C}_2 \cap V_2)$ and $D_2 = (\mathcal{C}_1 \cap V_2) \cup (\mathcal{C}_2 \cap V_1)$. However, it is possible that an offspring created this way contains some non-covered edges. These edges can only be a subset of the cut edges of the partition. We want to add as few vertices as possible to our solution to fix this. Hence, we add a minimum vertex cover of the bipartite graph induced by the non-covered cut edges to our vertex cover offspring. The minimum vertex cover in a bipartite graph can be computed using the Hopcroft-Karp algorithm [HK73]. Afterwards, we transform the vertex cover back to an independent set and follow our general approach by applying ARW local search to reach a local optimum.

Multi-way Combination. Our last two operations are multi-point crossover operations that extend the previous two operations. Both of them divide the graph into k blocks. Depending on the type of operation, a vertex or edge separator is used. We start with the description of the vertex separator approach where $V = V_1 \cup \ldots \cup V_k \cup \mathcal{S}$. The operation selects a number of parents. We then calculate the score for *every* possible parent, block pair (I_i, V_j), which is $|I_i \cap V_j|$—the number of the parent's solution vertices in the given block. We select the parent with the *highest score* for each of the blocks to compute the offspring. As before, we use a maximization step to make the solution maximal and afterwards apply ARW local search to ensure that our solution is a local optimum.

If we use an edge separator for the combination, we start with a k-way partition of the vertices $V = V_1 \cup \ldots \cup V_k$. This approach also computes scores for each pair of parent and block. This time the score of a pair is defined as the number of the vertex cover vertices of

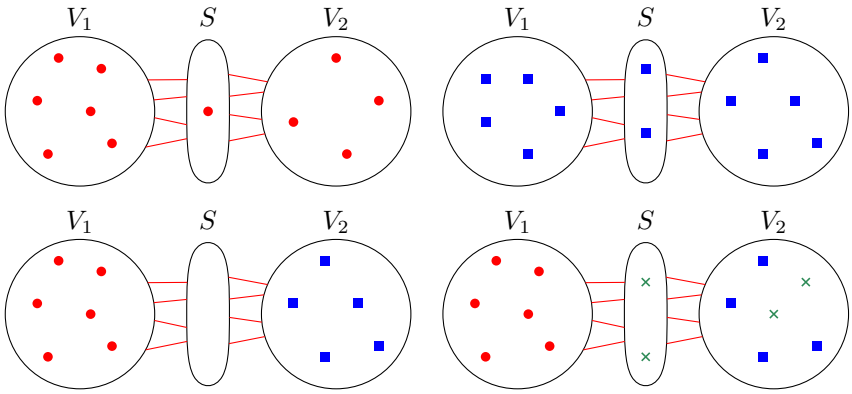


Figure 3.8: An example combine operation using a vertex separator $V = V_1 \cup V_2 \cup \mathcal{S}$. On top two input independent sets, I_1 and I_2, are shown. Bottom left: a possible offspring that uses the independent set of I_1 in block V_1 and the independent set of I_2 in block V_2. Bottom right: the improved offspring after ARW local search has been applied to improve the given solution and to add vertices from the separator to the independent set.

the complement of an independent set inside the given block. We select the parent with the *lowest score* for each of the blocks to compute the offspring. As in the simple vertex cover combine operation, it is possible that some cut edges are not covered. We use the simple greedy vertex cover algorithm to fix the offspring since the graph induced by the non-covered cut edges is not bipartite anymore. We then once again complement our vertex cover to get our final offspring.

3.2.2.d) Mutation Operations

After we performed a combine operation, we apply a mutation operation to introduce further diversification. Previous work [BK94; BZ03] uses bit-flipping for mutation, i.e., every bit in the representation of a solution has a certain probability of being flipped. We can not use this approach since our population only allows valid solutions. Instead, we perform forced insertions of new vertices into the solution and remove adjacent solution vertices if necessary similar to the perturbation routine of the ARW algorithm. Afterwards, we perform ARW local search to improve the perturbed solution.

3.2.2.e) Miscellaneous

Instead of computing a new partition for every combine operation, we hold a pool of partitions and separators that is computed once at the beginning of the algorithm's execution. A combine operation then picks a random partition or vertex separator from this precomputed pool. If the combine operations have been unsuccessful for a number of iterations, we

compute a fresh set of partitions. In our experiments, we used 200 unsuccessful combine operations as a threshold. We have to ensure that the partitions created for the combine operations are sufficiently different over multiple runs. However, although KaHIP is a randomized algorithm, small cuts in a graph may be similar due to the graph's inherent structure. To avoid similar cuts and increase the diversification of the partitions and vertex separators, we additionally give KaHIP a random imbalance $\epsilon \in_{\text{rnd}} [0.05, 0.75]$ to solve the partitioning problem.

In addition to our previously presented combine operations, we also tried a combine operation based on set intersection. This operation computes an offspring by keeping the vertices that are in both inputs which is by definition an independent set. Note that this is similar to the combine operation proposed by Mehrabi et al. [MMM09]. However, our experiments with the operation did not yield good results, thus we omit further investigations.

3.2.3 Reduction Algorithms

We now describe how we employ the reduction rules used by the algorithm of Akiba and Iwata [AI16]. Furthermore, we present how we use inexact reductions based on intermediate solutions to open up additional reduction space. Finally, we present an acceleration technique for our separator-based combine operations.

Faster Evolutionary Computation of Independent Sets. Our algorithm begins by applying the reduction rules by Akiba and Iwata to compute a reduced instance $\mathcal{K}$. We then apply the evolutionary algorithm on $\mathcal{K}$, thus boosting its performance due to the smaller size of the reduced instance. We stop the evolutionary algorithm after μ unsuccessful combine operations and choose an individual (independent set) $\mathcal{I}$ with the highest fitness in the population. This corresponds to an intermediate solution to the input problem, whose size we can compute based on some simple bookkeeping, i.e., without actually reconstructing the full solution in G. Instead of stopping the algorithm, we use $\mathcal{I}$ to further reduce the graph (as we next show) and repeat the process of applying exact reduction rules and using the evolutionary algorithm on the further reduced graph $\mathcal{K}'$ recursively.

Our *inexact* reduction technique enables new reductions by selecting a subset $\mathcal{U}$ of the independent set vertices in the individual $\mathcal{I}$ which has the highest fitness. These vertices and their neighbors are then removed from the reduced instance. Based on the intuition that high-degree vertices in $\mathcal{I}$ are unlikely to be in a large solution (consider for example the optimal independent set on a star graph), we choose λ vertices from $\mathcal{I}$ with the smallest degree as subset $\mathcal{U}$. Using a modified quick selection routine this can be done in linear time. Ties are broken randomly. It is easy to see that it is likely that some exact reduction techniques become applicable again. Another view on the inexact reduction is that we use the evolutionary algorithm to find vertices that are likely to be in a large independent set. The overall process is repeated until the reduced instance is empty (or a time limit is reached). We present pseudocode in Algorithm 3.3.

Additional Acceleration. We now propose a technique to accelerate separator-based combine operations, which are the centerpiece of the evolutionary portion of our algorithm. Recall that after performing a combine operation, we first use a greedy algorithm on the

Algorithm 3.3 : High-level overview of ReduMIS.

Data : $G = (V, E)$, solution size offset γ (initially zero)
Result : Best solution $\mathcal{I}$

if $n = 0$ **then return**
else
  // Apply exact reductions and intermediate solution
  $(\mathcal{K}, \theta) \leftarrow$ applyExactReductions(G) // Exact reductions, solution size offset θ
  $\mathcal{I} \leftarrow$ EvoMIS ($\mathcal{K}$) // Intermediate independent set
  if $|\mathcal{I}| + \gamma + \theta > |\mathcal{I}|$ **then** update $\mathcal{I}$

  // Apply inexact reductions
  select $\mathcal{U} \subseteq \mathcal{I}$ s.t. $|\mathcal{U}| = \lambda$, $\forall u \in \mathcal{U}, v \in \mathcal{I} \backslash \mathcal{U} : d_K(u) \leq d_K(v)$ // Fixed vertices
  $\mathcal{U} = \mathcal{U} \cup N(\mathcal{U})$ // Augment $\mathcal{U}$ with its neighbors
  $\mathcal{K}' \leftarrow \mathcal{K}[V_K \backslash \mathcal{U}]$ // Inexact reduced instance

  // Recurse on inexact reduced instance
  ReduMIS ($\mathcal{K}'$, $\gamma + \theta + |\mathcal{U}|$) // Recursive call with updated offsets
return $\mathcal{I}$

separator to maximize the offspring and then employ ARW local search to ensure that the output individual is locally optimal with respect to (1,2)-swaps. However, the ARW local search algorithm uses *all* independent set vertices for initialization. Since large subsets of the created individual are already locally maximal (due to the nature of combine operations, which takes as input locally maximal individuals), it is sufficient to initialize ARW local search with the independent set vertices in the separator (added by the greedy algorithm) and the solution vertices adjacent to the separator.

3.2.4 Experimental Evaluation

Methodology. We have implemented the algorithm described above using C++ and compiled all code using g++ version 4.6.3 with full optimizations turned on (`-O3` flag). Our implementation includes the reduction routines, local search, and the evolutionary algorithm. We used the fastsocial configuration of the KaHIP v0.6 graph partitioning package [SS13a] to obtain graph partitions and vertex separators necessary for the combine operations of the evolutionary algorithm. We mainly compare our algorithms against the ARW algorithm (since it has a relatively clear advantage in the experiments performed by Andrade et al. [ARW12]) and the exact algorithm by Akiba and Iwata [AI16]. For the exact algorithm by Akiba and Iwata [AI16], we use the same implementation as the authors, which we compile and run sequentially with Java 1.8.0_40. For the exact algorithm, we mark the running time with "-" when the instance could not be solved within ten hours or could not be solved due to stack overflow. Unless otherwise mentioned, we perform five independent runs of each

algorithm with different random seeds, where each algorithm is run sequentially with a ten-hour wall-clock time limit to compute its best solution.

We use two of the machines described in Section 2.3.2.c). In particular, Machine A is used for the experiments in Section 3.2.4.a) and in Section 3.2.4.b). Experiments for the ARW algorithm, the exact algorithm, and the original EvoMIS algorithm were also run on Machine A. Additionally, we use Machine B in Section 3.2.4.b) to solve the largest instances.

We present two kinds of data: (1) the solution size statistics aggregated over the five runs, including maximum and average values, and (2) convergence plots as described in Section 2.3.2.b). Our set of instances includes a subset of the social networks, meshes, graphs from finite element computations, sparse matrices, and large web graphs presented in Section 2.3.2.a). A full overview of the instances used is given in Appendix A. In particular, we conduct our experiments on all instances used by Lamm et al. [LSS15a] and extend it by additional large instances with up to billions of edges including web graphs and road networks. Finally, we also evaluate the three hardest instances that were solved by Akiba and Iwata [AI16].

Algorithm Configuration. After an extensive evaluation of the parameters, we fixed the population size to 250, the partition pool size to 30, the number of ARW iterations to 15000, and the number of blocks for the multi-way combine operations to 64. In each iteration, one of our four combine operations is picked uniformly at random. We fixed the convergence parameter μ to 1000 and the inexact reduction parameter λ to $0.1 \cdot |\mathcal{I}|$. However, our experiments indicate that our algorithm is not too sensitive about the precise choice of the parameters. We mark instances that have been used for the parameter tuning with a *.

3.2.4.a) Evaluation of EvoMIS

We now shortly summarize the main results of experiments concerning EvoMIS. We present detailed data in Tables 3.1–3.4. In 50 out of the 67 instances, we either improve or reproduce the maximum result computed by the ARW algorithm. However, EvoMIS only computes a maximum solution that is strictly larger than the maximum solution computed by the ARW algorithm in 21 cases. In 17 cases the maximum result of the ARW algorithm is larger than the maximum result of our algorithm. Thus, there is no clear advantage for either algorithm on the instances we have tested. Remarkably, when looking at the graphs obtained from the Florida Sparse Matrix collection, the ARW algorithm only outperforms our algorithm on one instance.

The mesh family that we use in this section was also used in the original paper [ARW12] that introduced the ARW algorithm. We like to stress that most of the maximum results of the ARW algorithm are strictly larger than the maximum values originally reported by [ARW12] (including the maximum values presented there of the algorithm by Grosso et al. [GLP08]). Except for four instances the same holds for EvoMIS. On these four instances, EvoMIS is worse than the original maximum value of the ARW algorithm. On the mesh family, in eight out of 14 cases our algorithm computes the best result ever reported in previous literature. On road networks and the largest graphs from the mesh family, as well as the Walshaw family, the ARW algorithm outperforms EvoMIS. Even with more time (i.e., a

Table 3.1: Results for social networks. Bold values represent the largest independent set size found by any algorithm tested.

Graph			ReduMIS		EvoMIS		ARW	
Name	n	Opt.	Avg.	Max.	Avg.	Max.	Avg.	Max.
enron	69 244	62 811	62 811	**62 811**	62 811	**62 811**	62 811	**62 811**
loc-Gowalla	196 591	112 369	112 369	**112 369**	112 369	**112 369**	112 369	**112 369**
citation	268 495	150 380	150 380	**150 380**	150 380	**150 380**	150 380	**150 380**
cnr-2000*	325 557	-	230 036	**230 036**	229 981	229 991	229 955	229 966
google	356 648	174 072	174 072	**174 072**	174 072	**174 072**	174 072	**174 072**
coPapers	434 102	47 996	47 996	**47 996**	47 996	**47 996**	47 996	**47 996**
skitter	554 930	-	328 626	**328 626**	328 519	328 520	328 609	328 619
amazon-2008	735 323	-	309 794	**309 794**	309 774	309 778	309 792	309 793
in-2004	1 382 908	896 762	896 762	**896 762**	896 581	896 585	896 477	896 562

Table 3.2: Results for road networks. Bold values represent the largest independent set size found by any algorithm tested.

Graph			ReduMIS		EvoMIS		ARW	
Name	n	Opt.	Avg.	Max.	Avg.	Max.	Avg.	Max.
ny*	264 346	-	131 502	**131 502**	131 384	131 395	131 481	131 485
bay	321 270	166 384	166 384	**166 384**	166 329	166 345	166 368	166 375
col	435 666	225 784	225 784	**225 784**	225 714	225 721	225 764	225 768
fla	1 070 376	549 637	549 637	**549 637**	549 093	549 106	549 581	549 587

whole day of computation), ARW still outperforms EvoMIS. Thus, one can not assume that the evolutionary algorithm will always outperform the iterated local search when given a larger time limit.

We also implemented the algorithm presented by Bäck and Khuri [BK94]. Their algorithm uses a two-point crossover as a combine operation, as well as a bit-flip approach or mutation. Solutions created by the combine and mutation operations can be invalid. Hence, a penalty approach is used to deal with invalid solution candidates. In the original paper, the algorithm is only tested on small synthetic or random instances ($\leq$ 200 vertices). We tested the algorithm on the four smallest graphs from the mesh family and gave it ten hours to compute a solution. However, the best valid solution created during the course of the algorithm *never* exceeded the size of the best solution after the initial population has been created. This is due to the fact that the two-point crossover and the mutation operations found valid solutions very rarely and the average solution quality of the population degrades over time. On average, the final solution quality of the algorithm is at least 20% worse than the final result of EvoMIS. Due to the low solution quality we observed, we did not perform additional experiments with this algorithm.

Table 3.3: Results for mesh type graphs. Bold values represent the largest independent set size found by any algorithm tested.

Graph			ReduMIS		EvoMIS		ARW	
Name	n	Opt.	Avg.	Max.	Avg.	Max.	Avg.	Max.
beethoven	4 419	-	2 004	**2 004**	2 004	**2 004**	2 004	**2 004**
cow	5 036	-	2 346	**2 346**	2 346	**2 346**	2 346	**2 346**
venus	5 672	-	2 684	**2 684**	2 684	**2 684**	2 684	**2 684**
fandisk	8 634	-	4 074	**4 075**	4 075	**4 075**	4 073	4 074
blob	16 068	-	7 250	**7 250**	7 249	**7 250**	7 249	**7 250**
gargoyle	20 000	-	8 852	8 852	8 853	**8 854**	8 852	8 853
face	22 871	-	10 217	**10 218**	10 218	**10 218**	10 217	10 217
feline	41 262	-	18 853	**18 854**	18 853	**18 854**	18 847	18 848
gameguy	42 623	-	20 726	**20 727**	20 726	**20 727**	20 670	20 690
bunny*	68 790	-	32 346	**32 348**	32 337	32 343	32 293	32 300
dragon	150 000	-	66 438	66 449	66 373	66 383	66 503	**66 505**
turtle	267 534	-	122 417	122 437	122 378	122 391	122 506	**122 584**
dragonsub	600 000	-	281 561	281 637	281 403	281 436	282 006	**282 066**
ecat	684 496	-	322 363	322 419	322 285	322 357	322 362	**322 529**
buddha	1 087 716	-	480 072	480 104	478 879	478 936	480 942	**480 969**

Table 3.4: Results for Walshaw benchmark graphs. Bold values represent the largest independent set size found by any algorithm tested.

Graph			ReduMIS		EvoMIS		ARW	
Name	n	Opt.	Avg.	Max.	Avg.	Max.	Avg.	Max.
crack	10 240	4 603	4 603	**4 603**	4 603	**4 603**	4 603	**4 603**
vibrobox	12 328	-	1 852	**1 852**	1 852	**1 852**	1 850	1 851
4elt	15 606	-	4 943	**4 944**	4 944	**4 944**	4 942	**4 944**
cs4	22 499	-	9 167	9 168	9 172	**9 177**	9 173	9 174
bcsstk30	28 924	1 783	1 783	**1 783**	1 783	**1 783**	1 783	**1 783**
bcsstk31	35 588	3 488	3 488	**3 488**	3 488	**3 488**	3 487	3 487
fe_pwt	36 519	-	9 309	9 309	9 309	**9 310**	9 310	**9 310**
brack2	62 631	21 418	21 418	**21 418**	21 417	21 417	21 416	21 416
fe_tooth	78 136	27 793	27 793	**27 793**	27 793	**27 793**	27 792	27 792
fe_rotor	99 617	-	22 010	22 016	22 022	22 026	21 974	**22 030**
598a*	110 971	-	21 814	21 819	21 826	21 829	21 891	**21 894**
fe_ocean	143 437	-	71 706	**71 716**	71 390	71 576	71 492	71 655
wave	156 317	-	37 054	37 060	37 057	**37 063**	37 023	37 040
auto	448 695	-	83 873	83 891	83 935	83 969	84 462	**84 478**

The Role of Graph Partitioning. To estimate the influence of good partitioning in this context, we performed an experiment in which partitions of the graph have been obtained by simple breadth-first searches. More precisely, we obtain a two-way partition of the graph using a breadth-first search starting from a random vertex. The search is stopped as soon as a specified number of vertices has been touched. Every vertex touched by the search is added to the first block, and every vertex not touched by the search is added to the second block. In our experiments, these low-quality partitions lead to significantly worse results than using a high-quality graph partitioner.

3.2.4.b) Boosting Algorithm Performance—ReduMIS

In this section, we compare the solution quality and performance of our reduction-based algorithm (ReduMIS) with EvoMIS, local search (ARW), and the exact algorithm by Akiba and Iwata [AI16]. Again, we present detailed data for ReduMIS, EvoMIS, and ARW in Tables 3.1–3.4. Results for the comparison of ReduMIS and the exact algorithm are given in Table 3.5. We briefly summarize the main results of our experiments.

First, ReduMIS *always* improves or preserves solution quality on all tested social or road networks. On four social networks (cnr-2000, skitter, amazon-2008, and in-2004) and all road networks, we compute a solution strictly larger than EvoMIS and ARW. Furthermore, for those social and road networks where we know the exact maximum independent set size, ReduMIS finds an optimal solution.

On mesh-like networks, ReduMIS computes solutions that are sometimes better and sometimes worse than our previous formulation of the evolutionary algorithm; however, ARW performs significantly better than both algorithms on large meshes (See Table 3.3). The same pattern can also be seen for the Walshaw benchmarks, many of which are finite-element meshes. As shown in Table 3.4, all algorithms give similar results on most instances, though ReduMIS outperforms ARW on the fe_ocean instance, and ARW significantly outperforms ReduMIS and EvoMIS on the auto instance. On graphs from the Florida Sparse Matrix collection, all algorithms give similar results. Therefore, we exclude them from our analysis and refer the reader to Appendix B.

The convergence plots in Figure 3.9 show that the running time of the evolutionary algorithm is reduced when using reductions, especially on the road and social networks. Once the first irreducible graph is computed, ReduMIS quickly outperforms ARW and EvoMIS. For additional convergence plots, we refer to Appendix C.

Next, we compare ReduMIS to the exact algorithm by Akiba and Iwata [AI16]. Table 3.5 lists instances that the exact algorithm can solve and compares its running time with the time it takes ReduMIS to reach a (maximum) independent set of the same size. Note that it is possible that the exact algorithm finds an exact solution before it is able to verify its correctness, which is only done at the end of their algorithm's execution. In general, the exact algorithm performs as expected: it either quickly solves an instance (typically in a few seconds) or it cannot solve the instance within the ten-hour limit. Our experiments indicate that the success of the exact algorithm is tied to the size of the reduced instance. For most instances, if the reduced instance is too large the algorithm does not finish within the ten-hour time limit. Since the exact reduction rules work well for social networks and road

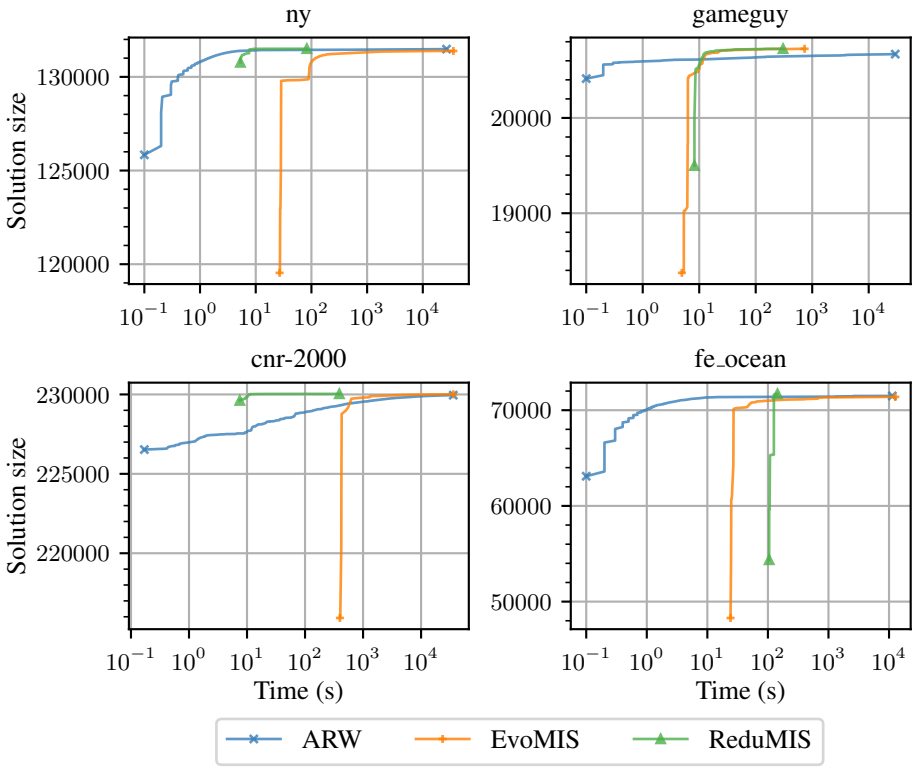


Figure 3.9: Convergence plots for ny (top left), gameguy (top right), cnr-2000 (bottom left) and fe_ocean (bottom right).

networks, the algorithm can solve many of these instances. However, the exact algorithm cannot solve any mesh graphs and fails to solve many other instances, since reductions do not produce a small reduced graph on these instances.

Even though our algorithm is not optimized for speed[1], the running time of our algorithm is still competitive on most instances. However, on some instances, our algorithm outperforms the exact algorithm by far. The largest speed-up is obtained on bcsstk30, where our algorithm is about four orders of magnitude faster. Conversely, there are several instances for which ReduMIS needs much more time; for example, Oregon-1 is solved 364 times faster by the exact algorithm. In addition to comparing against the benchmark instances, we also ran our algorithm on the hardest instances computed exactly in the paper by Akiba and Iwata [AI16]: as-Skitter-big, web-Stanford, and libimseti. In all cases, ReduMIS computes the optimal result on these instances as well. ReduMIS is a factor of 147 and 2 faster than

[1] For example, our evolutionary algorithm builds all the partitions needed for the combine operations when the algorithm starts.

Table 3.5: Running times for ReduMIS and the exact algorithm on the graphs that the exact algorithm could solve. The running time t_{ReduMIS} is the average time when ReduMIS finds the optimal solution. Instances marked with a † are the hardest instances solved exactly in [AI16]. Running times in bold are those where ReduMIS found the exact solution faster than the exact algorithm.

Graph	Opt.	t_{ReduMIS}	t_{exact}
a5esindl	30 004	0.07	0.07
as-Skitter-big†	1 170 580	1 262.46	2 838.030
bay	166 384	14.32	2.33
bcsstk30	1 783	**2.71**	31 152.16
bcsstk31	3 488	3.11	2.20
blockqp1	20 011	46.33	3.89
brack2	21 418	**9.43**	792.48
c-57	19 997	6.70	0.03
c-67	31 257	1.02	0.03
c-68	36 546	0.45	0.05
ca-HepPh	4 994	0.50	0.07
case9	7 224	3.23	0.54
cbuckle	1 097	3.83	1.34
citation	150 380	0.52	0.49
col	225 784	**27.93**	7 140.28
coPapers	47 996	3.21	1.45
crack	4 603	**0.05**	0.06
crankseg_2	1 735	**1.86**	2.63
cyl6	600	0.86	0.29
dixmaanl	20 000	**12.27**	13.62
Dubcova1	4 096	0.07	0.05
enron	62 811	3.08	0.06
fe_tooth	27 793	**0.20**	0.439
fla	549 637	**20.10**	22.50
in-2004	896 762	7.82	5.08
loc-Gowalla	112 369	0.79	0.33
libimseti†	127 294	28 375.4	1 729.05
olafu	735	3.84	1.44
Oregon-1	9 512	2.55	0.01
raefsky4	1 055	0.86	0.33
rajat07	4 971	**0.02**	0.05
skirt	2 383	0.14	0.14
TSOPF_FS_b300_c2	28 338	139.25	32.83
web-Google	174 072	2.95	0.83
web-Stanford†	163 390	**316.15**	46 450.11

the exact algorithm on web-Stanford and as-Skitter-big, respectively. However, we need a

Table 3.6: Results on huge instances. Value $n(\mathcal{K})$ denotes the number of nodes of the first exact reduced graph and $\overline{n}(\mathcal{K}'')$ denotes the average number of nodes of the first inexact reduced graph. Column ℓ presents the average recursion depth in which the best solution was found and t_{avg} denotes the average time when the solution was found. Entries marked with a * indicate that ARW could not solve the instance.

Graph	n	m	$n(\mathcal{K})$	$\overline{n}(\mathcal{K}'')$	Avg.	Max	ℓ	t_{avg}	Max_{ARW}
europe	≈18.0M	≈22.2M	11 879	826	9 267 810	9 267 811	1.4	2m	9 249 040
USA-road	≈23.9M	≈28.8M	169 808	8 926	12 428 075	12 428 086	2.0	38m	12 426 262
eu-2005	≈862K	≈16.1M	68 667	55 848	452 352	452 353	1.4	26m	451 813
uk-2002	≈19M	≈261M	241 517	182 213	11 951 998	11 952 006	4.6	213m	*
it-2004	≈41M	≈1.0G	1 602 560	1 263 539	25 620 513	25 620 651	1.4	26.1h	*
sk-2005	≈51M	≈1.8G	3 200 806	2 510 923	30 686 210	30 686 446	1.4	27.3h	*
uk-2007	≈106M	≈3.3G	3 514 783	-	67 285 232	67 285 438	1.0	30.4h	*

factor of 16 more time to find an optimal solution for libimseti. Our investigations revealed that libimseti has many high degree vertices, and therefore our inexact reductions (which force low degree vertices into the solution) are not as effective on this graph.

Additional Experiments. We now run our ReduMIS algorithm on the largest instances of our benchmark collection: road networks (europe, USA-road) and web graphs (eu-2005, uk-2002, it-2004, sk-2005, and uk-2007). For these experiments, we reduced the convergence parameter μ to 250 in order to speed up computation. On the three largest graphs it-2004, sk-2005 and uk-2007, we set the time limit of our algorithm to 24 hours after the first exact irreducible graph has been computed by the reduction routine. Table 3.6 gives detailed results of the algorithm including the size of the first exact irreducible graph. We note that the first (exactly) reduced graph is much smaller than the input graph: the reduction rules shrink the graph size by at least an order of magnitude. The largest reduction can be seen on the europe graph, for which the first reduced graph is more than three orders of magnitude smaller than the original graph. However, most of the reduced instances are still too large to be solved by the exact algorithm. As expected, applying our inexact reduction technique (i.e., fixing vertices into the solution and applying exact reductions afterwards) further reduces the size of the input graph. On road networks, inexact reductions reduce the graph size again by an order of magnitude. Moreover, the best solution found by our algorithm is not found on the first exact reduced graph $\mathcal{K}$, but in deeper recursion levels. In other words, the best solution found by our algorithm is found on an inexact reduced graph. We run ARW on the original instances as well, giving it as much time as our algorithm consumed to find its best solution. It could not handle the largest instances and computes smaller independent sets on the other instances. In the next section, we present variants of the ARW local search that use reduction rules to alleviate this.

3.3 On-the-fly Reductions

As already discussed in the previous section, exact algorithms for MIS and its related problems can take exponential time to find solutions, making *massive* graphs infeasible to solve in practice. Instead, heuristic algorithms such as local search are used to efficiently compute high-quality independent sets. For many practical instances, some local search algorithms even quickly find exact solutions [ARW12; GLP08]. While approaches like EvoMIS and ReduMIS described in Section 3.2 are able to compute very high-quality solutions, they still require significant time to compute them. Thus, we now present an advanced local search algorithm that quickly computes large independent sets by combining iterated local search with reduction rules that reduce the size of the search space without losing solution quality. We do so by either (1) computing a reduced instance and then applying local search on the reduced graph or (2) applying reduction rules on-the-fly during the local search, i.e., testing if reductions can be applied while vertices are swapped in and out of the solution. By doing so we significantly boost the performance of the local search algorithm, especially on huge sparse networks where reduction can be applied with great success. In addition to exact reduction techniques, we also apply inexact reductions that remove high-degree vertices from the graph. In particular, we show that cutting a small percentage of high-degree vertices from the graph minimizes performance bottlenecks of local search while maintaining high solution quality. Experiments indicate that our algorithm can outperform previous state-of-the-art algorithms both in terms of speed and quality.

Organization. The rest of this section is organized as follows. We describe the core components of our algorithm in Section 3.3.1. Here, we exemplify exact reduction techniques as well as our inexact cutting procedures. Then we present experiments to evaluate our algorithm in Section 3.3.2. Our experimental evaluation indicates that the reduction techniques indeed boost the performance of the ARW local search, especially on huge sparse networks that are hard to handle for the original algorithm (as seen in the previous Section).

3.3.1 Techniques for Accelerating Local Search

First, we note that while local search techniques such as ARW perform well on huge uniformly sparse mesh-like graphs, they perform poorly on complex networks, which are typically scale-free. We first discuss why local search performs poorly on huge complex networks, then introduce the techniques we use to address these shortcomings.

The first performance issue is related to vertex selection for perturbation. Many vertices are guaranteed to be in some maximum independent set. These include, for example, vertices with degree one. However, ARW treats such vertices like any other. During a perturbation step, these vertices may be forced out of the current solution, causing extra searches that may not improve the solution.

The second issue is that high-degree vertices may slow down ARW significantly. Most internal operations of ARW (including (1,2)-swaps) require traversing the adjacency lists of multiple vertices, which takes time proportional to their degree. High-degree vertices are only scanned if they have at most one solution neighbor (or belong to the solution

themselves). However, this can often be the case if high-degree vertices are interconnected as is the case in complex networks.

A third issue is caused by the particular implementation of the ARW algorithm. When performing a (1,2)-swap involving the insertion of a vertex v, the original ARW implementation (as tested by Andrade et al. [ARW12]) picks a pair of neighbors u, w of v at random among all valid ones. Although this technically violates the $\mathcal{O}(m)$ worst-case bound (which requires the first such pair to be taken), the effect is minimal on small-degree networks. On large complex networks, this can become a significant bottleneck.

To deal with the third issue, we simply modified the ARW code to limit the number of valid pairs considered to a small constant (100). Addressing the first two issues requires more involved techniques, i.e., reductions and *high-degree vertex cutting*, as we discuss next.

3.3.1.a) Exact Reductions

First, we investigate the use of reduction rules. Recall that computing a reduced instance is achieved by the repeated application of reductions to the input graph G until it cannot be reduced further, producing an instance $\mathcal{K}$. Even simple reduction rules can significantly reduce the graph size [Str16]. Indeed, in some cases, $\mathcal{K}$ may be empty—giving an exact solution without requiring any additional steps. We note that this is the case for many of the graphs used in the experiments by Akiba and Iwata [AI16]. Furthermore, any solution of $\mathcal{K}$ can be extended to a solution of the input.

The size of the reduced instance depends entirely on the structure of the input graph. In many cases, the reduced instance can still be too large, making it intractable to find an exact maximum independent set in practice (see Sect. 3.3.2). In this case "too large" can mean a few thousand vertices. However, for many graphs, the reduced instance is still significantly smaller than the input graph, and even though it is intractable for exact algorithms, local search algorithms such as ARW have been shown to find a maximum independent set quickly on small benchmark graphs. Therefore, it stands to reason that ARW would perform better on the reduced instances.

3.3.1.b) Inexact Reductions: Cutting High-Degree Vertices

To further boost local search, we investigate removing (cutting) high-degree vertices outright. This is a natural strategy: intuitively, vertices with very high degrees are unlikely to be in a large independent set (consider a maximum independent set of graphs with few high-degree vertices, such as a star graph or scale-free networks). In particular, many reduction rules show that low-degree vertices are guaranteed to be in some maximum independent sets, and applying these reductions results in a small reduced instance (see Section 3.2), where high-degree vertices are left behind. This is especially true for huge complex networks considered here, which generally have few high-degree vertices.

Besides intuition, there is much additional evidence to support this strategy. In particular, the natural greedy algorithm that repeatedly selects low-degree vertices to construct an independent set is typically within 1%–10% of the maximum independent set size for sparse

graphs [ARW12]. Moreover, several successful algorithms make choices that favor low-degree vertices. ReduMIS (see Section 3.2) forces low-degree vertices into an independent set in a multi-level algorithm, giving high-quality independent sets as a result. Exact branch-and-bound algorithms also order vertices so that vertices of high-degree are excluded first during search. In particular, optimal and near-optimal independent sets are typically found after high-degree vertices have been evaluated and excluded from search; however, it is then much slower to find the remaining solutions, since only low-degree vertices remain in the search. This behavior can be observed in the experiments of Batsyn et al. [Bat+14], where better initial solutions from local search significantly speed up exact search because non-promising parts of the search space can be cut.

We consider two strategies for removing high-degree vertices from the graph. When we cut by *absolute degree*, we remove the vertices with a degree higher than a threshold. In *relative degree* cutting, we iteratively remove the highest-degree vertices and their incident edges from the graph. This approach mirrors the greedy algorithm that repeatedly selects the smallest-degree vertices in the graph to be in an independent set until the graph is empty. We stop when a fixed fraction of all vertices is removed. This better ensures that vertices that are part of a cluster of high-degree vertices are not removed, since some of these vertices can be part of a large independent set.

3.3.1.c) Putting Things Together

We use reductions and cutting in two ways. First, we explore the standard technique of producing a reduced instance in advance and then run ARW on the reduced instance. Second, we investigate applying reductions on-the-fly as ARW runs.

Preprocessing. Our first algorithm (KerMIS) uses exact reductions in combination with relative degree cutting. It uses the full set of reductions from Akiba and Iwata [AI16], as described in Sect. 3.3.1. Note that we do not include isolated clique removal, as it was not included in their reductions. After computing a reduced instance, we then cut 1% of the highest-degree vertices using relative degree cutting, breaking ties randomly. We then run ARW on the resulting graph.

On-the-fly. Our second approach (OnlineMIS) applies a set of simple reductions on-the-fly. For this algorithm, we only use isolated clique removal (for degrees zero, one, and two), since it does not require the graph to be modified—we can just mark isolated vertices and their neighbors as removed during local search. In more detail, we first perform a quick *single pass* when computing the initial solution for ARW. We force isolated vertices into the initial solution and mark them and their neighbors as being removed. Note that this does not result in a fully reduced instance, as this pass may create more isolated vertices which are not (necessarily) found by this method. We further mark the top 1% of high-degree vertices as removed during this pass. As local search continues, whenever we check if a vertex can be inserted into the solution, we check if it is isolated and update the solution and graph similar to the single pass. Thus, OnlineMIS reduces the graph on-the-fly as local search proceeds. This is beneficial as one can quickly remove vertices that are part of an optimal solution while also outputting temporary solutions much faster compared to computing a

reduced graph in advance. Note that as local search continues this can result in a graph that can not be reduced further. However, there might also be additional structures that could be removed but are not found, since we only target vertices that are taken into account during local search.

3.3.2 Experimental Evaluation

Methodology. We implemented our algorithms (OnlineMIS and KerMIS), including the reduction techniques, using C++ and compiled all code using g++ version 4.6.3 with full optimizations turned on (`-O3` flag). We further compiled the original implementations of ARW and ReduMIS using the same settings. For ReduMIS, we use the same parameters as in Section 3.2.4 (convergence parameter $\mu = 1000000$, reduction parameter $\lambda = 0.1 \cdot |\mathcal{I}|$ and cutting percentage $\eta = 0.1 \cdot |\mathcal{K}|$). For all instances, we perform three independent runs of each algorithm with different random seeds. For small instances, we run each algorithm sequentially with a five-hour wall-clock time limit to compute its best solution. For huge graphs with tens of millions of vertices and at least one billion edges, we use a time limit of ten hours. Each run was performed on Machine B presented in Section 2.3.2.c). We consider a set of sparse networks, road networks, and meshes taken from the full set of instances described in Section 2.3.2.a). To be more specific, we picked a sample of large sparse networks, road networks, and meshes from Lamm et al. [Lam+17] presented in the previous section. This sample includes the largest web graphs and the three hardest instances from Akiba and Iwata [AI16]. We then extended this set of instances with additional huge web graphs with billions of edges which are the particular focus of this work.

3.3.2.a) Accelerated Solutions

We now illustrate the speed improvement over existing heuristic algorithms. First, we measure the speedup of OnlineMIS over other high-quality heuristic search algorithms. In Table 3.7, we report the maximum speedup of OnlineMIS over the state-of-the-art competitors. We compute the maximum speedup for an instance as follows. For each solution size i, we compute the speedup $s^i_{Alg} = t^i_{Alg}/t^i_{OnlineMIS}$ of OnlineMIS over algorithm Alg for that solution size. We then report the maximum speedup $s^{\max}_{Alg} = \max_i s^i_{Alg}$ for the instance. Later, we also examine the speedup for finding high-quality solutions that are close to the best solution, i.e., within 99.5% of the best solution.

As can be seen in Table 3.7, OnlineMIS always has a maximum speedup greater than one over every other algorithm. We first note that OnlineMIS is significantly faster than ReduMIS and KerMIS. There are 14 instances where OnlineMIS achieves a maximum speedup of over 100 over ReduMIS. KerMIS performs only slightly better than ReduMIS, with OnlineMIS achieving similar speedups on twelve instances. On meshes, KerMIS fares especially poor. On these instances OnlineMIS always finds a better solution than KerMIS (instances marked with *). On the bunny and feline instances, OnlineMIS achieves a maximum speedup of more than 10000 over KerMIS. Furthermore, on the venus mesh graph, KerMIS never matches the quality of a single solution from OnlineMIS. ARW is the closest competitor, and OnlineMIS only has two maximum speedups greater than 100. However, on eight instances,

Table 3.7: For each graph instance, we give the number of vertices n and the number of edges m. We further provide the maximum speedup for OnlineMIS over other heuristic search algorithms. For each solution size i, we compute the speedup $s^i_{Alg} = t^i_{Alg}/t^i_{OnlineMIS}$ of OnlineMIS over algorithm Alg for that solution size. We then report the maximum speedup $s^{\max}_{Alg} = \max_i s^i_{Alg}$ for the instance. When an algorithm never matches the final solution quality of OnlineMIS, we give the highest non-infinite speedup and mark it with a "⋆". A "∞" indicates that all speedups are infinite.

	Graph		Maximum Speedup of OnlineMIS		
Name	n	m	$s^{\max}_{ARW}$	$s^{\max}_{KerMIS}$	$s^{\max}_{ReduMIS}$
Huge instances					
it-2004	41 291 594	1 027 474 947	4.51	221.26	266.30
sk-2005	50 636 154	1 810 063 330	356.87⋆	201.68	302.64
uk-2007	105 896 555	1 154 392 916	11.63⋆	108.13	122.50
Social networks and web graphs					
amazon-2008	735 323	3 523 472	43.39⋆	13.75	50.75
as-Skitter-big	1 696 415	11 095 298	355.06⋆	2.68	7.62
dewiki-2013	1 532 354	33 093 029	36.22⋆	632.94	1 726.28
enwiki-2013	4 206 785	91 939 728	51.01⋆	146.58	244.64
eu-2005	862 664	22 217 686	5.52	62.37	217.39
hollywood-2011	2 180 759	114 492 816	4.35	5.51	11.24
libimseti	220 970	17 233 144	15.16⋆	218.30	1 118.65
ljournal-2008	5 363 260	49 514 271	2.51	3.00	5.33
orkut	3 072 441	117 185 082	1.82⋆	478.94⋆	8 751.62⋆
web-Stanford	281 903	1 992 636	50.70⋆	29.53	59.31
webbase-2001	118 142 155	854 809 761	3.48	33.54	36.18
wikilinks	25 890 800	543 159 884	3.88	11.54	11.89
youtube	1 134 890	543 159 884	6.83	1.83	7.29
Road networks					
europe	18 029 721	22 217 686	5.57	12.79	14.20
USA-road	23 947 347	28 854 312	7.17	24.41	27.84
Meshes					
buddha	1 087 716	1 631 574	1.16	154.04⋆	976.10⋆
bunny	68 790	103 017	3.26	16 616.83⋆	526.14
dragon	150 000	225 000	2.22⋆	567.39⋆	692.60⋆
feline	41 262	61 893	2.00⋆	13 377.42⋆	315.48
gameguy	42 623	63 850	3.23	98.82⋆	102.03
venus	5 672	8 508	1.17	∞	157.78⋆

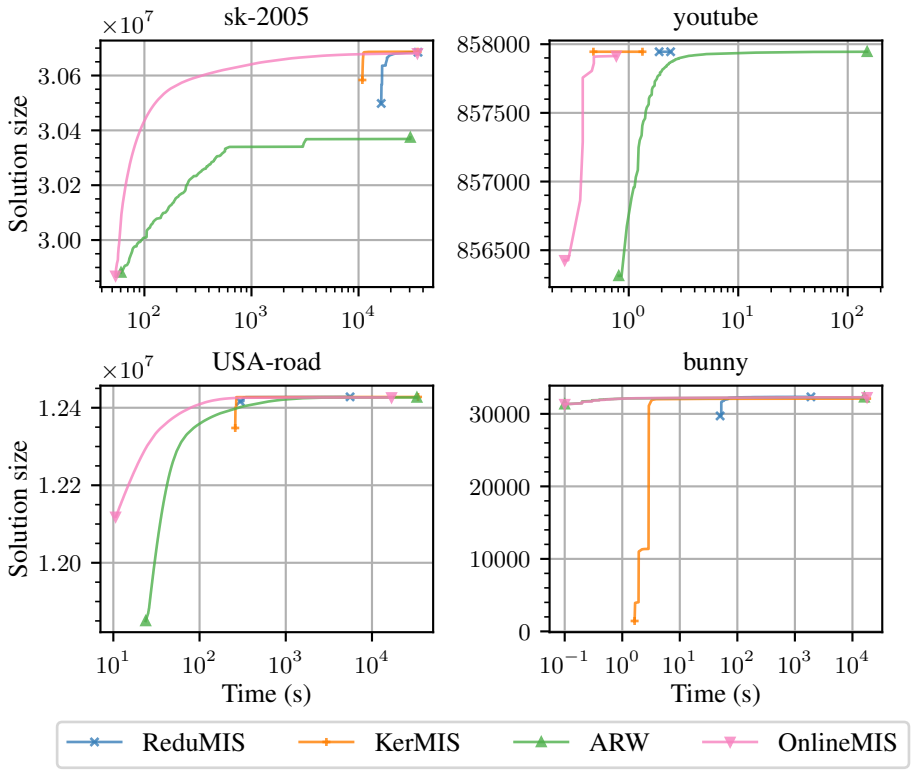


Figure 3.10: Convergence plots for sk-2005 (top left), youtube (top right), USA-road (bottom left) and bunny (bottom right).

OnlineMIS achieves a maximum speedup over ten, and on eleven instances ARW fails to match the final solution quality of OnlineMIS.

Figure 3.10 shows several representative convergence plots which illustrate the early solution quality of OnlineMIS compared to ARW, the closest competitor. For additional convergence plots, we refer to Appendix D. These convergence plots show average values over all three runs. On the non-mesh instances, OnlineMIS takes an early lead over ARW, though solution quality converges over time. Lastly, we examine the convergence plot for the bunny mesh graph. Reductions and high-degree cutting are not effective on meshes. Thus, ARW and OnlineMIS have similar initial solution sizes.

3.3.2.b) Time to High-Quality Solutions

We now look at the time it takes an algorithm to find a high-quality solution. We first determine the largest independent set found by any of the four algorithms, which represent

Table 3.8: For each algorithm, we give the average time t_{avg} to reach 99.5% of the best solution found by any algorithm. The fastest such time for each instance is marked in bold. We also give the size of the largest solution found by any algorithm and list the algorithms (abbreviated by first letter) that found this largest solution in the time limit. A "-" indicates that the algorithm did not find a solution of sufficient quality.

Graph Name	OnlineMIS t_{avg}	ARW t_{avg}	KerMIS t_{avg}	ReduMIS t_{avg}	Best IS Size	Best IS Algorithms
Huge instances:						
it-2004	**86.01**	327.35	7 892.04	9 448.18	25 620 285	R
sk-2005	**152.12**	-	10 854.46	16 316.59	30 686 766	K
uk-2007	**403.36**	3 789.74	23 022.26	26 081.36	67 282 659	K
Social networks and web graphs						
amazon-2008	**0.76**	1.26	5.81	15.23	309 794	K, R
as-Skitter-big	**1.26**	2.70	2.82	8.00	1 170 580	K, R
dewiki-2013	**4.10**	7.88	898.77	2 589.32	697 923	K
enwiki-2013	**10.49**	19.26	856.01	1 428.71	2 178 457	K
eu-2005	**1.32**	3.11	29.01	95.65	452 353	R
hollywood-2011	**1.28**	1.46	7.06	14.38	523 402	O, A, K, R
libimseti	**0.44**	0.45	50.21	257.29	127 293	R
ljournal-2008	**3.79**	8.30	10.20	18.14	2 970 937	K, R
orkut	**42.19**	49.18	2 024.36	-	839 086	K
web-Stanford	**1.58**	8.19	3.57	7.12	163 390	R
webbase-2001	**144.51**	343.86	2 920.14	3 150.05	80 009 826	R
wikilinks	**34.40**	85.54	348.63	358.98	19 418 724	R
youtube	**0.26**	0.81	0.48	1.90	857 945	A, K, R
Road networks						
europe	**28.22**	75.67	91.21	101.21	9 267 811	R
USA-road	**44.21**	112.67	259.33	295.70	12 428 105	R
Meshes						
buddha	**26.23**	26.72	119.05	1 699.19	480 853	A
bunny	**3.21**	9.22	-	70.40	32 349	R
dragon	**3.32**	4.90	5.18	97.88	66 502	A
feline	**1.24**	1.27	-	39.18	18 853	R
gameguy	15.13	**10.60**	60.77	12.22	20 727	R
venus	**0.32**	0.36	-	6.52	2 684	O, A, R

the best-known solutions [Lam+17], and compute how long it takes each algorithm to find an independent set within 99.5% of this size. The results are shown in Table 3.8. With a single exception, OnlineMIS is the fastest algorithm to be within 99.5% of the target solution.

Table 3.9: For each algorithm, we include average solution size and average time t_{avg} to reach it within a time limit (5 hours for normal graphs, 10 hours for huge graphs). Solutions in italics indicate that OnlineMIS computes a larger solution than ARW and bold marks the largest overall solution. A "-" in our indicates that the algorithm did not find a solution in the time limit.

Graph	OnlineMIS		ARW		KerMIS		ReduMIS	
Name	Avg.	t_{avg}	Avg.	t_{avg}	Avg.	t_{avg}	Avg.	t_{avg}
Huge instances:								
it-2004	25 610 697	35 324	25 612 993	33 407	25 619 988	35 751	**25 620 246**	35 645
sk-2005	*30 680 869*	34 480	30 373 880	11 387	**30 686 684**	34 923	30 684 867	35 837
uk-2007	*67 265 560*	35 982	67 101 065	8 702	**67 282 347**	35 663	67 278 359	35 782
Social networks and web graphs								
amazon-2008	*309 792*	6 154	309 791	12 195	309 793	818	**309 794**	153
as-Skitter-big	*1 170 560*	7 163	1 170 548	14 017	**1 170 580**	4	**1 170 580**	9
dewiki-2013	*697 789*	17 481	697 669	16 030	**697 921**	14 070	697 798	17 283
enwiki-2013	*2 178 255*	13 612	2 177 965	17 336	**2 178 436**	17 408	2 178 327	17 697
eu-2005	452 296	11 995	452 311	22 968	452 342	5 512	**452 353**	2 332
hollywood-2011	**523 402**	33	**523 402**	101	**523 402**	9	**523 402**	17
libimseti	*127 288*	8 250	127 284	9 308	**127 292**	102	**127 292**	16 747
ljournal-2008	2 970 236	428	2 970 887	16 571	**2 970 937**	36	**2 970 937**	41
orkut	***839 073***	17 764	839 001	17 933	839 004	19 765	806 244	34 197
web-Stanford	*163 384*	5 938	163 382	10 924	163 388	35	**163 390**	12
webbase-2001	79 998 332	35 240	80 002 845	35 922	80 009 041	30 960	**80 009 820**	31 954
wikilinks	19 404 530	21 069	19 416 213	34 085	19 418 693	23 133	**19 418 724**	854
youtube	857 914	< 1	**857 945**	93	**857 945**	< 1	**857 945**	2
Road networks								
europe	9 267 573	15 622	9 267 587	28 450	9 267 804	27 039	**9 267 809**	115
USA-road	12 426 557	10 490	12 426 582	31 583	12 427 819	32 490	**12 428 099**	4 799
Meshes								
buddha	480 795	17 895	**480 808**	17 906	480 592	16 695	479 905	17 782
bunny	32 283	13 258	32 287	13 486	32 110	14 185	**32 344**	1 309
dragon	***66 501***	15 203	66 496	14 775	66 386	16 577	66 447	3 456
feline	*18 846*	15 193	18 844	10 547	18 732	15 055	**18 851**	706
gameguy	20 662	6 868	20 674	12 119	20 655	7 467	**20 727**	191
venus	**2 684**	507	**2 684**	528	2 664	9	2 683	74

In fact, OnlineMIS finds such a solution at least twice as fast as ARW on 14 instances, and it is almost ten times faster on the largest instance, uk-2007. OnlineMIS is also orders of magnitude faster than ReduMIS (by a factor of at least 100 in seven cases). We also see

that KerMIS is faster than ReduMIS in 19 cases but much slower than OnlineMIS for all instances. It eventually finds the largest independent set (among all algorithms) for ten instances. This shows that the complete set of reductions is not always necessary, especially when the goal is to get a high-quality solution quickly. It also justifies our choice of cutting: the solution quality of KerMIS rivals (and sometimes even improves) that of ReduMIS.

3.3.2.c) Overall Solution Quality

Next, we show that OnlineMIS has high solution quality when given enough time for searching (5 hours for normal graphs, 10 hours for huge graphs). Although continuous improvement in quality is not the goal of OnlineMIS, in 11 instances OnlineMIS finds a larger independent set than ARW, and in four instances OnlineMIS finds the largest solution within the time limit. As seen in Table 3.9, OnlineMIS also finds a solution within 0.1% of the best solution found by any algorithm for all graphs. However, in general, OnlineMIS finds lower-quality solutions than ReduMIS, which might be caused by cutting high-degree vertices. Nonetheless, as this shows, the solution quality remains high even when cutting 1% of the vertices.

We further test KerMIS, which first fully reduces the graph using the advanced reductions from ReduMIS, removes 1% of the highest-degree vertices, and then runs ARW on the remaining graph. On eight instances, KerMIS finds a better solution than ReduMIS. However, reductions and cutting take a long time (over three hours for sk-2005, ten hours for uk-2007), and therefore KerMIS is much slower to get to a high-quality solution than OnlineMIS. Thus, our experiments show that the complete set of reductions is not always necessary, especially when the goal is to get a high-quality solution quickly. This also further justifies our choice of cutting, as the solution quality of KerMIS remains high. On the other hand, as-Skitter-big, ljournal-2008, and youtube are solved quickly with advanced reduction rules.

3.4 Exact Portfolio Algorithm

The techniques presented in the previous sections of this chapter are important contributions to heuristic algorithms that are able to find high-quality solutions on massive instances. In particular, the recursive removal of vertices that are likely to be in an independent set (inexact reductions) and simple but fast reductions in combination with local search have since been used in other successful state-of-the-art approaches [CLZ17; Zhe+20].

In the remaining sections of this chapter, we now focus on improving exact algorithms for finding maximum independent sets with the goal of developing algorithms that can rival the performance of heuristic approaches.

First, we present the algorithm that won the Parameterized Algorithms and Computational Experiments (PACE) 2019 Challenge, which focused on the vertex cover problem. In particular, we deployed a portfolio of algorithms using techniques from the literature on all three previously mentioned problems (MIS, MVC, and MC, see Section 3.1). These include reduction rules and branch-and-reduce for the minimum vertex cover problem [AI16], iterated local search for the maximum independent set problem [ARW12], and an exact

state-of-the-art branch-and-bound maximum clique algorithm [LJM17]. As mentioned in Section 2.1.3, finding a minimum vertex cover is complementary to finding a maximum independent set, i.e., for a maximum independent set $\mathcal{I} \subseteq V$, $V \setminus \mathcal{I}$ is a minimum vertex cover and vice versa. Thus, all previously mentioned techniques can be applied to MVC in a straightforward way. In addition to describing our techniques and algorithm in detail, we analyze the results of our extensive experiments on the data sets provided by the challenge. Not only do our experiments illustrate the power of the techniques spanning the literature, they also offer several new insights not yet seen before. In particular, reductions followed by branch-and-bound can outperform branch-and-reduce algorithms; seeding branch-and-reduce by an initial solution from a local search can significantly boost its performance; and, somewhat surprisingly, reductions are sometimes counterproductive: branch-and-bound algorithms can perform significantly worse on the reduced instance than on the original input graph.

Organization. We begin this section by outlining each of the techniques we use in our portfolio algorithm in Section 3.4.1. We then describe how we actually combine these techniques in our algorithm in Section 3.4.2. Lastly, in Section 3.4.3, we perform an experimental evaluation to show the impact of the individual components of our portfolio on the final number of instances solved.

3.4.1 Techniques

We now describe the techniques we use in our algorithm in detail. First, we use the branch-and-reduce algorithm of Akiba and Iwata [AI16]. Their algorithm exhaustively applies the full suite of reduction rules presented in Section 3.1.4 before branching and includes a number of advanced branching rules and lower bounds to prune the search (see Section 3.1.5). We also evaluated other sets of reduction rules, i.e., the ones used by Strash [Str16] and the weighted reductions presented in Section 4.2. However, they did not provide a significant advantage over the ones used by Akiba and Iwata [AI16] and were therefore not included.

Experiments by Strash [Str16] show that the full power of branch-and-reduce is only needed *very rarely* in real-world instances; reductions followed by a standard branch-and-bound algorithm is sufficient for many real-world instances. Furthermore, branch-and-reduce does not work well on many synthetic benchmark instances, where reduction rules are ineffective [AI16] and instead add significant overhead to branch-and-bound. We use a state-of-the-art branch-and-bound maximum clique algorithm (MoMC) by Li et al. [LJM17], which uses incremental MaxSAT reasoning to prune the search, and a combination of static and dynamic vertex ordering to select the vertex for branching. We run the clique algorithm on the complement graph, which results in a maximum independent set from which we then derive a minimum vertex cover. In preliminary experiments, we found that a reduced instance can sometimes be harder for the algorithm than the original input; therefore, we run the algorithm on the reduced instance and the original graph.

Finally, Batsyn et al. [Bat+14] showed that if the branch-and-bound search is primed with a high-quality solution from local search, then instances can be solved up to thousands of times faster. We use the iterated local search algorithm by Andrade et al. [ARW12] to

prime the branch-and-reduce algorithm with a high-quality initial solution. To the best of our knowledge, this has not been tried before. We implemented the algorithm to find a high-quality solution on *the reduced instance*. Calling local search on the reduced instance has been shown to produce a high-quality solution much faster than without applying reductions [CLZ17; Dah+16a].

3.4.2 Putting it all Together

Our algorithm first runs a preprocessing phase, followed by four phases of either branch-and-bound or branch-and-reduce.

Phase 1. (Preprocessing) Our algorithm starts by computing a reduced instance of the graph using the reductions by Akiba and Iwata [AI16]. From there, we use iterated local search to produce a high-quality solution I_{init} on the (hopefully smaller) reduced instance.

Phase 2. (Branch-and-Reduce, short) We prime a branch-and-reduce algorithm with the initial solution I_{init} and run it with a short time limit.

Phase 3. (Branch-and-Bound, short) If Phase 2 is unsuccessful, we run the MoMC [LJM17] clique algorithm on the complement of the reduced instance, also using a short time limit[2]. Sometimes reductions can make the problem harder for MoMC. Therefore, if the first call is unsuccessful, we also run MoMC on the complement of the original (unreduced) input with the same short time limit.

Phase 4. (Branch-and-Reduce, long) If we have still not found a solution, we run branch-and-reduce on the reduced graph using the initial solution I_{init} and a longer time limit. We opt for this second phase because, while most graphs amenable to reductions are solved very quickly with branch-and-reduce (less than a second), experiments by Akiba and Iwata [AI16] showed that other slower instances either finish in at most a few minutes or take significantly longer—more than the time limit allotted for the challenge. This second phase of branch-and-reduce aims to catch any instances that still benefit from the use of reductions.

Phase 5. (Branch-and-Bound, remaining time) If all previous phases were unsuccessful, we run MoMC on the original (unreduced) input graph until the end of the time limit. This last phase is meant to capture only the hardest-to-compute instances.

The algorithm time limits (discussed in Section 3.4.3) and ordering were carefully chosen so that the overall algorithm outputs solutions of the "easy" instances *quickly*, while still being able to solve hard instances.

[2]Note that repeatedly checking the time can slow down a highly optimized branch-and-bound algorithm considerably; we therefore simulate time checking by using a limit on the number of branches.

3.4.3 Experimental Evaluation

We now look at the impact of the algorithmic components on the number of instances solved. Here, we focus on the public instances of the PACE 2019 Challenge, Vertex Cover Track A, obtained from the PACE Challenge website[3] (see Section 2.3.2.a)). However, we also provide the results for the private instances [4]. In particular, we summarize the results comparing against the second and third best competing algorithms on the private instances during the challenge. In total, the PACE Challenge contained 200 instances. Note that the public instances used during the challenge are all odd-numbered, and the private instances are even-numbered.

Methodology. All of our experiments were run on Machine C described in Section 2.3.2.c). All algorithms were implemented in C++ and compiled with g++ version 6.3.0 using optimization flag `-O3`. Our source code is publicly available at GitHub[5]. Each algorithm was run sequentially with a time limit of 30 minutes—the time allotted to solve a single data set in the PACE 2019 Challenge. Even though we provide running times, our primary focus is on the total number of instances solved. We do this because the challenge gave a better score for solving more instances faster [DFH19].

Algorithm Configuration. We now explain the algorithm configuration we use in our experimental setup. In the following, MoMC runs the clique algorithm by Li et al. [LJM17] on the complement of the input graph; RMoMC applies reductions to the input graph exhaustively and then runs MoMC on the complement of the resulting reduced instance; LSBnR applies reductions exhaustively, then runs local search to obtain a high-quality solution on the reduced instance which is used as an initial bound in the branch-and-reduce algorithm that is run on the reduced instance; BnR applies reductions and then runs the branch-and-reduce algorithm on the reduced instance (no local search is used to improve an initial bound); FullA is the full algorithm as described in the previous section, using a short time limit of one second and a long time limit of thirty seconds.

3.4.3.a) Evaluation

Tables 3.10 and 3.11 give an overview of the public instances that each algorithm solved, including the reduced graph size and the minimum vertex cover size for those instances solved by any of the four algorithms. Overall, MoMC can solve 30 out of the 100 instances. Applying reductions first enables RMoMC to solve 68 instances. However, curiously, there are two instances (instances 131 and 157) that MoMC solves but that RMoMC can not solve. In these cases, reductions reduced the number of vertices but *increased* the number of edges. This is due to the alternative reduction, which in some cases can create more edges than initially present. This is why we choose to also run MoMC on the unreduced input graph in FullA (in order to solve those instances as well).

LSBnR solves 55 of the 100 instances. Priming the branch-and-reduce algorithm with an initial solution computed by local search has a significant impact: LSBnR solves 13 more

[3] `https://pacechallenge.org/files/pace2019-vc-exact-public-v2.tar.bz2`
[4] `https://doi.org/10.5281/zenodo.3354609`
[5] `https://github.com/KarlsruheMIS/pace-2019`

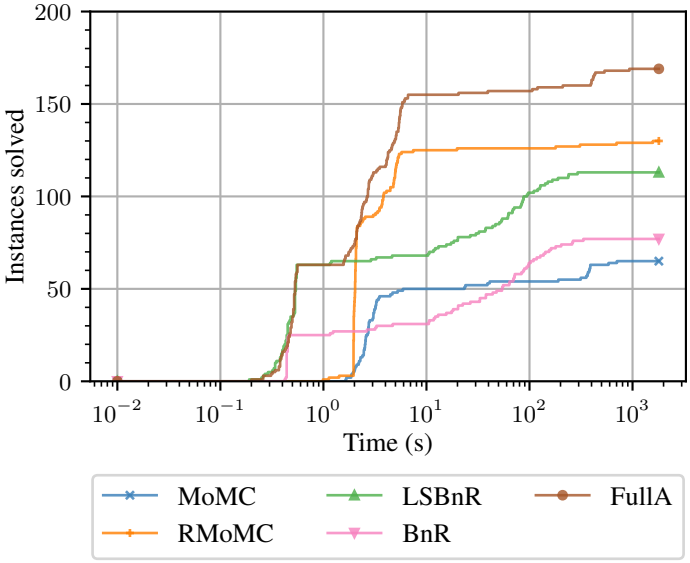


Figure 3.11: Number of instances solved over time by each algorithm over *all* instances. At each time step t, we count each instance solved by the algorithm in at most t seconds.

instances than BnR, which solve 42 instances. In particular, using local search to find an initial bound helps to solve large instances in which the initial reduction step does not reduce the graph fully. Surprisingly, RMoMC solves 26 instances that BnR does not (and even LSBnR is only able to solve one of these instances). To the best of our knowledge, this is the first time that reductions followed by branch-and-bound is shown to outperform branch-and-reduce significantly. Our full algorithm FullA solves 82 of the 100 instances and, as expected, dominates each of the other configurations. This can be further seen from the plot in Figure 3.11, which shows how many instances each algorithm solves over time (this includes all 100 public and 100 private instances of the challenge). Note that LSBnR and RMoMC solve more instances in narrow time gaps due to FullA's setup cost and running multiple algorithms. However, FullA quickly makes up for this and overtakes all algorithms at approximately eight seconds.

In addition to the 100 public instances, the PACE 2019 Challenge tests all submissions on 100 private instances. Tables 3.12 and 3.13 give detailed per instances results on those instances. The results are similar to the results on the public instances. On the private instances, MoMC can solve 35 out of the 100 instances, RMoMC solves 62, LSBnR solves 58, and BnR solves 35 instances. We note that our choice of using MoMC as our branch-and-bound algorithm is significant on these instances. Eight instances solved exclusively by our algorithm are solved in Phase 5, where MoMC is run until the end of the challenge time limit. Overall, our full algorithm FullA solved 87 of the 100 instances, which is ten more

Table 3.10: Detailed per instance results for public instances. The columns n and m refer to the number of vertices and edges of the input graph, n' and m' refer to the number of vertices and edges of the reduced graph after reductions have been applied exhaustively, and $|\mathcal{C}|$ refers to the size of the minimum vertex cover of the input graph. We list running times for when an algorithm successfully solved the given instance and "-" when it reached the time limit. The fastest running times for each instance are highlighted in bold

Graph	n	m	n'	m'	MoMC	RMoMC	LSBnR	BnR	FullA	$\lvert\mathcal{C}\rvert$
001	6 160	40 207	0	0	-	1.97	0.55	**0.42**	0.54	2 586
003	60 541	74 220	0	0	-	1.97	0.55	**0.44**	0.54	12 190
005	200	819	192	800	1.67	1.96	2.89	3.18	**1.60**	129
007	8 794	10 130	0	0	-	1.96	0.54	**0.44**	0.53	4 397
009	38 452	174 645	0	0	-	1.96	0.54	**0.43**	0.53	21 348
011	9 877	25 973	0	0	-	1.96	0.53	**0.45**	0.52	4 981
013	45 307	55 440	0	0	-	1.96	0.54	**0.45**	0.52	8 610
015	53 610	65 952	0	0	-	1.96	0.54	**0.45**	0.52	10 670
017	23 541	51 747	0	0	-	1.96	0.53	**0.44**	0.51	12 082
019	200	884	194	862	2.02	1.97	4.64	4.71	**1.64**	130
021	24 765	30 242	0	0	-	1.97	0.53	**0.44**	0.52	5 110
023	27 717	133 665	0	0	-	1.96	0.53	**0.44**	0.53	16 013
025	23 194	28 221	0	0	-	1.96	0.53	**0.44**	0.53	4 899
027	65 866	81 245	0	0	-	1.96	0.53	**0.45**	0.56	13 431
029	13 431	21 999	0	0	-	1.97	0.55	**0.45**	0.56	6 622
031	200	813	198	818	3.00	**1.97**	87.86	109.45	2.81	136
033	4 410	6 885	138	471	-	1.97	**1.19**	1.24	1.58	2 725
035	200	884	189	859	1.88	1.97	11.70	11.77	**1.76**	133
037	198	824	194	810	1.99	1.98	15.58	15.56	**1.85**	131
039	6 795	10 620	219	753	-	1.98	120.20	127.38	**1.75**	4 200
041	200	1 040	200	1 023	2.53	**1.99**	68.26	71.68	2.36	139
043	200	841	198	844	4.68	**2.02**	237.80	259.31	5.57	139
045	200	1 044	200	1 020	2.28	**2.00**	37.17	37.79	2.10	137
047	200	1 120	198	1 080	3.06	**2.03**	83.60	86.87	2.75	140
049	200	957	198	930	2.73	**2.00**	26.57	26.64	2.15	136
051	200	1 135	200	1 098	2.73	**2.01**	81.99	83.70	2.24	140
053	200	1 062	200	1 026	2.60	**2.03**	85.97	88.97	2.41	139
055	200	958	194	938	1.86	2.03	10.37	10.54	**1.82**	134
057	200	1 200	197	1 139	2.81	**2.04**	97.06	98.49	2.34	142
059	200	988	193	954	2.47	2.04	20.18	20.30	**2.02**	137
061	200	952	198	914	3.25	**2.04**	30.18	32.98	2.49	135
063	200	1 040	200	1 011	3.26	**2.05**	94.50	102.56	2.77	138
065	200	1 037	200	1 011	2.39	**2.06**	54.23	54.50	2.13	138
067	200	1 201	200	1 174	3.12	**2.07**	166.63	179.52	2.91	143
069	200	1 120	196	1 077	2.73	**2.07**	61.64	64.11	2.32	140
071	200	984	200	952	2.29	**2.09**	50.94	54.09	2.20	136
073	200	1 107	200	1 078	2.51	**2.09**	62.07	63.60	2.44	139
075	26 300	41 500	500	3 000	-	-	**0.39**	-	**0.39**	16 300
077	200	988	193	954	2.69	**2.10**	20.16	20.25	**2.10**	137
079	26 300	41 500	500	3 000	-	-	**0.38**	-	**0.38**	16 300
081	199	1 124	197	1 087	3.30	**2.10**	189.03	202.95	3.05	141
083	200	1 215	198	1 182	5.25	**2.39**	239.39	264.19	6.60	144
085	11 470	17 408	3 539	25 955	-	-	-	-	-	
087	13 590	21 240	441	1 512	-	**2.13**	-	-	3.52	8 400
089	57 316	77 978	16 834	54 847	-	-	-	-	-	
091	200	1 196	200	1 163	5.91	**3.84**	293.03	332.93	39.45	145
093	200	1 207	200	1 162	4.46	**2.11**	141.75	154.99	5.65	143
095	15 783	24 663	510	1 746	-	**2.54**	-	-	4.23	9 755
097	18 096	28 281	579	1 995	-	7.43	-	-	**6.63**	11 185
099	26 300	41 500	500	3 000	-	-	0.34	-	**0.32**	16 300

Table 3.11: Detailed per instance results for public instances. The columns n and m refer to the number of vertices and edges of the input graph, n' and m' refer to the number of vertices and edges of the reduced graph after reductions have been applied exhaustively, and $|\mathcal{C}|$ refers to the size of the minimum vertex cover of the input graph. We list running times for when an algorithm successfully solved the given instance and "-" when it reached the time limit. The fastest running times for each instance are highlighted in bold

Graph	n	m	n'	m'	MoMC	RMoMC	LSBnR	BnR	FullA	$\|\mathcal{C}\|$
101	26 300	41 500	500	3 000	-	-	0.45	-	**0.34**	16 300
103	15 783	24 663	513	1 752	-	**3.66**	-	-	4.62	9 755
105	26 300	41 500	500	3 000	-	-	**0.42**	-	0.47	16 300
107	13 590	21 240	435	1 500	-	**2.44**	-	-	3.37	8 400
109	66 992	90 970	20 336	66 350	-	-	-	-	-	
111	450	17 831	450	17 831	2.53	**0.99**	-	-	2.71	420
113	26 300	41 500	500	3 000	-	-	0.52	-	**0.47**	16 300
115	18 096	28 281	573	1 986	-	**4.78**	-	-	5.40	11 185
117	18 096	28 281	582	2 007	-	**5.12**	-	-	5.39	11 185
119	18 096	28 281	588	2 016	-	**4.77**	-	-	5.33	11 185
121	18 096	28 281	579	1 998	-	**4.95**	-	-	5.05	11 185
123	26 300	41 500	500	3 000	-	-	**0.45**	-	0.48	16 300
125	26 300	41 500	500	3 000	-	-	**0.41**	-	0.45	16 300
127	18 096	28 281	582	2 001	-	**4.97**	-	-	5.06	11 185
129	15 783	24 663	507	1 752	-	**3.81**	-	-	4.89	9 755
131	2 980	5 360	2 179	6 951	**382.52**	-	-	-	399.52	1 920
133	15 783	24 663	507	1 746	-	**3.49**	-	-	4.02	9 755
135	26 300	41 500	500	3 000	-	-	**0.48**	-	0.49	16 300
137	26 300	41 500	500	3 000	-	-	**0.46**	-	0.50	16 300
139	18 096	28 281	579	1 995	-	**4.84**	-	-	6.11	11 185
141	18 096	28 281	576	1 995	-	**5.27**	-	-	5.84	11 185
143	18 096	28 281	582	2 001	-	**4.87**	-	-	6.19	11 185
145	18 096	28 281	576	1 989	-	**4.83**	-	-	5.13	11 185
147	18 096	28 281	567	1 974	-	**4.46**	-	-	5.11	11 185
149	26 300	41 500	500	3 000	-	-	0.45	-	**0.43**	16 300
151	15 783	24 663	501	1 728	-	**4.09**	-	-	4.95	9 755
153	29 076	45 570	2 124	16 266	-	-	-	-	-	
155	26 300	41 500	500	3 000	-	-	0.43	-	**0.40**	16 300
157	2 980	5 360	2 169	6 898	**388.56**	-	-	-	434.79	1 920
159	18 096	28 281	582	2 004	-	**5.73**	-	-	5.83	11 185
161	138 141	227 241	41 926	202 869	-	-	-	-	-	
163	18 096	28 281	582	2 004	-	**5.06**	-	-	5.54	11 185
165	18 096	28 281	576	1 995	-	**5.09**	-	-	5.45	11 185
167	15 783	24 663	510	1 746	-	**3.80**	-	-	4.56	9 755
169	4 768	8 576	3 458	11 014	-	-	-	-	-	
171	18 096	28 281	576	1 989	-	**4.99**	-	-	5.45	11 185
173	56 860	77 264	17 090	55 568	-	-	-	-	-	
175	3 523	6 446	2 723	8 570	-	-	-	-	-	
177	5 066	9 112	3 704	11 797	-	-	-	-	-	
179	15 783	24 663	504	1 740	-	**3.21**	-	-	4.39	9 755
181	18 096	28 281	573	1 989	-	4.46	**0.19**	-	0.20	11 185
183	72 420	118 362	30 340	133 872	-	-	-	-	-	
185	3 523	6 446	2 723	8 568	-	-	-	-	-	
187	4 227	7 734	3 264	10 286	-	-	-	-	-	
189	7 400	13 600	5 802	18 212	-	-	-	-	-	
191	4 579	8 378	3 539	11 137	-	-	-	-	-	
193	7 030	12 920	5 510	17 294	-	-	-	-	-	
195	1 150	81 068	1 150	81 068	-	-	-	-	-	
197	1 534	127 011	1 534	127 011	-	-	-	-	-	
199	1 534	126 163	1 534	126 163	-	-	-	-	-	

Table 3.12: Detailed per instance results for private instances. The columns n and m refer to the number of vertices and edges of the input graph, n' and m' refer to the number of vertices and edges of the reduced graph after reductions have been applied exhaustively, and $|\mathcal{C}|$ refers to the size of the minimum vertex cover of the input graph. We list running times for when an algorithm successfully solved the given instance and "-" when it reached the time limit. The fastest running times for each instance are highlighted in bold.

Graph	n	m	n'	m'	MoMC	RMoMC	LSBnR	BnR	FullA	$\lvert\mathcal{C}\rvert$
002	51 795	63 334	0	0	-	1.97	0.55	**0.44**	0.54	10 605
004	8 114	26 013	0	0	-	1.96	0.55	**0.44**	0.54	2 574
006	200	751	188	716	1.64	1.96	**1.16**	**1.16**	1.57	126
008	7 537	72 833	0	0	-	1.96	0.54	**0.44**	0.53	3 345
010	199	774	189	756	2.04	1.96	3.25	3.27	**1.63**	127
012	53 444	68 044	0	0	-	1.96	0.54	**0.45**	0.52	10 918
014	25 123	31 552	0	0	-	1.96	0.54	**0.45**	0.52	5 111
016	153	802	153	802	-	-	-	-	-	
018	49 212	63 601	0	0	-	1.96	0.53	**0.44**	0.51	10 201
020	57 287	71 155	0	0	-	1.97	0.53	**0.44**	0.52	11 648
022	12 589	33 129	0	0	-	1.97	0.53	**0.44**	0.52	6 749
024	7 620	47 293	0	0	-	1.96	0.53	**0.44**	0.53	4 364
026	6 140	36 767	0	0	-	1.96	0.53	**0.44**	0.53	2 506
028	54 991	67 000	0	0	-	1.97	0.55	**0.45**	0.56	11 211
030	62 853	79 557	0	0	-	1.97	0.55	**0.46**	0.56	13 338
032	1 490	2 680	1 081	3 426	**41.09**	-	-	-	106.28	960
034	1 490	2 680	1 090	3 467	**39.08**	1 577.43	-	-	119.60	960
036	26 300	41 500	500	3 000	-	1.98	**0.53**	2.65	0.55	16 300
038	786	14 024	460	6 623	23.88	**2.29**	18.45	18.12	4.11	605
040	210	625	210	625	190.43	**179.29**	-	-	210.26	145
042	200	974	200	952	2.10	**1.99**	53.16	55.35	2.10	136
044	200	1 186	200	1 147	3.05	**1.99**	126.19	136.99	2.76	142
046	200	812	200	812	3.06	**2.02**	150.57	161.63	2.70	137
048	200	1 052	198	1 022	2.57	**2.03**	37.59	37.72	2.10	138
050	200	1 048	200	1 025	3.43	**2.01**	70.89	72.05	2.95	140
052	200	1 019	198	1 000	2.76	**2.02**	32.27	33.00	2.32	138
054	200	985	198	951	2.76	**2.03**	46.67	47.86	2.36	137
056	200	1 117	200	1 089	3.11	**2.03**	127.38	142.38	2.63	141
058	200	1 202	200	1 171	3.21	**2.04**	68.81	70.31	2.71	142
060	200	1 147	200	1 118	2.75	**2.04**	110.27	117.51	2.35	141
062	199	1 164	199	1 128	3.19	**2.05**	87.30	97.72	2.75	141
064	200	1 071	198	1 040	2.46	**2.05**	43.76	43.94	2.12	138
066	200	884	198	875	2.58	**2.07**	12.14	12.14	2.27	134
068	200	983	198	961	1.94	2.07	13.08	12.98	**1.92**	135
070	200	887	198	856	2.22	2.07	17.32	17.59	**2.04**	133
072	200	1 204	198	1 176	2.47	2.09	62.51	69.43	**1.98**	140
074	200	820	194	785	2.55	2.10	10.65	10.62	**2.02**	132
076	26 300	41 500	500	3 000	-	2.10	**0.39**	-	**0.39**	16 300
078	11 349	17 739	357	1 245	-	**2.11**	-	-	2.60	7 015
080	26 300	41 500	500	3 000	-	-	**0.38**	-	0.40	16 300
082	200	978	196	956	3.50	**2.09**	84.63	93.88	3.02	138
084	13 590	21 240	435	1 503	-	**2.08**	-	-	3.26	8 400
086	26 300	41 500	500	3 000	-	2.07	**0.34**	-	0.38	16 300
088	26 300	41 500	500	3 000	-	2.06	**0.34**	-	0.37	16 300
090	11 349	17 739	357	1 245	-	**2.05**	-	-	2.39	7 015
092	450	17 794	450	17 794	2.20	**2.04**	-	-	2.70	420
094	5 960	10 720	4 217	13 456	-	-	-	-	-	
096	26 300	41 500	500	3 000	-	-	**0.29**	-	0.37	16 300
098	26 300	41 500	500	3 000	-	-	**0.33**	-	0.51	16 300
100	26 300	41 500	500	3 000	-	1.42	**0.41**	22.52	0.51	16 300

Table 3.13: Detailed per instance results for private instances. The columns n and m refer to the number of vertices and edges of the input graph, n' and m' refer to the number of vertices and edges of the reduced graph after reductions have been applied exhaustively, and $|\mathcal{C}|$ refers to the size of the minimum vertex cover of the input graph. We list running times for when an algorithm successfully solved the given instance and "-" when it reached the time limit. The fastest running times for each instance are highlighted in bold

Graph	n	m	n'	m'	MoMC	RMoMC	LSBnR	BnR	FullA	$\|\mathcal{C}\|$
102	26 300	41 500	500	3 000	-	-	0.45	-	**0.38**	16 300
104	26 300	41 500	500	3 000	-	-	**0.35**	-	0.46	16 300
106	2 980	5 360	2 136	6 809	**389.26**	-	-	-	424.79	1 920
108	26 300	41 500	500	3 000	-	-	**0.25**	-	0.50	16 300
110	98 128	161 357	29 168	140 392	-	-	-	-	-	
112	18 096	28 281	576	1 992	-	**5.27**	-	-	5.61	11 185
114	15 783	24 663	504	1 740	-	**3.73**	-	-	4.09	9 755
116	26 300	41 500	500	3 000	-	-	0.51	-	**0.31**	16 300
118	26 300	41 500	500	3 000	-	-	**0.40**	-	0.45	16 300
120	70 144	116 378	6 029	38 285	-	-	-	-	-	
122	26 300	41 500	500	3 000	-	-	**0.44**	-	0.49	16 300
124	26 300	41 500	500	3 000	-	-	**0.48**	-	0.50	16 300
126	18 096	28 281	582	2 001	-	**5.23**	-	-	5.79	11 185
128	26 300	41 500	500	3 000	-	-	-	-	-	
130	26 300	41 500	500	3 000	-	-	**0.47**	-	0.49	16 300
132	15 783	24 663	513	1 755	-	**3.70**	-	-	4.24	9 755
134	26 300	41 500	500	3 000	-	-	**0.40**	-	0.48	16 300
136	18 096	28 281	585	2 007	-	**5.03**	-	-	5.49	11 185
138	18 096	28 281	576	1 992	-	**5.42**	-	-	5.57	11 185
140	26 300	41 500	500	3 000	-	-	**0.44**	-	0.48	16 300
142	2 980	5 360	2 180	6 946	**366.96**	-	-	-	399.39	1 920
144	26 300	41 500	500	3 000	-	-	**0.45**	-	0.46	16 300
146	26 300	41 500	500	3 000	-	-	0.45	-	**0.44**	16 300
148	26 300	41 500	500	3 000	-	-	**0.44**	-	0.46	16 300
150	26 300	41 500	500	3 000	-	-	0.43	-	**0.42**	16 300
152	13 590	21 240	438	1 506	-	2.27	**0.32**	-	0.38	8 400
154	15 783	24 663	504	1 737	-	**3.51**	-	-	4.23	9 755
156	450	17 809	450	17 809	2.56	**1.13**	-	-	2.65	420
158	15 783	24 663	507	1 746	-	**3.73**	-	-	4.65	9 755
160	18 096	28 281	576	1 989	-	**4.96**	-	-	5.33	11 185
162	50 635	83 075	13 066	63 758	-	-	-	-	-	
164	29 296	46 040	1 210	8 666	-	-	-	-	-	
166	3 278	5 896	2 400	7 643	**584.89**	-	-	-	-	
168	2 980	5 360	2 180	6 943	**369.18**	-	-	-	413.56	1 920
170	15 783	24 663	507	1 746	-	**3.04**	-	-	4.10	9 755
172	4 025	7 435	3 158	9 863	-	-	-	-	-	
174	2 980	5 360	2 180	6 955	**355.88**	-	-	-	393.39	1 920
176	15 783	24 663	501	1 734	-	**3.36**	-	-	4.19	9 755
178	18 096	28 281	573	1 995	-	**5.39**	-	-	5.41	11 185
180	15 783	24 663	501	1 731	-	**3.87**	-	-	4.75	9 755
182	26 300	41 500	500	3 000	-	-	0.27	-	**0.26**	16 300
184	6 290	11 560	4 904	15 397	-	-	-	-	-	
186	26 300	41 500	500	3 000	-	-	**0.26**	-	**0.26**	16 300
188	6 660	12 240	5 220	16 375	-	-	-	-	-	
190	3 875	7 090	2 997	9 424	-	-	-	-	-	
192	2 980	5 360	2 180	6 941	**378.92**	-	-	-	415.68	1 920
194	1 150	80 851	1 150	80 851	314.14	**309.04**	-	-	534.28	1 100
196	1 534	126 082	1 534	126 082	-	-	-	-	-	
198	1 150	80 072	1 150	80 072	705.24	**699.15**	-	-	930.91	1 100
200	1 150	80 258	1 150	80 258	23.39	**19.79**	-	-	20.49	1 100

instances than the second-place submission (peaty [PT19], solving 77), and eleven more than the third-place submission (bogdan [Zav19]), solving 76). Our algorithm dominates these other approaches: except for one graph, our algorithm solves all instances that peaty and bogdan can solve combined.

We briefly describe these two algorithms. The peaty algorithm uses reductions to compute a reduced graph of the input followed by an unpublished maximum weight clique solver on the complement of each of the connected components of the reduced graph to assemble a solution. The clique solver is similar to MaxCLQ by Li and Quan [LQ10] but is more general. Local search is used to obtain an initial solution. On the other hand, bogdan implemented a small suite of simple reductions (including vertex folding, isolated clique removal, and degree-1 removal) together with a recent maximum clique solver by Szabó and Zavalnij [SZ18].

By examining both the public and private instances, we see that a large part of the instances (instances below 100) can be solved by either RMoMC, LSBnR, or BnR. MoMC has problems solving some of those instances. Most "harder" instances (instances above 100) are solved by either RMoMC or LSBnR with very little overlap between both algorithms. As noted previously, some of the harder instances can only be solved by using MoMC. Our algorithm FullA combines all approaches so that we solve all instances that are solved by the individual algorithms. Finally, we like to note that our algorithm is able to solve many instances in less than ten seconds and solves many instances significantly faster than the time intervals used for classifying those instances with Gurobi [Gur21] (see Section 2.3.2.a)).

3.5 Targeted Branching Rules

As already presented multiple times throughout this chapter, reductions can have a large practical impact. Both heuristic and exact approaches benefit greatly from the use of reductions. However, most previous research aimed at improving the performance of branch-and-reduce algorithms so far has been focused on either proposing more practically efficient special cases of already existing rules [CLZ17; Dah+16a] or maintaining dependencies between reduction rules to reduce unnecessary checks [AC19; HSS19]. Nonetheless, improving other aspects of branch-and-reduce has been shown to benefit its performance [PvdG21]. In particular, the branching strategy has been shown to have a significant impact on the running time [AI16]. Up to now, the most frequently used branching strategy employed in many state-of-the-art algorithms selects branching vertices solely based on their degree. Other factors, such as the actual reduction rules used during the algorithm's execution, are rarely taken into account. However, recently there have been some attempts to incorporate such branching strategies for other problems, such as finding a maximum k-plex [Gao+18].

In this section, we propose and examine several novel strategies for selecting branching vertices. These strategies follow two main approaches motivated by existing research. First, branching on vertices that decompose the graph into several connected components that can be solved independently. Solving components individually has been shown to improve the performance of branch-and-reduce in practice significantly, especially when the size of the largest component is small [AC19]. Second, branching on vertices whose removal leads

to reduction rules becoming applicable again. In turn, this leads to a smaller reduced graph and thus improved performance. For each approach, we present several concrete strategies that vary in their complexity. Finally, we evaluate their performance by comparing them to the aforementioned default strategy used in the state-of-the-art algorithm by Akiba and Iwata [AI16]. For this purpose, we use of a broad spectrum of instances from different graph classes and applications. Our experiments indicate that our strategies are able to find an optimal solution faster than the default strategy on a large set of instances. In particular, our reduction-based packing rule is able to outperform the default strategy on 65% of all instances. Furthermore, our decomposition-based strategies achieve a speedup of 1.22 (over the default strategy) over all instances.

Organization. We begin this section by introducing important previous work on branching rules in exact algorithms for MIS and its complementary problems in Section 3.5.1. We then present our novel branching strategies based on decomposition and reductions in Section 3.5.2 and Section 3.5.3. Finally, Section 3.5.4 covers our experimental evaluation and highlights the impact of different branching strategies on state-of-the-art algorithms.

3.5.1 Previous Work on Branching Strategies

The most commonly used branching strategy for MIS and MVC is to select a vertex of maximum degree. Fomin et al. [FGK09] show that using a vertex of maximum degree that also minimizes the number of edges between its neighbors is optimal with respect to their complexity measure. The algorithm by Akiba and Iwata [AI16], which we augment with our new branching rules, also uses this strategy. Akiba and Iwata also compare this strategy against branching on a vertex of minimum degree or a random vertex. They show that both of these perform significantly worse than branching on a maximum degree vertex (also see Section 3.1.5).

Xiao and Nagamochi [XN17] also use this strategy in most cases. For dense subgraphs, however, they use an edge branching strategy: They branch on an edge $\{u, v\}$ where $|N(u) \cap N(v)|$ is sufficiently large (depending on the maximum degree of the graph) by excluding both u and v in one branch and applying the alternative reduction (see Section 3.5.3.b)) to $\{u\}$ and $\{v\}$ in the other branch.

Bourgeois et al. [Bou+12] use maximum degree branching as long as there are vertices of degree of at least five. Otherwise, they utilize specialized algorithms to solve subinstances with an average degree of three or four. Those algorithms perform a rather complex case analysis to find a suitable branching vertex. The analysis is based on exploiting structures that contain 3- or 4-cycles. Branching on specific vertices in such structures often enables additional reduction rules to be applied.

Chen et al. [CKX10] use the notion of *good pairs* that are advantageous for branching. They choose these good pairs by a set of rules which are omitted here. They combine these with so-called *tuples* of a set of vertices and the number of vertices from this set that must be included in a maximum independent set. This information can be used when branching on a vertex contained in that set to remove additional vertices from the graph. Akiba and

Iwata [AI16] use the same concept in their packing rule. Chen et al. combine good pairs, tuples, and high degree vertices for their branching strategy.

Most algorithms for MC (e.g., [ST14; Tom+13]) compute a greedy coloring that assigns a minimum integer (coloring number) to each vertex. They then branch on vertices with the highest high coloring numbers. More sophisticated MC algorithms use MaxSAT encodings to prune the set of branching vertices [LFX13; LJM17; LQ10]. Li et al. [LJX15] combine greedy coloring and MaxSAT reasoning to further reduce the number of branching vertices.

Another approach for MC is using the *degeneracy order* $v_1 < v_2 < \cdots < v_n$ where v_i is a vertex of the smallest degree in $G \setminus \{v_1, \dots v_{i-1}\}$. Carraghan and Pardalos [CP90] present an algorithm that branches on vertices in descending degeneracy order. Li et al. [LFX13] introduce another vertex ordering using iterative maximum independent set computations (which might be easier than MC on some graphs) and breaking ties according to the degeneracy order.

3.5.2 Decomposition Branching

Our first approach to improve the default branching strategy found in many state-of-the-art algorithms (including that of Akiba and Iwata [AI16]) is to decompose the graph into several connected components. Subsequently, processing these components individually has been shown to improve the performance of branch-and-reduce in practice [AC19]. The algorithm by Akiba and Iwata also makes use of this. To this end, we now present three strategies with varying computational complexity: articulation points, edge cuts, and nested dissections.

3.5.2.a) Articulation Points

First, we are concerned with finding single vertices that are able to decompose a graph into at least two connected components. Such points are called *articulation points* (or cut vertices). Articulation points can be computed in linear time $\mathcal{O}(n + m)$ using a simple depth-first search (DFS) algorithm (see Hopcroft and Tarjan [HT73] for a detailed description). In particular, a vertex v is an articulation point if it is either the root of the DFS tree and has at least two children or any non-root vertex that has a child u, such that no vertex in the subtree rooted at u has a back edge to one of the ancestors of v.

For our first branching strategy, we maintain a set of articulation points $A \subseteq V$. When selecting a branching vertex, we first discard all invalid vertices from A, i.e., vertices that were removed from the graph by a preceding reduction step. If this results in A becoming empty, a new set of articulation points is computed on the current graph in linear time. If no articulation points exist, we select a vertex based on the default branching strategy. Otherwise, if A contains at least one vertex, an arbitrary one from A is chosen as the branching vertex. Figure 3.12 illustrates branching on an articulation point.

Even though this strategy introduces only a small (linear) overhead, articulation points can be rare depending on the type of graph. This results in the default branching strategy being selected rather frequently. Our preliminary experiments also indicate that articulation points are rarely found at higher depth. However, due to their low overhead, we can justify searching for them whenever A becomes empty.

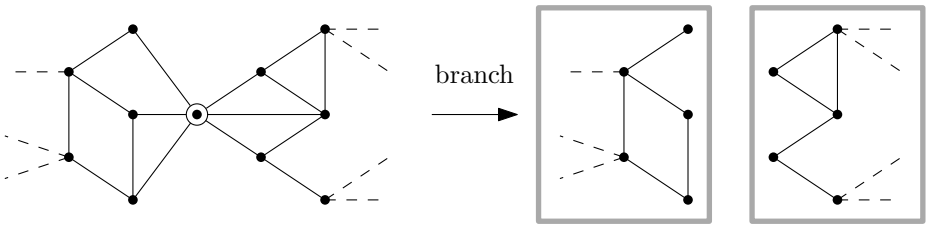


Figure 3.12: Branching on an articulation point (circled vertex) decomposes the graph into two connected components (gray boxes) that can be solved independently. Only the branch where the vertex is excluded from the independent set is shown.

3.5.2.b) Edge Cuts

To alleviate the restrictive nature of articulation points, we now propose a more flexible branching strategy based on *(minimal) edge cuts*. In general, we want to find small vertex separators, i.e., a set of vertices whose removal disconnects the graph. We do so by making use of vertex separators derived from minimum edge cuts.

As described in Section 2.1.2, a cut between two sets of vertices S and $T = V \setminus S$ is a set of edges $E(S,T) = \{\{u,v\} \in E \mid u \in S \wedge v \in T\}$. A cut is a *minimum cut* if it has minimal cardinality among all possible cuts of a graph. In practice, finding minimum cuts often yields trivial cuts with either S or T only consisting of a single vertex with the minimum degree. Thus, we are interested in finding *s-t-cuts*, i.e., cuts where S and T contain specific vertices $s,t \in V$. Finding these cuts can be done efficiently in practice, e.g., using a preflow push algorithm [GT88]. However, selecting the vertices s and t to ensure reasonably balanced cuts can be tricky. Natural choices include random vertices and vertices that are far apart in terms of their shortest path distance. However, our preliminary experiments indicate that selecting random vertices of maximum degree for s and t seems to produce the best results. Finally, to derive a vertex separator from a cut, one can compute an MVC on the bipartite graph induced by the cut, e.g., using the Hopcroft-Karp algorithm [HK73]. This separator can then be used to select branching vertices. In particular, we continuously branch on vertices from the separator.

Overall, our second strategy works similarly to the first one: We maintain a set of possible branching vertices that were selected by computing a minimum s-t-cut and turning it into a vertex separator. Vertices that were removed by reduction are discarded from this set, and once it is empty, a new cut computation is started. However, in contrast to the first strategy, finding a set of suitable branching vertices is much more likely.

In order to avoid separators that contain too many vertices and thus would require too many branching steps to disconnect the graph, we only keep those that do not exceed a certain size and balance threshold. The specific values for these thresholds are presented in Section 3.5.4. Finally, if no suitable separator is found, we use the default branching strategy. In this case, we do not try to find a new separator for a fixed number of branching steps, as finding one is both unlikely and costly.

3.5.2.c) Nested Dissection

Both of our previous strategies dynamically maintain a set of branching vertices. Even though this comes with the advantage that most of the computed vertices remain viable candidates for some branching steps, it introduces a noticeable overhead. To alleviate this, our last strategy uses a static ordering of possible branching vertices computed once at the beginning of the algorithm. For this purpose, we make use of a *nested dissection ordering* [Geo73].

A nested dissection ordering of the vertices of a graph G is obtained by recursively computing balanced bipartitions (A, B) and a vertex separator $\mathcal{S}$ that separates A and B. The actual ordering is then given by concatenating the orderings of A and B followed by the vertices of $\mathcal{S}$. Thus, if we select branching vertices based on the reverse of a nested dissection ordering, we continuously branch on vertices that disconnect the graph into balanced partitions. We compute such an ordering once after the initial reduction phase.

There are two main optimizations that we use when considering the nested dissection ordering. First, we limit the number of recursive calls during the nested dissection computation. We do so because we noticed that vertices at the end of the ordering seldom lead to a decomposition of the graph. This is due to the graph structure being changed by reductions, which can lead to separators becoming invalid. Furthermore, similar to the edge-cut-based strategy, we limit the size of separators considered during branching using a threshold. Again, this is done to ensure that we do not require too many branching steps to decompose the graph. The specific value for this size threshold is given in Section 3.5.4. If any separator in the nested dissection exceeds this threshold, we use the default strategy.

3.5.3 Reduction Branching

Our second main approach to selecting good branching vertices is to choose a vertex whose removal will enable the application of new reduction rules. During every reduction step, we find a list of candidate vertices to branch on. The following sections will demonstrate how we identify such branching candidate vertices with little computational overhead in practice. For ease of reading, we briefly repeat the relevant reduction rules introduced in Section 3.1.4. Out of the candidates found, we then select a vertex of maximum degree. If the degree of all candidate vertices lies below a threshold (defined in Section 3.5.4) or no candidate vertices were found, we fall back to branching on a vertex of maximum degree. The rationale here is that a vertex of large degree changes the graph's structure more than a vertex of small degree, even if that vertex is guaranteed to enable the application of a reduction rule. Also, our current strategies (except the packing-based rule in Section 3.5.3.d)) only allow the application of the targeted reduction rule in the excluding branch. However, in the including branch, all neighbors are removed from the graph as well since they already have an adjacent vertex in the solution. Thus, in both branches, multiple vertices are removed.

We also performed preliminary experiments with storing the candidate vertices in a priority queue without resetting after every branch. However, changes were too frequent for this approach to be faster because of the high amount of priority queue operations.

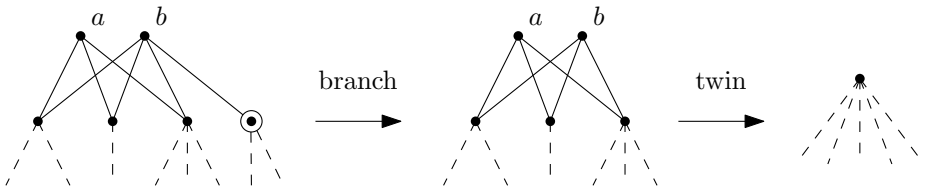


Figure 3.13: Vertices a and b are almost twins. After branching on the circled vertex they become twins (in the excluding branch) and can be reduced.

3.5.3.a) Almost Twins

The first reduction we target is the twin reduction by Xiao and Nagamochi [XN13].

Definition 3.6 (Twins)

In a graph $G = (V, E)$ two vertices u and v are called twins if $N(u) = N(v)$ and $d(u) = d(v) = 3$.

Theorem 3.7 (Twin Reduction)

In a graph $G = (V, E)$ let vertices u and v be twins. If there is an edge among $N(u)$, there is always a maximum independent set that includes $\{u, v\}$ and therefore excludes $N(u)$. Otherwise, let $G' = (V', E')$ be the graph with $V' = (V \setminus N[\{u, v\}]) \cup \{w\}$ where $w \notin V$ and $E' = (E \cap \binom{V'}{2}) \cup \{\{w, x\} \mid x \in N^2(u)\}$ and let $\mathcal{I}'$ be a maximum independent set in G'. Then, $\mathcal{I} = \begin{cases} \mathcal{I}' \cup \{u, v\} & \text{, if } w \notin \mathcal{I}' \\ (\mathcal{I}' \setminus \{w\}) \cup N(u) & \text{, else} \end{cases}$ is a maximum independent set in G.

We now define *almost twins* as follows:

Definition 3.8 (Almost Twins)

In a graph $G = (V, E)$ two non-adjacent vertices u and v are called almost twins if $d(u) = 4$, $d(v) = 3$ and $N(v) \subseteq N(u)$ (i.e., $N(u) = N(v) \cup \{w\}$).

Clearly, after removing w, u and v are twins, so we can apply the twin reduction. Finding almost twins can be done while searching for twins: The original algorithm checks for each vertex v of degree three whether there is a vertex $u \in N^2(v)$ with $d(u) = 3$ and $N(u) = N(v)$. We augment this algorithm by simultaneously also searching for $u \in N^2(v)$ with $d(u) = 4$ and $N(v) \subseteq N(u)$. This induces about the same computational cost for degree-4 vertices in $N^2(v)$ as for degree-3 vertices. While there might be instances where this causes high overhead, we expect the practical slowdown to be small. Figure 3.13 illustrates branching for almost twins.

3.5.3.b) Almost Funnels

Next, we consider the funnel reduction which is a special case of the alternative reduction by Xiao and Nagamochi [XN13].

Definition 3.9 (Alternative Sets)

In a graph $G = (V, E)$ two non-empty, disjoint subsets $A, B \subseteq V$ are alternatives if $|A| = |B|$ and there is a maximum independent set $\mathcal{I}$ in G such that $\mathcal{I} \cap (A \cup B)$ is either A or B.

Theorem 3.10 (Alternative Reduction)

In a graph $G = (V, E)$ let A and B be alternative sets. Let $G' = (V', E')$ the graph with $V' = V \setminus (A \cup B \cup (N(A) \cap N(B)))$ and $E' = \{\{u, v\} \in E \mid u, v \in V'\} \cup \{\{u, v\} \mid u \in N(A) \setminus N[B], v \in N(B) \setminus N[A]\}$ and let $\mathcal{I}'$ be a maximum independent set in G'. Then,

$$\mathcal{I} = \begin{cases} \mathcal{I}' \cup A & \text{, if } (N(A) \setminus N[B]) \cap \mathcal{I}' = \varnothing \\ \mathcal{I}' \cup B & \text{, else} \end{cases} \text{ is a maximum independent set in } G.$$

Note that the alternative reduction adds new edges between existing vertices of the graph, which might not be beneficial in every case. To counteract this, the algorithm by Akiba and Iwata [AI16] only uses special cases, one of which is the funnel reduction.

Definition 3.11 (Funnel)

In a graph $G = (V, E)$ two adjacent vertices u and v are called funnels if $G_{N(v) \setminus \{u\}}$ is a complete graph, i.e., if $N(v) \setminus \{u\}$ is a clique.

Theorem 3.12 (Funnel Reduction)

In a graph $G = (V, E)$ let u and v be funnels. Then, $\{u\}$ and $\{v\}$ are alternative sets.

Again, we define a structure that allows us to apply the funnel reduction after the removal of a single vertex:

Definition 3.13 (Almost Funnel)

In a graph $G = (V, E)$ two adjacent vertices u and v are called almost funnels if u and v are not funnels and there is a vertex w such that $N(v) \setminus \{u, w\}$ induces a clique.

By removing w, u and v become funnels. The original funnel algorithm checks whether u and v are funnels by iterating over the vertices in $N(v) \setminus \{u\}$ and checking whether they are adjacent to *all* previous vertices. Once a vertex is found that is not adjacent to all previous vertices, the algorithm concludes that u and v are not funnels and terminates. We augment this algorithm by not immediately terminating it in this case. Instead, we consider the following two cases: (1) The current vertex w is not adjacent to at least two of the previous vertices. In this case, we can check whether $N(v) \setminus \{u, w\}$ induces a clique. (2) w is adjacent to all but one previous vertex w'. In this case, both w and w' might be candidate branching vertices. Thus, we check whether $N(v) \setminus \{u, w\}$ or $N(v) \setminus \{u, w'\}$ induce a clique. This adds up to two additional clique checks (of slightly smaller size) performed in addition to the one clique check in the original algorithm.

3.5.3.c) Almost Unconfined

The core idea of the unconfined reduction by Xiao and Nagamochi [XN13; XN17] is to detect vertices not required for a maximum independent set. These vertices can therefore

be removed from the graph. To find these vertices, an algorithm is used to contradict the assumption that every maximum independent set contains the vertex.

Definition 3.14 (Child, Parent)

In a graph $G = (V, E)$ with an independent set $\mathcal{I}$, a vertex v is called a child of $\mathcal{I}$ if $|N(v) \cap \mathcal{I}| = 1$ and the unique neighbor of v in $\mathcal{I}$ is called the parent of v.

Algorithm 3.4 shows the algorithm used by Akiba and Iwata [AI16] to detect so-called *unconfined* vertices.

Algorithm 3.4 : Algorithm for finding unconfined vertices [XN13; XN17].

Data : $G = (V, E), v \in V$
Result : True if v is unconfined, False otherwise

```
1  S ← {v}
2  while True do
3      u ← u ∈ N(S) such that |N(u) ∩ S| = 1 and |N(u) \ N[S]| is minimized
4      if u = None then
5          return False
6      if N(u) \ N[S] = ∅ then
7          return True
8      if |N(u) \ N[S]| = 1 then
9          S ← S ∪ (N(u) \ N[S])
10     else
11         return False
```

Theorem 3.15 (Unconfined Reduction)

In a graph $G = (V, E)$, if Algorithm 3.4 returns true for an unconfined vertex v, there is always a maximum independent set that does not contain v.

Again, we define a vertex to be almost unconfined:

Definition 3.16 (Almost Unconfined)

In a graph $G = (V, E)$ a vertex v is called almost unconfined if v is not unconfined but there is a vertex w such that v is unconfined in $G \setminus \{w\}$.

Here, we only present an augmentation that detects *some* almost unconfined vertices. In particular, if at any point during the algorithm's execution there is only *one* extending child, i.e., a child u of S with $N(u) \setminus N[S] = \{w\}$, then removal of w makes v unconfined. During Algorithm 3.4, we collect all these vertices w and add them to the set of candidate branching vertices if the algorithm cannot already remove v. This only adds the overhead of temporarily storing the potential candidates and adding them to the actual candidate list if v is not removed.

3.5.3.d) Almost Packing

The core idea behind the packing rule by Akiba and Iwata [AI16] is that when the excluding branch of a vertex v is selected, one can assume that no maximum independent set contains v. Otherwise, if there is a maximum independent set that contains v, the algorithm finds it in the including branch of v. Based on the assumption that no maximum independent set includes a vertex v, constraints for the remaining vertices can be derived. For example, a maximum independent set that does not contain v has to include at least two neighbors of v. The corresponding constraint is $\sum_{u \in N(v)} x_u \geq 2$, where x_u is a binary variable indicating whether a vertex is included in the current solution. Otherwise, we find a solution of the same size in the branch including v. The algorithm creates such constraints when branching or reducing and updates them accordingly during the reductions and branching steps. When a vertex v is eliminated from the graph, x_v gets removed from all constraints. If v is included in the current solution, the corresponding right sides are also decreased by one.

A constraint $\sum_{v \in S \subset V} x_v \geq k$ can be utilized in two reductions. First, if k is equal to the number of variables $|S|$, all vertices from S have to be included in the current solution. If there are edges between vertices from S, then no valid solution can include all vertices from S, so the branch is pruned. Second, if there is a vertex v such that $|S| - |N(v) \cap S| < k$, then v has to be excluded from the current solution. If $k > |S|$, the constraint can not be fulfilled and the current branch is pruned.

In our branching strategy, we target both reductions. If there is a constraint $\sum_{v \in S \subset V} x_v \geq k$, where $|S| = k + 1$, excluding any vertex of S from the solution or including a vertex of S that has one neighbor in S enables the first reduction. Thus, we consider all vertices in S for branching. Note that including a vertex from S that has more than one neighbor in S makes the constraint unfulfillable and the branch is pruned.

If there is a constraint $\sum_{v \in S \subset V} x_v \geq k$ and a vertex v, such that $k = |S| - |N(v) \cap S|$, excluding any vertex of $S \setminus N(v)$ from the solution or including a vertex of $S \setminus N(v)$ that has at least one neighbor in $S \setminus N(v)$ enables the second reduction. Thus, we consider all vertices in $S \setminus N(v)$ for branching.

In contrast to our previous reduction-based branching rules, packing reductions can also be applied in the including branch in many cases. Detecting these branching candidates can be done with small constant overhead while performing the packing reduction.

3.5.4 Experimental Evaluation

In this section, we present the results of our experimental evaluation. The tables and figures presented show aggregated results. For detailed results of all instances, see Appendix E.

Methodology. We augment a C++adaptation of the algorithm by Akiba and Iwata [AI16] with our branching strategies and compile it with g++ version 9.3.0 using full optimizations (`-O3`). Our code is publicly available on GitHub[6]. We execute all our experiments on Machine B described in Section 2.3.2.c). All numbers reported are arithmetic means of three runs with a timeout of ten hours. Instances include the "easy" instances used for

[6] `https://github.com/Hespian/CutBranching`

the PACE Challenge, complements of maximum clique instances, and sparse networks taken from the set of instances described in Section 2.3.2.a). Detailed instance information can be found in Appendix A. Our original set of instances contained the first 80 PACE instances, 53 DIMACS instances, and 34 sparse networks. In particular, this set contained all corresponding instances used by Akiba and Iwata [AI16] and additional sparse networks. From the final set of instances, we excluded all instances that (1) required no branches, (2) on which all techniques had a running time of less than 0.1 seconds, or (3) on which no technique was able to find a solution within ten hours. The remaining set of instances comprises 48 PACE instances, 37 DIMACS instances, and 16 sparse networks.

Algorithm Configuration. We use a C++ adaptation of the implementation by Akiba and Iwata [AI16] in its default configuration as a basis for our algorithm. Preliminary experiments on a subset of our instances were used to find suitable values for the parameters of our techniques. In particular, we used the geometric mean over all instances of the speedup over the default branching strategy as a basis for the following decisions. For the technique based on edge cuts, we only use cuts that contain at most 25 vertices and where the smaller side of the cut contains at least ten percent of the remaining vertices. If no suitable separator is found, we skip ten branching steps. For computing nested dissections, we use InertialFlowCutter [Got+19] with the KaFFPa [SS13b] backend. The KaFFPa partitioner is configured to use the *strong* preset with a fixed seed of 42. For branching, we use three levels of nested dissections with a minimum balance of at least 40% of the vertices in the smaller part of each dissection. Furthermore, we only use the nested dissection if separators contain at most 50 vertices. For the reduction-based branching rules, we fall back to the default branching strategy if all candidates have a degree of less than $\Delta - k$. In the case of twin-, funnel- and unconfined-reduction-based branching strategies, we choose k as 2. For the packing-reduction-based branching rule, k is set to 5, and for the combined branching rule, k is set to 4.

3.5.4.a) Decomposition Branching

Figure 3.14 shows performance profiles (see Section 2.3.2.b)) of the running time and number of branches of our decomposition-based branching strategies: Let $\mathcal{T}$ be the set of all techniques we want to compare, I the set of instances and $t_T(I)$ the running time/number of branches of technique $T \in \mathcal{T}$ on instance $I \in I$. The y-axis shows for each technique T the fraction of instances for which $t_T(I) \leq \tau \cdot \min_{T' \in \mathcal{T}} t_{T'}(I)$, where τ is shown on the x-axis.

The running time plot in Figure 3.14 shows that for most instances, the default strategy of branching on a vertex of maximum degree outperforms our decomposition-based approaches. However, for instances that have suitable candidates for decomposition, such as sparse networks, significant speedups compared to the default strategy can be observed. To be more specific, by assigning a time of ten hours (our timeout threshold) for unfinished instances, we achieve a total speedup[7] of 2.15 to 2.29 over maximum degree branching on sparse networks (see Table 3.14 and Appendix E). In particular, there is one instance

[7] calculated by dividing the running times to solve all instances for two algorithms, excluding instances unsolved by both algorithms

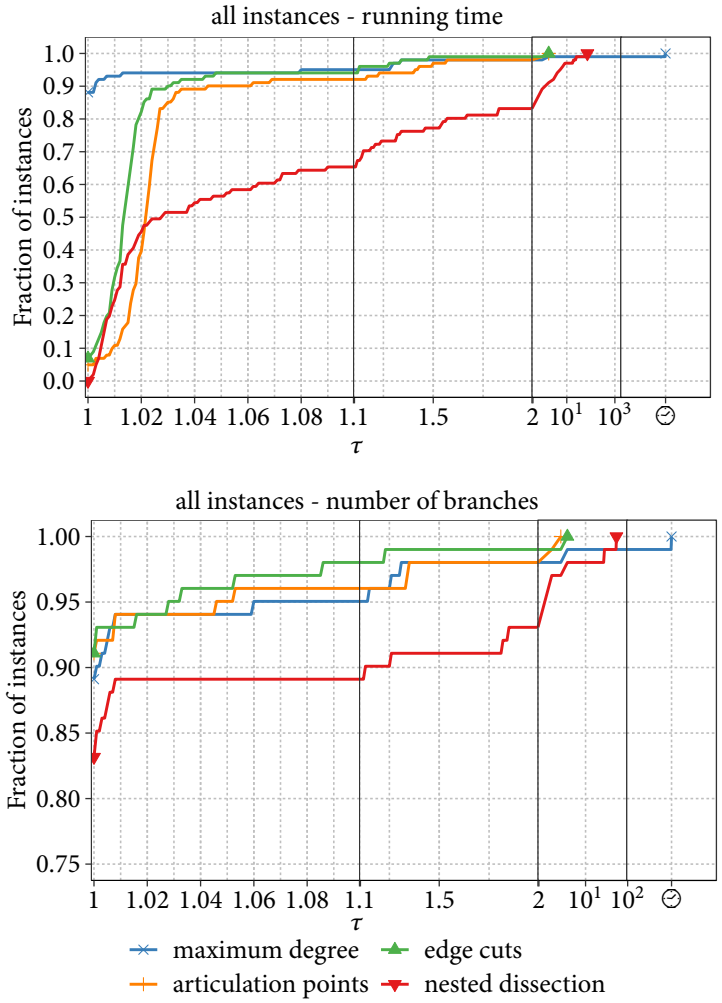


Figure 3.14: Performance profiles for decomposition-based branching strategies

(web-Stanford) that causes a timeout with the default strategy but can be solved in eight (articulation points) to 43 (nested dissections) seconds using a decomposition-based approach. Table 3.14 shows that our technique using edge cuts seems to be the most beneficial, achieving an overall speedup of 22% over maximum degree. Finally, Figure 3.14 shows that most running times are only slightly slower than the default strategy, with a few instances showing a speedup. This is mainly because the number of branches required to solve the instances does not change in most cases, and most of the running time difference is caused by the overhead from searching for branching vertices.

Table 3.14: Speedup of decomposition-based techniques over maximum degree branching.

Technique	PACE	DIMACS	Sparse net.	All Instances
articulation points	0.99	0.99	2.17	1.20
edge cuts	1.00	0.99	**2.29**	**1.22**
nested dissections	1.00	0.99	2.15	1.21

Table 3.15: Speedup of reduction-based techniques over maximum degree branching.

Technique	PACE	DIMACS	Sparse net.	All Instances
Twin	1.00	1.00	0.97	0.99
Funnel	1.14	0.99	0.98	1.02
Unconfined	0.79	1.00	0.86	0.92
Packing	**1.34**	**1.04**	**1.31**	**1.16**
Combined	1.14	1.03	1.30	1.12

3.5.4.b) Reduction Branching

Figure 3.15 shows the performance profiles for our reduction-based branching strategies. We see that targeting the packing reduction results in the fastest time for the most number of instances. In fact, targeting the packing reduction performs better than maximum degree branching on all but 3 PACE instances, achieving a speedup of 34% (see Table 3.15 and Appendix E) on these instances. On the DIMACS instances, performance is closer to that of maximum degree with an overall speedup of 4%. On sparse networks, packing is only faster than maximum degree branching on 6 out of 16 instances but still achieves an overall speedup of 31% due to being considerably faster on some longer running instances. The performance of our packing-based technique might be explained by its property of enabling a reduction in both the including and the excluding branch. In contrast, our other reduction-based techniques only allow a reduction in the excluding branch.

Our funnel-based technique is faster than maximum degree branching for all but 4 of the PACE instances, resulting in a speedup of 14% on these instances but only a 2% speedup over all instances due to slightly slower running times on the other instance classes. We also show results for a strategy that targets all reduction rules described in Section 3.5.3 (called *combined*). Even though this approach leads to the second-lowest number of branches for most instances, the time required to identify candidate vertices for all reduction rules causes too big of an overhead to be competitive. In fact, preliminary experiments showed that the number of branches is still small for a technique that only combines twin-, funnel-, and unconfined-based branching. Optimizing the algorithms to identify candidate vertices more quickly could make this combined strategy competitive.

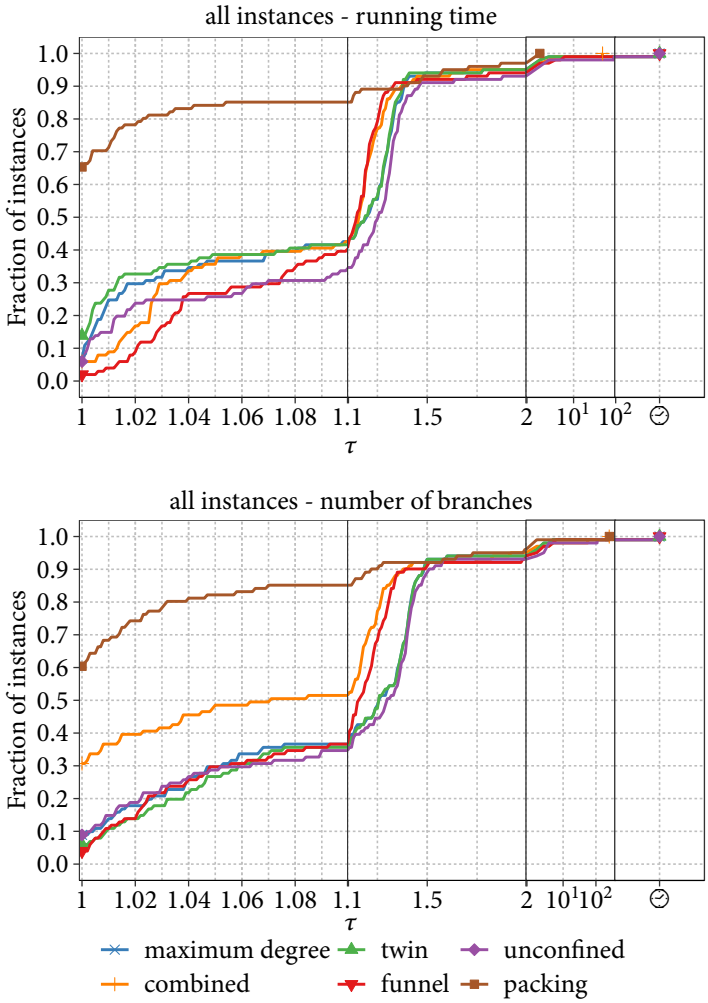


Figure 3.15: Performance profiles for reduction-based branching strategies

3.6 Conclusion and Future Work

In this chapter, we presented multiple algorithms that make use of practically efficient reduction techniques to find high-quality maximal or maximum independent sets. These algorithms scale to graphs with billions of edges and are able to solve many previously infeasible instances.

Heuristic Approaches. We first presented multiple heuristic approaches, including a very natural evolutionary algorithm and combinations of local search with reductions. Our

evolutionary algorithm uses novel combine operations that are based on graph partitioning and local search. In contrast to previous evolutionary algorithms for the problem, our operations are able to guarantee that the created offspring are valid. After computing a reduced instance, we use a preliminary solution of the evolutionary algorithm to further reduce the graph size by identifying and removing vertices likely to be in large independent sets. This further opens the reduction space (i.e., more exact reduction routines can be applied) so that we can proceed recursively. This speeds up computations drastically *and* preserves or even improves final solution quality. Similar inexact reduction procedures as the one introduced here are an important component of other state-of-the-art heuristic algorithms [CLZ17; Gao+17].

Our advanced local search approaches also make use of both exact and inexact reductions. To be more specific, we have shown that applying reductions on-the-fly during local search quickly leads to high-quality independent sets. Additionally, by reducing with advanced reduction rules, we can also improve local search for high-quality independent sets in the long run—rivaling the current best heuristic algorithms for complex networks. Finally, cutting a small percentage of high-degree vertices has little impact on the quality of independent sets found during local search. However, this inexact reduction can drastically improve performance by removing bottlenecks of the local search.

Overall, our approaches scale to instances that were previously infeasible and introduce techniques that can easily be applied to other problems. Additional reductions can easily be added to our algorithms and thus provide an interesting possibility for future improvement. Using more advanced reduction rules that still run in (near-)linear time has also been shown to be very efficient in practice [CLZ17]. Thus, exploring the usage of new reductions and efficient special cases of existing ones also provide interesting opportunities. Determining which reductions offer the best balance between solution quality and speed is an important topic that has recently gained more attention [Str16; SHG20].

More concrete future work includes a coarse-grained parallelization of our algorithms that could be combined with the parallel reduction routine of Hespe et al. [HSS19]. Furthermore, solving connected components separately and in parallel could also boost the performance of our algorithms further [AC19]. Lastly, it may also be interesting to use an exact algorithm when the reduced graph size falls below a certain threshold.

Exact Approaches. We presented two approaches that aim to close the gap between exact and heuristic algorithms in terms of speed and the number of feasible instances. First, we examined our winning solver of the PACE 2019 Challenge, Vertex Cover Track. Our algorithm uses a portfolio of techniques, including an aggressive reduction strategy with a large set of known reduction rules, local search, branch-and-reduce, and a branch-and-bound algorithm. Of particular interest is that several of our techniques were not from the literature on the vertex cover problem: they were originally published to solve the maximum independent set and maximum clique problems. Moreover, the two closest competitors in the challenge also applied reductions and a clique solver. Our results emphasize that reductions are an important tool to boost the performance of branch-and-bound and that local search is highly effective in increasing the performance of branch-and-reduce.

We then presented several novel branching strategies for the maximum independent set problem with the aim of further improving the performance of branch-and-reduce algorithms. Our strategies either follow a decomposition-based or reduction-rule-based approach. The decomposition-based strategies try to find vertices that are likely to decompose the graph into two or more connected components. Even though these strategies often come with a non-negligible overhead, they work well for graphs with a suitable structure, such as social networks. For instances that still favor the default branching strategy of branching on the vertex of the highest degree, our reduction-rule-based strategies provide a smaller but more consistent speedup. These rules aim to facilitate the application of reduction rules, leading to smaller graphs that can be solved more quickly.

Again, our exact algorithms scale to large instances and are able to provide better performance on a broad set of instances compared to other state-of-the-art approaches. Our portfolio algorithm can easily be extended by newly developed approaches, and our branching strategies are likely adaptable for additional reduction rules. Thus, integrating those in our algorithms provides an opportunity to improve their performance. Our PACE solver shows that approaches for the complementary problems are also beneficial in increasing the number of feasible instances. However, deciding which algorithm (or problem) to pick for a specific instances still remains an open question. Additionally, we have seen that sometimes an unreduced instance is actually easier to solve than a reduced one. The questions of whether reductions are always beneficial and what constitutes a "hard" reduced instance also remain open questions.

To further improve the performance of our branching strategies, we are actively investigating which particular strategy to use for a given instance. For this purpose, we are examining multiple graph characteristics and using machine learning approaches to evaluate which characteristics are suitable indicators for selecting a branching strategy. We are also implementing a more sophisticated and incremental way of tracking when a vertex becomes a potential branching vertex for our reduction-based strategies.

Chapter 4

4 Maximum Weight Independent Sets

We propose new and practically efficient reduction rules and transformations. These rules are used to build reduction procedures and exact approaches for the maximum weight independent set problem. We begin by proposing generalizations of popular reduction rules for the unweighted case and integrate them into a branch-and-reduce algorithm. We also propose so-called meta reductions that serve as an underlying theoretical framework. We then move on to transformations that are able to increase the graph size temporarily. This counterintuitive approach is integrated into a reduction procedure that can find the smallest graphs currently possible. In turn, it can boost the performance of existing algorithms drastically.

References. This chapter is based on the conference papers [Lam+19] (ALENEX 2019) published jointly with Christian Schulz, Darren Strash, Robert Williger, and Huashuo Zhang, [Gel+21] (ALENEX 2021) published jointly with Alexander Gellner, Christian Schulz, Darren Strash, and Bogdán Zaválnij, and the survey paper [Abu+20] published jointly with Faisal Abu-Khzam, Matthias Mnich, Alexander Noe, Christian Schulz, and Darren Strash. Large parts of this chapter were copied verbatim from these papers or the corresponding technical reports [Lam+18; Gel+20].

We now outline the specific contributions the author of this dissertation made to each of these publications. For the conference paper [Lam+19], the author, together with Christian Schulz and Darren Strash, is one of the main authors of the paper with editing done by Robert Williger. The author made major contributions to multiple of the practically efficient weighted reduction rules presented in Section 4.2.3. Note that the meta reduction rules (neighbor removal and neighborhood folding) are due to Darren Strash and Huashuo Zhang. In order to be self-contained, we include these rules. However, for the corresponding proofs, we refer to Lamm et al. [Lam+19]. Furthermore, the author made major contributions to the engineering aspects of the branch-and-reduce algorithms presented in Section 4.2.2. Implementation and evaluations were done by Robert Williger under the supervision of Christian Schulz, Darren Strash, and the author.

The conference paper [Gel+21] was written by the author of this dissertation with editing done by Alexander Gellner, Christian Schulz, Darren Strash, and Bogdán Zaválnij. The three struction variants proposed in this work (Section 4.3.1) are due to Bogdán Zaválnij, and we refer to Gellner et al. [Gel+21] for their corresponding proofs. The author made major contributions to multiple of the practically efficient struction variants (Section 4.3.2) and their incorporating algorithms (Section 4.3.3). This includes both the non-increasing and cyclic blow-up algorithm. Implementation and evaluations were done by Alexander Gellner under the supervision of Christian Schulz, Darren Strash, Bogdán Zaválnij, and the author.

Finally, for the survey paper [Abu+20], the author is the main author of the sections concerned with weighted independent sets, vertex covers, and cliques.

Motivation. The maximum weight independent set problem is an NP-hard problem [GJ79] that has attracted much attention in the combinatorial optimization community due to its difficulty and importance in many fields. In particular, it is used in areas such as map labeling [GNR16; Bar+16], vehicle routing [Don+22], coding theory [NKÖ97; Bro+90], combinatorial auctions [WH15b], alignment of biological networks [AKK11], workload scheduling for energy-efficient scheduling of disks [CKR11], computer vision [ML12], and protein structure prediction [Mas+10].

As a concrete example, weighted independent sets are vital in labeling strategies for maps [GNR16; Bar+16], where the objective is to maximize the importance of visible non-overlapping labels on a map. To be more specific, one is given a set of objects that have labels associated with them, e.g., geographic places, points of interest, or search requests [Bar+16]. Labels are weighted by their importance, e.g., a city's population or area, and are represented by a geometric shape (usually a rectangle). Two labels have a conflict if their geometric shapes intersect. To maximize the importance of non-overlapping labels, this problem can be reduced to a label conflict graph, i.e., the graph whose vertices are labels (weighted by their importance) and two vertices are connected if their labels overlap.

Maximum weight independent sets are also used for solving long-haul vehicle routing problems [Don+22; Don+21]. Given a set of routes consisting of a driver, a set of loads assigned to the driver, and a weight, the goal is to find a feasible subset of routes with maximum weight. A subset of routes is feasible if no two routes in it share a driver or a load. Again, this problem can be modeled as a conflict graph whose vertices are routes with their associated weight. Additionally, two vertices are connected if their corresponding routes share a route or a load. A maximum weight independent set in this graph then corresponds to a feasible subset of routes with maximum weight.

Instances used for these applications often contain hundreds of thousands up to millions of vertices and edges. As for the unweighted case, solving MWIS on these graphs using existing exact methods can be slow or infeasible in realistic time frames.

Overview. In this chapter, we take a look at how the results for the maximum cardinality independent set problem presented in the previous chapter can be extended and applied to its weighted counterpart. To this end, we begin by presenting relevant related work for the maximum weight independent set problem and its related problems with a focus on reductions in Section 4.1.

The two main contributions of this chapter are then presented in Sections 4.2 and 4.3. We begin by covering the first practically efficient branch-and-reduce algorithm for the maximum weight independent set problem. This algorithm uses novel reductions that contain both weighted generalizations of the rules popularized by Akiba and Iwata [AI16] and entirely new ones. We also include so-called meta reductions that serve as a theoretical framework for many of these rules. The resulting algorithm is able to solve a large number of instances up to two orders of magnitude faster than even existing heuristic algorithms. We also show that for instances that still remain infeasible, combining our reduction procedure with local search produces higher-quality solutions than local search alone.

The second approach we introduce in this chapter uses first of its kind, practically efficient variants of the struction rule by Ebenegger et al. [EHd84]. In contrast to existing approaches, the resulting algorithm leverages the full potential of the struction by temporarily allowing for the graph to increase in size. This increase in size may lead to smaller reduced graphs in the long run, as our experimental evaluation shows. We integrate this algorithm into the previously mentioned branch-and-reduce algorithm. Our experiments show that the small graphs produced by our algorithm enable many previously infeasible instances to be solved exactly. Additionally, we are able to solve instances up to orders of magnitude faster than previous algorithms. We then conclude this chapter with an outlook on interesting open problems and future work in Section 4.4.

Overall, this chapter covers important contributions for both heuristic and exact algorithms for the maximum weight independent set problem. Our algorithms drastically increase the scale and speed at which instances can be solved in practice. Additionally, they help heuristic algorithms to produce higher-quality solutions than previously possible.

4.1 Related Work

Due to the significant practical results achieved for the unweighted case presented in the previous chapter, there has been an increasing interest in generalizing reduction techniques for the maximum weight independent set (MWIS) and minimum weight vertex cover (MWVC) problems. In this section, we cover relevant related work, including both heuristic and exact approaches for MWIS and MWVC. As before, the maximum weight clique problem (MWC) will be treated separately due to the overhead required to transform between MWC and MWIS, i.e., computing complementary graphs.

4.1.1 Exact Approaches

Much research has been devoted to improving exact branch-and-bound algorithms for MWIS and its complementary problems. These improvements include different pruning methods and sophisticated branching schemes [Öst02; BY86; Bab94; WH06].

Warren and Hicks [WH06] proposed three combinatorial branch-and-bound algorithms that are able to solve DIMACS and weighted random graphs quickly. These algorithms use weighted clique covers to generate upper bounds that reduce the search space via pruning. Furthermore, they all use a branching scheme proposed by Balas and Yu [BY86]. In particular, their first algorithm is an extension and improvement of a method by Babel [Bab94]. Their second one uses a modified version of the algorithm by Balas and Yu that uses clique covers that borrow structural features from the ones by Babel [Bab94]. Finally, their third approach is a hybrid of both previous algorithms. Overall, their algorithms are able to solve instances with hundreds of vertices quickly.

Butenko and Trukhanov [BT07] proposed a reduction rule for MWIS called the critical weighted independent set reduction. A critical weighted set is a subset of vertices such that the difference between its weight and the weight of its neighboring vertices is maximal for all such sets. They can be found in polynomial time via minimum cuts. The authors then

show that every critical weighted independent set is part of a maximum weight independent set. We will cover this reduction rule in greater detail in Section 4.2.1.

As noted by Larson [Lar07], it is possible that the initial critical set found by Butenko and Trukhanov might be empty in the unweighted case. To prevent this case, Larson [Lar07] proposed an algorithm that finds a *maximum* (unweighted) critical independent set. His algorithm accumulates vertices that are in some critical set and removes their neighborhoods. Additionally, he provides a method to check if a given vertex is part of some critical set. Later Iwata [IOY14] has shown how to remove a large collection of vertices from a maximum matching all at once; however, it is not known if these reductions are equivalent.

Wang et al. [Wan+19] also make use of reduction rules for vertices with a degree of at most two as a preprocessing step for a branch-and-bound algorithm. Additionally, they evaluate different degree-based heuristics for selecting branching vertices and use pruning based on the best solution found so far.

The first practically efficient branch-and-reduce algorithm for MWIS that is able to solve large real-world instances was proposed by Lamm et al. [Lam+19] and is covered by the work detailed in Section 4.2. They develop a comprehensive set of practically efficient reduction rules. This includes generalizations of previous weighted and unweighted reduction rules, as well as two so-called meta reductions that serve as a general framework for the other rules.

Based on the work by Lamm et al. [Lam+19], Xiao et al. [Xia+21] propose additional generalized reduction rules. These include two reduction rules based on heavy sets. An independent set $\mathcal{I}$ is called a heavy set if for any independent set $\mathcal{I}'$ in the induced subgraph $G[N(\mathcal{I})]$ it holds that $w(N(\mathcal{I}') \cup \mathcal{I}) \geq w(\mathcal{I}')$ [Xia+21]. They also provide weighted generalizations of unconfined vertices and alternative sets (see Section 3.1.4). Finally, they introduce two reduction rules based on simultaneous sets. A simultaneous set is a vertex set such that all vertices in this set are either in a maximum weight independent set or not [Xia+21]. They then combine these rules with existing ones into an efficient preprocessing procedure that is used as part of a branch-and-reduce algorithm. Their experimental evaluation indicates that they are able to reduce both running time and reduced graph sizes significantly compared to the algorithm by Lamm et al. [Lam+19] on both SNAP and OSM instances.

Huang et al. [HXC21] also introduce a set of weighted reduction rules based on previous works [Lam+19; Xia+21]. They use these rules in combination with a variety of branching strategies in a branch-and-reduce algorithm and analyze its running complexity using the measure-and-conquer technique [FGK09]. By doing so, they show that MWIS on graphs with average degree x can be solved in $\mathcal{O}^*(1.1443^{0.624x-0.872)n})$.

Zheng et al. [Zhe+20] propose a heuristic and exact approach that both make use of reduction rules for vertices of degree at most two. Their exact approach is a branch-and-reduce algorithm that applies these reduction rules recursively. However, the authors do not provide any details on the bounds used for pruning or branching strategies used in the algorithm. Their heuristic approach is inspired by the reducing-peeling framework of Chang et al. [CLZ17]. Thus, it exhaustively applies their reduction rules and subsequently removes high-degree vertices to extend the space of possible reductions.

Gellner et al. [Gel+21] proposed new practically efficient variants of the struction rule by Ebenegger et al. [EHd84]. Their algorithm is able to produce the smallest-known reduced

graphs, solves more instances than previous exact approaches, and has a running time that is comparable to heuristic algorithms. It will be covered in the work presented in Section 4.3.

Finally, there are exact procedures that are either based on other extensions of branch-and-bound, e.g., [Reb+11; War+05; War03] or on the reformulation into other NP-hard problems, for which a variety of algorithms already exist. For instance, Xu et al. [XKK16] recently developed an algorithm called SBMS, which calculates an optimal solution for a given MWVC instance by solving a series of SAT instances.

4.1.2 Heuristic Approaches

For the unweighted case, the iterated local search algorithm by Andrade et al. [ARW12] (Section 3.1.6) is one of the most successful approaches in practice. Their algorithm was improved significantly by applying (2,1)- and (k,1)-swaps as perturbation steps [JH15], omitting high-degree vertices [Dah+16a], and using reduction rules [Dah+16a; Lam+17].

Several local search algorithms have been proposed for the maximum weight independent set problem [Pul09; LCH17; NPS18; Cai+18; Li+20]. Local search approaches are often able to obtain high-quality solutions on medium to large instances that are not solvable using exact algorithms. We now cover some of the most recent state-of-the-art local search algorithms in greater detail.

The hybrid iterated local search (HILS) by Nogueira et al. [NPS18] adapts the ARW algorithm for weighted graphs. In addition to weighted $(1,2)$-swaps, it also uses $(\omega,1)$-swaps that add one vertex v into the current solution and exclude its ω neighbors. These two types of neighborhoods are explored separately using variable neighborhood descent (VND). Additionally, they introduce a dynamic perturbation mechanism that regulates intensification and diversification. In practice, their algorithm finds all known optimal solutions on well-known benchmark instances within milliseconds and outperforms other state-of-the-art local searches.

Cai et al. [Cai+18] proposed a heuristic algorithm for MWVC that is able to derive high-quality solutions for a variety of large real-world instances. Their algorithm is based on a local search algorithm by Li et al. [LCH17] and uses iterative removal and maximization of a valid vertex cover. Vertices are removed and inserted using two scoring functions: a gain function and a loss function. In particular, the baseline algorithm removes two vertices in each iteration. The first vertex is chosen using the minimal loss value; the second one is selected by the best from multiple selection heuristic [Cai15]. They then introduce two dynamic approaches for selecting between different scoring functions for the gain and loss values. To be more precise, their first approach selects between two different scoring functions by counting the number of non-improving steps. Their second approach dynamically adjusts the number of vertices removed in a single iteration based on their total degree.

Recently, Li et al. [Li+20] used a set of four reduction rules during the initial construction phase of a local search algorithm. In particular, they use weighted reduction rules that are able to remove degree-1 and degree-2 vertices. They then use these reduction rules exhaustively at the beginning of their algorithm to obtain an improved initial solution. They also adapt the configuration checking approach [CSS11] to MWVC, which is used to reduce cycling, i.e., returning to a solution that has been visited recently. Finally, they develop a

technique called self-adaptive-vertex-removing, which dynamically adjusts the number of removed vertices per iteration. Their local search algorithm called NuMWVC is able to compute high-quality solutions on a large variety of instances. This includes many instances commonly used for the unweighted case, which have been given vertex weights drawn from a uniform distribution.

In order to overcome the shortcomings of both heuristic and exact approaches, new algorithms that combine reduction rules with local search were proposed recently [Dah+16a; Lam+16]. A very successful approach using this approach is the reducing-peeling framework proposed by Chang et al. [CLZ17], which is based on the techniques proposed by Lamm et al. [Lam+16]. Their algorithm works by computing a reduced graph using practically efficient reduction rules in linear and near-linear time. Additionally, they provide an extension of their reduction rules that is able to compute good initial solutions for the reduced instance. In particular, they greedily select vertices that are unlikely to be in a large independent set, thereby opening up the reduction space again. Thus, they are able to significantly improve the performance of the ARW local search algorithm that is applied to the reduced graph.

Gu et al. [Gu+21] propose a framework for computing near-maximum weight independent sets based on reducing-peeling. To this end, they introduce two types of practically efficient reductions: low-degree reductions and high-degree reductions. Low-degree reductions aim to efficiently select vertices that are contained in a maximum weight independent set and remove vertices with degree one and two. High-degree reductions aim to further reduce the graph by eliminating vertices with degree greater than two and can be applied by computing common neighbors for the two endpoints of every edge. Furthermore, the authors evaluate three different tie-breaking mechanisms that are used for eliminating vertices when no further reductions can be applied. Their experimental evaluation on 23 sparse instances indicates that their algorithm is able to compute high-quality solutions orders of magnitude faster than previous approaches.

Finally, Dong et al. [Don+22] recently presented a new local search algorithm that uses a wide range of simple and efficient local search operations. Their algorithm is an iterative local search based on the Greedy Randomized Adaptive Search Procedure (GRASP) metaheuristic [FR89; FR95; RR14]. They also introduce a new variant of path relinking [LM99; RR14] and a new alternating augmenting path local search that allows them to escape local optima and improve the performance of their algorithm. Their experimental evaluation compares their algorithm against the hybrid iterated local search HILS by Nogueria et al. [NPS18] on a large set of instances stemming from vehicle routing and map labeling, as well as randomly weighted computer, social, and road networks. For the map labeling and randomly weighted instances, they further use the reduction algorithm by Lamm et al. [Lam+19] and run their algorithm on the resulting reduced graphs. They show that their algorithm is able to outperform HILS on large vehicle routing instances while also being competitive on the remaining instances.

4.1.3 Maximum Weight Clique

As for the maximum cardinality clique problem (MC), there exist multiple works on efficient branch-and-bound algorithms for its weighted counterpart MWC [Öst01; Öst02; Kum08;

Shi+17; JLM17; Li+18], many of which aim to improve upper bounds that are used for pruning unnecessary branches. Due to its success for MC, MaxSat reasoning has also been used in multiple MWC approaches [Fan+14; LQ10; Jia+18]. The two-stage MaxSat reasoning approach by Jiang et al. [Jia+18], in particular, has been shown to be very efficient on a large set of real-world graphs, including DIMACS instances [JT96], instances taken from the Network Data Repository [RA15], as well as instances taken from different application domains of MWC.

There also is a wide range of heuristics and local search algorithms available for the maximum clique problem [CL16; WH13; WHG12; WCY16; ZHG17; Fan+17; Wan+20; Cai+21]. Many of the best performing of these algorithms also make use of reduction rules [CL16; Wan+20; Cai+21]. Thus, we now discuss these approaches in greater detail.

Cai et al. [CL16; Cai+21] give a heuristic algorithm that interleaves between clique finding and reductions and is also able to verify that a computed solution is exact. In particular, their algorithm uses a novel construct-and-cut method for clique finding. In each iteration of this method, the solution is extended by a vertex selected using a new benefit estimation function and a dynamic variant of the best from multiple selection heuristic [Cai15]. Once a new best solution is found, their algorithm then tries to apply reductions and reduce the graph size. If this results in the graph becoming empty, the resulting solution is optimal. To be more specific, they use three reduction rules that are able to remove a vertex v by computing three different upper bounds on the weight of any clique containing v. Their algorithm finds better solutions than the state-of-the-art on large sparse graphs while also often requiring significantly less time. Furthermore, it is able to verify the optimality of these solutions for about half of these graphs tested. We briefly note that their algorithm and reductions are targeted at sparse graphs, and therefore their reductions would likely work well for MWIS on *dense* graphs—but not on the sparse graphs we mainly consider in this dissertation. To the best of our knowledge, their algorithm is the first practical MWC algorithm using reductions.

The previously mentioned reduction rules of Cai et al. [CL16; Cai+21] are also used in the work of Wang et al. [Wan+20]. In particular, they propose a local search algorithm that uses a new variant of configuration checking called strong configuration checking and a perturbation heuristic called walk perturbation. They also propose a second algorithm tailored for large graphs that improves the previous algorithm by incorporating the best from multiple selection heuristic [Cai15] and a modified version of the aforementioned reduction rules by Cai and Lin [CL16]. The resulting algorithms perform significantly better than previous heuristic algorithms and are able to compute optimal solutions for a large set of instances.

4.2 Generalized Reduction Rules

As seen in Section 3.1.5, branch-and-reduce is a powerful technique for the maximum independent set problem. However, significantly fewer approaches have been proposed and evaluated for its weighted counterpart. This is in part due to a lack of known effective reductions. In particular, while the maximum independent set problem has many known

reductions, we are only aware of a few practical reductions for the maximum weight independent set problem that existed prior to our work. These include the weighted critical independent set reduction by Butenko and Trukhanov [BT07] and the struction by Ebenegger et al. [EHd84]. Both of these have only been tested on small synthetic instances and not on large instances, which are the focus of this work.

There is only one other reduction-like procedure of which we are aware, although it is neither called so directly nor is it explicitly implemented as a reduction. Nogueria et al. [NPS18] introduced the notion of a $(\omega, 1)$-swap in their local search algorithm, which swaps a vertex into a solution if its neighbors in the current solution have a smaller total weight. This swap is not guaranteed to select a vertex that is part of a maximum weight independent set; however, we show how to transform it into a reduction that does.

In this section, we develop a full suite of new reductions for MWIS and provide extensive experiments to show their effectiveness in practice on real-world graphs of up to millions of vertices and edges. While existing exact algorithms are only able to solve graphs with hundreds of vertices, our experiments show that our approach is able to exactly solve real-world label conflict graphs with thousands of vertices and other larger networks with synthetically generated vertex weights—all of which are infeasible for state-of-the-art algorithms. Further, our branch-and-reduce algorithm is able to solve many instances up to two orders of magnitude faster than existing local search algorithms—solving the majority of instances within 15 minutes. For those instances remaining infeasible, we show that combining reductions with local search produces higher-quality solutions than local search alone.

Finally, we develop new *meta* reductions, which are general rules that subsume traditional reductions. We show that weighted variants of popular unweighted reductions can be explained by two general (and intuitive) rules—which use maximum weight independent set search as a subroutine. This yields a simple theoretical framework covering many reductions.

Organization. The rest of the section is organized as follows. We first present the critical weighted independent set reduction in Section 4.2.1. We then describe the overall structure of our branch-and-reduce algorithm in Section 4.2.2. The full set of reductions for the maximum weight independent set problem that are employed by us is described in Section 4.2.3. An extensive experimental evaluation of our method is presented in Section 4.2.4.

4.2.1 Critical Weighted Independent Set Reduction

A subset $U_c \subseteq V$ is called a *critical set* if $|U_c| - |N(U_c)| = \max\{|U| - |N(U)| : U \subseteq V\}$. Likewise, an independent set $\mathcal{I}_c \subseteq V$ is called a *critical independent set* if $|\mathcal{I}_c| - |N(\mathcal{I}_c)| = \max\{|\mathcal{I}| - |N(\mathcal{I})| : \mathcal{I} \text{ is an independent set of } G\}$. Butenko and Trukhanov [BT07] show that any critical independent set is a subset of a maximum independent set. They then continue to develop a reduction that uses critical independent sets which can be computed in polynomial time. In particular, they start by finding a critical set in G by using a reduction to the maximum matching problem in a bipartite graph [Age94] . In turn, this problem can then be solved in $\mathcal{O}\left(n\sqrt{m}\right)$ time using the Hopcroft-Karp algorithm [HK73]. They then obtain a critical independent set by setting $\mathcal{I}_c = U_c \setminus N(U_c)$. Finally, they can remove $\mathcal{I}_c$ and $N(\mathcal{I}_c)$ from G.

We now briefly describe the critical *weighted* independent set reduction, which is one of the few reductions that has appeared in the literature for the maximum weight independent set problem. Similar to the unweighted case, a subset $U_c \subseteq V$ is called a *critical weighted set* if $w(U_c) - w(N(U_c)) = \max\{w(U) - w(N(U)) : U \subseteq V\}$. A weighted independent set $\mathcal{I}_c \subseteq V$ is called a *critical weighted independent set* (CWIS) if $w(\mathcal{I}_c) - w(N(\mathcal{I}_c)) = \max\{w(\mathcal{I}) - w(N(\mathcal{I})) : \mathcal{I}$ is an independent set of $G\}$. Butenko and Trukhanov [BT07] show that any CWIS is a subset of a maximum weight independent set. Additionally, they propose a weighted critical set reduction which works similar to its unweighted counterpart. However, instead of computing a maximum matching in a bipartite graph, a critical weighted set is obtained by solving the selection problem [Age94]. The *selection problem* [Bal70; Rhy70] consists of a finite set of points S, a cost $c_s > 0$ for each $s \in S$, a finite set Σ of subsets σ of points from S, and a profit p_σ for each $\sigma \in \Sigma$. A *selection* is a set of subsets from Σ together with the set of points from S contained in the subsets. The goal of the selection problem then is to choose a selection that maximizes the sum of the profits minus the sum of the costs of the points. This problem is equivalent to finding a minimum cut in a bipartite graph [Bal70; Rhy70].

Theorem 4.1 (CWIS Reduction)
Let $U \subseteq V$ be a critical weighted independent set of G. Then U is in some maximum weight independent set of G. We set $G' = G \setminus N[U]$ and $\alpha_w(G) = \alpha_w(G') + w(U)$.

For a proof of correctness, see the paper by Butenko and Trukhanov [BT07].

4.2.2 Efficient Branch-and-Reduce

After covering the necessary preliminaries, we now describe our branch-and-reduce algorithm in full detail. This includes the pruning and branching techniques that we use, as well as other algorithm details. An overview of our algorithm can be found in Algorithm 4.1. To keep the description simple, the pseudocode describes the algorithm such that it outputs the weight of a maximum weight independent set in the graph. However, our algorithm is implemented to actually output the maximum weight independent set. We now describe a high-level overview of our algorithm before covering the individual parts in greater detail later. Overall, our algorithm follows the steps used by Akiba and Iwata [AI16] (Section 3.1.5).

Throughout the algorithm, we maintain the current solution weight and the best solution weight. Our algorithm applies a set of reduction rules before branching on a vertex. We describe these reductions in the following section. Initially, we run a local search algorithm on the reduced graph to compute a lower bound on the solution weight, which later helps to prune the search space. We then prune the search by excluding unnecessary parts of the branch-and-bound tree to be explored. If the graph is not connected, we separately solve each connected component. If the graph is connected, we branch into two cases by applying a branching rule. If our algorithm does not finish within a certain time limit, we use the best solution and improve it using a greedy approach. More precisely, our algorithm sorts the vertices in decreasing order of their weight and adds vertices in that order if feasible. We give a detailed description of the subroutines below.

Algorithm 4.1 : Branch-and-reduce algorithm for MWIS.

Data : $G = (V, E)$, current solution weight c, best solution weight $\mathcal{W}$
Result : best solution weight $\mathcal{W}$

```
1  Solve G, c, W
2      (G, c) ← Reduce(G, c)
3      if W = 0 then
4          W ← c + ARW(G)
5      if c + UpperBound(G) ≤ W then
6          return W
7      if G is empty then
8          return max{W, c}
9      if G is not connected then
10         forall G_i ∈ Components(G) do
11             c ← c + Solve(G_i, 0, 0)
12         return max(W, c)
13     (G_1, c_1), (G_2, c_2) ← Branch(G, c)
14     W ← Solve(G_1, c_1, W)          // Run 1st case, update current best solution
15     W ← Solve(G_2, c_2, W)          // Use updated W to shrink the search space
16     return W
```

Incremental Reductions. We start by running all reductions that are described in the following section. Following the lead of previous works [Str16; CLZ17; HSS19], we apply our reductions *incrementally*. For each reduction rule, we check if it is applicable to any vertex of the graph. After the checks for the current reduction are completed, we continue with the next reduction if the current reduction has not changed the graph. If the graph has been changed, we go back to the first reduction rule and repeat. Most of the reductions we introduce in the following section are *local*: if a vertex changes, then we do not need to check the entire graph to apply the reduction again; we only need to consider the vertices whose neighborhood has changed since the reduction was last applied. The critical weighted independent set reduction defined above is the only *global* reduction we use; it always considers all vertices in the graph.

For each of the local reductions, there is a queue of changed vertices associated to the rule. Every time a vertex or its neighborhood is changed, it is added to the queues of all reductions. When a reduction is applied, only the vertices in its associated queue have to be checked for applicability. After the checks are finished for a particular reduction, its queue is cleared. Initially, the queues of all reductions are filled with every vertex in the graph.

Pruning. Exact branch-and-bound algorithms for MWIS often use weighted clique covers to compute an upper bound for the optimal solution [WH06]. A weighted clique cover of G is a collection of (possibly overlapping) cliques $\mathcal{Q}_1, \ldots, \mathcal{Q}_k \subseteq V$, with associated weights

$W_1, \ldots, W_k$ such that $\mathcal{Q}_1 \cup \mathcal{Q}_2 \cup \cdots \cup \mathcal{Q}_k = V$, and for every vertex $v \in V$, $\sum_{i : v \in \mathcal{Q}_i} W_i \geq w(v)$, i.e., each vertex is covered by a set of cliques whose total weight is larger than the weight of the vertex. The weight of a clique cover is defined as $\sum_{i=1}^{k} W_i$ and provides an upper bound on $\alpha_w(G)$. This holds because the intersection of a clique and any independent set of G is either a single vertex or empty. The objective then is to find a clique cover of small weight. This can be done using an algorithm similar to the coloring method of Brelaz [Bré79]. However, this method can become computationally expensive since its running time is dependent on the maximum weight of the graph [WH06]. Thus, we use a faster method to compute a weighted clique cover similar to the one used in Akiba and Iwata [AI16]. In particular, we begin by sorting the vertices in descending order of their weight (ties are broken by selecting the vertex with higher degree). Next, we initiate an empty set of cliques Q. We then iterate over the sorted vertices and search for the clique with the maximum weight to which it can be added. If there are no candidates for insertion, we insert a new single vertex clique to Q and assign it the weight of the vertex. Afterwards, the vertex is marked as processed, and we continue with the next one. Computing a weighted clique cover using this algorithm has a linear running time independent of the maximum weight. Thus, we are able to obtain a bound much faster. However, this algorithm produces a higher weight clique cover than the method of Brelaz [Bré79; AI16].

In addition to computing an upper bound, we also add an additional lower bound using a heuristic approach. To be more specific, we run a modified version of the ARW local search by Andrade et al. [ARW12], which is able to handle vertex weights, for a fixed fraction of our total running time. This lower bound is computed once after we apply our reductions initially and then again when splitting the graph into connected components.

Connected Components. Solving the maximum weight independent set problem for a graph G is equal to solving the problem for all c connected components $G_1, \ldots, G_c$ of G and then combining the solution sets $\mathcal{I}_1, \ldots, \mathcal{I}_c$ to form a solution $\mathcal{I}$ for G: $\mathcal{I} = \bigcup_{i=1}^{c} \mathcal{I}_i$. We leverage this property by checking the connectivity of G after each completed round of reduction applications. If the graph disconnects due to branching or reductions, we then apply our branch-and-reduce algorithm recursively to each of the connected components and combine their solutions afterwards. This technique can reduce the size of the branch-and-bound tree significantly on some instances.

Branching. Our algorithm has to pick a branching order for the remaining vertices in the graph. Initially, vertices are sorted in non-decreasing order by degree, with ties broken by weight. Throughout the algorithm, the next vertex to be chosen is the highest vertex in the ordering. Similar to Akiba and Iwata [AI16], we first process the excluding branch and then the including branch. This way, we quickly eliminate the largest neighborhoods and make the problem "simpler".

4.2.3 Weighted Reduction Rules

We now develop a comprehensive set of reduction rules for the maximum weight independent set problem. We first introduce two *meta* reductions, which we then use to instantiate many efficient reductions similar to already-known unweighted reductions.

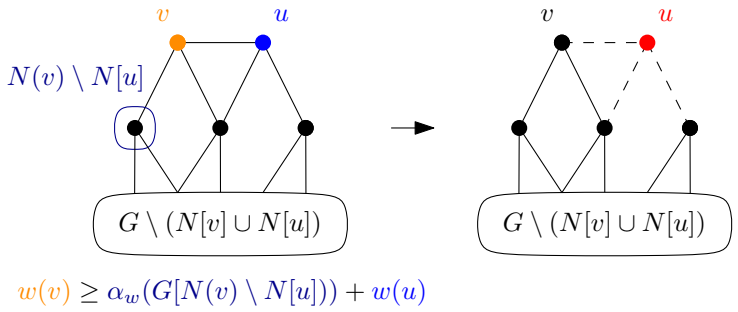


Figure 4.1: Example application of neighbor removal to a vertex v (orange). The vertex u (blue) can be removed from the graph and excluded from the independent set.

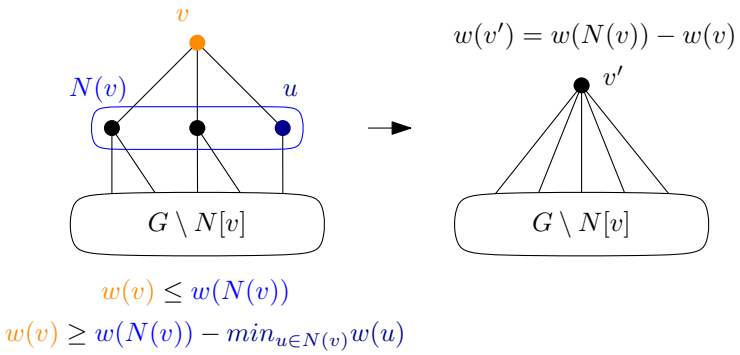


Figure 4.2: Example application neighborhood folding to a vertex v (orange). The neighborhood $N[v]$ is contracted to a new vertex v'.

4.2.3.a) Meta Reductions

There are two operations that are commonly used in reductions: *vertex removal* and *vertex folding*. In the following reductions, we show general ways to detect when these operations can be applied in the neighborhood of a vertex.

Neighbor Removal. In our first meta reduction, we show how to determine if a neighbor can be outright removed from the graph. We call this reduction the *neighbor removal reduction*. (See Figure 4.1.)

Theorem 4.2 (Neighbor Removal Reduction)

Let $v \in V$. For any $u \in N(v)$, if $\alpha_w(G[N(v) \setminus N[u]]) + w(u) \leq w(v)$, then u can be removed from G, as there is some maximum weight independent set of G that excludes u, and $\alpha_w(G) = \alpha_w(G \setminus \{u\})$.

Neighborhood Folding. For our second meta reduction, we show a general condition for folding a vertex with its neighborhood. We first briefly describe the intuition behind the

reduction. Consider v and its neighborhood $N(v)$. If $N(v)$ has a unique independent set $\mathcal{I}_{N(v)}$ with a weight larger than $w(v)$, then we only need to consider two types of independent sets: independent sets that contain v or ones that contain $\mathcal{I}_{N(v)}$. Any other independent set in $N(v)$ can be swapped for v and achieve a higher overall weight. By folding v with $\mathcal{I}_{N(v)}$, we can solve the remaining graph and then decide which of the two options will give a maximum weight independent set of the graph. (See Figure 4.2.)

Theorem 4.3 (Neighborhood Folding Reduction)
Let $v \in V$ and suppose that $N(v)$ is independent, i.e., the vertices in $N(v)$ are not adjacent. If $w(N(v)) \geq w(v)$, but $w(N(v)) - \min_{u \in N(v)}\{w(u)\} \leq w(v)$, then fold v and $N(v)$ into a new vertex v' with weight $w(v') = w(N(v)) - w(v)$. Let $\mathcal{I}'$ be a maximum weight independent set of G', then we construct a maximum weight independent set $\mathcal{I}$ of G as follows: If $v' \in \mathcal{I}'$ then $\mathcal{I} = (\mathcal{I}' \setminus \{v'\}) \cup N(v)$. Otherwise, if $v \in \mathcal{I}'$ then $\mathcal{I} = \mathcal{I}' \cup \{v\}$. Furthermore, $\alpha_w(G) = \alpha_w(G') + w(v)$.

However, these reductions require solving MWIS on the neighborhood of a vertex and, therefore, may be as expensive as computing a maximum weight independent set on the input graph. We now show how special cases of our meta reductions can be used to build practically efficient reductions.

4.2.3.b) Practically Efficient Reductions

Neighborhood Removal. In their HILS local search algorithm, Nogueria et al. [NPS18] introduced the notion of a $(\omega, 1)$-swap, which swaps a vertex v into a solution if its neighbors in the current solution $\mathcal{I}$ have weight $w(N(v) \cap \mathcal{I}) < w(v)$. This can be transformed into what we call the *neighborhood removal reduction*.

Theorem 4.4 (Neighorhood Removal Reduction)
For any $v \in V$, if $w(v) \geq w(N(v))$ then v is in some maximum weight independent set of G. Let $G' = G \setminus N[v]$ and $\alpha_w(G) = \alpha_w(G') + w(v)$.

Proof. Since $w(N(v)) \leq w(v)$, $\forall u \in N(v)$ we have that

$$\alpha_w(G[N(v) \cap N(u)]) + w(u) \leq w(N(v)) \leq w(v).$$

Then we can remove all $u \in N(v)$ and are left with v in its own component. Calling this graph G', we have that v is in some maximum weight independent set and $\alpha_w(G) = \alpha_w(G') + w(v)$. □

For the remaining reductions, we assume that the neighborhood removal reduction has already been applied. Thus, $\forall v \in V$, $w(v) < w(N(v))$.

Weighted Isolated Clique. Similar to the (unweighted) isolated clique reduction, we now argue that an isolated vertex is in some maximum weight independent set—if it has the highest weight in its clique (see Figure 4.3).

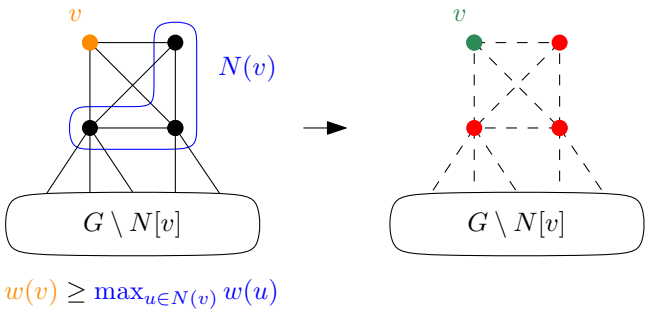


Figure 4.3: Example application of the weighted isolated clique reduction to vertex v (orange). The neighborhood $N[v]$ can be removed from the graph. The vertex v is added to the independent set (and $N(v)$ is excluded).

Theorem 4.5 (Weighted Isolated Clique Reduction)
Let $v \in V$ be isolated and $w(v) \geq \max_{u \in N(v)} w(u)$. Then v is in some maximum weight independent set of G. Let $G' = G \setminus N[v]$ and $\alpha_w(G) = \alpha_w(G') + w(v)$.

Proof. Since $N(v)$ is a clique, $\forall u \in N(v)$ we have that

$$\alpha_w(G[N(v) \cap N(u)]) \leq \alpha_w(N(v)) = \max_{u \in N(v)} \{w(u)\} \leq w(v).$$

As was done for neighborhood removal, remove all $u \in N(v)$ producing G' and $\alpha_w(G) = \alpha_w(G') + w(v)$. □

Isolated Weight Transfer. Given its weight restriction, the weighted isolated clique removal reduction may be ineffective. Therefore, we introduce a reduction that supports more liberal vertex removal. An example application of this reduction rule is given in Figure 4.4.

Theorem 4.6 (Isolated Weight Transfer)
Let $v \in V$ be isolated and suppose that the set of isolated vertices $S(v) \subseteq N(v)$ is such that $\forall u \in S(v)$, $w(v) \geq w(u)$. We

(i) remove all $u \in N(v)$ with $w(u) \leq w(v)$; the remaining neighbors are denoted by $N'(v)$,

(ii) remove v and $\forall x \in N'(v)$, set its new weight to $w'(x) = w(x) - w(v)$ and let the resulting graph be denoted by G'. Then $\alpha_w(G) = w(v) + \alpha_w(G')$ and a maximum weight independent set $\mathcal{I}$ of G can be constructed from a maximum weight independent set $\mathcal{I}'$ of G' as follows: if $\mathcal{I}' \cap N'(v) = \varnothing$ then $\mathcal{I} = \mathcal{I}' \cup \{v\}$. Otherwise, $\mathcal{I} = \mathcal{I}'$.

Proof. For (i), note that it is safe to remove all $u \in N(v)$ with $w(u) \leq w(v)$ since these vertices meet the criteria for the neighbor removal reduction. All vertices $x \in N'(v)$ that remain have weight $w(x) > w(v)$ and are not isolated.

Case 1 ($\mathcal{I}' \cap N'(v) = \varnothing$): Let $\mathcal{I}'$ be a maximum weight independent set of G', we show that if $\mathcal{I}' \cap N'(v) = \varnothing$ then $\mathcal{I} = \mathcal{I}' \cup \{v\}$. To prove this, we show that $w(v) + \alpha_w(G \setminus N[v]) \geq \alpha_w(G \setminus \{v\})$.

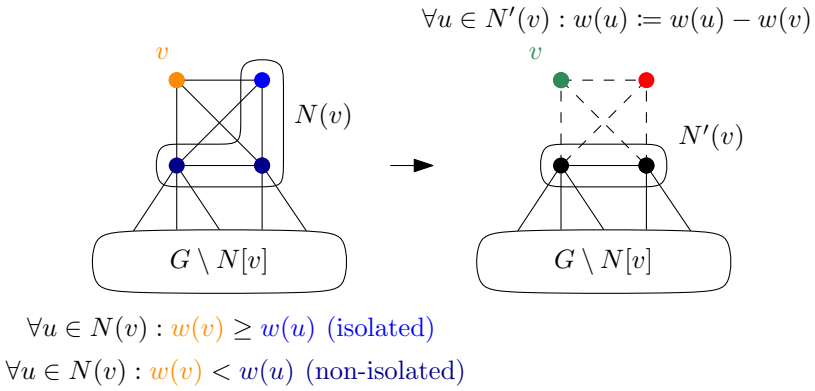


Figure 4.4: Example application of isolated weight transfer on vertex v (orange). The vertex v and its isolated neighbor can be removed from the graph. The vertex v is added to the independent set (and its isolated neighbor is excluded).

Let $x \in N'(v)$. Since $x \notin \mathcal{I}'$, it follows that

$$\begin{aligned} w(v) + \alpha_w(G') &= w(v) + \alpha_w(G'[V' \setminus N'(v)]) \\ &= w(v) + \alpha_w(G \setminus N[v]) \end{aligned}$$

and

$$\begin{aligned} w(v) + \alpha_w(G') &\geq w(v) + w'(x) + \alpha_w(G'[V' \setminus N[x]]) \\ &= w(v) + w(x) - w(v) + \alpha_w(G'[V' \setminus N[x]]) \\ &= w(x) + \alpha_w(G \setminus N[x]). \end{aligned}$$

Thus, for any $x \in N'(v)$, it holds that

$$\begin{aligned} w(v) + \alpha_w(G') &= w(v) + \alpha_w(G \setminus N[v]) \\ &\geq w(x) + \alpha_w(G \setminus N[x]) \end{aligned}$$

and, therefore, the heaviest independent set containing v is at least the weight of the heaviest independent containing *any* neighbor of v. Concluding, we have that

$$w(v) + \alpha_w(G \setminus N[v]) \geq \alpha_w(G \setminus \{v\})$$

and therefore $\mathcal{I} = \mathcal{I}' \cup \{v\}$ is a maximum weight independent set of G.

Case 2 ($\mathcal{I}' \cap N'(v) \neq \emptyset$): Let $\mathcal{I}'$ be a maximum weight independent set of G', we prove that if $\mathcal{I}' \cap N'(v) \neq \emptyset$ then $\mathcal{I} = \mathcal{I}'$. To show this, let $\{x\} = \mathcal{I}' \cap N'(v)$. Define G'' as the graph resulting from increasing the weight of $N'(v)$ by $w(v)$, i.e., $\forall\, u \in N'(v)$ we set $w''(u) = w'(u) + w(v) = w(u)$. We first show that $\mathcal{I}'' = \mathcal{I}'$ is a maximum weight independent set of G'. Therefore, assume that $\mathcal{I}^*$ is a maximum weight independent set

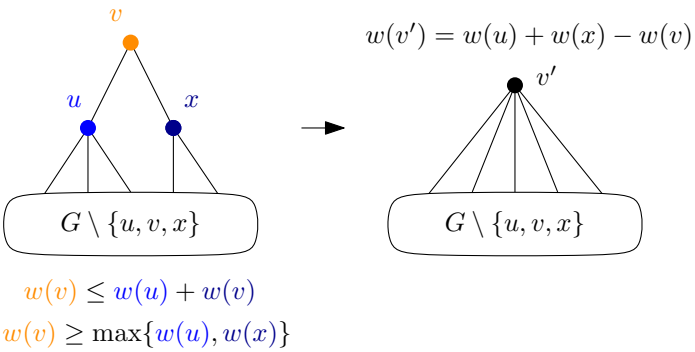


Figure 4.5: Example application of the weighted vertex fold reduction to vertex v (orange). Vertices v, u, and w are contracted to the new vertex v'.

of G'' with $w(\mathcal{I}^*) > w(\mathcal{I}')$ that does not contain x. However, then $\mathcal{I}^*$ is also a better maximum weight independent set on G' which contradicts our initial assumption. Finally, we have that $w(\mathcal{I}'') = w(\mathcal{I}') + w(v)$, since exactly one vertex in $N'(v)$ is in $\mathcal{I}'$.

Next, we define G''' as the graph resulting from adding back v to $G''$and show that $\mathcal{I}''' = \mathcal{I}''$ is a maximum weight independent set of G'''. For this purpose, we assume that $\mathcal{I}^*$ is a maximum weight independent set of G''' with $w(\mathcal{I}^*) > w(\mathcal{I}''')$. Then, $v \in \mathcal{I}^*$ since we only added this vertex to G''. Likewise, $x \notin \mathcal{I}^*$ since its a neighbor of v.

Since $w(\mathcal{I}^*) > w(\mathcal{I}''')$, we have that:

$$\begin{aligned} w(\mathcal{I}^* \setminus \{v\}) &= w(\mathcal{I}^*) - w(v) \\ &> w(\mathcal{I}'') - w(v) \\ &= w(\mathcal{I}') + w(v) - w(v) \\ &= w(\mathcal{I}'). \end{aligned}$$

However, since $\mathcal{I}^* \setminus \{v\}$ does neither include v nor any neighbor of v it is also an independent set of G' that is larger than $\mathcal{I}'$. This contradicts our initial assumption and thus $\mathcal{I}''' = \mathcal{I}'' = \mathcal{I}'$. Furthermore, since $G''' = G$, we have that $\mathcal{I}''' = \mathcal{I} = \mathcal{I}'$. □

Weighted Vertex Fold. Similar to the unweighted vertex fold reduction, we show that we can fold vertices with two non-adjacent neighbors—however, not all weight configurations permit this. An example application of this reduction rule is given in Figure 4.5.

Theorem 4.7 (Weighted Vertex Fold Reduction)

Let $v \in V$ have $d(v) = 2$, such that v's neighbors u, x are not adjacent. If $w(v) < w(u) + w(x)$ but $w(v) \geq \max\{w(u), w(x)\}$, then we fold v, u, x into vertex v' with weight $w(v') = w(u) + w(x) - w(v)$ forming a new graph G'. Then $\alpha_w(G) = \alpha_w(G') + w(v)$. Let $\mathcal{I}'$ be a maximum weight independent set of G'. If $v' \in \mathcal{I}'$, then $\mathcal{I} = (\mathcal{I}' \setminus \{v'\}) \cup \{u, x\}$ is a maximum weight independent set of G. Otherwise, $\mathcal{I} = \mathcal{I}' \cup \{v\}$ is a maximum weight independent set of G.

Proof. Apply neighborhood folding to v. □

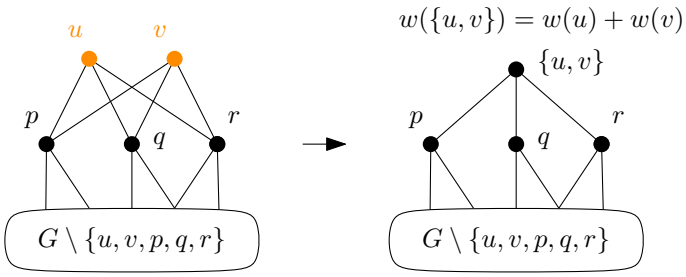


Figure 4.6: Illustrating the proof of the weighted twin reduction. The reduction is applied to vertices u and v (orange) which are contracted to vertex $\{u, v\}$.

Weighted Twin. The twin reduction, as described by Akiba and Iwata [AI16], works for twins with three common neighbors. We describe our variant in the same terms but note that the reduction supports an arbitrary number of common neighbors.

Theorem 4.8 (Weighted Twin Reduction)

Let vertices u and v have neighborhoods $N(u) = N(v) = \{p, q, r\}$ such that p, q, and r are not adjacent. We have two cases:

(i) If $w(\{u, v\}) \geq w(\{p, q, r\})$, then u and v are in some maximum weight independent set of G. Let $G' = G \setminus N[\{u, v\}]$.

(ii) If $w(\{u, v\}) < w(\{p, q, r\})$, but $w(\{u, v\}) > w(\{p, q, r\}) - \min_{x \in \{p,q,r\}} w(x)$, then we can fold u, v, p, q, r into a new vertex v' with weight $w(v') = w(\{p, q, r\}) - w(\{u, v\})$ and call this graph G'. Let $\mathcal{I}'$ be a maximum weight independent set of G'. Then we construct a maximum weight independent set $\mathcal{I}$ of G as follows: if $v' \in \mathcal{I}'$ then $\mathcal{I} = (\mathcal{I}' \setminus \{v'\}) \cup \{p, q, r\}$, if $v' \notin \mathcal{I}'$ then $\mathcal{I} = \mathcal{I}' \cup \{u, v\}$.

Furthermore, $\alpha_w(G) = \alpha_w(G') + w(\{u, v\})$.

Proof. Just as in the unweighted case, either u and v are simultaneously in a maximum weight independent set or some subset of p, q, r is. First, fold u and v into a new vertex $\{u, v\}$ with weight $w(\{u, v\})$. To show (i), apply the neighborhood removal reduction to vertex $\{u, v\}$. For (ii), since $N(\{u, v\})$ is independent, we apply the neighborhood folding reduction to $\{u, v\}$, giving the claimed result (see Figure 4.6).

Weighted Dominance. Lastly, we give a weighted variant of the dominance reduction (see Figure 4.7 for an example application of this rule).

Theorem 4.9 (Weighted Dominance Reduction)

Let $u, v \in V$ be vertices such that $N[u] \supseteq N[v]$ (i.e., u dominates v). If $w(u) \leq w(v)$, there is a maximum weight independent set in G that excludes u and $\alpha_w(G) = \alpha_w(G \setminus \{u\})$. Therefore, u can be removed from the graph.

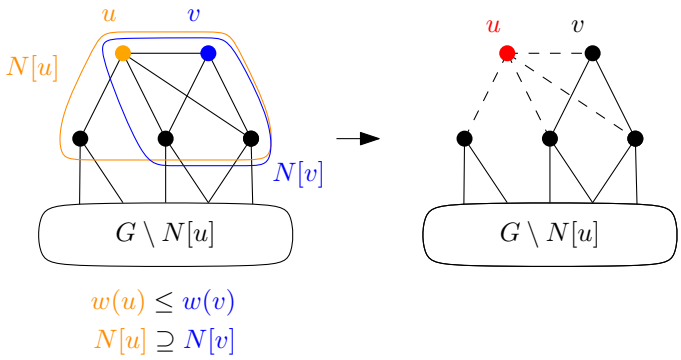


Figure 4.7: Example application of the weighted dominance reduction to vertex u (orange) and v (blue). Vertex u can be removed and excluded from the independent set since it is dominated by vertex v.

Proof. We show by a cut-and-paste argument that there is a maximum weight independent set of G excluding u. Let $\mathcal{I}$ be a maximum weight independent set of G. If u is not in $\mathcal{I}$, then we are done. Otherwise, suppose $u \in \mathcal{I}$. Then it must be the case that $w(v) = w(u)$. Otherwise, $\mathcal{I}' = (\mathcal{I} \setminus \{u\}) \cup \{v\}$ is an independent set with a weight larger than $\mathcal{I}$. Thus, $\mathcal{I}'$ is a maximum weight independent set of G excluding u, and $\alpha_w(G) = \alpha_w(G \setminus \{u\})$. □

4.2.4 Experimental Evaluation

We now compare the performance of our branch-and-reduce algorithm to existing state-of-the-art algorithms on large real-world graphs. We also examine how our reduction rules can drastically improve the quality of existing heuristic approaches.

Methodology. All of our experiments were run on Machine B described in Section 2.3.2.c). All algorithms were implemented in C++ and compiled with g++ version 4.8.4 using optimization flag `-O3`. Each algorithm was run sequentially for a total of 1000 seconds[1]. We present two kinds of data: (1) the best solution found by each algorithm and the time (in seconds) required to obtain it and (2) convergence plots (see Section 2.3.2.b)). The experiments for heuristic algorithms were performed with five different random seeds. Our set of instances includes a large set of real-world label conflict graphs and randomly weighted sparse networks also used by Akiba and Iwata [AI16]. Vertex weights are drawn uniformly at random from the interval $[1, 200]$. Basic properties of our benchmark instances can be found in Appendix A. We also present the reduced instance sizes in Appendix F. For the label conflict graphs, we use the same instances used in the experiments by Cai et al. [Cai+18]. We omit instances with less than 1000 vertices, as these are easy to solve, and our focus is on large-scale networks.

[1] Results with more than 1000 seconds are due to initial reductions taking longer than the time limit.

Algorithms Compared. We use two different variants of our branch-and-reduce algorithm. The first variant, called BnR_{full}, uses our *full set* of reductions each time we branch. The second variant, called $\text{BnR}_{\text{dense}}$, omits the more costly reductions and also terminates the execution of the remaining reductions faster than BnR_{full}. In particular, this configuration completely omits the weighted critical set reductions from both the initialization and recursion. Additionally, we also omit the weighted clique reduction from the first reduction call and use a faster version that only considers triangles during recursion. Finally, we do not use generalized neighborhood folding during recursion. This configuration finds solutions more quickly on dense graphs.

We also include the state-of-the-art heuristics HILS by Nogueria et al. [NPS18] and both versions of DynWVC by Cai et al. [Cai+18]. We do not include any exact algorithms besides our own (e.g., [BT07; WH06]) as their codes are not available. Also, note that these exact algorithms are either not tested in the weighted case [BT07] or the largest instances reported consist of a few hundred vertices [WH06].

To further evaluate the impact of reductions on existing algorithms, we also propose combinations of the heuristic approaches with reductions (Red + HILS and Red + DynWVC). We do so by first computing a reduced graph using our (full) set of reductions and then running the existing algorithms on the resulting graph.

4.2.4.a) Comparison with State-of-the-Art

A representative sample of our experimental results for the OSM and SNAP networks is presented in Table 4.1. For a full overview of all instances, we refer to Table 27 (OSM) and Table 28 (SNAP), respectively. For each instance, we list the best solution computed by each algorithm w_{Alg} and the time in seconds required to find it t_{Alg}. For each data set, we highlight the best solution found across all algorithms in bold. Additionally, if any version of our algorithm is able to find an exact solution, the corresponding row is highlighted in gray. Each table also includes the aggregated number of instances (including ones that are not part of the sample) our algorithm is able to solve. For those instances that were solved by us, we also include the number of instances on which the heuristic algorithms computed a solution of the same weight. Finally, recall that our algorithm computes a solution on unsolved instances once the time limit is reached by additionally running a greedy algorithm as post-processing.

Examining the OSM graphs, BnR is able to solve 15 out of the 34 instances we tested. However, HILS is also able to compute a solution with the same weight on all of these instance. Furthermore, HILS obtains a higher or similar quality solution than both versions of DynWVC and BnR for all remaining unsolved instances. Overall, HILS is able to find the best solution on all OSM instances that we tested. Additionally, on most of these instances, it does so significantly faster than all of its competitors. Note that neither HILS nor DynWVC provides any optimality guarantees (in contrast to BnR).

Looking at both versions of DynWVC, we see that DynWVC1 performs better than DynWVC2, which is also reported by Cai et al. [Cai+18]. Comparing both variants of our branch-and-reduce algorithm, we see that they are able to solve the same instances.

Table 4.1: Best solution found by each algorithm and the time (in seconds) required to compute it. The global best solution is highlighted in bold. Rows are highlighted in gray if BnR is able to find an exact solution.

Graph	$t_{\max}$	$w_{\max}$	$t_{\max}$	$w_{\max}$	$t_{\max}$	$w_{\max}$
OSM instances	DynWVC1		HILS		$\text{BnR}_{\text{dense}}$	
alabama-AM3	464.02	185 527	0.73	**185 744**	15.79	185 707
florida-AM2	1.14	**230 595**	0.04	**230 595**	0.03	**230 595**
georgia-AM3	0.88	**222 652**	0.05	**222 652**	4.88	214 918
kansas-AM3	46.87	**87 976**	0.84	**87 976**	11.35	87 925
maryland-AM3	1.34	**45 496**	0.02	**45 496**	3.34	**45 496**
massachusetts-AM3	435.31	145 863	2.73	**145 866**	12.87	145 617
utah-AM3	136.15	98 802	0.08	**98 847**	64.04	**98 847**
vermont-AM3	119.63	63 234	0.95	**63 302**	95.81	55 584
Solved instances						44.12% (15/34)
Optimal weight		60.00% (9/15)		100.00% (15/15)		
SNAP instances	DynWVC2		HILS		BnR_{full}	
as-skitter	576.93	123 105 765	998.75	122 539 706	746.93	**123 904 741**
ca-AstroPh	108.35	796 535	46.76	**796 556**	0.03	**796 556**
email-EuAll	179.26	**25 330 331**	501.09	**25 330 331**	0.19	**25 330 331**
p2p-Gnutella08	0.19	**435 893**	0.25	**435 893**	0.01	**435 893**
roadNet-TX	1 000.78	77 525 099	1 697.13	76 366 577	33.49	**78 606 965**
soc-LiveJournal1	1 001.23	277 824 322	12 437.50	280 559 036	270.96	**283 948 671**
web-Google	683.63	56 190 870	994.58	55 954 155	3.16	**56 313 384**
wiki-Talk	991.31	235 874 419	996.02	235 852 509	3.36	**235 875 181**
Solved instances						80.65% (25/31)
Optimal weight		28.00% (7/25)		68.00% (17/25)		

Nonetheless, $\text{BnR}_{\text{dense}}$ is able to compute better solutions on roughly half of the remaining instances. Additionally, it almost always requires significantly less time compared to BnR_{full}.

For the SNAP networks, we see that BnR solves 25 of the 31 instances we tested [2]. Most notable, on seven of these instances where either HILS or DynWVC1 also finds a solution with optimal weight, it does so up to two orders of magnitude faster. This difference in performance compared to the OSM networks can be explained by the significantly lower graph density and less uniform degree distribution of the SNAP networks. These structural differences seem to allow our reduction rules to be applicable more often, resulting in a significantly smaller reduced instance (as seen in Appendix F). This is similar to the behavior of unweighted branch-and-reduce [AI16]. Therefore, except for a single instance, our algorithm is able to find the best solution on *all* graphs tested.

[2] Using a longer time limit of 48 hours we are able to solve 27 our of 31 instances.

Table 4.2: Best solution found by each algorithm and the time (in seconds) required to compute it. $s_{\text{base}} = t_{\text{base}}/t_{\text{modified}}$ denotes the speedup between the modified and base versions of each local search. The global best solution is highlighted in bold. Rows are highlighted in gray if BnR is able to find an exact solution.

Graph	$t_{\max}$	$w_{\max}$	$t_{\max}$	$w_{\max}$	s_{base}	$t_{\max}$	$w_{\max}$
OSM instances	DynWVC1		Red+DynWVC1			$\text{BnR}_{\text{dense}}$	
alabama-AM3	464.02	185 527	370.80	185 727	1.25	15.79	185 707
florida-AM2	1.14	**230 595**	0.03	**230 595**	44.19	0.03	**230 595**
georgia-AM3	0.88	**222 652**	2.64	**222 652**	0.33	4.88	214 918
kansas-AM3	46.87	**87 976**	13.59	**87 976**	3.45	11.35	87 925
maryland-AM3	1.34	**45 496**	2.07	**45 496**	0.65	3.34	**45 496**
massachusetts-AM3	435.31	145 863	10.68	**145 866**	40.75	12.87	145 617
utah-AM3	136.15	98 802	168.07	**98 847**	0.81	64.04	**98 847**
vermont-AM3	119.63	63 234	62.85	63 280	1.90	95.81	55 584
Solved instances							44.12% (15/34)
Optimal weight		60.00% (9/15)		93.33% (14/15)			
SNAP instances	DynWVC2		Red+DynWVC2			$\text{BnR}_{\text{dense}}$	
as-skitter	576.93	123 105 765	85.60	123 995 808	6.74	746.93	123 904 741
ca-AstroPh	108.35	796 535	0.02	**796 556**	4 962.17	0.03	**796 556**
email-EuAll	179.26	**25 330 331**	0.12	**25 330 331**	1 548.08	0.19	**25 330 331**
p2p-Gnutella08	0.19	**435 893**	0.00	**435 893**	46.98	0.01	**435 893**
roadNet-TX	1 000.78	77 525 099	771.05	78 601 813	1.30	33.49	**78 606 965**
soc-LiveJournal1	1 001.23	277 824 322	996.68	283 973 997	1.00	270.96	283 948 671
web-Google	683.63	56 190 870	3.30	56 313 349	207.26	3.16	**56 313 384**
wiki-Talk	991.31	235 874 419	2.30	**235 875 181**	430.22	3.36	**235 875 181**
Solved instances							80.65% (25/31)
Optimal weight		28.00% (7/25)		84.00% (21/25)			

Comparing the heuristic approaches, both versions of DynWVC perform better than HILS on most instances, with DynWVC2 often finding better solutions than DynWVC1. Nonetheless, HILS finds higher weight solutions than DynWVC1 and DynWVC2.

4.2.4.b) The Power of Weighted Reductions

We now examine the effect of using reductions to improve existing heuristic algorithms. For this purpose, we compare the combined approaches Red + HILS and Red + DynWVC with their base versions and our branch-and-reduce algorithm. Our sample of results for the OSM and SNAP networks are given in Tables 4.2 and 4.3, which show the results for DynWVC and HILS, respectively. In addition to the data used in our state-of-the-art comparison, we now also report speedups between the modified and base versions of each local search.

Table 4.3: Best solution found by each algorithm and the time (in seconds) required to compute it. $s_{\text{base}} = t_{\text{base}}/t_{\text{modified}}$ denotes the speedup between the modified and base versions of each local search. The global best solution is highlighted in bold. Rows are highlighted in gray if BnR is able to find an exact solution.

Graph	$t_{\max}$	$w_{\max}$	$t_{\max}$	$w_{\max}$	s_{base}	$t_{\max}$	$w_{\max}$
OSM instances	HILS		Red + HILS			$\text{BnR}_{\text{dense}}$	
alabama-AM3	0.73	**185 744**	4.05	**185 744**	0.18	15.79	185 707
florida-AM2	0.04	**230 595**	0.03	**230 595**	1.75	0.03	**230 595**
georgia-AM3	0.05	**222 652**	2.43	**222 652**	0.02	4.88	214 918
kansas-AM3	0.84	**87 976**	2.06	**87 976**	0.41	11.35	87 925
maryland-AM3	0.02	**45 496**	2.07	**45 496**	0.01	3.34	**45 496**
massachusetts-AM3	2.73	**145 866**	2.92	**145 866**	0.93	12.87	145 617
utah-AM3	0.08	**98 847**	2.10	**98 847**	0.04	64.04	**98 847**
vermont-AM3	0.95	63 302	2.95	**63 312**	0.32	95.81	55 584
Solved instances							44.12% (15/34)
Optimal weight	100.00% (15/15)		100.00% (15/15)				
SNAP instances	HILS		Red + HILS			$\text{BnR}_{\text{dense}}$	
as-skitter	998.75	122 539 706	845.70	**123 996 322**	1.18	746.93	123 904 741
ca-AstroPh	46.76	**796 556**	0.02	**796 556**	2 142.48	0.03	**796 556**
email-EuAll	501.09	**25 330 331**	0.12	**25 330 331**	4 327.82	0.19	**25 330 331**
p2p-Gnutella08	0.25	**435 893**	0.00	**435 893**	63.80	0.01	**435 893**
roadNet-TX	1 697.13	76 366 577	946.32	78 602 984	1.79	33.49	**78 606 965**
soc-LiveJournal1	12 437.50	280 559 036	761.51	**283 975 036**	16.33	270.96	283 948 671
web-Google	994.58	55 954 155	3.01	**56 313 384**	330.28	3.16	**56 313 384**
wiki-Talk	996.02	235 852 509	2.30	**235 875 181**	432.26	3.36	**235 875 181**
Solved instances							80.65% (25/31)
Optimal weight	68.00% (17/25)		88.00% (22/25)				

Additionally, we give the percentage of instances solved by BnR and the percentage of solutions with optimal weight found by the heuristic algorithms compared to BnR. For a full overview of all instances, we refer to Appendix G.

When looking at the speedups for the SNAP graphs, we can see that using reductions allows local search to find optimal solutions orders of magnitude faster. Additionally, they are now able to find an optimal solution more often than without reductions. In particular, DynWVC2 achieves an increase of 56% of optimal solutions when using reductions. Overall, we achieve a speedup of up to three orders of magnitude for the SNAP instances. Thus, the additional costs for computing the reduced graph can be neglected for these instances. However, for the OSM instances, our reduction rules are less applicable, and computing the reduced graph comes at a significant cost compared to the unmodified local searches.

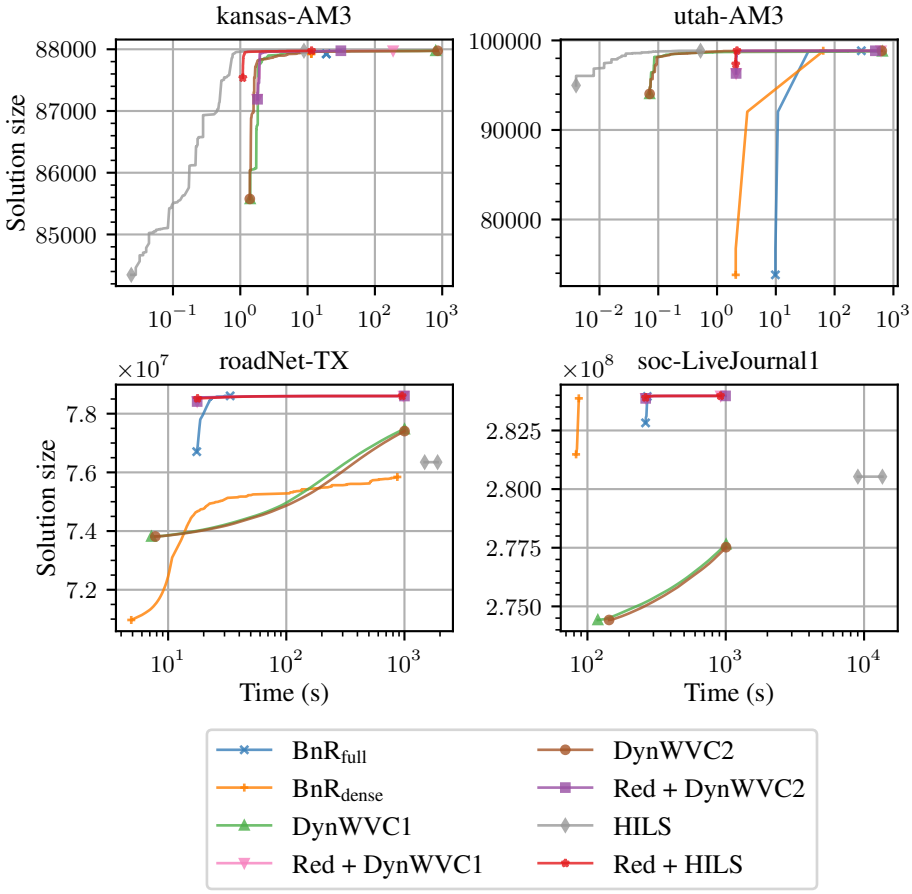


Figure 4.8: Solution quality over time for two OSM instances (top) and two SNAP instances (bottom).

To further examine the influence of using reductions, Figure 4.8 shows the solution quality over time for all algorithms and four instances. For additional convergence plots, we refer to Appendix H. For the OSM instances, we can see that initially DynWVC and HILS are able to find good quality solutions much faster compared to their combined approaches. However, once the reduced graph has been computed, regular DynWVC and HILS are quickly outperformed by the hybrid algorithms.

A more drastic change can be seen for the SNAP instances. Instances where both DynWVC and HILS examine poor performance, Red + DynWVC and Red + HILS now rival our branch-and-reduce algorithm and give near-optimal solutions in less time. Thus, using

reductions for instances that are too large for traditional heuristic approaches allows for a drastic improvement.

4.3 Increasing Transformations

As seen in the previous section, reduction rules are often able to calculate very small irreducible graphs on a wide number of (weighted) instances. Thus, branch-and-reduce algorithms are able to solve many instances up to two orders of magnitude faster than existing local search algorithms. However, these algorithms still fail to compute reduced graphs that are small enough on some instances [Lam+19]. This is mainly due to the specialized nature of these reduction rules that search for very specific subgraphs that can be removed.

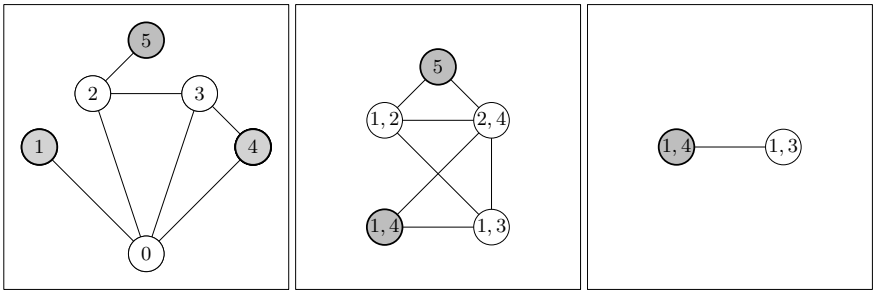
In this chapter, we introduce new generalized reduction and transformation rules to address this issue. Unlike previous narrowly-defined reduction rules, we engineer transformation rules that can also *increase* the size of the input. Surprisingly, this can simplify the problem and open up the reduction space to yield even smaller irreducible graphs later throughout the algorithm's execution [Ale+03].

More precisely, we engineer practically efficient variants of the stability number reduction rule (called struction). To the best of our knowledge, these are the first practical implementations of the weighted struction rule that are able to handle a large variety of real-world instances. Our algorithm exploits the full potential of the struction rule by also allowing the application of structions that may increase the number of vertices. These new rules are integrated into the state-of-the-art branch-and-reduce algorithm by Lamm et al. [Lam+19] presented in the previous section – with and without the property that a transformation rule can increase the size of the input. Extensive experiments indicate that our algorithm calculates significantly smaller irreducible graphs than current state-of-the-art approaches, and preprocessing with our transformations enables branch-and-reduce to solve many instances that were previously infeasible to solve to optimality.

Organization. We begin this section with an overview of the work related to the struction by Ebenegger et al. [EHd84] in Section 4.3.1. Our practically efficient struction variants are introduced in Section 4.3.2. Section 4.3.3 then shows how these variants can be used to build an efficient preprocessing algorithm that can be integrated into state-of-the-art branch-and-reduce algorithms. Finally, we close this section with an extensive experimental evaluation in Section 4.3.4.

4.3.1 Struction

Originally the struction (STability number RedUCTION) was introduced by Ebenegger et al. [EHd84] and was later improved by Alexe et al. [Ale+03]. This method is a graph transformation for unweighted graphs that can be applied to an arbitrary vertex and reduces the stability number by exactly one. Thus, by successive application of the struction, the independence number of a graph can be determined. To be more specific, when applying a struction to a vertex, the vertex and its neighborhood are removed from the graph, and the independence number of the graph is reduced by one. However, new vertices and edges



(a) Original Graph (b) After first struction (c) After second struction

Figure 4.9: Example application of unweighted structions to reduce a graph. The first struction is applied to vertex "0", resulting in the removal of its closed neighborhood and the insertion of vertices "1,2", "1,3", "1,4", and "2,4". The second and third struction are applied to "5" and "1,4", removing their closed neighborhoods and creating no vertices. Vertices included in the solution when undoing the structions are highlighted in gray.

might be inserted, leading to a potential increase in the graph size. An example application of the unweighted struction is given in Figure 4.9. Ebenegger et al. also show that there is an equivalence between finding a maximum weight independent set and maximizing a pseudo Boolean function, i.e., a real-valued function with Boolean variables, which allows deriving the struction as a special case. Finally, the authors present a generalization of the struction to weighted graphs, which will be discussed later.

On this basis, theoretical algorithms with polynomial time complexity for special graph classes have been developed [HMdW85a; HMdW85b; Ale+03]. These algorithms use a set of additional reduction rules and a careful selection of vertices on which the struction can be applied.

Hoke and Troyon [HT94] developed another form of the weighted struction, using the same equivalence found by Ebenegger et al. [EHd84]. In particular, they derive the *revised weighted struction*. However, this type of struction can only be applied to claw-free graphs, i.e., graphs without an induced three-leaf star. This transformation also removes a vertex v and its neighborhood but is able to create fewer new vertices since these are only created for pairs of non-adjacent neighbors whose combined weight is greater than the weight of v.

As far as we are aware, prior to this work, only a few experiments with struction variants existed and were limited to only small instances: Ebenegger et al. [EHd84] and Alexe et al. [Ale+03] evaluated the struction only on small graphs with less than a hundred vertices for the unweighted case. Furthermore, for the weighted case, none of the previously proposed struction variants has been evaluated so far [EHd84; HT94; Ale+03].

Original Weighted Struction. We now present the original weighted struction introduced by Ebenegger et al. [EHd84], on which we base our struction variants. In general, we apply a struction to a *center vertex* v and denote its neighborhood by $N(v) = \{1, 2, ..., p\}$. All variants we use remove v from the graph G, producing a new graph G', and reduce the weighted independence number of the graph G by its weight, i.e., $\alpha_w(G) = \alpha_w(G') + w(v)$.

For ease of presentation, we first introduce a method called *layering*. Layering describes the partitioning of a given set M that contains vertices $v_{x,y}$, that are indexed by two parameters $x \in X, y \in Y$. The sets X, Y either contain vertices or vertex sets. For $k \in X$, a layer L_k contains all vertices having k as the first parameter, i.e., $L_k = \{v_{x,y} \in M : x = k\}$. Conversely, the *layer* of a vertex $v_{x,y}$ is $L(v_{x,y}) = k = x$.

In order to apply the original struction by Ebenegger et al. [EHd84], the center vertex v must have minimum weight among its closed neighborhood. The struction is then applied by removing v and creating new vertices for each pair i, j of non-adjacent vertices in $N(v)$. To guarantee that we can obtain a maximum weight independent set $\mathcal{I}$ of G using a maximum weight independent set $\mathcal{I}'$ of G' with $w(\mathcal{I}) = w(\mathcal{I}') + w(v)$, we also insert edges between the new and original vertices. We now provide a formal definition of the original struction.

Theorem 4.10 (Original Struction)

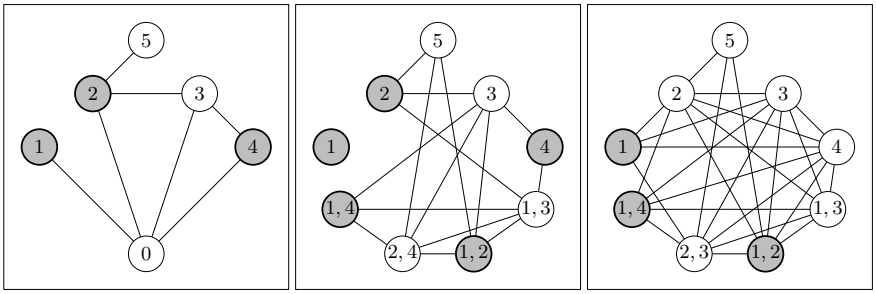
Let $v \in V$ be a vertex with minimum weight $w(v)$ among its closed neighborhood. Transform the graph as follows:

- *Remove v, lower weight of each neighbor by $w(v)$.*
- *For each pair of non-adjacent neighbors $x < y$, create a new vertex $v_{x,y}$ with weight $w(v_{x,y}) := w(v)$.*
- *Insert edges between $v_{q,x}, v_{r,y}$ if either x and y are adjacent or they belong to different layers, i.e., $q \neq r$.*
- *Each vertex $v_{x,y}$ is also connected to vertex $w \in V \setminus \{v\}$ adjacent to either x or y.*

For a maximum weight independent set $\mathcal{I}'$ of G', we obtain a maximum weight independent set $\mathcal{I}$ of G as follows: If $\mathcal{I}' \cap N(v) = \varnothing$ applies, we have $\mathcal{I} = \mathcal{I}' \cup \{v\}$. Otherwise, we remove the new vertices, i.e., $\mathcal{I} = \mathcal{I}' \cap V$. Furthermore, we have $\alpha_w(G) = \alpha_w(G') + w(v)$. A proof that the original struction is a valid reduction can be found in the paper by Ebenegger et al. [EHd84]. An example application of the original struction is given in Figure 4.10 (b), where the center vertex is "0", $L_1 = \{$"1,2", "1,3", "1,4"$\}$ and $L_2 = \{$"2,4"$\}$.

4.3.2 New Weighted Struction Variants

We now introduce three new struction variants: First, we deal with the fact that using the original weighted struction, a maximum weight independent set in the transformed graph might consist of more vertices than in the original graph. We do so by using different weight assignments for the new vertices and inserting additional edges. Second, we present a generalization of the revised weighted struction that can be applied to vertices of general graphs without the need to fulfill specific weight constraints. However, this variant creates



(a) Original Graph (b) Original Struction (c) Modified Struction

Figure 4.10: Example application of the original struction and modified struction on vertex $v = 0$. Vertices representing the same independent set in the different graphs are highlighted in gray.

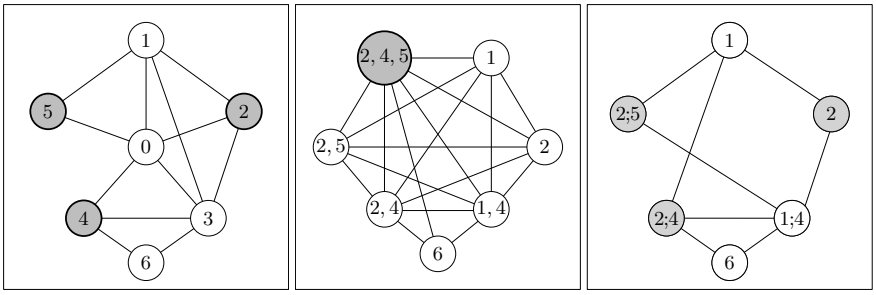
new vertices for independent sets in the neighborhood of a vertex v whose weight is greater than $w(v)$. Finally, we alleviate this issue by creating new vertices only for a specific subset of these independent sets in the third variant.

Modified Weighted Struction. One caveat of the original struction is that the *number of vertices* that are part of a maximum weight independent set in the transformed graph is generally larger than in the original graph. The modified struction tries to alleviate this issue by ensuring that the number of vertices of a maximum weight independent set stays the same in both graphs. This is done by using a different weight assignment and inserting additional edges. In particular, the newly created vertex for each pair of non-adjacent neighbors $x, y \in N(v)$ with $x < y$ is now assigned weight $w(v_{x,y}) = w(y)$ (instead of $w(v)$). Furthermore, in addition to the edges created in the original struction, each neighbor $k \in N(v)$ is connected to each vertex $v_{x,y}$ belonging to a different layer than k. Finally, $N(v)$ is extended to a clique by adding edges between vertices $x, y \in N(v)$. For a maximum weight independent set $\mathcal{I}'$ of G', we now obtain a maximum weight independent set $\mathcal{I}$ of G as follows: If $\mathcal{I}' \cap N(v) = \varnothing$ applies, we have $\mathcal{I} = \mathcal{I}' \cup \{v\}$. Otherwise, we obtain $\mathcal{I}$ by replacing each new vertex $v_{x,y} \in \mathcal{I}'$ with the original vertex v_y, i.e., $\mathcal{I} = (\mathcal{I}' \cap V) \cup \{v_y \mid v_{x,y} \in \mathcal{I}' \setminus V\}$. As for the original struction, we have $\alpha_w(G) = \alpha_w(G') + w(v)$. An example application of the modified struction is given in Figure 4.10 (c), where the center vertex is "0", $L_1 = \{$"1,2", "1,3", "1,4"$\}$ and $L_2 = \{$"2,4"$\}$.

Lemma 4.11 (Modified Weighted Struction, Independence Number)

After using the modified weighted struction, the equality $\alpha_w(G) = \alpha_w(G') + w(v)$ holds.

Extended Weighted Struction. The extended struction removes the weight restriction for the vertex v in the former variants. Unlike the previous two structions, this variant considers independent sets of arbitrary size in the neighborhood $N(v)$. In fact, we create new vertices for each independent set in $G[N(v)]$ if its weight is greater than v. Note that this can result in up to $\mathcal{O}\left(2^{d(v)}\right)$ new vertices.



(a) Original Graph (b) Extended Struction (c) Extended Reduced Struction

Figure 4.11: Application of extended struction and extended reduced struction on center vertex $v = 0$. Vertices representing the same independent set in the different graphs are highlighted in gray. We assume some weight constraints in the original graph for the construction in (b) and (c): $w(1) > w(0)$, $w(2) > w(0)$, and $w(3) + w(4) + w(5) \leq w(0)$.

Theorem 4.12 (Extended Weighted Struction)

Let $v \in V$ be an arbitrary vertex and $\mathcal{C}$ the set of all independent sets c in $G[N(v)]$ with $w(c) > w(v)$. We derive the transformed graph G' as follows: First, remove v together with its neighborhood and create a new vertex v_c with weight $w(v_c) = w(c) - w(v)$ for each independent set $c \in \mathcal{C}$. Each vertex v_c is then connected to each non-neighbor $w \in \overline{N}(v)$ adjacent to at least one vertex in c. Finally, the vertices v_c are connected with each other, forming a clique. For a maximum weight independent set $\mathcal{I}'$ of G' we obtain a maximum weight independent set $\mathcal{I}$ of G as follows: If $\mathcal{I}' \setminus V = \{v_c\}$ replace v_c with the vertices of the corresponding independent set c, i.e., $\mathcal{I} = (\mathcal{I}' \cap V) \cup c$. Otherwise, $\mathcal{I} = \mathcal{I}' \cup \{v\}$. Furthermore, we have $\alpha_w(G) = \alpha_w(G') + w(v)$.

An example application of the extended struction can be found in Figure 4.11 (b), where the center vertex is "0", and the set of independent sets (i.e., the set of new vertices) is $C = \{$"1", "1,4", "2", "2,4", "2,4,5", "2,5"$\}$.

Lemma 4.13 (Extended Weighted Struction, Independence Number)

After using the extended weighted struction, $\alpha_w(G) = \alpha_w(G') + w(v)$ holds.

Extended Reduced Weighted Struction. The extended reduced struction is a variant of the extended struction, which can potentially reduce the number of newly created vertices. For this purpose, only independent sets with weight "just" greater than $w(v)$ are considered. Let $\mathcal{C}$ be the set of all independent sets in $G[N(v)]$. We define a subset of independent sets $C \subseteq \mathcal{C}$ as follows. For an arbitrary but fixed ordering of the vertices, C consists of independent sets of weight greater than $w(v)$ that have no prefix set (according to the ordering) with weight greater than $w(v)$. That is, $C = \{c \in \mathcal{C} \mid w(c) - w(M(c)) \leq w(v)\}$ where $M(c)$ is the vertex of $N(v)$ of the highest index in the ordering.

We then use the same construction as for the extended struction but only create new vertices for the set C. The resulting set of vertices is denoted by V_C. However, since this construction might not be valid anymore, we also add additional vertices that are connected to each other by using layering. To be more specific, for each pair of an independent set $c \in C$ and a vertex $y \in N(v)$, we create a vertex $v_{c,y}$ with weight $w(v_{c,y}) = w(y)$, if c can be extended by y, i.e., y is not adjacent to any vertex $v' \in c$. We denote this set of vertices $v_{c,y}$ by V_E. We then insert edges between two vertices $v_{c,y}, v_{c',y'}$ if they either belong to different layers or y and y' have been adjacent. Moreover, each vertex $v_{c,y}$ is connected to each non-neighbor $w \in \overline{N}(v)$ if w has been connected to either y or a vertex $x \in c$. Finally, we connect each vertex v_c to each vertex $v_{c',y}$ belonging to a different layer than c. For a maximum weight independent set $\mathcal{I}'$ of G', we obtain a maximum weight independent set $\mathcal{I}$ of G as follows: If $\mathcal{I}' \cap V_C = \varnothing$ applies, we set $\mathcal{I} = \mathcal{I}' \cup \{v\}$. Otherwise, there is a single vertex $v_c \in \mathcal{I}' \cap V_C$ that we replace with the vertices of its independent set c. Moreover, we replace each vertex $v_{c,y} \in \mathcal{I}' \cap V_E$ with the vertex v_y. Altogether we have $\mathcal{I} = (\mathcal{I}' \cap V) \cup c \cup \{v_y \mid v_{c,y} \in \mathcal{I}' \cap V_E\}$ and $\alpha_w(G) = \alpha_w(G') + w(v)$. An example application of the extended struction can be found in Figure 4.11 (c), where "0" is the center vertex, $V_C = \{$"1", "2"$\}$, $L_1 = \{$"1;4"$\}$ and $L_2 = \{$"2;4", "2;5"$\}$.

Lemma 4.14 (Extended Reduced Weighted Struction, Independence Number)
After using the extended reduced weighted struction, $\alpha_w(G) = \alpha_w(G') + w(v)$ holds.

4.3.3 Practically Efficient Structions

We now propose our two novel preprocessing algorithms for MWIS based on the struction variants presented in the previous section. Additionally, we present how we integrate the different structions into the algorithm of Lamm et al. [Lam+19] presented in Section 4.2, both as a preprocessing step and as a reduce step during the algorithm's execution, to compute exact solutions more quickly. This takes an initial step towards a generalized *branch-and-transform* framework.

Since the different forms of the weighted struction do not necessarily reduce the number of vertices, we divide them (and the existing reduction rules) into three different classes that are used throughout this section. For *decreasing transformations* (reductions), the transformed graph G' has fewer vertices than the original graph G. Note that all reduction rules presented in Section 4.2.3 belong to this class. We also derive special cases of the structions which belong to this type. Transformations where the number of vertices in the original graph stays the same but that reduce the size and weight of MWIS in the transformed graph are called *plateau transformations*. While plateau transformations cannot reduce the size of the graph, they can potentially produce new subgraphs, which can then be reduced by other (decreasing) transformations. Finally, *increasing transformations* are transformations whose resulting graph has more vertices than the original graph. Similar to plateau transformations, the idea is to reduce the resulting graph by further reduction rules and transformations. However, since increasing structions can lead to a larger transformed graph, it is difficult to integrate them into algorithms that only apply non-increasing transformations.

4.3.3.a) Non-Increasing Reduction Algorithm

In this section, we show how to obtain decreasing and plateau transformations from the four different forms of structions presented in Section 4.3.1. Based on these, an incremental preprocessing algorithm is proposed.

In general, when applying any form of struction, the number of vertices of the transformed graph G' depends on the number of removed and newly created vertices. However, it is difficult to estimate the number of resulting vertices in advance since it varies depending on the number of possible independent sets present in the neighborhood of the center vertex. Thus, when applying a struction variant, we generally keep track of the number of vertices that will be created. If this number exceeds a given maximum value n_{max}, we discard the corresponding struction to ensure that not too many vertices are created.

We begin by taking a closer look at the structurally similar original weighted struction and modified weighted struction. These variants reduce the number of vertices by at most one since they only remove the center vertex v. Therefore, decreasing or plateau structions can be obtained by setting $n_{max} = 0$ or $n_{max} = 1$. However, note that this type of decreasing struction is already covered by the isolated weight transfer presented in Section 4.2.3.

Looking at the two remaining struction variants, we see that they not only remove the center vertex v but also its neighborhood $N(v)$. Thus, the size of the graph can be reduced by up to $d(v) + 1$ vertices. Decreasing or plateau structions can be obtained by using the corresponding struction variant with $n_{max} = d(v)$ or $n_{max} = d(v) + 1$.

The resulting rules can then easily be integrated into the reduction algorithm of the previous branch-and-reduce algorithm. However, since all struction variants are very general reduction rules, they tend to be expensive in terms of running time. Therefore, we apply them after the faster localized reduction rules but before the even more expensive critical set reduction. To be more specific, we use the following reduction rule order: neighborhood removal, vertex fold, isolated clique, dominance, twin, isolated weight transfer, neighborhood folding, decreasing struction, plateau struction, and critical weighted independent set.

4.3.3.b) Cyclic Blow-Up Algorithm

Next, we extend the previous (non-increasing) algorithm to also make use of increasing structions. The main idea is to alternate between computing an irreducible graph using the previous algorithm and then applying increasing structions while ensuring that the graph size does not increase too much. The reasoning for this is that even though the graph size might increase, this can generate new and potentially reducible subgraphs, thus leading to an overall decrease in the graph size.

In the following, we say that a graph $\mathcal{K}$ is better than a graph $\mathcal{K}'$ if it has fewer vertices. However, our algorithm can easily be adapted to match other quality criteria. Pseudocode for our algorithm is given in Algorithm 4.2.

Our algorithm maintains two graphs $\mathcal{K}$ and $\mathcal{K}^\star$. $\mathcal{K}^\star$ is the best graph found so far, i.e., the graph with the least number of vertices. $\mathcal{K}$ is the current graph, which we try to reduce to get a better graph $\mathcal{K}^\star$. Both graphs, $\mathcal{K}$ and $\mathcal{K}^\star$ are initialized with the graph obtained by applying the non-increasing reduction algorithm of the previous section. The algorithm

Algorithm 4.2 : Cyclic blow-up algorithm.

Data : $G = (V, E)$, unsuccessful iteration threshold $X \in [1, \infty)$, maximum blowup $\alpha \in [1, \infty)$
Result : smallest reduced graph $\mathcal{K}^\star$

```
1  CyclicBlowUp (G, X, α)
2      K ← Reduce(G)
3      K* ← K
4      count ← 0
5      while n(K) ≤ α · n(K*) and count < X do
6          K' ← BlowUp(K)
7          if K' = K then
8              return K*
9          K'' ← Reduce(K')
10         K ← Accept(K'', K)
11         if n(K) < n(K*) then
12             K* ← K
13         else
14             count ← count+1
15     return K*
```

then alternates between two phases, a *blow-up phase* and a *reduction phase*. During the blow-up phase, a set of increasing structions is applied to $\mathcal{K}$, resulting in a new graph $\mathcal{K}'$. $\mathcal{K}'$ is then reduced using the non-increasing algorithm, resulting in a graph $\mathcal{K}''$. Next, we have to decide whether to use $\mathcal{K}''$ or $\mathcal{K}$ for the next iteration. Note that it can be advantageous to accept a graph $\mathcal{K}''$ even if it has more vertices than $\mathcal{K}$ to avoid local minima. Nonetheless, we decided only to keep $\mathcal{K}''$ if it has fewer vertices than $\mathcal{K}$, as this strategy provided better results during preliminary experiments. Finally, since we might not completely reduce the graph, we use a termination criterion, which will be discussed later.

Blow-Up Phase. The starting point of the blow-up phase is an irreducible graph, where no more reductions (including decreasing structions) can be applied. Next, we select a vertex v from a candidate set C. This candidate set consists of all vertices in the current graph which have not been explicitly excluded from selection during the algorithm's computation. Vertex selection is a crucial part of our algorithm. Depending on the selected vertex, the struction might create a large number of new vertices and the size of the transformed graph can increase drastically. Thus, we will discuss possible selection strategies later.

Next, we apply a struction to the selected vertex v. As for our previous algorithm, we keep track of the number of newly created and deleted vertices during this step. In particular, if the struction would result in more than n_{max} vertices, it is aborted. In this case, the vertex v is excluded from the candidate set. The vertex v will become viable again as soon

as the corresponding struction would create a different transformed graph, i.e., when its neighborhood $N(v)$ changed.

After having applied a struction, we then proceed with the subsequent reduction phase. It might also be possible to apply more than one struction during a blow-up phase. However, one has to be careful not to let the size of the graph grow too large.

Vertex Selection Strategies. The goal of the vertex selection procedure is to find an increasing struction that results in a new graph, which can then be reduced to an overall smaller graph. In general, it is very difficult to estimate in advance to what extent the transformed graph can be reduced without actually performing the reduction phase. Therefore, most of the following strategies aim to increase the size of the graph by only a few vertices. The number of newly created vertices is determined by the number of independent sets in the neighborhood of v having a total weight greater than the weight of v. In general, determining this number is NP-hard [PV06] and thus often infeasible to compute in practice. In contrast, a much simpler selection strategy would be to choose vertices uniformly at random. However, this can lead to structions that significantly increase the graph size.

Thus, in order to limit the size increase of a struction, we decided to use an approximation of the exact number of independent sets in the neighborhood of v. In particular, we only consider independent sets up to a size of two. This results in a lower bound L for the number of independent sets [PV06], which can be computed in $\mathcal{O}(\Delta^2)$ time. Since the lower bound L can be far smaller than the actual number of newly created vertices, we use an additional *tightness check*: This check is passed if less than $L' = \lceil \beta \cdot L \rceil$ new vertices with $\beta \in (1, \infty)$ are created by the corresponding struction. Our strategy then works as follows: We select a vertex v with a minimal increase and perform the tightness check. If it fails, we know that at least L' new vertices are created by the corresponding struction. Therefore, L' forms a tighter bound for the number of new vertices, and we reinsert v to C using the bound L'. We then repeat this process until we find a vertex that passes the tightness check. Overall, this allows us to discard structions that create a large number of vertices earlier than if we directly computed the exact number of independent sets in the neighborhood of a vertex v.

Termination Criteria. In general, the size of $\mathcal{K}$ can decrease very slowly or even exhibit oscillatory behavior. This can cause the algorithm to take a long time to improve $\mathcal{K}^\star$ or even not improve it at all. For this purpose, one needs an appropriate termination criterion. First, we want to avoid that the size of the current graph $\mathcal{K}$ distances too much from that of the best graph $\mathcal{K}^\star$. Therefore, we abort the algorithm as soon as the size of the current graph exceeds the size of the best graph by a factor $\alpha \in [1, \infty)$, that is if $n(\mathcal{K}) > \alpha \cdot n(\mathcal{K}^\star)$. Additionally, we also count the number of unsuccessful iterations, i.e., iterations in which the new graph has been rejected. Our second criterion aborts the algorithm if this value exceeds some constant $X \in [1, \infty)$.

4.3.4 Experimental Evaluation

We now evaluate the impact and performance of our preprocessing algorithms. First, we compare the performance of our algorithms with the two configurations used for the branch-and-reduce algorithm presented in Section 4.2. For this purpose, we examine the sizes of

the reduced graphs, the number of instances solved, as well as the time required to do so. Second, we perform a broader comparison with other state-of-the-art algorithms, including heuristic approaches. We do so to highlight the performance our exact approaches are able to achieve even when compared to heuristics that are unable to prove the optimality of their computed solutions.

Methodology. We ran all the experiments on Machine B described in Section 2.3.2.c). All algorithms were implemented in C++ and compiled with g++ version 7.5.0 using optimization flag `-O3`. All algorithms were executed sequentially with a time limit of 1000 seconds. The experiments for heuristic algorithms were performed with five different random seeds. We present maximum values and cactus plots as described in Section 2.3.2.b). Our set of instances includes all previously introduced real-world label conflict graphs and randomly weighted sparse networks (see Section 4.2.4). Additionally, we extend this set of benchmark instances by also considering the mesh, finite element, and maximum weight clique instances presented in Section 2.3.2.a). We do so to get a clearer picture of the applicability of weighted reductions to different graph classes. Note that the complements of the maximum weight clique instances are only somewhat sparse—and most are irreducible by our techniques. This behavior has already been observed by Akiba and Iwata [AI16] on similar instances. Therefore, we will omit these instances from our experiments. An overview of all instances considered is given in Appendix A. Finally, in addition to the cactus plots presented in this section, we also present convergence plots in Appendix K.

Algorithm Configuration. For our evaluation, we use both the non-increasing algorithm and the cyclic blow-up algorithm. In particular, we use two different configurations of the cyclic blow-up algorithm: The first configuration, called C_{strong}, aims to achieve small reduced graphs. For this purpose, we set the number of unsuccessful blow-up phases to $X = 64$, the number of vertices that a struction is allowed to create to $n_{max} = 2048$, and the maximum struction degree (the degree up to which we can apply structions) $d_{max} = 512$. In our preliminary experiments, this configuration was always able to compute the smallest reduced graphs. The second configuration, called C_{fast}, aims to achieve a good tradeoff between the reduced graph size and the time required to compute an exact solution. Thus we set $X = 25$, $n_{max} = 512$ and $d_{max} = 64$. Finally, all our algorithms use $\beta = 2$ for the tightness-check during vertex selection, as well as the extended weighted struction, as this struction variant achieved the best performance during preliminary experiments.

To measure the impact of our preprocessing methods on existing approaches, we add each configuration to the branch-and-reduce algorithm. This results in three algorithms, which we call Cyclic-Fast, Cyclic-Strong, and NonIncreasing in the following. Note that each algorithm uses the corresponding configuration only for its initial preprocessing, whereas subsequent graph reductions only use decreasing transformations. Finally, we have replaced the ARW local search used in the original algorithm to compute lower bounds with the hybrid iterated local search (HILS) of Nogueira et al. [NPS18]. This resulted in slightly better runtime during preliminary experiments but had no impact on the number of instances that were solved.

We begin by comparing our three algorithms with the two configurations, called BnR_{full} and $\text{BnR}_{\text{dense}}$, of the branch-and-reduce algorithm presented in Section 4.2. Our comparison

is divided into two parts: First, we consider the sizes of the irreducible graphs after the initial reduction phase. Second, we compare the number of solved instances and the time required to solve them. A complete overview of the reduced graph sizes and running times for each algorithm is given in Appendix I. As for the previous section, tables present a representative sample of our experimental results. For a full overview of all instances, we refer to Table 27 (OSM) and Table 28 (SNAP), respectively. For each instance, we list the best solution computed by each algorithm w_{Alg} and the time in seconds required to find it t_{Alg}. For each data set, we highlight the best solution found across all algorithms in bold. Each table also includes aggregated values for the full set of instances (including ones that are not part of the sample). In particular, for the tables related to the irreducible graph sizes, we present the number of completely reduced instances. For the tables related to the number of solved instances, we present the total number of solved instances. Additionally, for those instances that are solvable by an exact algorithm, we also present the number of instances on which the heuristic algorithm was able to compute a solution of the same weight.

Table 4.4 shows the sizes of the irreducible graphs after the initial reduction phase. Note that we omit $\text{BnR}_{\text{dense}}$ as it always calculates equally sized or larger graphs than BnR_{full}.

4.3.4.a) Comparison with Branch-and-Reduce

First, we note that except for fe_ocean, Cyclic-Strong always produces the smallest reduced graphs. For this particular instance, the usage of the struction limits the efficiency of the critical set reduction, resulting in a larger reduced graph. Furthermore, the greatest improvement can be found on the mesh instances, which are all completely reduced using Cyclic-Strong. In comparison, BnR_{full} is not able to obtain an empty graph on a single of these instances and ends up with reduced graphs consisting of up to thousands of vertices. Overall, Cyclic-Strong is able to achieve an empty reduced graph on 60 of the 87 instances tested – an additional 48 instances compared to the 22 empty graphs computed by BnR_{full}.

If we compare the reduced graphs of Cyclic-Strong and Cyclic-Fast, we see that they always have the same size on the mesh instances. However, the size of the reduced instances computed by Cyclic-Fast on the other instance families is up to a few thousand vertices larger. On the OSM instances, for example, Cyclic-Fast calculates a reduced graph that has the same size as the one computed by Cyclic-Strong on only 16 out of 34 instances, with the largest difference being 2216 vertices.

Next, we examine the number of solved instances and the time required to solve them. For this purpose, Figure 4.12 shows cactus plots for the number of solved instances over time. First, we can see that Cyclic-Strong was able to solve the most instances overall (68 out of 87 instances). To be more specific, Cyclic-Strong was able to solve an additional 11 instances compared to BnR_{full} and $\text{BnR}_{\text{dense}}$. Of these newly solved instances, six are from the OSM family, three from the SNAP family, and one additional instance from the FE family.

Comparing the time that our algorithms require to solve the instances with BnR_{full} and $\text{BnR}_{\text{dense}}$, we can see improvements on almost all instances. Our Cyclic-Fast algorithm is able to find solutions up to an order of magnitude faster than BnR_{full} and $\text{BnR}_{\text{dense}}$ on five mesh instances, 13 OSM instances, and three SNAP instances. On the two OSM instances pennsylvania-AM3 and utah-AM3 and the SNAP instance roadNet-CA, we are up to two

Table 4.4: Smallest irreducible graph found by each algorithm (in number of vertices $n(\mathcal{K})$ of reduced graph $\mathcal{K}$) and the time (in seconds) required to compute it. The smallest irreducible graph for each instance is highlighted in bold. Rows are highlighted in gray if one of our algorithms is able to obtain an empty graph.

Graph	$n(\mathcal{K})$	t_r	$n(\mathcal{K})$	t_r	$n(\mathcal{K})$	t_r	$n(\mathcal{K})$	t_r	$n(\mathcal{K})$	t_r
OSM instances	$\text{BnR}_{\text{dense}}$		BnR_{full}		NonIncreasing		Cyclic-Fast		Cyclic-Strong	
alabama-AM2	173	0.06	173	0.07	**0**	0.01	**0**	0.01	**0**	0.01
district-of-columbia-AM2	6 360	11.86	6 360	14.39	5 606	0.85	1 855	2.51	**1 484**	84.91
florida-AM3	1 069	31.52	1 069	35.20	814	0.13	661	0.44	**267**	42.26
georgia-AM3	861	8.99	861	10.14	796	0.08	587	0.69	**425**	12.84
greenland-AM3	3 942	3.81	3 942	24.77	3 953	3.94	**3 339**	10.27	**3 339**	54.44
new-hampshire-AM3	247	4.99	247	5.69	164	0.02	**0**	0.07	**0**	0.09
rhode-island-AM2	1 103	0.55	1 103	0.68	845	0.17	**0**	0.53	**0**	4.57
utah-AM3	568	8.21	568	8.97	396	0.03	**0**	0.09	**0**	0.40
Empty graphs	0.0% (0/34)		0.0% (0/34)		11.8% (4/34)		41.2% (14/34)		50.0% (17/34)	
SNAP instances	$\text{BnR}_{\text{dense}}$		BnR_{full}		NonIncreasing		Cyclic-Fast		Cyclic-Strong	
as-skitter	26 584	25.82	8 585	36.69	3 426	4.75	2 782	5.50	**2 343**	6.80
ca-AstroPh	**0**	0.02	**0**	0.02	**0**	0.02	**0**	0.03	**0**	0.03
email-EuAll	**0**	0.08	**0**	0.09	**0**	0.06	**0**	0.09	**0**	0.07
p2p-Gnutella06	**0**	0.01	**0**	0.01	**0**	0.01	**0**	0.01	**0**	0.01
roadNet-PA	133 814	2.43	35 442	7.73	300	1.05	**0**	1.19	**0**	1.14
soc-LiveJournal1	60 041	236.88	29 508	213.74	4 319	22.27	3 530	24.13	**1 314**	37.77
web-Google	2 810	1.57	1 254	2.42	361	1.75	**46**	1.88	**46**	7.97
wiki-Vote	477	0.03	**0**	0.02	**0**	0.02	**0**	0.02	**0**	0.02
Empty graphs	58.1% (18/31)		67.7% (21/31)		67.7% (21/31)		80.6% (25/31)		80.6% (25/31)	
mesh instances	$\text{BnR}_{\text{dense}}$		BnR_{full}		NonIncreasing		Cyclic-Fast		Cyclic-Strong	
buddha	380 315	5.56	107 265	26.19	86	1.83	**0**	1.87	**0**	1.91
dragon	51 885	0.89	12 893	1.34	**0**	0.18	**0**	0.19	**0**	0.21
ecat	239 787	4.07	26 270	10.09	274	2.12	**0**	2.12	**0**	2.14
Empty graphs	0.0% (0/15)		0.0% (0/15)		66.7% (10/15)		100.0% (15/15)		100.0% (15/15)	
FE instances	$\text{BnR}_{\text{dense}}$		BnR_{full}		NonIncreasing		Cyclic-Fast		Cyclic-Strong	
fe_ocean	141 283	1.05	**0**	5.94	138 338	8.90	138 134	9.61	138 049	10.78
fe_sphere	15 269	0.21	15 269	1.47	2 961	0.34	147	0.62	**0**	0.75
Empty graphs	0.0% (0/7)		14.3% (1/7)		0.0% (0/7)		28.6% (2/7)		42.9% (3/7)	

orders of magnitude faster. We attribute this increase in performance to the much smaller reduced graph size, as often a smaller graph size tends to result in finding a solution faster. Additionally, the generalized neighborhood folding reduction that is used in BnR_{full} and $\text{BnR}_{\text{dense}}$ tends to increase the running time. Thus, we decided to omit this reduction rule from our algorithm.

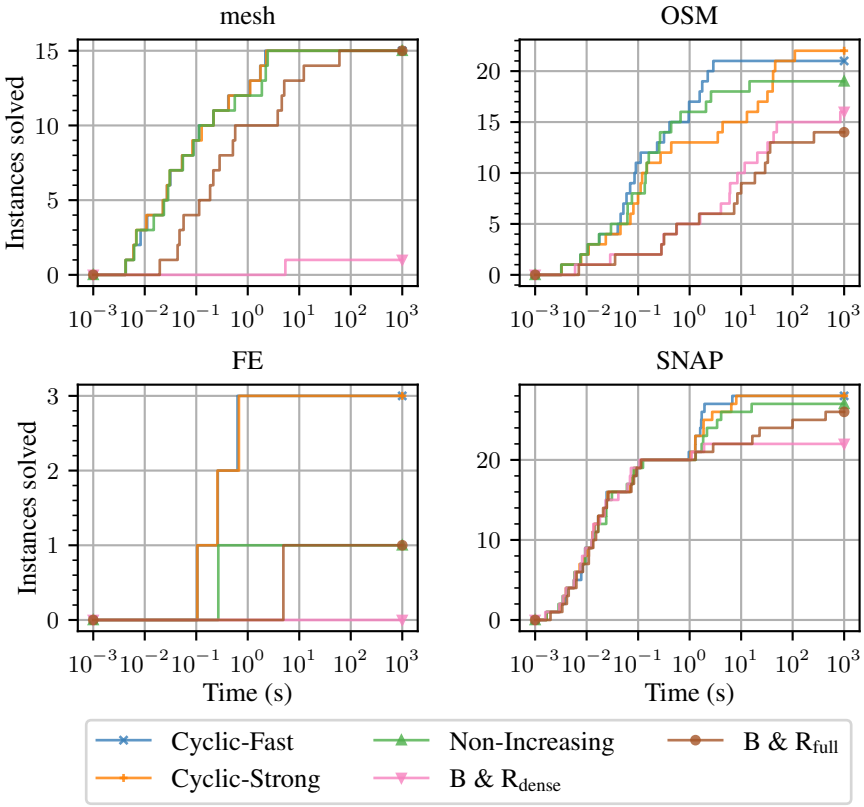


Figure 4.12: Cactus plots for the different instance families and evaluated algorithms.

4.3.4.b) Comparison with Heuristic Approaches

In the following, we provide a comparison of our algorithms with heuristic state-of-the-art approaches. For this purpose, we also include the two local searches DynWVC and HILS and the two branch-and-reduce algorithms BnR_{full} and $\text{BnR}_{\text{dense}}$. For DynWVC, we use both configurations DynWVC1 and DynWVC2 described by Cai et al. [Cai+18]. For all algorithms, we compare both the best achievable weighted independent set and their convergence behavior regarding solution quality. An overview of the maximum weight and the minimum time required to obtain it is given in Table 4.5. Furthermore, for each exact algorithm, the number of solved instances is shown, whereas for heuristic algorithms, the number of instances on which they are also able to find a solution with optimal weight is given. However, note that the heuristic algorithms tested are not able to verify the optimality of the solution they computed. For the individual instance families, we list either DynWVC1 or DynWVC2, depending on which of the two configurations provides better performance.

Table 4.5: Best solution found by each algorithm and the time (in seconds) required to compute it. The global best solution is highlighted in bold. Rows are highlighted in gray if one of our exact algorithms is able to solve the corresponding instances.

Graph	t_{max}	w_{max}	t_{max}	w_{max}	t_{max}	w_{max}	t_{max}	w_{max}
OSM instances	DynWVC2		HILS		Cyclic-Fast		Cyclic-Strong	
alabama-AM2	0.24	174 269	0.03	**174 309**	0.01	**174 309**	0.01	**174 309**
district-of-columbia-AM2	915.18	208 977	400.69	**209 132**	4.21	**209 132**	84.21	209 131
florida-AM3	862.04	237 120	3.98	**237 333**	1.57	**237 333**	40.97	**237 333**
georgia-AM3	1.31	**222 652**	0.04	**222 652**	0.98	**222 652**	12.97	**222 652**
greenland-AM3	640.46	14 010	1.18	**14 011**	10.95	**14 011**	58.24	14 008
new-hampshire-AM3	1.63	**116 060**	0.03	**116 060**	0.05	**116 060**	0.08	**116 060**
rhode-island-AM2	13.90	184 576	0.24	**184 596**	0.41	**184 596**	4.37	**184 596**
utah-AM3	136.90	**98 847**	0.07	**98 847**	0.09	**98 847**	0.27	**98 847**
Solved instances						61.8% (21/34)		64.7% (22/34)
Optimal weight		68.2% (15/22)		100.0% (22/22)				
SNAP instances	DynWVC2		HILS		Cyclic-Fast		Cyclic-Strong	
as-skitter	383.97	123 273 938	999.32	122 658 804	346.69	124 137 148	354.71	**124 137 365**
ca-AstroPh	125.05	797 480	13.47	**797 510**	0.02	**797 510**	0.02	**797 510**
email-EuAll	132.62	**25 286 322**	338.14	**25 286 322**	0.07	**25 286 322**	0.07	**25 286 322**
p2p-Gnutella06	186.97	548 611	1.29	**548 612**	0.01	**548 612**	0.01	**548 612**
roadNet-PA	469.18	60 990 177	999.94	60 037 011	0.96	**61 731 589**	1.04	**61 731 589**
soc-LiveJournal1	999.99	279 231 875	1 000.00	255 079 926	51.33	284 036 222	44.19	**284 036 239**
web-Google	324.65	56 206 250	995.92	56 008 278	1.72	**56 326 504**	6.44	**56 326 504**
wiki-Vote	0.32	**500 079**	10.34	**500 079**	0.02	**500 079**	0.02	**500 079**
Solved instances						90.3% (28/31)		90.3% (28/31)
Optimal weight		28.6% (8/28)		57.1% (16/28)				
mesh instances	DynWVC2		HILS		Cyclic-Fast		Cyclic-Strong	
buddha	797.35	56 757 052	999.94	55 490 134	1.75	**57 555 880**	1.77	**57 555 880**
dragon	981.51	7 944 042	996.01	7 940 422	0.21	**7 956 530**	0.22	**7 956 530**
ecat	542.87	36 129 804	999.91	35 512 644	2.19	**36 650 298**	2.29	**36 650 298**
Solved instances						100.0% (15/15)		100.0% (15/15)
Optimal weight		0.0% (0/15)		0.0% (0/15)				
FE instances	DynWVC1		HILS		Cyclic-Fast		Cyclic-Strong	
fe_ocean	983.53	**7 222 521**	999.57	7 069 279	18.85	6 591 832	19.04	6 591 537
fe_sphere	875.87	616 978	843.67	616 528	0.63	**617 816**	0.67	**617 816**
Solved instances						42.9% (3/7)		42.9% (3/7)
Optimal weight		0.0% (0/3)		0.0% (0/3)				

Finally, we omit BnR_{full}, $\text{BnR}_{\text{dense}}$, and NonIncreasing, as these are outperformed by either Cyclic-Fast or Cyclic-Strong, as presented in the previous section. A complete overview of the solution sizes and running times for each algorithm is given in Appendix J.

Considering the OSM family, we see that our algorithms are able to compute an optimal solution on 22 of the 34 instances tested. However, HILS was also able to calculate a solution of optimal size on all these instances. Considering the OSM family, we can see that HILS calculates optimal solutions on all 22 of the 34 instances that can be solved by our algorithm

Cyclic-Strong. In contrast, DynWVC is only able to do so on 15 of the 22 instances. On ten of the remaining 12 instances which our algorithms are not able to solve, HILS is able to calculate the best solution. Finally, when comparing the time required to compute the best solution, we find that HILS generally performs better than the other algorithms.

For the SNAP instances, we have already seen that Cyclic-Fast and Cyclic-Strong can solve 28 of the 31 instances optimally. In contrast, HILS can only calculate optimal solutions on 16 of these 28 instances. Additionally, DynWVC is able to obtain optimal solutions on eight of the solved instances. For the three unsolved instances, Cyclic-Strong computes the best solution on as-skitter and soc-LiveJournal1, while DynWVC1 obtains it on soc-pokec-relationships. In terms of running time, we see that both DynWVC and HILS are often orders of magnitude slower than our algorithms in achieving their optimal solution.

On the mesh instances, we can observe a similar pattern as for the SNAP instances. Our algorithms Cyclic-Fast and Cyclic-Strong are able to solve all instances optimally and always need less than three seconds to obtain them. On the other hand, none of the evaluated local searches is able to compute an optimal solution on a single instance and are slower than our algorithms by orders of magnitude.

Finally, on the FE family, neither DynWVC nor HILS is able to obtain a solution of equal weight on any of the three instances solved by our algorithms. However, considering the unsolved instances, our algorithms are only able to compute the best solution on the instance fe_body. On all remaining instances, one of the two DynWVC configurations calculates the best solution.

4.4 Conclusion and Future Work

In this chapter, we presented novel approaches and reduction rules to find optimal maximum weight independent sets. Our approaches are able to scale to significantly larger graphs than previously feasible for (exact) state-of-the-art algorithms. Our new reductions can also be integrated into existing approaches to drastically improve their performance.

To be more specific, we presented the first practically efficient branch-and-reduce algorithm for the maximum weight independent set problem. The core of this algorithm is a full suite of novel reduction rules. This suite includes generalizations of commonly used reduction rules for the unweighted problem and reduction rules that have no clear unweighted counterpart. We also presented meta reductions that can be used as a theoretical framework for many of these rules. Our reduction rules are then integrated into a branch-and-reduce algorithm optimized for the maximum weight independent set problem. We performed extensive experiments to show the effectiveness of the resulting algorithm in practice on real-world graphs of up to millions of vertices and edges. Our experimental evaluation shows that our algorithm can solve many large real-world instances very quickly. Additionally, our experiments show a highly positive effect on existing local search algorithms when they are paired with our reductions.

Next, we presented multiple algorithms that make use of novel transformations based on the struction method. In general, the struction method can be classified as a transformation that reduces the independence number of a graph. One caveat of the struction method

is that it does not guarantee that the size of the graph reduces. We thus introduced three different types of structions that aim to reduce the number of newly constructed vertices. In addition, we derived special cases of these struction, e.g., by limiting the number of new vertices, that are efficient in practice. We also derived versions that can easily be integrated into existing approaches by making sure that *no* new vertices are created. Our experimental evaluation indicates that our techniques outperform existing algorithms on a wide variety of instances. In particular, except for a single instance, our algorithms produce the smallest known reduced graphs and, when performed with branch-and-reduce, solve more instances than existing exact algorithms—even solving instances faster than heuristic approaches.

Overall, as for our unweighted approaches presented in Chapter 3, our approaches scale to significantly larger instances than previously feasible. Additionally, we increased the number of instances that can be solved exactly in a reasonable amount of time— often even rivaling the performance of heuristic algorithms. Finally, our algorithms can easily be extended by additional reduction rules, branching strategies, or upper and lower bounds.

As for the unweighted case, it still remains an important open problem which reductions provide an optimal balance between speed and quality for a given instance. Integrating inexact reductions to improve the performance of heuristic approaches might also be a promising opportunity for future work. Due to the results achieved by using increasing transformations, it would be of particular interest to develop increasing transformations that are efficient enough to be used throughout the recursion of the branch-and-reduce algorithm. This could result in a more general *branch-and-transform* framework. Evaluating reductions that are similar to the struction, like the conic reduction [Loz00] or clique reduction [LP09], might also yield beneficial results. Finally, it would be interesting to see if the reduction rules for the weighted case can be optimized for the usage on unweighted instances and provide a significant benefit.

Chapter 5

5 Maximum Cuts

We engineer a new suite of efficient reduction rules for the NP-hard problem of finding maximum cardinality cuts of a graph. Previous works mainly focused on theoretically efficient reduction algorithms. However, the practical efficiency of these algorithms remains an unexplored problem. We thus propose a set of practically efficient reduction rules and demonstrate their significant impact for solving highly challenging benchmark instances. Additionally, we note that our reduction rules subsume most of the previously published rules. Our experiments indicate that the performance of current state-of-the-art approaches can be improved by up to multiple orders of magnitude when using our reduction rules.

References. This chapter is based on the conference paper [Fer+20] (ALENEX 2020) published jointly with Damir Ferizovic, Demian Hespe, Matthias Mnich, Christian Schulz, and Darren Strash. Together with Demian Hespe, the author of this dissertation is one of the main authors of the paper, with editing done by Damir Ferizovic, Matthias Mnich, Christian Schulz, and Darren Strash. The author made smaller contributions to the reductions presented in Section 5.2, which were mainly developed by Damir Ferizovic and Demian Hespe. In order to be self-contained, we include all rules and refer to Ferizovic et al. [Fer+20] for their corresponding proofs. Furthermore, the author made major contributions to the engineering aspects of the algorithm (Section 5.3). Implementation was done by Damir Ferizovic. Evaluations were done by Damir Ferizovic and the author of this dissertation. Large parts of this section were copied verbatim from the paper or the technical report [Fer+19].

Motivation. The maximum cardinality cut problem (MaxCut) is to partition the vertex set of a given graph $G = (V, E)$ into two sets $S \subseteq V$ and $V \setminus S$ so as to maximize the total number of edges between those two sets. Such a partition is called a *maximum cut*. Computing a maximum cut of a graph is a well-known NP-hard problem [GJS74] in the area of computer science. While signed and weighted variants are often considered throughout the literature [Bar82; Bar96; Bar+88; Chi+07; dSHK13; Har59; HLW02], the unweighted case still has lots of potential for improvement, and solving it quickly is of importance to all variants. MaxCut variants have many applications including social network modeling [Har59], statistical physics [Bar82], portfolio risk analysis [HLW02], VLSI design [Bar+88; Chi+07], network design [Bar96], and image segmentation [dSHK13]. Instances used for these applications can contain tens of thousands of vertices and edges.

For example, maximum weight cuts can be used in image segmentation to separate regions with similar average colors [dSHK13; DGS18]. For this purpose, an image is divided into regions with similar colors. Each of these regions is represented by a vertex in a graph.

Vertices are connected if the regions of the image are adjacent to each other. Additionally, edges are weighted based on the dissimilarity of the average colors of the regions of both endpoints. To be more specific, the weight of an edge is larger if the average colors are more dissimilar. Regions that are not separated by a maximum weight cut then represent regions with similar average colors and can be merged.

Maximum weight cuts can also be used for solving the via minimization problem in VLSI design and printed circuit board design [Bar+88]. In printed circuit board design, a "via" is a hole that has to be drilled in order to create a feasible layer assignment that assigns crossing wire segments to different layers. Since vias introduce additional costs during fabrication and can lead to the malfunction of a circuit board, one wants to find an assignment that minimizes the number of vias. To solve this problem, a reduced layout graph is used where vertices correspond to connected components of critical segments, i.e., wire segments on which no via can be placed. Vertices are connected by an edge if any two critical segments within the components are connected by a free segment, i.e., a wire segment on which a via can be placed. Edges are weighted based on the difference $a - b$, where a is the number of vias needed when both endpoints are placed on the same layer, and b is the number of vias needed when both endpoints are placed on different layers. A maximum weight cut then corresponds to a partitioning of the vertices that minimizes the number of vias.

Many practical approaches exist to compute a maximum cut [RRW10; Ben+11; Gar+14] or a large (inexact) cut [Wan+13; BH13; Koc+13; AO09]. Curiously, reductions, which have shown promising results for large instances of other fundamental NP-hard problems [Abu+07; HSS19; Lam+17], were previously not used in implementations of MaxCut approaches. To the best of our knowledge, no research has been done on the efficiency of reductions for MaxCut with the particular goal of achieving small reduced graphs *in practice*.

Overview. In this chapter, we extend the scope of practically efficient reductions from the maximum (weight) independent set problem presented in the previous chapters to MaxCut. We begin by presenting recent related work on MaxCut that includes two commonly used lower bounds in Section 5.1.

The main contributions of this chapter are then presented in Sections 5.2 and 5.3. In particular, we engineer a new suite of efficient reduction rules that subsume most of the previously published rules and demonstrate their significant impact on benchmark data sets, including synthetic instances and data sets from the VLSI and image segmentation application domains. Our experiments reveal that current state-of-the-art approaches can be sped up by up to *multiple orders of magnitude* when combined with our reduction rules. On social and biological networks, reductions enable us to solve four instances that were previously unsolved within a ten-hour time limit with state-of-the-art approaches; three of these instances are now solved in less than two seconds. Finally, we conclude this chapter with an outline of important open problems and opportunities for future work in Section 5.5.

Overall, this chapter serves as an example of how reductions can be useful for other NP-hard problems, where their usage has not been explored as thoroughly as for independent sets. As for the problems presented in detail in previous chapters, our algorithms are able to increase both the number *and* scale of instances that can be solved efficiently in practice.

5.1 Related Work

There are multiple works on improving fixed-parameter algorithms for MaxCut [Cro+13; CJM15; EM18; MSZ18]. Even though these works introduce a large set of reduction rules for effectively reducing a MaxCut instance, they are mostly focused on theoretical properties [Cro+13; CJM15; EM18; MSZ18; Pri05; Far+17]. These reductions typically have some constraints on the subgraphs they can be applied to, like being clique forests or clique-cycle forests. However, there are reduction rules for the general case that take exponential time in the maximum cut size [MR99]. There are other reduction rules that are fairly simplistic and focus on very narrow cases [Pri05]. For a thorough examination of existing reductions, we refer to Ferizovic [Fer19].

Many fixed-parameter algorithms for MaxCut furthermore utilize well-known lower bounds for the size of the maximum cut. By doing so, the corresponding algorithms have to decide whether a given MaxCut instance has a cut of size $k + l$ where $k \in \mathbb{N}_0$ is a parameter and l is the lower bound. In the following, we cover two commonly used lower bounds: the Edwards-Erdős bound and the spanning tree bound.

Edwards-Erdős Bound. For a connected graph, the Edwards-Erdős bound [Edw73; Edw75] is defined as $EE(G) = m/2 + (n-1)/4$. A linear-time algorithm that computes a cut satisfying the Edwards-Erdős bound for any given graph is provided by Van Ngoc and Tuza [vNT93]. Given a graph G and integer $k \in \mathbb{N}_0$, the *Max Cut Above Edwards-Erdős* (MaxCutAEE) problem asks if G admits a cut of size $EE(G) + k$. All reduction rules for MaxCutAEE require a set $S \subseteq V$ set such that $G \setminus S$ is a clique forest. Etscheid and Mnich [EM18] propose an algorithm that computes such a set S of at most $3k$ vertices in time $\mathcal{O}(k \cdot (n+m))$.

Spanning Tree Bound. Another bound is based on utilizing the spanning forest of a graph [MSZ18]. For a given $k \in \mathbb{N}_0$, a maximum cut of size $n - 1 + k$ is searched for. This decision problem is denoted as MaxCutAST (*Max Cut Above Spanning Tree*). For sparse graphs, this bound is larger than the Edwards-Erdős bound. The reductions for the problem require a set $S \subset V$ such that $G \setminus S$ is a clique-cycle forest.

Note, that neither of the two bounds presented here is better for all graphs. Madathil et al. [MSZ18] show that when $n - 1 > m/2 + (n+1)/4$, the spanning tree bound is preferable. On the other hand, the Edwards-Erdős is preferable for dense graphs.

In terms of practical results, multiple approaches, both exact [RRW10; KM06; KMR14; KMR17; Gus+20; HP21; Hrg+19; RKS22; Cha+22; LAS19] and heuristic [DGS18; BMZ02; Koc+13], have been proposed in the literature. We note that many of these approaches primarily target dense instances of the more general problem WeightedMaxCut [RRW10; KMR14; KMR17; Gus+20; Hrg+19; HP21]. Thus, we only provide a brief overview of these approaches. Note that we also include the exact algorithm by Rendl et al. [RRW10] in our experimental evaluation to examine the effects of our preprocessing procedure on these types of algorithms.

5.1.1 Exact Approaches

Rendl et al. [RRW10] presented an overview of various techniques used by previous branch-and-bound algorithms. Furthermore, the authors present their own bounding procedure, which uses the semidefinite relaxation for WeightedMaxCut. Their experimental evaluation on a large set of instances (see Section 2.3.2.a)) with up to 343 vertices indicates that their algorithm outperforms previous approaches.

Krislock et al. [KMR17] also presented an exact branch-and-bound algorithm using a novel bounding procedure that can be used for WeightedMaxCut. The authors compared the performance of their algorithm to BiqMac by Rendl et al. [RRW10] on a set of 328 largely dense instances with less than 500 vertices from the BiqMac library [Wie18]. Their algorithm is able to solve around 75% of instances (85% for hard instances) in less time. We note that in addition to WeightedMaxCut, their algorithm can also be used for other problems, including MWIS.

Recent parallel branch-and-bound algorithms are given by Hrga et al. [Hrg+19], Gusmeroli et al. [Gus+20], and Hrga and Povh [HP21]. The most recent approach by Hrga and Povh [HP21] uses a distributed memory parallelization to efficiently process multiple branches of the branch-and-bound tree concurrently. In addition to comparing their sequential algorithm against previous state-of-the-art approaches, they also evaluated the scalability of their approach on up to 48 workers and showed good strong-scaling behavior.

In the following, we cover recent state-of-the-art approaches that focus on larger (and sparse) instances, which our work focuses on. Note that many of the following approaches also target the more general problem WeightedMaxCut.

Recently, Charfreitag et al. [Cha+22] proposed an exact algorithm for WeightedMaxCut tailored towards large sparse instances. In particular, their branch-and-cut algorithm uses innovations based on integer linear programming and polyhedral combinatorics. They compared their algorithm against various state-of-the-art approaches, including the previously presented works by Krislock et al. [KMR17] and Gusmeroli et al. [Gus+20], which are targeted towards dense instances. Their evaluation indicates that their algorithm outperforms these approaches on a large set of sparse instances taken from multiple real-world applications. The authors also evaluated the usage of the reduction techniques presented in this chapter. However, except for a few promising instances, performance gains from reductions were negligible.

Lange et al. [LAS19] propose a preprocessing algorithm that uses novel reduction rules for WeightedMaxCut and minimum weight multicuts, i.e., finding a k-way partition that minimizes the total cut. Their reduction rules target single edges, triangles, as well as more general connected subgraphs. The authors show that their preprocessing algorithm using these rules is able to drastically reduce the graph size for WeightedMaxCut instances stemming from statistical physics. Rehfeldt et al. [RKS22] extend this set of reductions and integrate them into a parallel branch-and-cut algorithm. They compare their algorithm against Gurobi [Gur21] and the algorithm of Charfreitag et al. [Cha+22] on a large set of dense and sparse instances. The authors also note that recent versions of Gurobi are able to outperform many state-of-the-art approaches, including the algorithms by Rendl et al. [RRW10], Gusmeroli et al. [Gus+20; Hrg+19], and Hrga and Pvoh [HP21].

5.1.2 Heuristic Approaches

Dunning et al. [DGS18] evaluated 37 different heuristic approaches from existing works on a large set of over 3000 instances. Their evaluation indicates that an algorithm by Burer et al. [BMZ02] is among the best state-of-the-art heuristics available. Furthermore, they provide a machine learning-based algorithm portfolio called MqLib that selects the most well-suited set of heuristics for a given problem instance. However, Wang and Hao [WH22] note that their portfolio does not include several advanced algorithms available. Finally, their algorithm is not able to determine if a computed cut is actually the maximum cut.

Wang and Hao [WH22] provide an overview of state-of-the-art metaheuristic algorithms for the closely related quadratic unconstrained binary optimization problem. This includes fast local search and population-based methods. They further evaluate multiple state-of-the-art approaches on a set of 54 WeightedMaxCut instances with varying sizes between 800 and 2000 vertices. In particular, they examine two algorithms based on path relinking [Wan+12], a diversification-driven tabu search [GLH10], and two variants of an automatic algorithm [dSR18] trained on different inputs. Their evaluation indicates that the path relinking approach performs best in terms of the best solutions obtained, whereas the automatic approach performs best in terms of average solution values.

5.2 Practically Efficient Reductions

We now introduce our new reduction rules. The main feature of our new rules is that they do not depend on the computation of a clique-forest to determine if they can be applied. Furthermore, our new rules subsume almost all rules from previous works [Cro+13; CJM15; EM18; MSZ18; Far+17] except for reduction rules 10 and 11 by Crowston et al. [Cro+13]. Hence, our algorithm will only apply the rules proposed in this section. For proofs and further details, including how to obtain a maximum cut of the unreduced instance, we refer to Ferizovic [Fer19]. For an overview of how rules are subsumed, we refer to Table 5.1.

Table 5.1: Reduction rules subsumed by our new rules. An "x" in row *a* and column *b* means that rule *a* subsumes rule *b*. If there are multiple "x"s in a column (say, rows *a* and *b* in column *c*), then rules *a* and *b* combined subsume rule *c*.

Source	[Far+17]	[CJM15]			[Cro+13]		[EM18]	[MSZ18]							
Rule	A	5	6	7	8	9	9	6	7	8	9	10	11	12	13
1	x	x		x		x	x	x	x	x	x			x	x
2								x		x	x	x	x		
5			x												
$7_{w=1}$			x		x										

Reducing Cliques. Our first reduction rule targets cliques that have few external vertices, i.e., cliques where the number of external vertices is at most half the number of vertices.

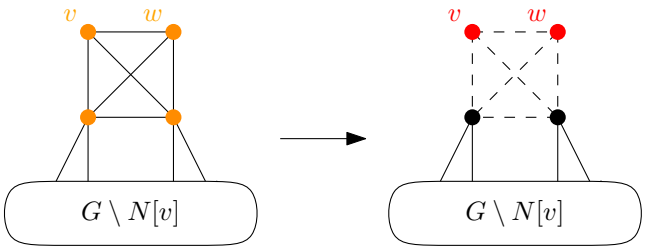


Figure 5.1: Example application of reduction rule 1 to the clique $N[v]$ (orange). The vertices v and w (and all edges $E(G[N[v]])$ can be removed from the graph.

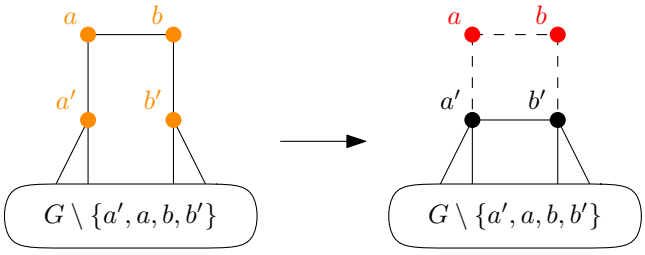


Figure 5.2: Example application of reduction rule 2 to the induced 3-path $\langle a', a, b, b' \rangle$ (orange). The vertices a and b (and their incident edges) can be removed from the graph. An additional edge $\{a', b'\}$ is added to the graph.

If such a clique is found, we are able to remove all its internal vertices and their incident edges. Figure 5.1 shows an example application of this reduction rule.

Theorem 5.1 (Reduction Rule 1)

Let $G = (V, E)$ be a graph and let $S \subseteq V$ induce a clique in G. If $|C_{ext(G)}(S)| \leq \lceil |S|/2 \rceil$, then $\beta(G) = \beta(G') + \beta(K_{|S|})$ for $G' = (V \setminus C_{int(G)}(S), E \setminus E(G[S]))$.

Reducing Paths. Our second reduction rule is able to reduce induced 3-paths $\langle a', a, b, b' \rangle$. For such a path, we can remove both a and b (and their incident edges) and instead insert an edge $\{a', b'\}$. Figure 5.2 shows an example application of this reduction rule.

Theorem 5.2 (Reduction Rule 2)

Let $\langle a', a, b, b' \rangle$ be an induced 3-path in a graph G with $N(a) = \{a', b\}$ and $N(b) = \{a, b'\}$. Construct G' from G by adding a new edge $\{a', b'\}$ and removing the vertices a and b. Then $\beta(G) = \beta(G') + 2$.

Extending Near-Cliques. Since we are able to remove certain cliques with reduction rule 1, we now present a transformation that is able to extend near-cliques under certain conditions. To be more specific, we can extend a near-clique if its number of vertices is odd or it contains more than two internal vertices. The newly constructed clique then might be reduced by reduction rule 1, which can lead to an overall decrease in size.

Theorem 5.3 (Reduction Rule 3)

Let $G = (V, E)$ be a graph and let $S \subseteq V$ induce a near-clique in G. Let G' be the graph obtained from G by adding the missing edge e' so that S induces a clique in G'. If $|S|$ is odd or $|C_{int(G)}(S)| > 2$, then $\beta(G) = \beta(G')$.

Clique Edge Removal. Since some cliques are irreducible by currently known rules, it may be beneficial to also apply reduction rule 3 "in reverse". Although this "reverse" reduction neither reduces the vertex set nor (as our experiments suggest) leads to applications of other rules, it can undo unfruitful additions of edges made by reduction rule 3 and may remove other edges from the graph.

Theorem 5.4 (Reduction Rule 4)

Let G be a graph and let $S \subseteq V$ induce a clique in G. If $|S|$ is odd or $C_{int(G)}(S) > 2$, an edge between two vertices of $C_{int(G)}(S)$ is removable. That is, $\beta(G) = \beta(G')$ for $G' = (V, E \setminus \{e\})$, $e \in E(G[C_{int(G)}(S)])$.

Common Clique. The next reduction rule is closely related to the upcoming generalization of reduction rule 8 by Crowston et al. [Cro+13]. It is able to further reduce the case where $|X| = |N(X)|$ for a clique X of G. In comparison, the generalization of reduction rule 8 is able to handle the case $|X| > |N(X)|$. Due to the degree to which these rules are similar, they are also merged together in our implementation, as the techniques to handle both are the same and will be discussed later on. An example application of this reduction rule is given in Figure 5.3.

Theorem 5.5 (Reduction Rule 5)

Let $X \subseteq V$ induce a clique in a graph G, where $|X| = |N(X)| \geq 1$ and $N(X) = N(x) \setminus X$ for all $x \in X$. Create G' from G by removing an arbitrary vertex of X. Then $\beta(G) = \beta(G') + |X|$.

Weighted Path Compression. The following reduction rule is our only rule whose application turns unweighted instances into instances of WeightedMaxCut. Our experiments show that this can reduce the graph size *by up to half*. This reduction to a weighted instance is noteworthy, given that existing approaches for MaxCut usually support weighted instances.

Theorem 5.6 (Reduction Rule 6)

Let $G = (V, E, w)$ be a graph with an edge weighting function $w : E \to \mathbb{Z}$, and $\langle a, b, a' \rangle$ be an induced 2-path with $N(b) = \{a, a'\}$. Let e_1 be the edge between vertex a and b; let e_2 be the one between b and a'. Construct G' from G by deleting vertex b and adding a new edge $\{a, a'\}$ with $w'(\{a, a'\}) = \max\{w(e_1), w(e_2)\} - \max\{0, w(e_1) + w(e_2)\}$. Then $\beta(G, w) = \beta(G', w) + \max\{0, w(e_1) + w(e_2)\}$.

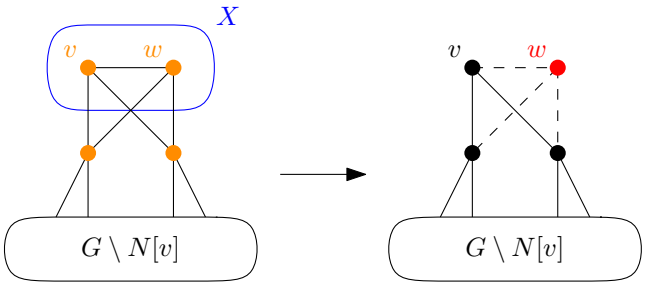


Figure 5.3: Example application of reduction rule 5 to the clique $N[v]$ (orange). One of the vertices of $X = \{v, w\}$ (blue) can be removed from the graph. In this example w is removed.

Generalization of Crowston et al. [Cro+13]. Our last two rules (reduction rules $7_{w=1}$ and 7) generalize reduction rule 8 by Crowston et al., which we restate for completeness.

Theorem 5.7 (Reduction Rule 8 [Cro+13])

Let $G = (V, E, l)$ be a signed graph, $S \subseteq V$ a set of vertices such that $G[V \setminus S]$ is a clique forest, and C a block in $G[V \setminus S]$. If there is a $X \subseteq C_{int(G[V \setminus S])}(C)$ such that $|X| > \frac{|C|+|N(X) \cap S|}{2} \geq 1$ and for all $x \in X$: $N_l^+(x) \cap S = N_l^+(X) \cap S$ and $N_l^-(x) \cap S = N_l^-(X) \cap S$. Construct the graph G' from G by removing any two vertices $x_1, x_2 \in X$. Then $\beta(G') - EE(G') = \beta(G) - EE(G)$.

This reduction rule requires a vertex set S such that $G \setminus S$ is a clique forest. Different choices of S then lead to different applications of this rule. Our generalizations do not require such a set anymore and can find *all* possible applications for any choice of S.

Theorem 5.8 (Reduction Rule $7_{w=1}$)

Let X be the vertex set of a clique in G with $|X| > \max\{|N(X)|, 1\}$ and $N(X) = N(x) \setminus X$ for all $x \in X$. Construct the graph G' by deleting two arbitrary vertices $x_1, x_2 \in X$ from G. Then $\beta(G) = \beta(G') + |N(x_1)|$.

Theorem 5.9 (Reduction Rule 7)

Let $X \subseteq V$ induce a clique in a signed graph (G, l) such that $\forall e \in E(G[X]) : l(e) =$ "$-$" and $|X| > \max\{|N(X)|, 1\}$, $N_l^+(X) = N_l^+(x) \setminus X$, and $N_l^-(X) = N_l^-(x) \setminus X$ for all $x \in X$. Construct a new graph G' by removing two arbitrary vertices $x_1, x_2 \in X$ from G. Then $\beta(G) = \beta(G') + |N(x_1)|$.

Weighted Generalizations. In order to also reduce weighted instances to some degree, we use a simple weighted generalization of two reduction rules. That is, we extend their applicability from an unweighted subgraph to a subgraph where all edges have the same weight $c \in \mathbb{R}$. We do this for reduction rules 1 and 3 as they are relatively easy to generalize.

Theorem 5.10 (Reduction Rule $1_{w=c}$)
Let $G = (V, E, w)$ be a weighted graph and let $S \subseteq V$ induce a clique with $w(e) = c$ for every edge $e \in E(G[S])$ for some constant $c \in \mathbb{R}$. Let $G' = (V \setminus C_{int(G)}(S), E \setminus E(G[S]), w')$ with $w'(e) = w(e)$ for every $e \in E(G')$. If $|C_{ext(G)}(S)| \leq \lceil \frac{|S|}{2} \rceil$, then $\beta_w(G) = \beta_w(G') + c \cdot \beta(K_{|S|})$.

Theorem 5.11 (Reduction Rule $3_{w=c}$)
Let $G = (V, E, w)$ be a weighted graph and let $S \subseteq V$ induce a near-clique in G. Furthermore, let $w(e) = c$ for every edge $e \in E(G[S])$ for some constant $c \in \mathbb{R}$. Let G' be the graph obtained from G by adding the edge e' so that S induces a clique in G'. Set $w'(e') = c$, and $w'(e) = w(e)$ for $e \in E$. If $|S|$ is odd or $|C_{int(G)}(S)| > 2$, then $\beta_w(G) = \beta_w(G')$.

5.3 Implementation

We now discuss our overall reduction algorithm for MaxCut in detail. This algorithm uses transformations between multiple variants of MaxCut, including weighted and signed. High-level pseudocode is given in Algorithm 5.1. We also provide implementation details for individual reduction rules and how we avoid unnecessary checks for their applicability.

Our algorithm begins by generating an unweighted instance by replacing every weighted edge $e \in E$ with weight w with an unweighted subgraph with at most $\mathcal{O}(w)$ vertices and edges [Fer19]. Note that if the weights are large this transformation is costly, and the resulting graph may not have size polynomial in the input. Afterwards, we apply our full set of unweighted reduction rules: 1, $7_{w=1}$ (together with 5), 2, and 3 (in this order). As already mentioned earlier, reduction rule $7_{w=1}$ is the unweighted version of 7. We then create a signed instance of the graph by exhaustively executing weighted path compression using reduction rule 6 with the restriction that the resulting weights are -1 or $+1$. We then exhaustively apply reduction rule 7. Once the signed reductions are done, we apply reduction rule 6 to fully compress all paths into weighted edges. This is then succeeded by reduction rules $3_{w=c}$ and $1_{w=c}$.

Once no further reduction rules are applicable, we transform the instance into an unweighted one and apply reduction rule 4 in order to further reduce the number of edges. If a weighted algorithm is to be used on the reduced graph, we exhaustively perform reduction rule 6 to produce a weighted reduced graph. Note that different permutations of the order in which reduction rules are applied can lead to different results. However, preliminary experiments showed that the difference in kernel size and running time is usually small, and the order we chose performed best.

In the following, we cover implementation details for finding candidates for the individual reduction rules. For this purpose, we group the reduction rules by how they are applied. Note that we store the edges of our graphs in a hash table to allow efficient edge lookups.

Reduction Rules 2 and 6. We can find all candidates for reduction rule 2 and 6 in linear time $\mathcal{O}(n)$ by iterating over all vertices and checking if a vertex v has degree two. For reduction rule 2, we additionally have to check if (1) v has a neighbor w that also has degree two and (2) that the other neighbor of v and w are not the same vertex.

Algorithm 5.1 : High-level overview of reduction algorithm for MaxCut. Adapted from Ferizovic [Fer19].

Data : $G = (V, E)$, boolean *weightedResult*
Result : Reduced graph $\mathcal{K}$

```
1  MaxCutReduction (G, weightedResult)
2      success ← True
3      while success do
4          success ← False
5          MakeUnweighted(G)
6          if ApplyUnweighted(G)        // Rule 1, 7_{w=1}, 5, 2 and 3; Return True on change
7           then
8              success ← True
9          MakeSigned(G)                                   // Restricted rule 6
10         if ApplySigned(G)                          // Rule 7; Return True on change
11          then
12             success ← True
13         MakeWeighted(G)                                              // Rule 6
14         if ApplyWeighted(G)          // Rule 3_{w=c} and 1_{w=c}; Return True on change
15          then
16             success ← True
17     MakeUnweighted(G)
18     ApplyRule4(G)
19     if weightedResult then
20         MakeWeighted(G)
21     return G
```

Reduction Rules 1, 3 and 4. We can find all candidates for reduction rule 1, 3 and 4 in $\mathcal{O}(n \cdot \Delta^2)$ by iterating over all vertices. Depending on the rule, we then have to check if different conditions hold when scanning a vertex v.

For reduction rules 1 and 4, we first mark all vertices that are external. This can be done by iterating over the neighborhood of a vertex v and determining the minimum degree $d_{\min}$ in $N[v]$. If $d(w) \neq d_{\min}$, then $w \in N[v]$ is an external vertex. Afterwards, we iterate over all vertices that have not been marked as external. For a vertex v, we then decide if $N[v]$ induces a clique in time $\mathcal{O}(|N[v]|^2) = \mathcal{O}(\Delta^2)$. Finally, we determine the set of internal and external vertices of the clique by checking if the degree of a vertex $w \in N[v]$ is equal to $d(v)$.

Finding all candidates for reduction rule 3 can also be done in time $\mathcal{O}(n \cdot \Delta^2)$ by scanning all vertices. We first determine if a vertex v has a neighborhood $N(v)$ such that (1) $N(v)$ is a clique and (2) there is no vertex $w \in N(v)$ such that $d(w) \leq d(v)$. This can be done in $\mathcal{O}(|N(v)|^2)$ time. Next, we determine the neighbor with minimal degree $w_{\min}$, iterate

over $N(w_{\min})$, and test if there is a vertex $w' \in N(w_{\min})$ such that $N(v) = N(w')$. We then check if $N[v] \cup \{w'\}$ has an odd size or contains at least two internal vertices. Finally, we determine if there is a single missing edge whose insertion results in $N[v] \cup \{w'\}$ becoming a clique in $\mathcal{O}(|N[v] \cup \{w'\}|)$ time.

Reduction Rules 5 and 7. The following algorithm identifies all candidates of reduction rule 5 (and 7) in linear time. First, we sort the adjacencies of all vertices. That is, for every vertex $v \in V$, the vertices in $N(v)$ are sorted according to their identifier. For this, we create an auxiliary array of size n that contains empty lists. We then traverse the vertices $w \in N(v)$ for every vertex $v \in V$ and insert each pair (v, w) in a list identified by indexing the auxiliary array with w. We then iterate once over the array from the lowest identifier to the highest and recreate the graph with sorted adjacencies. In total, this process takes $\mathcal{O}(n + m)$ time. However, in later applicability checks, we disregard sorting the adjacencies in linear time. Instead, we use a comparison based sort on the adjacencies.

Afterwards, for any clique X of G, we have to check if all pairs (x_1, x_2) of vertices from X satisfy $N[x_1] = N[x_2]$ (neighborhood condition). Our algorithm uses tries [Fre60; Bri59] to find all candidates. A *trie* supports two operations: Insert(key,val) and Retrieve(key). The key parameter is an array of integers, and val is a single integer. Retrieve returns all inserted values by Insert that have the same key. Both of these operations require time linear in the length of the key. Internally, a trie stores the inserted elements as a tree, where every node corresponds to one integer of the key, and every prefix is stored only once. This means that two keys sharing a prefix share the same path through the trie until the position where they differ. Values are stored in the leaves of this tree using an array of integers.

For each vertex $v \in V$, we use the ordered set $N[v]$ as key and v as val. Notice that $N(v)$ is already sorted. The key $N[v]$ can thus be computed through the insertion of v into the sequence $N(v)$ in time $\mathcal{O}(|N[v]|)$. After we perform Insert($N[v]$,v) for every vertex $v \in V$, each trie leaf contains all vertices that satisfy the condition of reduction rule 5, i.e., they have the same closed neighborhoods. This means that every vertex pair (x_1, x_2) in the vertex set X stored in a trie leaf satisfies the neighborhood condition. We then verify whether X is a clique in $\mathcal{O}(|X|^2)$ time. Since all vertices in X also have the same closed neighborhood, we know that they induce a clique. Note that each such set X is considered exactly once, and the graph is fully partitioned, i.e., a vertex $v \in V$ is only part of one set X. Thus this requires $\mathcal{O}(n + m)$ time in total. As the last step, we check whether $|X| = |N(X)| \geq 1$.

Using an almost equivalent approach as we did for reduction rule 5, we can find all candidates of reduction rule 7. Our implementation combines searching candidates for these rules into a single routine.

Timestamping. Therefore, we only use this technique for reduction rules 1 and 7.

Next, we describe how to avoid unnecessary checks to see if reduction rules apply. For this purpose, let the time of the most recent change in the neighborhood of a vertex v be $T : V \to \mathbb{N}_0$, and let the variable $t \in \mathbb{N}$ describe the current time. Initially, $T(v) = 0$ for all $v \in V(G)$ and $t = 1$. Every time a reduction rule performs a change on $N(v)$, we set $T(v) = t$ and increment t. For each individual reduction rule r, we also maintain a timestamp $t_r \in \mathbb{N}_0$ (initialized with 0), indicating the upper bound up to which all vertices have already been

processed. Hence, all vertices $v \in V(G)$ with $T(v) \leq t_r$ do not need to be checked again by reduction rule r.

Note that timestamping only works for "local" reduction rules—the rules whose applicability can be determined by investigating the neighborhood of a vertex. Therefore, we only use this technique for reduction rules 1 and 7. In preliminary experiments, we did not find this method to be significantly faster (or slower) than others (for example, keeping a queue of vertices with changed neighborhoods and then considering these vertices for all reductions). However, unlike a queue, it allows us to limit possible redundant applications of individual reductions as much as possible. A priority queue is an ideal choice for processing timestamped vertices; however, in experiments, we found it sufficient to collect all vertices with timestamps greater than t_r and attempt to apply reduction rule r on them all.

5.4 Experimental Evaluation

We now examine how state-of-the-art heuristic and exact approaches can benefit from using our reduction algorithm. Particular attention will be placed on the benefits for large-scale instances that are hard to solve. Furthermore, we evaluate the performance of the individual reduction rules on a large set of synthetic instances.

Methodology and Setup. All of our experiments were run on Machine B described in Section 2.3.2.c). All algorithms were implemented in C++ and compiled using g++ version 7.3.0 using optimization flag `-O3`. We use the following state-of-the-art WeightedMaxCut approaches for comparisons: the exact algorithms LocalSolver [Ben+11] (heuristically finds a large cut and can then verify if it is maximum), BiqMac [RRW10] as well as the heuristic algorithm MqLib [DGS18]. MqLib is unable to determine on its own when it reaches a maximum cut and always exhausts the given time limit. We also evaluated an implementation of the reduction rules used by Etscheid and Mnich [EM18], seeing that our new rules find smaller kernels in shorter time. In the following, for a graph $G = (V, E)$, $\mathcal{K}$ denotes the graph after all reductions have been applied exhaustively. For this purpose, we examine the following efficiency metric: we denote the *reduction efficiency* by $e(G) = 1 - n(\mathcal{K})/n$. Note that $e(G)$ is equal to 1 when all vertices are removed after applying all reduction rules and 0 if no vertices are removed.

For our experiments, we use different datasets, including random graphs, sparse real-world networks, and denser instances from the BiqMac Library (see Section 2.3.2.a)). Note that for a large set of instances tested, reductions are not able to provide any significant reduction in size. For example, the rudy instances feature a uniform edge distribution and an overall average degree of at least 3.5. We provide results for these instances in Appendix L. Since our focus is on evaluating the performance of reduced graphs against unreduced ones, we therefore exclude instances that can not be reduced from our evaluation.

5.4.1 Performance of Individual Rules

To analyze the impact of each individual reduction rule, we measure the size of the reduced graph our algorithm procedures before and after their removal. Figure 5.4 shows our results

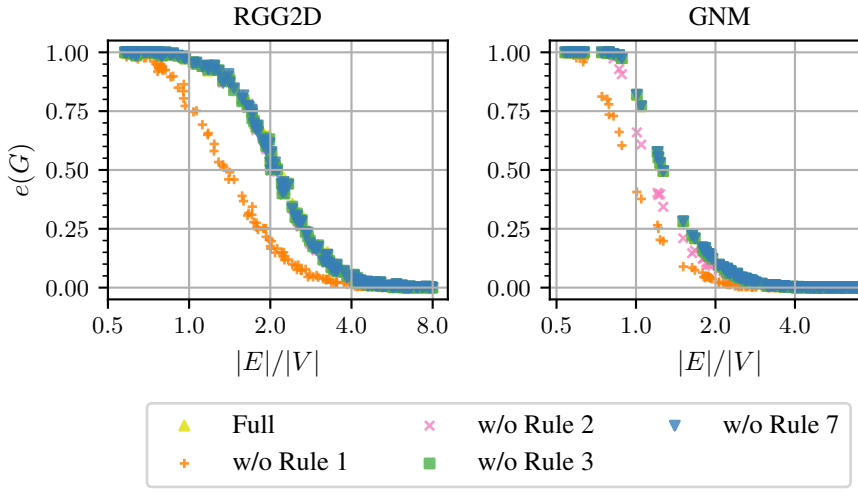


Figure 5.4: Comparison of the reduction efficiency of our full algorithm to the efficiency of our algorithm without a particular reduction rule tested on 150 synthetic instances with varying densities for each graph model.

on RGG2D and GNM graphs with 2048 vertices and varying densities. We have settled on those two types of graphs as they represent different ends on the spectrum of reduction efficiency, as seen in Figure 5.5. In particular, our preliminaries experiments indicated that reductions perform well on instances that are sparse and have a non-uniform degree distribution. Such properties are given by the random geometric graph model used for generating the RGG2D instances. Likewise, reductions perform poorly on the uniform random graphs that make up the GNM instances. We excluded reduction rule 4 from these experiments as it only removes edges and thus leads to no difference in reduction efficiency.

Looking at Figure 5.4, we can see that reduction rule 1 gives the most significant reduction in size. Its absence always diminishes the result more than any other rule. In particular, we see a difference in efficiency of up to 0.47 (RGG2D) and 0.41 (GNM) when removing reduction rule 1. The second most impactful rule for the RGG2D instances is reduction rule 7, with a difference of only up to 0.04. For the GNM instances, reduction rule 2 is second with a difference of up to 0.17. Whereas reduction rules 3 and 7 lead to no difference in efficiency on these instances. Thus, we can conclude that depending on the graph type, different reduction rules have varying importance. Furthermore, our simple reduction rule 1 seems to have the most significant impact on the overall reduction efficiency. Note that this is in line with the theoretical results from Table 5.1, which states that reduction rule 1 covers most of the previously published reduction rules, and reduction rule 2 still covers many but fewer rules from previous work.

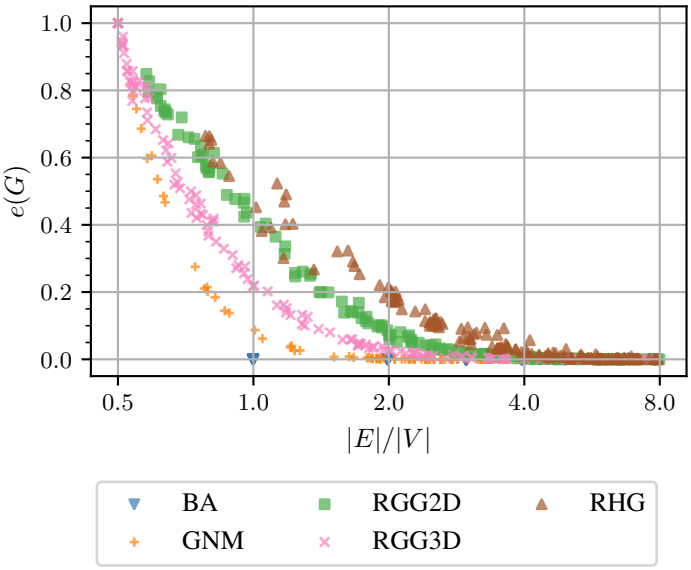


Figure 5.5: Reduction efficiency of our full algorithm tested on 150 synthetic instances with varying densities for each graph model.

5.4.2 Exactly Computing a Maximum Cut

To examine the improvements reductions bring for medium-sized instances, we compare the time required to obtain a maximum cut for both the reduced and the original instance. We performed these experiments using both LocalSolver and BiqMac. Note that we did not use MqLib as it is not able to verify the optimality of the cut it computes. The results of our experiments for the set of real-world instances are given in Table 5.2 (with weighted path compression) and Table 5.3 (without weighted path compression). Since the image segmentation instances are already weighted, they are omitted from Table 5.3.

First, we notice that reductions are able to provide moderate to significant speedups for all instances that we have tested. In particular, we observe a speedup between 1.04 and 228.91 for instances that were previously solvable by LocalSolver. Likewise, for the instances that BiqMac is able to process, we achieve a speedup of up to three orders of magnitude. Additionally, we allow these algorithms to now compute a maximum cut in less than 17 minutes for a majority of instances that have previously been infeasible.

To examine the impact of allowing a weighted reduced graph, we now compare the performance our algorithm using weighted path compression (Table 5.2) with the unweighted version (Table 5.3). We can see that by including weighted path compression, we can achieve significantly better speedups, especially for the sparse real-world instances by Rossi and Ahmed [RA15]. For example, on ego-facebook, we are able to achieve a speedup of 228.91 with compression and 11.83 without.

Table 5.2: Time to compute a maximum cut on the original instance G and the reduced instance $\mathcal{K}$ for LocalSolver (LS) and BiqMac (BM). Times t_{Alg} are given in seconds. Reductions are accounted for within the timings for $\mathcal{K}$. Values in brackets provide the speedup and are derived from $t_{Alg}(G)/t_{Alg}(\mathcal{K})$. Times labeled with "-" exceeded the ten-hour time limit, and an "f" indicates the algorithm crashed.

Graph	n	$e(G)$	$t_{\text{LS}}(G)$	$t_{\text{LS}}(\mathcal{K})$		$t_{\text{BM}}(G)$	$t_{\text{BM}}(\mathcal{K})$	
ca-CSphd	1 882	0.99	24.07	0.32	[75.40]	-	0.06	[∞]
ego-facebook	2 888	1.00	20.09	0.09	[228.91]	-	0.01	[∞]
ENZYMES_g295	123	0.86	1.22	0.33	[3.70]	0.82	0.13	[6.57]
road-euroroad	1 174	0.79	-	-	-	-	-	-
bio-yeast	1 458	0.81	-	-	-	-	32 726.75	[∞]
rt-twitter-copen	761	0.85	-	834.71	[∞]	-	1.77	[∞]
bio-diseasome	516	0.93	-	4.91	[∞]	-	0.07	[∞]
ca-netscience	379	0.77	-	956.03	[∞]	-	0.67	[∞]
soc-firm-hi-tech	33	0.36	4.67	1.61	[2.90]	0.09	0.06	[1.41]
g000302	317	0.21	0.58	0.49	[1.17]	1.88	0.74	[2.53]
g001918	777	0.12	1.47	1.41	[1.04]	31.11	17.45	[1.78]
g000981	110	0.28	10.73	4.73	[2.27]	531.47	21.53	[24.68]
g001207	84	0.19	1.10	0.16	[6.88]	53.20	0.06	[962.38]
g000292	212	0.03	0.45	0.45	[1.01]	0.43	0.37	[1.14]
imgseg_271031	900	0.99	10.66	0.19	[55.94]	-	0.17	[∞]
imgseg_105019	3 548	0.93	234.01	22.68	[10.32]	f	13 748.62	[∞]
imgseg_35058	1 274	0.37	34.93	24.71	[1.41]	-	-	-
imgseg_374020	5 735	0.82	1 739.11	72.23	[24.08]	f	-	-
imgseg_106025	1 565	0.68	159.31	34.05	[4.68]	-	-	-

Finally, it is also noteworthy that we get significant improvements for the weighted instances from VLSI design and image segmentation. We also examined the performance of each individual reduction rule and noticed that this is solely due to reduction rule $1_{w=c}$. These findings could improve the work by de Sous et al. [dSHK13], which also affects the work by Dunning et al. [DGS18]. In conclusion, our novel reduction rules give us a simple but powerful tool for speeding up existing state-of-the-art approaches for computing maximum cuts. Moreover, as mentioned previously, even our simple weighted path compression by itself is able to have a significant impact.

5.4.3 Analysis on Large Instances

We now examine the performance of our reduction algorithm and its impact on existing heuristic approaches for large graph instances with up to millions of vertices. For this purpose, we compared the cut size over time achieved by LocalSolver and MqLib with and without our reductions. Note that we did not use BiqMac as it was not able to handle

Table 5.3: Time to compute a maximum cut on the original instance G and the reduced instance $\mathcal{K}$ for LocalSolver (LS) and BiqMac (BM). Times t_{Alg} are given in seconds. Reductions are accounted for within the timings for $\mathcal{K}$. Values in brackets provide the speedup and are derived from $t_{Alg}(G)/t_{Alg}(\mathcal{K})$. Times labeled with "-" exceeded the ten-hour time limit, and an "f" indicates the algorithm crashed. Weighted path compression by reduction rule 6 *is not* used at the end – the reduced graph is unweighted.

Graph	n	$e(G)$	$t_{\text{LS}}(G)$	$t_{\text{LS}}(\mathcal{K})$		$t_{\text{BM}}(G)$	$t_{\text{BM}}(\mathcal{K})$	
ca-CSphd	1 882	0.98	24.79	1.12	[22.23]	-	0.32	[∞]
ego-facebook	2 888	0.93	20.39	1.72	[11.83]	967.99	1.42	[682.04]
ENZYMES_g295	123	0.82	1.83	0.36	[5.09]	0.96	0.37	[2.60]
road-euroroad	1 174	0.69	-	-	-	-	-	-
bio-yeast	1 458	0.72	-	-	-	-	-	-
rt-twitter-copen	761	0.80	-	409.47	[∞]	-	101.14	[∞]
bio-diseasome	516	0.93	-	6.66	[∞]	-	0.35	[∞]
ca-netscience	379	0.67	-	4 116.61	[∞]	-	2.10	[∞]
soc-firm-hi-tech	33	0.30	4.92	2.34	[2.10]	0.29	0.31	[0.94]
g000302	317	0.10	0.71	0.50	[1.41]	1.28	0.89	[1.44]
g001918	777	0.06	1.67	1.51	[1.10]	14.90	11.69	[1.27]
g000981	110	0.22	11.32	1.97	[5.74]	0.98	0.44	[2.23]
g001207	84	0.17	1.56	0.15	[10.11]	0.47	0.37	[1.28]
g000292	212	0.01	0.69	0.51	[1.35]	0.56	0.62	[0.91]

instances with more than 3000 vertices. Our results using a three-hour time limit for each algorithm are given in Table 5.4. Furthermore, we present convergence plots in Figure 5.6.

First, we note that the time to compute the actual reduced graph is relatively small. In particular, we are able to compute a reduced instance for a graph with 14 million vertices and edges in just over six minutes. Furthermore, we achieve an efficiency between 0.18 and 0.91 across all tested instances. When looking at the convergence plots (Figure 5.6), we can observe that for multiple instances the additional preprocessing time of reductions is quickly compensated by a significantly steeper increase in cut size compared to the unreduced version. Additionally, for instances where a reduced instance can be computed very quickly, such as web-google, we find a better solution almost instantaneously. In contrast, for ca-coauthors-dblp reductions are not able to provide significant benefits. In general, the results achieved by reductions followed by the local search heuristic are *always* better than just using the local search heuristic alone. However, the final improvement on the size of the largest cut found by LocalSolver and MqLib is generally small for the given time limit of three hours.

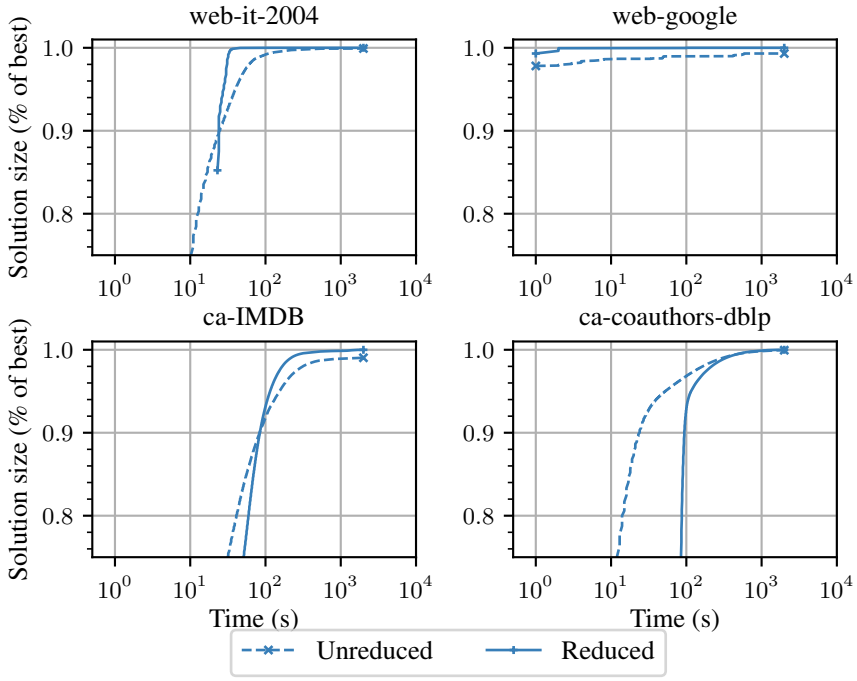


Figure 5.6: Convergence of LocalSolver on large instances. The dashed line represents the size of the cut for the unreduced graph, the full line does so for the reduced graph.

Table 5.4: Evaluation of large graph instances. A three-hour time limit was used, and five iterations were performed. t_{ker} is the time needed for computing a reduced instance. The columns Diff_{LS} and Diff_{MQ} indicate the percentage by which the size of the largest computed cut is larger on the reduced graph compared to the unreduced one for LocalSolver and MqLib, respectively.

Graph	n	d_{avg}	$e(G)$	$t_{\text{ker}}(G)$	Diff_{LS}	Diff_{MQ}
inf-road_central	14 081 816	1.20	0.59	362.32	inf%	2.70%
inf-power	4 941	1.33	0.62	0.04	1.64%	0.45%
web-google	1 299	2.13	0.79	0.01	0.69%	0.19%
ca-MathSciNet	332 689	2.47	0.63	8.02	1.33%	0.55%
ca-IMDB	896 305	4.22	0.42	27.55	0.97%	0.32%
web-Stanford	281 903	7.07	0.18	105.17	0.34%	0.30%
web-it-2004	509 338	14.09	0.91	22.10	0.08%	0.02%
ca-coauthors-dblp	540 486	28.20	0.25	72.39	0.05%	0.04%

5.5 Conclusion and Future Work

In this chapter, we proposed and engineered a new reduction algorithm for MaxCut. For this purpose, we presented reduction rules that are not reliant on specific subgraphs such as clique-forest and can thus be applied efficiently in practice. Note that these rules are able to subsume many of the reduction rules presented in previous works.

Our extensive experiments show that our reduction algorithm has a significant impact on handling MaxCut instances in practice. In particular, our experiments reveal that current state-of-the-art approaches can be sped up by up to *multiple orders of magnitude* when combined with our reduction algorithm. Since our reduction rules are often able to compute small reduced graphs, we are able to drastically increase the scale of instances these algorithms are able to handle. Finally, we evaluated the efficiency of the individual rules on different types of synthetic graphs. Thus, we were able to identify that one of our most simplistic rules often provides the largest impact on reduction efficiency.

Since our reduction algorithm makes use of different variants of MaxCut, including unweighted, weighted, and signed, new reduction rules for these problems can easily be integrated into it. Practically efficient reduction rules for WeightedMaxCut are of particular interest since they can be used to compute a weighted reduced graph directly without relying on unweighted and signed reduction rules. Furthermore, technical aspects of our reduction algorithm could be improved to further boost its performance. This could include an improved dependency checking mechanism across different reduction rules or a parallelization of their application.

Chapter 6

6 Conclusion

After presenting our contributions and results for the individual problems in the previous chapters, we now conclude with a brief and unified summary of our work. We also summarize general areas for promising future research on reductions in practice (and theory).

6.1 Summary

In this dissertation, we presented several contributions that make use of reductions to build and improve state-of-the-art algorithms for three different NP-hard optimization problems. This includes a variety of both heuristic and exact approaches that combine reductions with a wide spectrum of techniques, including graph partitioning, evolutionary algorithms, local search, branch-and-reduce, and algorithm portfolios. In addition, we provided important theoretical contributions by proposing new reduction and transformation rules for problems that have many applications, including map labeling [GNR16; Bar+16], vehicle routing [Don+22], social network analysis [Put+15], or disk scheduling [CKR11]. Particular attention was put on tailoring implementations of these rules that can be efficiently applied in practice.

We evaluated our algorithms on a large set of instances stemming from multiple domains and applications. In general, our experiments show that our algorithms are able to significantly increase both the scale and speed at which instances can be processed in practice (by up to orders of magnitude). This allows us to compute exact or high-quality solutions for many previously infeasible instances. We were also able to win the Parameterized Algorithms and Computational Experiments (PACE) 2019 Challenge [DFH19]. Additionally, to the best of our knowledge, our algorithms were the first to establish the use of certain concepts, such as inexact reductions, that have since been used in other state-of-the-art approaches [CLZ17; Zhe+20]. Finally, as our experiments with existing approaches indicate, our algorithms can easily be integrated into other algorithms to improve their performance.

Our algorithms are either available as standalone libraries or as part of the libraries KaMIS[1], WeGotYouCovered[2] (maximum cardinality and weight independent sets), and DMAX[3] (maximum cardinality cuts). The KaMIS library, in particular, has been deployed in multiple use-cases, including map labeling [Klu+19], DNA word design [LR21], microtransfer printing [FHL19], or satellite scheduling [EK20].

[1] https://karlsruhemis.github.io/
[2] https://github.com/KarlsruheMIS/pace-2019
[3] https://algo2.iti.kit.edu/schulz/maxcut/

6.2 Outlook

As already outlined in the conclusions for the individual problems, there exist several open problems that are prevalent across different problems. We now present a unified outlook for promising opportunities for these open problems.

Parallel algorithms that make use of reductions are still largely unexplored for multiple problems. This includes shared-memory algorithms similar to the reduction algorithm by Hespe et al. [HSS19] but also distributed approaches that pose their own set of problems. However, due to the locality of many reduction rules, i.e., they only need to explore small neighborhoods, we assume that an efficient distributed implementation that does not require an excessive amount of communication is possible.

The particular selection of reduction rules that perform well on specific instances also remains a largely open problem that has recently received more attention [SHG20]. We see two interesting opportunities: (1) which reduction rules can be applied *efficiently* on an instance to achieve a good balance between speed and quality, and (2) which reduction rules provide the largest reduction in size at the cost of increased running time. Machine learning provides an interesting possibility to tackle this issue. For example, graph characteristics such as the degree distribution or clustering coefficient could be used to train a classifier that picks an appropriate set of reduction rules. Closely related to this is the problem of determining what actually makes a reduced instance hard to handle for both heuristic and exact approaches.

We presented a set of transformations for maximum weight independent sets that is able to temporarily increase the graph size in favor of a potentially smaller graph in the long run. Due to the results achieved by this approach, we see an opportunity in exploring this technique for other problems. This could lead to a more general *branch-and-transform* framework that might be able to achieve better results than the currently prevalent branch-and-reduce algorithms, where reductions usually reduce the graph size.

Finally, reduction rules have been successfully used to improve theoretical running time bounds [HXC21]. Thus, it remains an interesting topic to analyze the theoretical benefit of newly proposed reductions.

Appendix

A Instance Details

We present the number of vertices n and edges m for the instances used in our experiments. We also provide sources and the specific sections they are used in.

Table 1: Properties of large web graphs.

Graph	n	m	Source	Section
eu-2005	862 664	16 138 486	[BV04; Bol+11]	3.2,3.3
dewiki-2013	1 532 354	33 093 029	[BV04; Bol+11]	3.3
hollywood-2011	2 180 759	114 492 816	[BV04; Bol+11]	3.3
orkut	3 072 441	117 185 082	[BV04; Bol+11]	3.3
enwiki-2013	4 206 785	91 939 728	[BV04; Bol+11]	3.3
ljournal-2008	5 363 260	49 514 271	[BV04; Bol+11]	3.3
uk-2002	18 520 486	261 787 258	[BV04; Bol+11]	3.2
wikilinks	25 890 800	543 159 884	[Kun13]	3.3
it-2004	41 291 594	1 027 474 947	[BV04; Bol+11]	3.2,3.3
sk-2005	50 636 154	1 810 063 330	[BV04; Bol+11]	3.2,3.3
uk-2007	105 896 555	3 301 876 564	[BV04; Bol+11]	3.2,3.3
webbase-2001	118 142 155	854 809 761	[BV04; Bol+11]	3.3

Table 2: Properties of SNAP instances.

Graph	n	m	Source	Section
as-skitter	1 696 415	11 095 298	[LK14]	3.5,4.2,4.3
ca-AstroPh	18 772	198 050	[LK14]	3.5,4.2,4.3
ca-CondMat	23 133	93 439	[LK14]	3.5,4.2,4.3
ca-GrQc	5 242	14 484	[LK14]	4.2,4.3
ca-HepPh	12 008	118 489	[LK14]	4.2,4.3
ca-HepTh	9 877	25 973	[LK14]	4.2,4.3
email-Enron	36 692	182 831	[LK14]	3.5,4.2,4.3
email-EuAll	265 214	420 045	[LK14]	3.5,4.2,4.3
musae-twitch_DE	9 498	153 138	[LK14]	3.5
musae-twitch_FR	6 549	112 666	[LK14]	3.5
musae-twitch_ES	4 648	59 382	[LK14]	3.5
musae-twitch_RU	4 385	37 304	[LK14]	3.5
musae-twitch_EN	7 126	35 324	[LK14]	3.5
musae-twitch_PT	1 912	31 299	[LK14]	3.5
musae-github	37 700	289 003	[LK14]	3.5
musae-facebook	22 470	171 002	[LK14]	3.5
deezer_europe	28 281	92 752	[LK14]	3.5
lastfm_asia	7 624	27 806	[LK14]	3.5
p2p-Gnutella04	10 876	39 994	[LK14]	3.5,4.2,4.3
p2p-Gnutella05	8 846	31 839	[LK14]	4.2,4.3
p2p-Gnutella06	8 717	31 525	[LK14]	3.5,4.2,4.3
p2p-Gnutella08	6 301	20 777	[LK14]	4.2,4.3
p2p-Gnutella09	8 114	26 013	[LK14]	3.5,4.2,4.3
p2p-Gnutella24	26 518	65 369	[LK14]	4.2,4.3
p2p-Gnutella25	22 687	54 705	[LK14]	3.5,4.2,4.3
p2p-Gnutella30	36 682	88 328	[LK14]	4.2,4.3
p2p-Gnutella31	62 586	147 892	[LK14]	4.2,4.3
roadNet-CA	1 965 206	2 766 607	[LK14]	4.2,4.3
roadNet-PA	1 088 092	1 541 898	[LK14]	4.2,4.3
roadNet-TX	1 379 917	1 921 660	[LK14]	4.2,4.3
soc-Epinions1	75 879	405 740	[LK14]	4.2,4.3
soc-LiveJournal1	4 847 571	42 851 237	[LK14]	3.5,4.2,4.3
soc-Slashdot0811	77 360	469 180	[LK14]	4.2,4.3
soc-Slashdot0902	82 168	504 230	[LK14]	4.2,4.3
soc-pokec-relationships	1 632 803	22 301 964	[LK14]	4.2,4.3
web-BerkStan	685 230	6 649 470	[LK14]	3.5,4.2,4.3
web-Google	875 713	4 322 051	[LK14]	3.5,4.2,4.3
web-NotreDame	325 729	1 090 108	[LK14]	3.5,4.2,4.3
web-Stanford	281 903	1 992 636	[LK14]	3.2,3.3,3.5,4.2,4.3
wiki-Talk	2 394 385	4 659 565	[LK14]	4.2,4.3
wiki-Vote	7 115	100 762	[LK14]	4.2,4.3
loc-Gowalla	196 591	950 327	[LK14]	3.2
youtube	1 134 890	2 987 623	[LK14]	3.3

Table 3: Properties of additional social networks.

Graph	n	m	Source	Section
enron	69 244	254 449	[RA15]	3.2
libimseti	220 970	17 233 144	[RA15]	3.2,3.3,3.5
citation	268 495	1 156 647	[Bad+18]	3.2
cnr-2000	325 557	2 738 969	[Bad+18]	3.2
google	356 648	2 093 324	[Bad+18]	3.2
baidu-relatedpages	415 641	2 374 044	[RA15]	3.5
petster-fs-dog	426 820	8 543 549	[RA15]	3.5
coPapers	434 102	16 036 720	[Bad+18]	3.2
skitter	554 930	5 797 663	[Bad+18]	3.2
amazon-2008	735 323	3 523 472	[Bad+18]	3.2,3.3
in-2004	1 382 908	13 591 473	[Bad+18]	3.2,3.5
as-Skitter-big	1 696 415	11 095 298	[Kun13]	3.2,3.3
hudong-internallink	1 984 484	14 428 382	[RA15]	3.5

Table 4: Properties of DIMACS instances.

Graph	n	m	Source	Section
C125.9	125	787	[JT96]	3.5
MANN_a27	378	702	[JT96]	3.5
MANN_a45	1 035	1 980	[JT96]	3.5
brock200_1	200	5 066	[JT96]	3.5
brock200_2	200	10 024	[JT96]	3.5
brock200_3	200	7 852	[JT96]	3.5
brock200_4	200	6 811	[JT96]	3.5
gen200_p0.9_44	200	1 990	[JT96]	3.5
gen200_p0.9_55	200	1 990	[JT96]	3.5
hamming8-4	256	11 776	[JT96]	3.5
johnson16-2-4	120	1 680	[JT96]	3.5
keller4	171	5 100	[JT96]	3.5
p_hat1000-1	1 000	377 247	[JT96]	3.5
p_hat1000-2	1 000	254 701	[JT96]	3.5
p_hat1500-1	1 500	839 327	[JT96]	3.5
p_hat300-1	300	33 917	[JT96]	3.5
p_hat300-2	300	22 922	[JT96]	3.5
p_hat300-3	300	11 460	[JT96]	3.5
p_hat500-1	500	93 181	[JT96]	3.5
p_hat500-2	500	61 804	[JT96]	3.5
p_hat500-3	500	30 950	[JT96]	3.5
p_hat700-1	700	183 651	[JT96]	3.5
p_hat700-2	700	122 922	[JT96]	3.5
san1000	1 000	249 000	[JT96]	3.5
san200_0.7_1	200	5 970	[JT96]	3.5
san200_0.7_2	200	5 970	[JT96]	3.5
san200_0.9_1	200	1 990	[JT96]	3.5
san200_0.9_2	200	1 990	[JT96]	3.5
san200_0.9_3	200	1 990	[JT96]	3.5
san400_0.5_1	400	39 900	[JT96]	3.5
san400_0.7_1	400	23 940	[JT96]	3.5
san400_0.7_2	400	23 940	[JT96]	3.5
san400_0.7_3	400	23 940	[JT96]	3.5
sanr200_0.7	200	6 032	[JT96]	3.5
sanr200_0.9	200	2 037	[JT96]	3.5
sanr400_0.5	400	39 816	[JT96]	3.5
sanr400_0.7	400	23 931	[JT96]	3.5

Table 5: Properties of road networks.

Graph	n	m	Source	Section
luxembourg	114 599	119 666	[DGJ09]	3.5
ny	264 346	733 846	[DGJ09]	3.2,3.5
bay	321 270	397 415	[DGJ09]	3.2,3.5
col	435 666	521 200	[DGJ09]	3.2,3.5
fla	1 070 376	1 343 951	[DGJ09]	3.2,3.5
europe	18 029 721	22 217 686	[Del+09]	3.2,3.3
USA-road	23 947 347	28 854 312	[DGJ09]	3.2,3.3

Table 6: Properties of OSM instances.

Graph	n	m	Source	Section
alabama-AM2	1 164	38 772	[Cai+18]	4.2,4.3
alabama-AM3	3 504	619 328	[Cai+18]	4.2,4.3
district-of-columbia-AM1	2 500	49 302	[Cai+18]	4.2,4.3
district-of-columbia-AM2	13 597	3 219 590	[Cai+18]	4.2,4.3
district-of-columbia-AM3	46 221	55 458 274	[Cai+18]	4.2,4.3
florida-AM2	1 254	33 872	[Cai+18]	4.2,4.3
florida-AM3	2 985	308 086	[Cai+18]	4.2,4.3
georgia-AM3	1 680	148 252	[Cai+18]	4.2,4.3
greenland-AM3	4 986	7 304 722	[Cai+18]	4.2,4.3
hawaii-AM2	2 875	530 316	[Cai+18]	4.2,4.3
hawaii-AM3	28 006	98 889 842	[Cai+18]	4.2,4.3
idaho-AM3	4 064	7 848 160	[Cai+18]	4.2,4.3
kansas-AM3	2 732	1 613 824	[Cai+18]	4.2,4.3
kentucky-AM2	2 453	1 286 856	[Cai+18]	4.2,4.3
kentucky-AM3	19 095	119 067 260	[Cai+18]	4.2,4.3
louisiana-AM3	1 162	74 154	[Cai+18]	4.2,4.3
maryland-AM3	1 018	190 830	[Cai+18]	4.2,4.3
massachusetts-AM2	1 339	70 898	[Cai+18]	4.2,4.3
massachusetts-AM3	3 703	1 102 982	[Cai+18]	4.2,4.3
mexico-AM3	1 096	94 262	[Cai+18]	4.2,4.3
new-hampshire-AM3	1 107	36 042	[Cai+18]	4.2,4.3
north-carolina-AM3	1 557	473 478	[Cai+18]	4.2,4.3
oregon-AM2	1 325	115 034	[Cai+18]	4.2,4.3
oregon-AM3	5 588	5 825 402	[Cai+18]	4.2,4.3
pennsylvania-AM3	1 148	52 928	[Cai+18]	4.2,4.3
rhode-island-AM2	2 866	590 976	[Cai+18]	4.2,4.3
rhode-island-AM3	15 124	25 244 438	[Cai+18]	4.2,4.3
utah-AM3	1 339	85 744	[Cai+18]	4.2,4.3
vermont-AM3	3 436	2 272 328	[Cai+18]	4.2,4.3
virginia-AM2	2 279	120 080	[Cai+18]	4.2,4.3
virginia-AM3	6 185	1 331 806	[Cai+18]	4.2,4.3
washington-AM2	3 025	304 898	[Cai+18]	4.2,4.3
washington-AM3	10 022	4 692 426	[Cai+18]	4.2,4.3
west-virginia-AM3	1 185	251 240	[Cai+18]	4.2,4.3

Table 7: Properties of mesh instances.

Graph	n	m	Source	Section
beethoven	4 419	6 491	[San+08]	3.2,4.3
blob	16 068	24 102	[San+08]	3.2,4.3
buddha	1 087 716	1 631 574	[San+08]	3.2,3.3,4.3
bunny	68 790	103 017	[San+08]	3.2,3.3,4.3
cow	5 036	7 366	[San+08]	3.2,4.3
dragon	150 000	225 000	[San+08]	3.2,3.3,4.3
dragonsub	600 000	900 000	[San+08]	3.2,3.3,4.3
ecat	684 496	1 026 744	[San+08]	3.2,4.3
face	22 871	34 054	[San+08]	3.2,4.3
fandisk	8 634	12 818	[San+08]	3.2,4.3
feline	41 262	61 893	[San+08]	3.2,3.3,4.3
gameguy	42 623	63 850	[San+08]	3.2,3.3,4.3
gargoyle	20 000	30 000	[San+08]	3.2,4.3
turtle	267 534	401 178	[San+08]	3.2,4.3
venus	5 672	8 508	[San+08]	3.2,3.3,4.3

Table 8: Properties of FE instances.

Graph	n	m	Source	Section
fe_4elt2	11 143	32 818	[SWC04]	4.3
fe_sphere	16 386	49 152	[SWC04]	4.3
fe_pwt	36 519	144 794	[SWC04]	3.2,4.3
fe_body	45 087	163 734	[SWC04]	4.3
fe_tooth	78 136	452 591	[SWC04]	3.2,4.3
fe_rotor	99 617	662 431	[SWC04]	3.2,4.3
fe_ocean	143 437	409 593	[SWC04]	3.2,4.3

Table 9: Properties of additional Walshaw benchmark instances.

Graph	n	m	Source	Section
crack	10 240	30 380	[SWC04]	3.2
vibrobox	12 328	165 250	[SWC04]	3.2
4elt	15 606	45 878	[SWC04]	3.2
cs4	22 499	43 858	[SWC04]	3.2
bcsstk30	28 924	1 007 284	[SWC04]	3.2
bcsstk31	35 588	572 914	[SWC04]	3.2
brack2	62 631	366 559	[SWC04]	3.2
598a	110 971	741 934	[SWC04]	3.2
wave	156 317	1 059 331	[SWC04]	3.2
auto	448 695	3 314 611	[SWC04]	3.2

Table 10: Properties of Florida Sparse Matrix collection instances.

Graph	n	m	Source	Section
Oregon-1	11 174	23 409	[DH11]	3.2
ca-HepPh	12 006	118 489	[DH11]	3.2
skirt	12 595	91 961	[DH11]	3.2
cbuckle	13 681	331 417	[DH11]	3.2
cyl6	13 681	350 280	[DH11]	3.2
case9	14 453	72 171	[DH11]	3.2
rajat07	14 842	24 571	[DH11]	3.2
Dubcova1	16 129	118 440	[DH11]	3.2
olafu	16 146	499 505	[DH11]	3.2
bodyy6	19 366	57 691	[DH11]	3.2
raefsky4	19 779	654 416	[DH11]	3.2
smt	25 710	1 863 737	[DH11]	3.2
pdb1HYS	36 417	2 154 174	[DH11]	3.2
c-57	37 833	183 682	[DH11]	3.2
copter2	55 476	352 238	[DH11]	3.2
TSOPF_FS_b300_c2	56 813	4 376 395	[DH11]	3.2
c-67	57 975	236 980	[DH11]	3.2
dixmaanl	60 000	119 999	[DH11]	3.2
blockqp1	60 012	300 011	[DH11]	3.2
Ga3As3H12	61 349	2 954 799	[DH11]	3.2
GaAsH6	61 349	1 660 230	[DH11]	3.2
cant	62 208	1 972 466	[DH11]	3.2
ncvxqp5	62 500	187 483	[DH11]	3.2
crankseg_2	63 838	7 042 510	[DH11]	3.2
c-68	64 810	565 996	[DH11]	3.2

Table 11: Properties of the first half of public PACE instances.

Graph	n	m	Source	Section
001	6 160	40 207	[DFH19]	3.4,3.5
003	60 541	74 220	[DFH19]	3.4,3.5
005	200	819	[DFH19]	3.4,3.5
007	8 794	10 130	[DFH19]	3.4,3.5
009	38 452	174 645	[DFH19]	3.4,3.5
011	9 877	25 973	[DFH19]	3.4,3.5
013	45 307	55 440	[DFH19]	3.4,3.5
015	53 610	65 952	[DFH19]	3.4,3.5
017	23 541	51 747	[DFH19]	3.4,3.5
019	200	884	[DFH19]	3.4,3.5
021	24 765	30 242	[DFH19]	3.4,3.5
023	27 717	133 665	[DFH19]	3.4,3.5
025	23 194	28 221	[DFH19]	3.4,3.5
027	65 866	81 245	[DFH19]	3.4,3.5
029	13 431	21 999	[DFH19]	3.4,3.5
031	200	813	[DFH19]	3.4,3.5
033	4 410	6 885	[DFH19]	3.4,3.5
035	200	884	[DFH19]	3.4,3.5
037	198	824	[DFH19]	3.4,3.5
039	6 795	10 620	[DFH19]	3.4,3.5
041	200	1 040	[DFH19]	3.4,3.5
043	200	841	[DFH19]	3.4,3.5
045	200	1 044	[DFH19]	3.4,3.5
047	200	1 120	[DFH19]	3.4,3.5
049	200	957	[DFH19]	3.4,3.5
051	200	1 135	[DFH19]	3.4,3.5
053	200	1 062	[DFH19]	3.4,3.5
055	200	958	[DFH19]	3.4,3.5
057	200	1 200	[DFH19]	3.4,3.5
059	200	988	[DFH19]	3.4,3.5
061	200	952	[DFH19]	3.4,3.5
063	200	1 040	[DFH19]	3.4,3.5
065	200	1 037	[DFH19]	3.4,3.5
067	200	1 201	[DFH19]	3.4,3.5
069	200	1 120	[DFH19]	3.4,3.5
071	200	984	[DFH19]	3.4,3.5
073	200	1 107	[DFH19]	3.4,3.5
075	26 300	41 500	[DFH19]	3.4,3.5
077	200	988	[DFH19]	3.4,3.5
079	26 300	41 500	[DFH19]	3.4,3.5
081	199	1 124	[DFH19]	3.4
083	200	1 215	[DFH19]	3.4
085	11 470	17 408	[DFH19]	3.4
087	13 590	21 240	[DFH19]	3.4
089	57 316	77 978	[DFH19]	3.4
091	200	1 196	[DFH19]	3.4
093	200	1 207	[DFH19]	3.4
095	15 783	24 663	[DFH19]	3.4
097	18 096	28 281	[DFH19]	3.4
099	26 300	41 500	[DFH19]	3.4

Table 12: Properties of the second half of public PACE instances.

Graph	n	m	Source	Section
101	26 300	41 500	[DFH19]	3.4
103	15 783	24 663	[DFH19]	3.4
105	26 300	41 500	[DFH19]	3.4
107	13 590	21 240	[DFH19]	3.4
109	66 992	90 970	[DFH19]	3.4
111	450	17 831	[DFH19]	3.4
113	26 300	41 500	[DFH19]	3.4
115	18 096	28 281	[DFH19]	3.4
117	18 096	28 281	[DFH19]	3.4
119	18 096	28 281	[DFH19]	3.4
121	18 096	28 281	[DFH19]	3.4
123	26 300	41 500	[DFH19]	3.4
125	26 300	41 500	[DFH19]	3.4
127	18 096	28 281	[DFH19]	3.4
129	15 783	24 663	[DFH19]	3.4
131	2 980	5 360	[DFH19]	3.4
133	15 783	24 663	[DFH19]	3.4
135	26 300	41 500	[DFH19]	3.4
137	26 300	41 500	[DFH19]	3.4
139	18 096	28 281	[DFH19]	3.4
141	18 096	28 281	[DFH19]	3.4
143	18 096	28 281	[DFH19]	3.4
145	18 096	28 281	[DFH19]	3.4
147	18 096	28 281	[DFH19]	3.4
149	26 300	41 500	[DFH19]	3.4
151	15 783	24 663	[DFH19]	3.4
153	29 076	45 570	[DFH19]	3.4
155	26 300	41 500	[DFH19]	3.4
157	2 980	5 360	[DFH19]	3.4
159	18 096	28 281	[DFH19]	3.4
161	138 141	227 241	[DFH19]	3.4
163	18 096	28 281	[DFH19]	3.4
165	18 096	28 281	[DFH19]	3.4
167	15 783	24 663	[DFH19]	3.4
169	4 768	8 576	[DFH19]	3.4
171	18 096	28 281	[DFH19]	3.4
173	56 860	77 264	[DFH19]	3.4
175	3 523	6 446	[DFH19]	3.4
177	5 066	9 112	[DFH19]	3.4
179	15 783	24 663	[DFH19]	3.4
181	18 096	28 281	[DFH19]	3.4
183	72 420	118 362	[DFH19]	3.4
185	3 523	6 446	[DFH19]	3.4
187	4 227	7 734	[DFH19]	3.4
189	7 400	13 600	[DFH19]	3.4
191	4 579	8 378	[DFH19]	3.4
193	7 030	12 920	[DFH19]	3.4
195	1 150	81 068	[DFH19]	3.4
197	1 534	127 011	[DFH19]	3.4
199	1 534	126 163	[DFH19]	3.4

Table 13: Properties of the first half of private PACE instances.

Graph	n	m	Source	Section
002	51 795	63 334	[DFH19]	3.4,3.5
004	8 114	26 013	[DFH19]	3.4,3.5
006	200	751	[DFH19]	3.4,3.5
008	7 537	72 833	[DFH19]	3.4,3.5
010	199	774	[DFH19]	3.4,3.5
012	53 444	68 044	[DFH19]	3.4,3.5
014	25 123	31 552	[DFH19]	3.4,3.5
016	153	802	[DFH19]	3.4,3.5
018	49 212	63 601	[DFH19]	3.4,3.5
020	57 287	71 155	[DFH19]	3.4,3.5
022	12 589	33 129	[DFH19]	3.4,3.5
024	7 620	47 293	[DFH19]	3.4,3.5
026	6 140	36 767	[DFH19]	3.4,3.5
028	54 991	67 000	[DFH19]	3.4,3.5
030	62 853	79 557	[DFH19]	3.4,3.5
032	1 490	2 680	[DFH19]	3.4,3.5
034	1 490	2 680	[DFH19]	3.4,3.5
036	26 300	41 500	[DFH19]	3.4,3.5
038	786	14 024	[DFH19]	3.4,3.5
040	210	625	[DFH19]	3.4,3.5
042	200	974	[DFH19]	3.4,3.5
044	200	1 186	[DFH19]	3.4,3.5
046	200	812	[DFH19]	3.4,3.5
048	200	1 052	[DFH19]	3.4,3.5
050	200	1 048	[DFH19]	3.4,3.5
052	200	1 019	[DFH19]	3.4,3.5
054	200	985	[DFH19]	3.4,3.5
056	200	1 117	[DFH19]	3.4,3.5
058	200	1 202	[DFH19]	3.4,3.5
060	200	1 147	[DFH19]	3.4,3.5
062	199	1 164	[DFH19]	3.4,3.5
064	200	1 071	[DFH19]	3.4,3.5
066	200	884	[DFH19]	3.4,3.5
068	200	983	[DFH19]	3.4,3.5
070	200	887	[DFH19]	3.4,3.5
072	200	1 204	[DFH19]	3.4,3.5
074	200	820	[DFH19]	3.4,3.5
076	26 300	41 500	[DFH19]	3.4,3.5
078	11 349	17 739	[DFH19]	3.4,3.5
080	26 300	41 500	[DFH19]	3.4,3.5
082	200	978	[DFH19]	3.4
084	13 590	21 240	[DFH19]	3.4
086	26 300	41 500	[DFH19]	3.4
088	26 300	41 500	[DFH19]	3.4
090	11 349	17 739	[DFH19]	3.4
092	450	17 794	[DFH19]	3.4
094	5 960	10 720	[DFH19]	3.4
096	26 300	41 500	[DFH19]	3.4
098	26 300	41 500	[DFH19]	3.4
100	26 300	41 500	[DFH19]	3.4

Table 14: Properties of the second half of private PACE instances.

Graph	n	m	Source	Section
102	26 300	41 500	[DFH19]	3.4
104	26 300	41 500	[DFH19]	3.4
106	2 980	5 360	[DFH19]	3.4
108	26 300	41 500	[DFH19]	3.4
110	98 128	161 357	[DFH19]	3.4
112	18 096	28 281	[DFH19]	3.4
114	15 783	24 663	[DFH19]	3.4
116	26 300	41 500	[DFH19]	3.4
118	26 300	41 500	[DFH19]	3.4
120	70 144	116 378	[DFH19]	3.4
122	26 300	41 500	[DFH19]	3.4
124	26 300	41 500	[DFH19]	3.4
126	18 096	28 281	[DFH19]	3.4
128	26 300	41 500	[DFH19]	3.4
130	26 300	41 500	[DFH19]	3.4
132	15 783	24 663	[DFH19]	3.4
134	26 300	41 500	[DFH19]	3.4
136	18 096	28 281	[DFH19]	3.4
138	18 096	28 281	[DFH19]	3.4
140	26 300	41 500	[DFH19]	3.4
142	2 980	5 360	[DFH19]	3.4
144	26 300	41 500	[DFH19]	3.4
146	26 300	41 500	[DFH19]	3.4
148	26 300	41 500	[DFH19]	3.4
150	26 300	41 500	[DFH19]	3.4
152	13 590	21 240	[DFH19]	3.4
154	15 783	24 663	[DFH19]	3.4
156	450	17 809	[DFH19]	3.4
158	15 783	24 663	[DFH19]	3.4
160	18 096	28 281	[DFH19]	3.4
162	50 635	83 075	[DFH19]	3.4
164	29 296	46 040	[DFH19]	3.4
166	3 278	5 896	[DFH19]	3.4
168	2 980	5 360	[DFH19]	3.4
170	15 783	24 663	[DFH19]	3.4
172	4 025	7 435	[DFH19]	3.4
174	2 980	5 360	[DFH19]	3.4
176	15 783	24 663	[DFH19]	3.4
178	18 096	28 281	[DFH19]	3.4
180	15 783	24 663	[DFH19]	3.4
182	26 300	41 500	[DFH19]	3.4
184	6 290	11 560	[DFH19]	3.4
186	26 300	41 500	[DFH19]	3.4
188	6 660	12 240	[DFH19]	3.4
190	3 875	7 090	[DFH19]	3.4
192	2 980	5 360	[DFH19]	3.4
194	1 150	80 851	[DFH19]	3.4
196	1 534	126 082	[DFH19]	3.4
198	1 150	80 072	[DFH19]	3.4
200	1 150	80 258	[DFH19]	3.4

Table 15: Properties of rudy MaxCut instances.

Graph	n	m	Source	Section
g05_100	100	2 475	[Wie18]	5.4
g05_60	60	885	[Wie18]	5.4
g05_80	80	1 580	[Wie18]	5.4
pm1d_100	100	4 901	[Wie18]	5.4
pm1d_80	80	3 128	[Wie18]	5.4
pm1s_100	100	495	[Wie18]	5.4
pm1s_80	79	316	[Wie18]	5.4
pw01_100	100	495	[Wie18]	5.4
pw05_100	100	2 475	[Wie18]	5.4
pw09_100	100	4 455	[Wie18]	5.4
w01_100	100	470	[Wie18]	5.4
w05_100	100	2 356	[Wie18]	5.4
w09_100	100	4 245	[Wie18]	5.4

Table 16: Properties of medium-sized MaxCut instances.

Graph	n	m	Source	Section
ca-CSphd	1 882	1 740	[RA15]	5.4
ego-facebook	2 888	2 981	[RA15]	5.4
ENZYMES_g295	123	278	[RA15]	5.4
road-euroroad	1 174	1 417	[RA15]	5.4
bio-yeast	1 458	1 948	[RA15]	5.4
rt-twitter-copen	761	1 029	[RA15]	5.4
bio-diseasome	516	1 188	[RA15]	5.4
ca-netscience	379	914	[RA15]	5.4
soc-firm-hi-tech	33	125	[RA15]	5.4
g000302	317	476	[DGS18]	5.4
g001918	777	1 239	[DGS18]	5.4
g000981	110	188	[DGS18]	5.4
g001207	84	149	[DGS18]	5.4
g000292	212	381	[DGS18]	5.4
imgseg_271031	900	1 027	[DGS18]	5.4
imgseg_105019	3 548	4 325	[DGS18]	5.4
imgseg_35058	1 274	1 806	[DGS18]	5.4
imgseg_374020	5 735	4 427	[DGS18]	5.4
imgseg_106025	1 565	2 629	[DGS18]	5.4

Table 17: Properties of large-sized MaxCut instances.

Graph	n	m	Source	Section
inf-road_central	14 081 816	16 933 413	[RA15]	5.4
inf-power	4 941	6 594	[RA15]	5.4
web-google	1 299	2 773	[RA15]	5.4
ca-MathSciNet	332 689	820 644	[RA15]	5.4
ca-IMDB	896 305	3 782 463	[RA15]	5.4
web-Stanford	281 903	1 992 636	[RA15]	5.4
web-it-2004	509 338	7 178 413	[RA15]	5.4
ca-coauthors-dblp	540 586	15 245 729	[RA15]	5.4

B Additional Results for Inexact Iterative Reductions

We present additional results for the comparison of the heuristic MIS algorithms ReduMIS, EvoMIS, and ARW (see Section 3.2.4.b)) on instances from the Florida Sparse Matrix collection. For each algorithm, we give the maximum and average solution sizes achieved over five runs with different random seeds. The global best solution is highlighted in bold.

Table 18: Results for graphs from Florida Sparse Matrix collection.

Graph			ReduMIS		EvoMIS		ARW	
Name	n	Opt.	Avg.	Max.	Avg.	Max.	Avg.	Max.
Oregon-1	11 174	9 512	9 512	**9 512**	9 512	**9 512**	9 512	**9 512**
ca-HepPh	12 006	4 994	4 994	**4 994**	4 994	**4 994**	4 994	**4 994**
skirt	12 595	2 383	2 383	**2 383**	2 383	**2 383**	2 383	**2 383**
cbuckle	13 681	1 097	1 097	**1 097**	1 097	**1 097**	1 097	**1 097**
cyl6	13 681	600	600	**600**	600	**600**	600	**600**
case9	14 453	7 224	7 224	**7 224**	7 224	**7 224**	7 224	**7 224**
rajat07	14 842	4 971	4 971	**4 971**	4 971	**4 971**	4 971	**4 971**
Dubcova1	16 129	4 096	4 096	**4 096**	4 096	**4 096**	4 096	**4 096**
olafu	16 146	735	735	**735**	735	**735**	735	**735**
bodyy6	19 366	-	6 229	6 232	6 232	**6 233**	6 226	6 228
raefsky4	19 779	1 055	1 055	**1 055**	1 055	**1 055**	1 053	1 053
smt	25 710	-	782	**782**	782	**782**	780	780
pdb1HYS	36 417	-	1 077	**1 078**	1 078	**1 078**	1 070	1 071
c-57	37 833	19 997	19 997	**19 997**	19 997	**19 997**	19 997	**19 997**
copter2	55 476	-	15 192	15 194	15 192	**15 195**	15 186	15 194
TSOPF_FS_b300_c2	56 813	28 338	28 338	**28 338**	28 338	**28 338**	28 338	**28 338**
c-67	57 975	31 257	31 257	**31 257**	31 257	**31 257**	31 257	**31 257**
dixmaanl	60 000	20 000	20 000	**20 000**	20 000	**20 000**	20 000	**20 000**
blockqp1	60 012	20 011	20 011	**20 011**	20 011	**20 011**	20 011	**20 011**
Ga3As3H12	61 349	-	8 068	8 132	7 839	**8 151**	8 061	8 124
GaAsH6	61 349	-	8 567	**8 589**	8 562	8 572	8 519	8 575
cant	62 208	-	6 260	**6 260**	6 260	**6 260**	6 255	6 255
ncvxqp5*	62 500	-	24 504	24 523	24 526	24 537	24 580	**24 608**
crankseg_2	63 838	1 735	1 735	**1 735**	1 735	**1 735**	1 735	**1 735**
c-68	64 810	36 546	36 546	**36 546**	36 546	**36 546**	36 546	**36 546**

C Convergence Plots for Inexact Iterative Reductions

We present additional convergence plots for the heuristic MIS algorithms ReduMIS, EvoMIS, and ARW used in our experimental evaluation in Section 3.2.4. The results for each algorithm are event-based geometric average values over five runs with different random seeds.

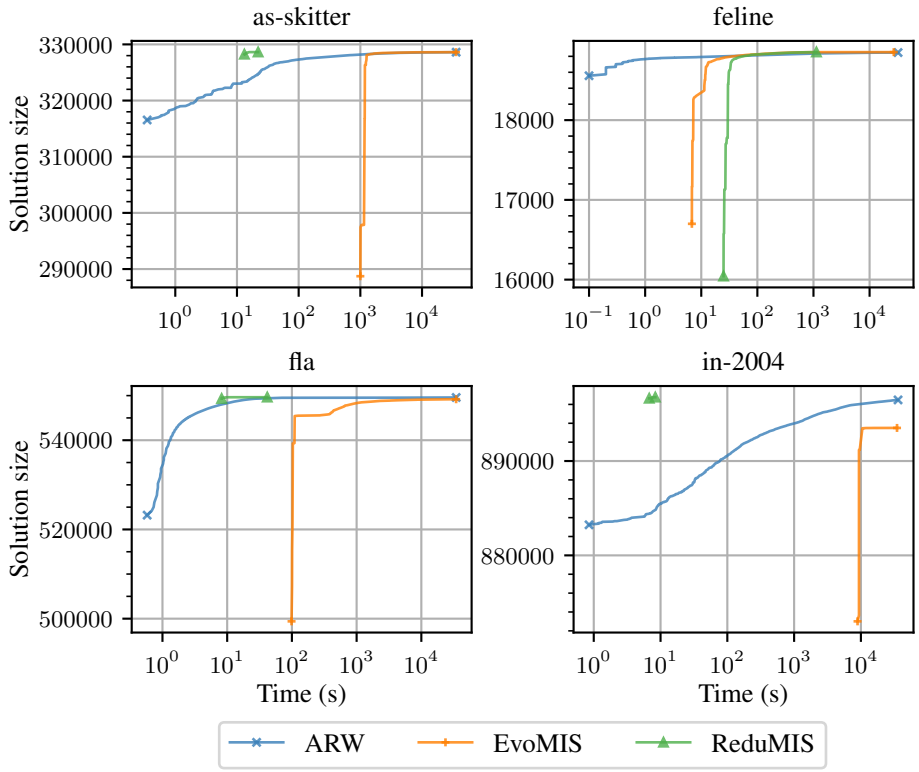


Figure 1: Convergence plots for as-skitter (top left), feline (top right), fla (bottom left), and in-2004 (bottom right).

D Convergence Plots for On-the-fly Reductions

In the following, we present additional convergence plots for our experimental evaluation in Section 3.3.2. This includes the heuristic MIS algorithms ReduMIS, KerMIS, ARW, and OnlineMIS. The results for each algorithm are event-based geometric average values over three runs with different random seeds.

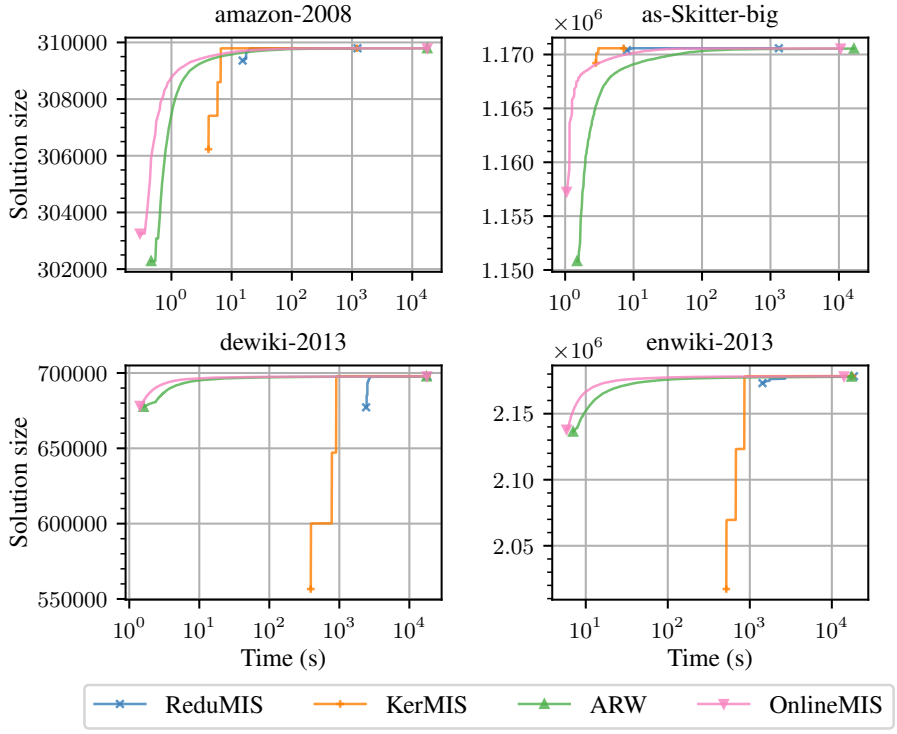


Figure 2: Convergence plots for amazon-2008 (top left), as-Skitter-big (top right), dewiki-2013 (bottom left), and enwiki-2013 (bottom right).

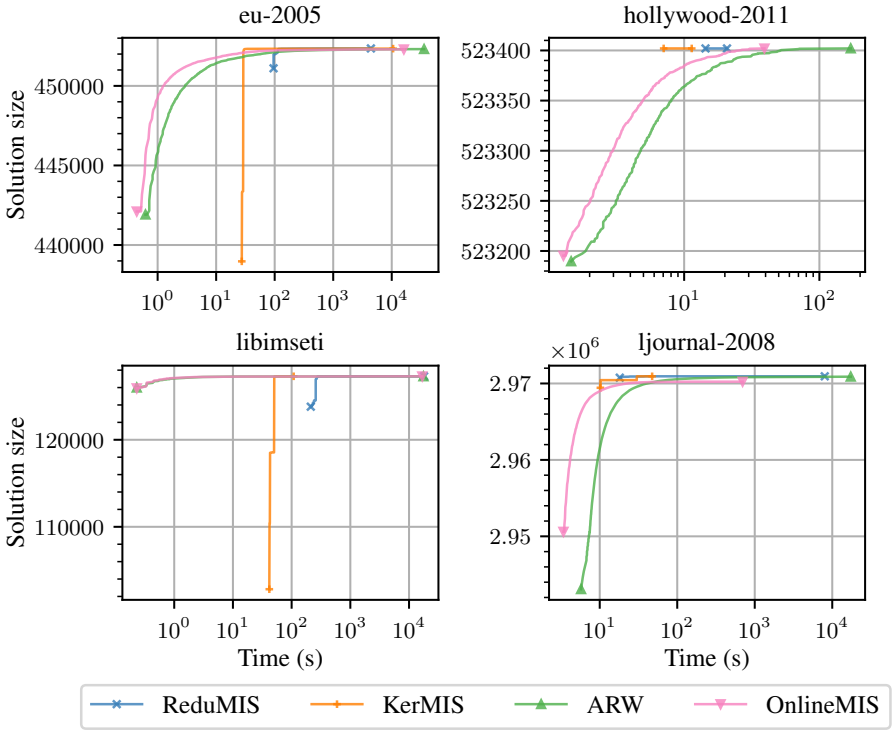


Figure 3: Convergence plots for eu-2005 (top left), hollywood-2011 (top right), libimseti (bottom left), and ljournal-2008 (bottom right).

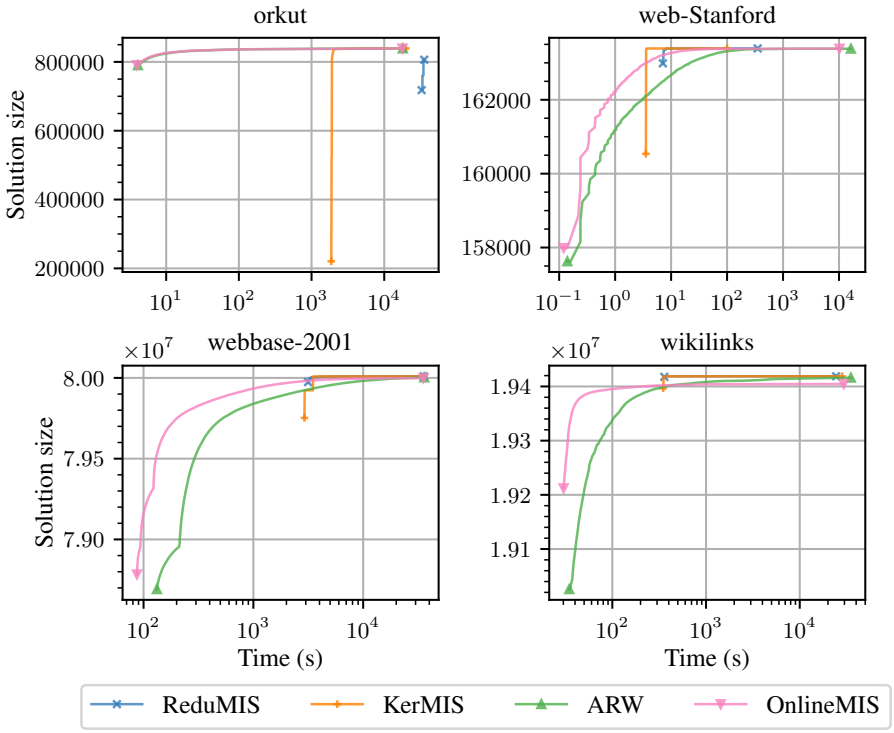


Figure 4: Convergence plots for orkut (top left), web-Stanford (top right), webbase-2001 (bottom left), and wikilinks (bottom right).

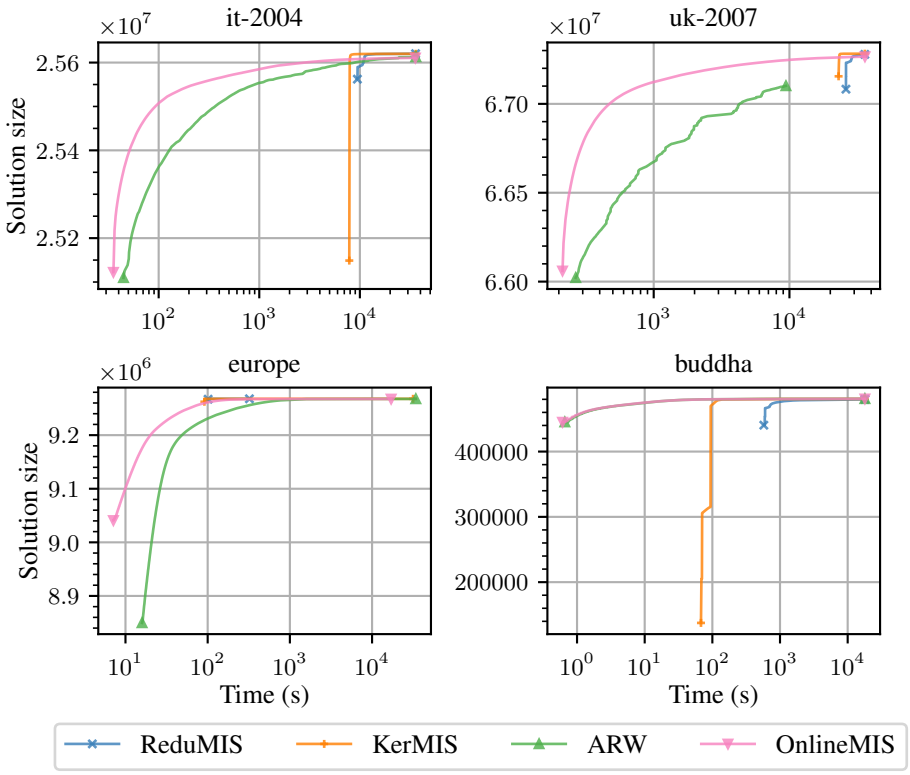


Figure 5: Convergence plots for it-2004 (top left), uk-2007 (top right), europe (bottom left), and buddha (bottom right).

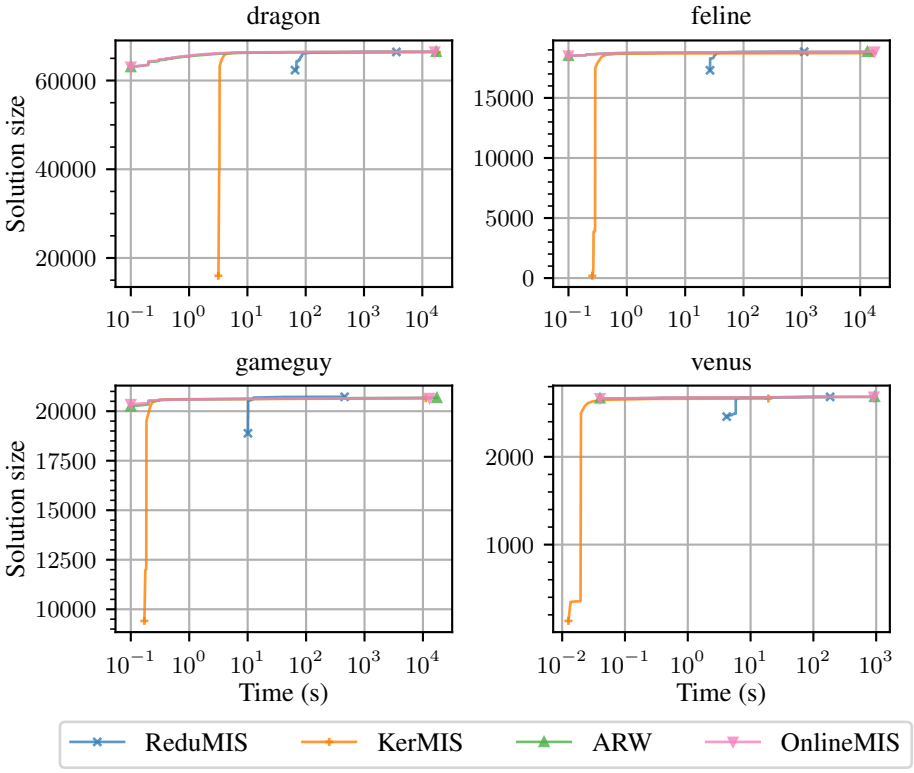


Figure 6: Convergence plots for dragon (top left), feline (top right), gameguy (bottom left), and venus (bottom right).

E Detailed Results for Targeted Branching Rules

In the following, we present detailed results of our experimental evaluation of branching strategies for exact MIS algorithms (Section 3.5.4). The tables show arithmetic means of running times t (in seconds) over three runs with different random seeds and the speedup s. Speedups are computed by dividing the running time of maximum degree branching by the running time of the respective technique. Timeouts are assigned a running time of ten hours. We also present the aggregated speedup s_{total} computed by dividing the running time of both algorithms over all instances (omitting instances where both algorithms do not finish within our time limit). A value is highlighted in bold if it is the best one within a row.

Table 19: Detailed results for our decomposition-based strategies on sparse networks.

Graph	max. deg.	articulation	edge cuts	nested dis.
Sparse net.	t	t (s)	t (s)	t (s)
as-skitter	**2 058.32**	2 100.57 (0.98)	2 071.06 (0.99)	2 068.46 (1.00)
baidu-relatedpages	**0.82**	0.88 (0.94)	0.86 (0.96)	7.22 (0.11)
bay	1.68	1.87 (0.90)	**1.31 (1.28)**	23.43 (0.07)
col	5 019.93	4 737.48 (1.06)	**3 872.65 (1.30)**	5 101.46 (0.98)
fla	25.33	**23.47 (1.08)**	24.58 (1.03)	329.42 (0.08)
hudong-internallink	**0.99**	1.55 (0.64)	1.46 (0.68)	1.99 (0.50)
in-2004	**5.22**	5.46 (0.96)	5.37 (0.97)	16.18 (0.32)
libimseti	**1 497.59**	1 507.54 (0.99)	1 503.49 (1.00)	1 704.53 (0.88)
musae-twitch_DE	**20 906.93**	21 470.00 (0.97)	20 987.30 (1.00)	20 949.83 (1.00)
musae-twitch_FR	**37.13**	37.81 (0.98)	37.32 (1.00)	41.55 (0.89)
petster-fs-dog	**6.82**	10.21 (0.67)	8.67 (0.79)	12.47 (0.55)
soc-LiveJournal1	**9.87**	11.50 (0.86)	11.06 (0.89)	23.91 (0.41)
web-BerkStan	**134.22**	360.88 (0.37)	138.84 (0.97)	207.92 (0.65)
web-Google	**0.61**	0.85 (0.71)	0.68 (0.89)	1.46 (0.41)
web-NotreDame	12.10	**9.07 (1.33)**	12.11 (1.00)	48.83 (0.25)
web-Stanford	>36 000	**8.38 (>4 294.84)**	27.41 (>1 313.18)	42.80 (>841.16)
s_{total}	1.00	2.17	**2.29**	2.15

Table 20: Detailed results for our decomposition-based strategies on PACE instances.

Graph	max. deg.	articulation	edge cuts	nested dis.
PACE	t	t (s)	t (s)	t (s)
05	**1.97**	2.00 (0.98)	2.00 (0.99)	2.44 (0.81)
06	**0.85**	0.87 (0.98)	0.87 (0.98)	1.33 (0.64)
10	**2.24**	2.27 (0.99)	2.26 (0.99)	2.66 (0.84)
16	25 836.77	26 175.23 (0.99)	**25 763.30 (1.00)**	25 865.40 (1.00)
19	**3.17**	3.22 (0.98)	3.18 (0.99)	3.63 (0.87)
31	**74.37**	76.03 (0.98)	75.45 (0.99)	74.82 (0.99)
33	**1.01**	1.03 (0.98)	1.02 (0.99)	40.09 (0.03)
35	**7.64**	7.84 (0.97)	7.77 (0.98)	8.13 (0.94)
36	**1.84**	1.87 (0.98)	1.85 (0.99)	3.93 (0.47)
37	**10.27**	10.48 (0.98)	10.47 (0.98)	10.75 (0.96)
38	12.33	11.24 (1.10)	**3.25 (3.79)**	15.35 (0.80)
39	**93.79**	96.82 (0.97)	95.96 (0.98)	95.21 (0.99)
40	**4 690.64**	4 794.37 (0.98)	4 758.15 (0.99)	4 712.57 (1.00)
41	**48.56**	49.84 (0.97)	49.39 (0.98)	49.35 (0.98)
42	**37.32**	38.11 (0.98)	37.91 (0.98)	37.87 (0.99)
43	**175.11**	178.81 (0.98)	177.26 (0.99)	175.24 (1.00)
44	**92.90**	95.13 (0.98)	94.28 (0.99)	93.40 (0.99)
45	**25.41**	26.01 (0.98)	25.73 (0.99)	25.90 (0.98)
46	**109.55**	111.95 (0.98)	111.00 (0.99)	110.22 (0.99)
47	**58.47**	59.70 (0.98)	59.38 (0.98)	59.22 (0.99)
48	**25.28**	25.80 (0.98)	25.60 (0.99)	25.80 (0.98)
49	**17.80**	18.19 (0.98)	18.10 (0.98)	18.30 (0.97)
50	**48.87**	50.01 (0.98)	49.56 (0.99)	49.40 (0.99)
51	**56.70**	58.00 (0.98)	57.63 (0.98)	57.52 (0.99)
52	**22.16**	22.68 (0.98)	22.53 (0.98)	22.69 (0.98)
53	**59.88**	61.42 (0.97)	60.77 (0.99)	60.42 (0.99)
54	**32.08**	32.89 (0.98)	32.73 (0.98)	32.67 (0.98)
55	**6.83**	6.97 (0.98)	6.92 (0.99)	7.32 (0.93)
56	**97.00**	99.09 (0.98)	98.31 (0.99)	97.80 (0.99)
57	**66.01**	67.76 (0.97)	67.18 (0.98)	66.83 (0.99)
58	**48.12**	48.83 (0.99)	48.72 (0.99)	48.63 (0.99)
59	**13.30**	13.60 (0.98)	13.54 (0.98)	13.80 (0.96)
60	**79.56**	81.58 (0.98)	80.94 (0.98)	80.23 (0.99)
61	**21.91**	22.31 (0.98)	22.26 (0.98)	22.36 (0.98)
62	**66.22**	68.48 (0.97)	67.40 (0.98)	66.80 (0.99)
63	**69.06**	70.55 (0.98)	69.91 (0.99)	69.35 (1.00)
64	**29.58**	30.07 (0.98)	29.99 (0.99)	30.09 (0.98)
65	**36.84**	37.53 (0.98)	37.28 (0.99)	37.29 (0.99)
66	**8.06**	8.28 (0.97)	8.23 (0.98)	8.63 (0.93)
67	**122.74**	124.79 (0.98)	124.25 (0.99)	123.38 (0.99)
68	**8.79**	8.92 (0.99)	8.86 (0.99)	9.24 (0.95)
69	**43.11**	44.13 (0.98)	43.85 (0.98)	43.63 (0.99)
70	**11.79**	12.00 (0.98)	11.97 (0.99)	12.25 (0.96)
71	**36.20**	36.83 (0.98)	36.66 (0.99)	36.64 (0.99)
72	**46.44**	47.47 (0.98)	46.91 (0.99)	46.86 (0.99)
73	**43.02**	44.07 (0.98)	43.77 (0.98)	43.65 (0.99)
74	**7.06**	7.24 (0.97)	7.14 (0.99)	7.49 (0.94)
77	**13.30**	13.65 (0.97)	13.51 (0.98)	13.79 (0.96)
s_{total}	**1.00**	0.99	1.00	1.00

Table 21: Detailed results for our decomposition-based strategies on DIMACS instances.

Graph	max. deg.	articulation	edge cuts	nested dis.
DIMACS	t	t (s)	t (s)	t (s)
C125.9	**0.98**	1.01 (0.97)	1.00 (0.98)	1.43 (0.69)
MANN_a27	**0.48**	0.49 (0.98)	0.49 (0.98)	0.98 (0.49)
MANN_a45	**73.80**	75.24 (0.98)	74.93 (0.98)	74.70 (0.99)
brock200_1	**137.34**	140.20 (0.98)	137.56 (1.00)	140.01 (0.98)
brock200_2	**4.59**	4.69 (0.98)	4.70 (0.98)	10.07 (0.46)
brock200_3	22.06	22.33 (0.99)	**21.92 (1.01)**	26.39 (0.84)
brock200_4	**28.34**	28.72 (0.99)	28.35 (1.00)	32.48 (0.87)
gen200_p0.9_44	**152.61**	156.30 (0.98)	154.50 (0.99)	153.49 (0.99)
gen200_p0.9_55	**131.24**	134.64 (0.97)	133.04 (0.99)	132.58 (0.99)
hamming8-4	**19.29**	19.65 (0.98)	19.49 (0.99)	25.38 (0.76)
johnson16-2-4	**39.87**	41.17 (0.97)	40.21 (0.99)	40.33 (0.99)
keller4	**2.62**	2.68 (0.98)	2.65 (0.99)	4.37 (0.60)
p_hat1000-1	**860.24**	868.71 (0.99)	870.04 (0.99)	906.24 (0.95)
p_hat1000-2	**33 035.45**	33 656.50 (0.98)	33 508.10 (0.99)	33 247.45 (0.99)
p_hat1500-1	**8 935.77**	9 015.15 (0.99)	9 015.74 (0.99)	8 994.28 (0.99)
p_hat300-1	**3.70**	3.79 (0.98)	3.82 (0.97)	23.94 (0.15)
p_hat300-2	**5.53**	5.66 (0.98)	5.63 (0.98)	21.76 (0.25)
p_hat300-3	189.58	191.06 (0.99)	**188.96 (1.00)**	196.89 (0.96)
p_hat500-1	**38.63**	39.26 (0.98)	39.41 (0.98)	59.29 (0.65)
p_hat500-2	**96.36**	97.82 (0.99)	97.58 (0.99)	107.29 (0.90)
p_hat500-3	**14 860.70**	14 895.15 (1.00)	14 979.65 (0.99)	14 909.35 (1.00)
p_hat700-1	163.30	**162.84 (1.00)**	163.17 (1.00)	177.34 (0.92)
p_hat700-2	**906.32**	917.87 (0.99)	914.96 (0.99)	917.50 (0.99)
san1000	**895.34**	902.64 (0.99)	903.38 (0.99)	920.28 (0.97)
san200_0.7_1	**10.85**	11.01 (0.98)	10.90 (1.00)	14.45 (0.75)
san200_0.7_2	0.33	0.34 (0.95)	**0.32 (1.01)**	2.34 (0.14)
san200_0.9_1	**13.93**	14.37 (0.97)	14.08 (0.99)	14.94 (0.93)
san200_0.9_2	**34.15**	34.77 (0.98)	34.35 (0.99)	34.90 (0.98)
san200_0.9_3	**1 069.00**	1 094.54 (0.98)	1 078.09 (0.99)	1 071.31 (1.00)
san400_0.5_1	**9.21**	9.35 (0.98)	9.36 (0.98)	16.76 (0.55)
san400_0.7_1	**1 125.52**	1 139.20 (0.99)	1 131.38 (0.99)	1 130.07 (1.00)
san400_0.7_2	3 062.38	**3 053.97 (1.00)**	3 083.59 (0.99)	3 073.66 (1.00)
san400_0.7_3	**4 411.82**	4 464.53 (0.99)	4 447.19 (0.99)	4 423.16 (1.00)
sanr200_0.7	**48.35**	49.51 (0.98)	48.71 (0.99)	52.13 (0.93)
sanr200_0.9	**679.25**	696.41 (0.98)	688.51 (0.99)	680.29 (1.00)
sanr400_0.5	**373.40**	374.20 (1.00)	374.26 (1.00)	380.08 (0.98)
sanr400_0.7	**29 766.80**	30 390.80 (0.98)	30 270.10 (0.98)	30 001.55 (0.99)
s_{total}	**1.00**	0.99	0.99	0.99

Table 22: Detailed results for our reduction-based strategies on sparse networks.

Graph	max. deg.	Twin	Funnel	Unconfined	Packing	Combined
Sparse net.	t	t (s)	t (s)	t (s)	t (s)	t (s)
as-skitter	2 058.32	2 054.41 (1.00)	1 849.79 (1.11)	1 977.94 (1.04)	**1 681.87 (1.22)**	1 704.73 (1.21)
baidu-relatedpages	0.82	**0.80 (1.02)**	0.84 (0.97)	0.85 (0.97)	0.83 (0.98)	0.93 (0.88)
bay	**1.68**	1.68 (1.00)	8.22 (0.20)	4.71 (0.36)	1.89 (0.89)	8.38 (0.20)
col	**5 019.93**	5 752.08 (0.87)	5 416.72 (0.93)	8 187.80 (0.61)	9 370.05 (0.54)	5 924.10 (0.85)
fla	**25.33**	25.41 (1.00)	45.62 (0.56)	76.60 (0.33)	34.78 (0.73)	42.75 (0.59)
hudong-internallink	**0.99**	1.31 (0.76)	1.27 (0.78)	1.21 (0.82)	1.55 (0.64)	1.12 (0.88)
in-2004	5.22	**4.88 (1.07)**	5.25 (0.99)	10.85 (0.48)	5.50 (0.95)	10.73 (0.49)
libimseti	1 497.59	1 452.17 (1.03)	1 620.09 (0.92)	**1 440.71 (1.04)**	1 476.25 (1.01)	1 706.07 (0.88)
musae-twitch_DE	20 906.93	20 996.87 (1.00)	21 190.67 (0.99)	22 650.53 (0.92)	**19 345.03 (1.08)**	23 006.50 (0.91)
musae-twitch_FR	37.13	37.04 (1.00)	38.58 (0.96)	41.15 (0.90)	**35.60 (1.04)**	42.46 (0.87)
petster-fs-dog	6.82	**6.62 (1.03)**	8.16 (0.84)	8.66 (0.79)	9.68 (0.70)	9.20 (0.74)
soc-LiveJournal1	9.87	**6.64 (1.49)**	9.57 (1.03)	9.49 (1.04)	11.33 (0.87)	10.69 (0.92)
web-BerkStan	134.22	135.47 (0.99)	**122.30 (1.10)**	146.94 (0.91)	123.60 (1.09)	174.07 (0.77)
web-Google	0.61	**0.53 (1.15)**	0.69 (0.87)	0.68 (0.89)	0.78 (0.78)	0.68 (0.89)
web-NotreDame	**12.10**	12.63 (0.96)	15.23 (0.79)	12.38 (0.98)	14.09 (0.86)	17.52 (0.69)
web-Stanford	>36 000	>36 000	>36 000	>36 000	**17 886.35 (>2.01)**	17 989.97 (>2.00)
s_{total}	1.00	0.97	0.98	0.86	**1.31**	1.30

Table 23: Detailed results for our reduction-based strategies on PACE instances.

Graph	max. deg.	Twin	Funnel	Unconfined	Packing	Combined
PACE	t	t (s)	t (s)	t (s)	t (s)	t (s)
05	1.97	1.96 (1.01)	1.99 (0.99)	2.04 (0.97)	**1.66 (1.19)**	2.11 (0.93)
06	0.85	0.85 (1.00)	0.74 (1.15)	0.92 (0.92)	**0.67 (1.27)**	0.81 (1.05)
10	2.24	2.23 (1.01)	2.23 (1.00)	2.32 (0.97)	**1.88 (1.19)**	2.06 (1.09)
16	25 836.77	25 856.57 (1.00)	22 446.13 (1.15)	34 642.13 (0.75)	**18 511.88 (1.40)**	22 590.78 (1.14)
19	3.17	3.14 (1.01)	2.90 (1.09)	3.25 (0.98)	**2.60 (1.22)**	3.04 (1.04)
31	74.37	74.31 (1.00)	58.14 (1.28)	73.23 (1.02)	55.99 (1.33)	**54.11 (1.37)**
33	1.01	**1.01 (1.00)**	1.15 (0.88)	1.14 (0.89)	1.02 (0.99)	1.29 (0.79)
35	7.64	7.63 (1.00)	7.37 (1.04)	7.90 (0.97)	**6.54 (1.17)**	7.75 (0.99)
36	**1.84**	1.86 (0.99)	11.44 (0.16)	162.22 (0.01)	1.90 (0.97)	75.52 (0.02)
37	10.27	10.31 (1.00)	10.27 (1.00)	10.63 (0.97)	**8.21 (1.25)**	10.90 (0.94)
38	12.33	12.36 (1.00)	11.08 (1.11)	11.40 (1.08)	11.44 (1.08)	**10.05 (1.23)**
39	93.79	93.99 (1.00)	**32.43 (2.89)**	127.32 (0.74)	93.99 (1.00)	98.25 (0.95)
40	4 690.64	4 689.28 (1.00)	4 285.37 (1.09)	4 530.07 (1.04)	4 176.59 (1.12)	**4 131.79 (1.14)**
41	48.56	48.42 (1.00)	42.00 (1.16)	48.66 (1.00)	**36.87 (1.32)**	38.74 (1.25)
42	37.32	37.19 (1.00)	35.69 (1.05)	37.60 (0.99)	**28.55 (1.31)**	36.07 (1.03)
43	175.11	174.63 (1.00)	158.08 (1.11)	172.91 (1.01)	**130.75 (1.34)**	154.96 (1.13)
44	92.90	92.97 (1.00)	82.64 (1.12)	94.37 (0.98)	**69.68 (1.33)**	90.20 (1.03)
45	25.41	25.37 (1.00)	25.29 (1.01)	26.20 (0.97)	**19.83 (1.28)**	26.38 (0.96)
46	109.55	109.47 (1.00)	92.61 (1.18)	108.01 (1.01)	**79.76 (1.37)**	82.72 (1.32)
47	58.47	58.18 (1.00)	53.01 (1.10)	59.16 (0.99)	**42.32 (1.38)**	52.28 (1.12)
48	25.28	25.21 (1.00)	22.65 (1.12)	25.72 (0.98)	**18.56 (1.36)**	22.93 (1.10)
49	17.80	17.76 (1.00)	16.43 (1.08)	19.02 (0.94)	**12.97 (1.37)**	16.18 (1.10)
50	48.87	48.90 (1.00)	46.07 (1.06)	49.75 (0.98)	**37.70 (1.30)**	47.09 (1.04)
51	56.70	56.58 (1.00)	51.45 (1.10)	57.63 (0.98)	**43.45 (1.31)**	50.32 (1.13)
52	22.16	22.12 (1.00)	20.56 (1.08)	22.99 (0.96)	**15.78 (1.40)**	20.82 (1.06)
53	59.88	59.88 (1.00)	54.78 (1.09)	60.43 (0.99)	**46.87 (1.28)**	55.74 (1.07)
54	32.08	32.02 (1.00)	29.29 (1.10)	32.89 (0.98)	**26.55 (1.21)**	27.76 (1.16)
55	6.83	6.80 (1.00)	6.50 (1.05)	6.99 (0.98)	**5.23 (1.31)**	6.35 (1.08)
56	97.00	96.45 (1.01)	88.78 (1.09)	98.09 (0.99)	**70.18 (1.38)**	81.46 (1.19)
57	66.01	65.97 (1.00)	57.60 (1.15)	65.90 (1.00)	**49.95 (1.32)**	52.45 (1.26)
58	48.12	47.74 (1.01)	45.82 (1.05)	48.56 (0.99)	**35.94 (1.34)**	46.62 (1.03)
59	13.30	13.30 (1.00)	12.73 (1.04)	13.72 (0.97)	**10.61 (1.25)**	12.30 (1.08)
60	79.56	79.36 (1.00)	71.73 (1.11)	80.70 (0.99)	**59.65 (1.33)**	71.85 (1.11)
61	21.91	21.91 (1.00)	20.47 (1.07)	22.28 (0.98)	**17.50 (1.25)**	21.06 (1.04)
62	66.22	66.18 (1.00)	59.16 (1.12)	67.83 (0.98)	**49.87 (1.33)**	59.64 (1.11)
63	69.06	68.81 (1.00)	61.23 (1.13)	70.81 (0.98)	**53.40 (1.29)**	58.65 (1.18)
64	29.58	29.38 (1.01)	26.96 (1.10)	29.46 (1.00)	**22.35 (1.32)**	26.78 (1.10)
65	36.84	36.72 (1.00)	33.42 (1.10)	37.93 (0.97)	**28.23 (1.30)**	31.17 (1.18)
66	8.06	8.06 (1.00)	7.47 (1.08)	8.21 (0.98)	**6.21 (1.30)**	7.97 (1.01)
67	122.74	122.34 (1.00)	113.33 (1.08)	123.58 (0.99)	**95.55 (1.28)**	112.43 (1.09)
68	8.79	8.75 (1.00)	8.92 (0.99)	8.94 (0.98)	**6.69 (1.31)**	8.57 (1.03)
69	43.11	43.11 (1.00)	38.46 (1.12)	44.18 (0.98)	**33.88 (1.27)**	39.86 (1.08)
70	11.79	11.73 (1.00)	10.09 (1.17)	12.22 (0.96)	**9.71 (1.21)**	9.76 (1.21)
71	36.20	35.91 (1.01)	32.22 (1.12)	35.37 (1.02)	**27.23 (1.33)**	33.39 (1.08)
72	46.44	46.18 (1.01)	41.66 (1.11)	46.68 (0.99)	**36.28 (1.28)**	41.86 (1.11)
73	43.02	43.00 (1.00)	40.38 (1.07)	43.77 (0.98)	**31.91 (1.35)**	43.51 (0.99)
74	7.06	7.06 (1.00)	6.67 (1.06)	7.86 (0.90)	**5.48 (1.29)**	6.96 (1.01)
77	13.30	13.25 (1.00)	12.74 (1.04)	13.80 (0.96)	**10.61 (1.25)**	12.31 (1.08)
s_{total}	1.00	1.00	1.14	0.79	**1.34**	1.14

Table 24: Detailed results for our reduction-based strategies on DIMACS instances.

Graph	max. deg.	Twin	Funnel	Unconfined	Packing	Combined
DIMACS	t	t (s)	t (s)	t (s)	t (s)	t (s)
C125.9	0.98	0.98 (1.00)	0.92 (1.07)	0.98 (1.00)	**0.85 (1.15)**	0.91 (1.08)
MANN_a27	0.48	0.48 (1.00)	0.57 (0.85)	0.52 (0.92)	**0.48 (1.01)**	0.59 (0.82)
MANN_a45	73.80	73.76 (1.00)	83.81 (0.88)	78.58 (0.94)	**71.86 (1.03)**	85.47 (0.86)
brock200_1	137.34	136.98 (1.00)	140.15 (0.98)	137.32 (1.00)	**135.14 (1.02)**	138.64 (0.99)
brock200_2	4.59	4.60 (1.00)	4.71 (0.98)	4.59 (1.00)	**4.58 (1.00)**	4.70 (0.98)
brock200_3	22.06	21.78 (1.01)	22.38 (0.99)	21.85 (1.01)	**21.76 (1.01)**	22.46 (0.98)
brock200_4	28.34	**28.15 (1.01)**	29.09 (0.97)	28.16 (1.01)	28.25 (1.00)	29.24 (0.97)
gen200_p0.9_44	152.61	152.40 (1.00)	136.94 (1.11)	169.47 (0.90)	**132.81 (1.15)**	149.63 (1.02)
gen200_p0.9_55	131.24	131.20 (1.00)	125.61 (1.04)	127.51 (1.03)	102.10 (1.29)	**50.64 (2.59)**
hamming8-4	19.29	19.30 (1.00)	19.78 (0.98)	**19.12 (1.01)**	19.35 (1.00)	19.67 (0.98)
johnson16-2-4	39.87	39.79 (1.00)	41.63 (0.96)	41.40 (0.96)	**38.70 (1.03)**	43.09 (0.93)
keller4	2.62	2.62 (1.00)	2.68 (0.98)	2.63 (1.00)	**2.58 (1.02)**	2.65 (0.99)
p_hat1000-1	860.24	**859.74 (1.00)**	870.92 (0.99)	873.91 (0.98)	862.77 (1.00)	871.60 (0.99)
p_hat1000-2	33 035.45	33 314.15 (0.99)	32 999.15 (1.00)	32 812.80 (1.01)	**30 913.22 (1.07)**	31 202.52 (1.06)
p_hat1500-1	8 935.77	**8 935.50 (1.00)**	9 009.69 (0.99)	8 954.18 (1.00)	8 958.19 (1.00)	9 046.97 (0.99)
p_hat300-1	3.70	3.69 (1.00)	3.78 (0.98)	3.69 (1.00)	**3.68 (1.00)**	3.78 (0.98)
p_hat300-2	5.53	5.53 (1.00)	5.68 (0.97)	5.54 (1.00)	**5.48 (1.01)**	5.63 (0.98)
p_hat300-3	189.58	187.77 (1.01)	189.16 (1.00)	185.68 (1.02)	**175.01 (1.08)**	179.53 (1.06)
p_hat500-1	38.63	38.70 (1.00)	39.36 (0.98)	39.03 (0.99)	**38.61 (1.00)**	39.34 (0.98)
p_hat500-2	96.36	96.39 (1.00)	97.87 (0.98)	96.21 (1.00)	**95.08 (1.01)**	96.96 (0.99)
p_hat500-3	14 860.70	14 887.15 (1.00)	14 624.90 (1.02)	14 765.90 (1.01)	**13 429.92 (1.11)**	13 712.38 (1.08)
p_hat700-1	163.30	**160.75 (1.02)**	163.63 (1.00)	160.81 (1.02)	163.24 (1.00)	163.31 (1.00)
p_hat700-2	906.32	908.46 (1.00)	914.56 (0.99)	906.78 (1.00)	**866.08 (1.05)**	879.99 (1.03)
san1000	**895.34**	898.16 (1.00)	906.21 (0.99)	901.40 (0.99)	913.29 (0.98)	932.29 (0.96)
san200_0.7_1	10.85	**10.78 (1.01)**	11.01 (0.99)	10.91 (0.99)	10.93 (0.99)	11.06 (0.98)
san200_0.7_2	0.33	0.32 (1.04)	0.33 (0.98)	**0.31 (1.07)**	0.32 (1.01)	0.33 (0.99)
san200_0.9_1	13.93	13.90 (1.00)	13.35 (1.04)	**4.94 (2.82)**	12.03 (1.16)	12.13 (1.15)
san200_0.9_2	34.15	33.87 (1.01)	21.46 (1.59)	12.32 (2.77)	15.80 (2.16)	**10.01 (3.41)**
san200_0.9_3	1 069.00	1 068.17 (1.00)	1 016.33 (1.05)	639.01 (1.67)	843.40 (1.27)	**600.71 (1.78)**
san400_0.5_1	9.21	9.21 (1.00)	9.37 (0.98)	**9.13 (1.01)**	9.24 (1.00)	9.37 (0.98)
san400_0.7_1	1 125.52	**1 121.99 (1.00)**	1 146.32 (0.98)	1 125.12 (1.00)	1 132.10 (0.99)	1 151.14 (0.98)
san400_0.7_2	3 062.38	3 063.23 (1.00)	3 066.62 (1.00)	3 463.29 (0.88)	**3 048.94 (1.00)**	3 489.72 (0.88)
san400_0.7_3	4 411.82	4 405.26 (1.00)	4 487.18 (0.98)	**4 398.18 (1.00)**	4 497.81 (0.98)	4 521.80 (0.98)
sanr200_0.7	48.35	**48.34 (1.00)**	50.09 (0.97)	48.41 (1.00)	48.49 (1.00)	50.25 (0.96)
sanr200_0.9	679.25	679.65 (1.00)	633.59 (1.07)	664.95 (1.02)	**531.48 (1.28)**	567.49 (1.20)
sanr400_0.5	373.40	**370.59 (1.01)**	376.93 (0.99)	377.71 (0.99)	370.72 (1.01)	376.10 (0.99)
sanr400_0.7	29 766.80	29 838.40 (1.00)	30 466.35 (0.98)	29 844.65 (1.00)	**29 473.60 (1.01)**	30 242.80 (0.98)
s_{total}	1.00	1.00	0.99	1.00	**1.04**	1.03

F Kernel Sizes for Generalized Reduction Rules

We now present the number of vertices n and edges m, as well as the number of vertices $n(\mathcal{K}_{\text{dense}})$ and $n(\mathcal{K}_{\text{full}})$ of the reduced instances computed by both variants (BnR_{full} and $\text{BnR}_{\text{dense}}$) of our branch-and-reduce algorithm for MWIS. Instances are those used in our experimental evaluation in Section 4.2.4.

Table 25: Reduced instance sizes of OSM instances.

Graph	n	m	$n(\mathcal{K}_{\text{dense}})$	$n(\mathcal{K}_{\text{full}})$
alabama-AM2	1 164	38 772	173	173
alabama-AM3	3 504	619 328	1 614	1 614
district-of-columbia-AM1	2 500	49 302	800	800
district-of-columbia-AM2	13 597	3 219 590	6 360	6 360
district-of-columbia-AM3	46 221	55 458 274	33 367	33 367
florida-AM2	1 254	33 872	41	41
florida-AM3	2 985	308 086	1 069	1 069
georgia-AM3	1 680	148 252	861	861
greenland-AM3	4 986	7 304 722	3 942	3 942
hawaii-AM2	2 875	530 316	428	428
hawaii-AM3	28 006	98 889 842	24 436	24 436
idaho-AM3	4 064	7 848 160	3 208	3 208
kansas-AM3	2 732	1 613 824	1 605	1 605
kentucky-AM2	2 453	1 286 856	442	442
kentucky-AM3	19 095	119 067 260	16 871	16 871
louisiana-AM3	1 162	74 154	382	382
maryland-AM3	1 018	190 830	187	187
massachusetts-AM2	1 339	70 898	196	196
massachusetts-AM3	3 703	1 102 982	2 008	2 008
mexico-AM3	1 096	94 262	620	620
new-hampshire-AM3	1 107	36 042	247	247
north-carolina-AM3	1 557	473 478	1 178	1 178
oregon-AM2	1 325	115 034	35	35
oregon-AM3	5 588	5 825 402	3 670	3 670
pennsylvania-AM3	1 148	52 928	315	315
rhode-island-AM2	2 866	590 976	1 103	1 103
rhode-island-AM3	15 124	25 244 438	13 031	13 031
utah-AM3	1 339	85 744	568	568
vermont-AM3	3 436	2 272 328	2 630	2 630
virginia-AM2	2 279	120 080	237	237
virginia-AM3	6 185	1 331 806	3 867	3 867
washington-AM2	3 025	304 898	382	382
washington-AM3	10 022	4 692 426	8 030	8 030
west-virginia-AM3	1 185	251 240	991	991

Table 26: Reduced instance sizes of SNAP instances.

Graph	n	m	$n(\mathcal{K}_{\text{dense}})$	$n(\mathcal{K}_{\text{full}})$
as-skitter	1 696 415	22 190 596	27 318	9 180
ca-AstroPh	18 772	396 100	0	0
ca-CondMat	23 133	186 878	0	0
ca-GrQc	5 242	28 968	0	0
ca-HepPh	12 008	236 978	0	0
ca-HepTh	9 877	51 946	0	0
email-Enron	36 692	367 662	0	0
email-EuAll	265 214	728 962	0	0
p2p-Gnutella04	10 876	79 988	0	0
p2p-Gnutella05	8 846	63 678	0	0
p2p-Gnutella06	8 717	63 050	0	0
p2p-Gnutella08	6 301	41 554	0	0
p2p-Gnutella09	8 114	52 026	0	0
p2p-Gnutella24	26 518	130 738	0	0
p2p-Gnutella25	22 687	109 410	11	0
p2p-Gnutella30	36 682	176 656	10	0
p2p-Gnutella31	62 586	295 784	0	0
roadNet-CA	1 965 206	5 533 214	233 083	63 926
roadNet-PA	1 088 092	3 083 796	135 536	38 080
roadNet-TX	1 379 917	3 843 320	151 570	39 433
soc-Epinions1	75 879	811 480	6	0
soc-LiveJournal1	4 847 571	85 702 474	61 690	29 779
soc-Slashdot0811	77 360	938 360	0	0
soc-Slashdot0902	82 168	1 008 460	20	0
soc-pokec-relationships	1 632 803	44 603 928	927 214	902 748
web-BerkStan	685 230	13 298 940	37 004	17 482
web-Google	875 713	8 644 102	2 892	1 178
web-NotreDame	325 729	2 180 216	14 038	6 760
web-Stanford	281 903	3 985 272	14 280	2 640
wiki-Talk	2 394 385	9 319 130	0	0
wiki-Vote	7 115	201 524	246	237

G Detailed Results for Generalized Reduction Rules

In the following, we present detailed results of our experimental evaluation of exact and heuristic algorithms for MWIS (Section 4.2.4). We include the heuristic algorithms HILS and DynWVC, their extended variants Red + HILS and Red + DynWVC, as well as our exact algorithms $\text{BnR}_{\text{dense}}$ and BnR_{full}. Detailed tables show running times $t_{\max}$ (in seconds) and solution weights $w_{\max}$ for the best solution computed by each algorithm. The results for heuristic algorithms are running times for the best solution achieved over five runs with different random seeds. The best solution weight for each instance is highlighted in bold. Rows are highlighted in gray if either $\text{BnR}_{\text{dense}}$ or BnR_{full} is able to find an exact solution.

Table 27: Detailed results of base algorithms on OSM instances.

	DynWVC1		DynWVC2		HILS		$\text{BnR}_{\text{dense}}$		BnR_{full}	
Graph	$t_{\max}$	$w_{\max}$	$t_{\max}$	$w_{\max}$	$t_{\max}$	$w_{\max}$	$t_{\max}$	$w_{\max}$	$t_{\max}$	$w_{\max}$
alabama-AM2	0.62	174 241	26.83	174 297	0.04	**174 309**	0.40	**174 309**	0.79	**174 309**
alabama-AM3	464.02	185 527	887.55	185 652	0.73	**185 744**	15.79	185 707	80.78	185 707
district-of-columbia-AM1	12.64	**196 475**	11.40	**196 475**	0.26	**196 475**	1.97	**196 475**	4.13	**196 475**
district-of-columbia-AM2	272.37	208 942	596.62	208 954	717.75	**209 132**	20.03	147 450	233.70	147 450
district-of-columbia-AM3	949.96	224 289	782.62	223 780	989.68	**227 598**	553.84	92 784	918.07	92 714
florida-AM2	1.14	**230 595**	0.72	**230 595**	0.04	**230 595**	0.03	**230 595**	0.02	**230 595**
florida-AM3	553.56	237 127	181.58	237 081	2.76	**237 333**	20.52	**237 333**	324.38	226 767
georgia-AM3	0.88	**222 652**	1.29	**222 652**	0.05	**222 652**	4.88	214 918	14.35	214 918
greenland-AM3	73.16	**14 011**	51.09	14 008	1.72	**14 011**	14.52	13 152	47.25	13 069
hawaii-AM2	4.85	125 273	3.20	125 276	0.33	**125 284**	3.59	**125 284**	10.89	**125 284**
hawaii-AM3	898.64	140 596	904.15	140 486	332.32	**141 035**	288.58	106 251	1 177.95	129 812
idaho-AM3	76.55	**77 145**	85.35	**77 145**	1.49	**77 145**	866.90	77 010	61.26	76 831
kansas-AM3	46.87	**87 976**	44.26	**87 976**	0.84	**87 976**	11.35	87 925	18.99	87 925
kentucky-AM2	5.12	**97 397**	7.39	**97 397**	0.47	**97 397**	11.35	**97 397**	42.05	**97 397**
kentucky-AM3	932.32	100 463	722.69	100 430	802.03	**100 507**	172.30	91 864	3 346.94	96 634
louisiana-AM3	0.32	60 005	0.27	60 002	0.03	**60 024**	3.38	**60 024**	20.17	**60 024**
maryland-AM3	1.34	**45 496**	0.87	**45 496**	0.02	**45 496**	3.34	**45 496**	11.08	**45 496**
massachusetts-AM2	0.37	**140 095**	0.09	**140 095**	0.02	**140 095**	0.46	**140 095**	0.48	**140 095**
massachusetts-AM3	435.31	145 863	154.61	145 863	2.73	**145 866**	12.87	145 617	23.97	145 631
mexico-AM3	0.14	**97 663**	46.86	**97 663**	0.04	**97 663**	14.25	**97 663**	289.14	**97 663**
new-hampshire-AM3	0.22	**116 060**	0.42	**116 060**	0.03	**116 060**	3.25	**116 060**	8.75	**116 060**
north-carolina-AM3	796.26	49 716	285.91	**49 720**	0.08	**49 720**	10.45	49 562	11.55	49 562
oregon-AM2	0.22	**165 047**	0.25	**165 047**	0.04	**165 047**	0.04	**165 047**	0.09	**165 047**
oregon-AM3	393.23	175 046	126.97	175 060	3.36	**175 078**	351.99	174 334	474.15	164 941
pennsylvania-AM3	0.09	**143 870**	0.15	**143 870**	0.04	**143 870**	9.98	**143 870**	38.76	**143 870**
rhode-island-AM2	6.66	184 562	24.74	184 576	0.40	**184 596**	10.70	184 543	16.79	184 543
rhode-island-AM3	54.99	201 553	609.14	201 344	43.34	**201 758**	399.33	162 639	931.05	163 080
utah-AM3	136.15	98 802	233.52	**98 847**	0.08	**98 847**	64.04	**98 847**	285.22	**98 847**
vermont-AM3	119.63	63 234	88.35	63 238	0.95	**63 302**	95.81	55 584	443.88	55 577
virginia-AM2	0.89	295 794	1.32	295 668	0.12	**295 867**	0.93	**295 867**	0.77	**295 867**
virginia-AM3	289.23	307 867	883.75	307 845	3.75	**308 305**	109.20	306 985	786.05	233 572
washington-AM2	2.00	**305 619**	15.60	**305 619**	0.62	**305 619**	2.44	**305 619**	2.20	**305 619**
washington-AM3	79.77	313 808	401.59	313 827	13.88	**314 288**	248.77	271 747	532.25	271 747
west-virginia-AM3	1.10	**47 927**	0.87	**47 927**	0.08	**47 927**	14.38	**47 927**	854.73	**47 927**

Table 28: Detailed results of base algorithms on SNAP instances.

	DynWVC1		DynWVC2		HILS		$\text{BnR}_{\text{dense}}$		BnR_{full}	
Graph	$t_{\max}$	$w_{\max}$	$t_{\max}$	$w_{\max}$	$t_{\max}$	$w_{\max}$	$t_{\max}$	$w_{\max}$	$t_{\max}$	$w_{\max}$
as-skitter	997.39	123 412 428	576.93	123 105 765	998.75	122 539 706	641.38	123 172 824	746.93	**123 904 741**
ca-AstroPh	207.99	796 467	108.35	796 535	46.76	**796 556**	0.03	**796 556**	0.03	**796 556**
ca-CondMat	71.54	1 143 431	222.30	1 143 471	45.07	**1 143 480**	0.02	**1 143 480**	0.02	**1 143 480**
ca-GrQc	1.75	**289 481**	0.82	**289 481**	0.60	**289 481**	0.00	**289 481**	0.00	**289 481**
ca-HepPh	26.36	579 624	17.31	579 662	11.44	**579 675**	0.02	**579 675**	0.02	**579 675**
ca-HepTh	9.87	560 630	12.64	560 642	94.19	**560 662**	0.01	**560 662**	0.01	**560 662**
email-Enron	295.02	2 457 460	910.50	2 457 505	79.40	**2 457 547**	0.04	**2 457 547**	0.03	**2 457 547**
email-EuAll	180.92	**25 330 331**	179.26	**25 330 331**	501.09	**25 330 331**	0.13	**25 330 331**	0.19	**25 330 331**
p2p-Gnutella04	2.46	667 496	866.88	667 503	2.64	**667 539**	0.01	**667 539**	0.01	**667 539**
p2p-Gnutella05	24.23	**556 559**	3.54	**556 559**	0.60	**556 559**	0.01	**556 559**	0.01	**556 559**
p2p-Gnutella06	532.67	547 585	1.38	547 586	1.47	**547 591**	0.01	**547 591**	0.01	**547 591**
p2p-Gnutella08	0.21	**435 893**	0.19	**435 893**	0.25	**435 893**	0.00	**435 893**	0.01	**435 893**
p2p-Gnutella09	0.23	**568 472**	0.22	**568 472**	0.15	**568 472**	0.01	**568 472**	0.01	**568 472**
p2p-Gnutella24	10.83	1 970 325	9.81	1 970 325	4.06	**1 970 329**	0.02	**1 970 329**	0.02	**1 970 329**
p2p-Gnutella25	2.22	**1 697 310**	6.33	**1 697 310**	1.64	**1 697 310**	0.01	**1 697 310**	0.02	**1 697 310**
p2p-Gnutella30	10.06	2 785 926	22.66	2 785 922	7.36	**2 785 957**	0.02	**2 785 957**	0.03	**2 785 957**
p2p-Gnutella31	169.03	4 750 622	43.15	4 750 632	34.33	**4 750 671**	0.13	**4 750 671**	0.04	**4 750 671**
roadNet-CA	1 001.61	109 028 140	1 000.88	109 023 976	3 312.19	108 167 310	931.36	106 500 027	774.56	**111 408 830**
roadNet-PA	720.57	60 940 033	787.59	60 940 033	998.56	59 915 775	988.62	58 927 755	32.06	**61 686 106**
roadNet-TX	1 001.45	77 498 612	1 000.78	77 525 099	1 697.13	76 366 577	870.62	75 843 903	33.49	**78 606 965**
soc-Epinions1	617.40	5 668 054	625.89	5 668 180	694.51	5 668 382	0.07	**5 668 401**	0.11	**5 668 401**
soc-LiveJournal1	1 001.31	277 850 684	1 001.23	277 824 322	12 437.50	280 559 036	86.66	283 869 420	270.96	**283 948 671**
soc-Slashdot0811	809.97	5 650 118	477.14	5 650 303	767.51	5 650 644	0.10	**5 650 791**	0.18	**5 650 791**
soc-Slashdot0902	783.10	5 953 052	272.11	5 953 235	786.70	5 953 436	0.13	**5 953 582**	0.21	**5 953 582**
soc-pokec-relationships	999.99	82 522 272	1 001.42	**82 640 035**	2 482.18	82 381 583	287.40	82 595 492	1 404.57	75 717 984
web-BerkStan	347.17	43 595 139	372.33	43 593 142	994.73	43 319 988	22.58	43 138 612	831.75	**43 766 431**
web-Google	759.75	56 193 138	683.63	56 190 870	994.58	55 954 155	2.08	**56 313 384**	3.16	**56 313 384**
web-NotreDame	963.44	**25 975 765**	875.22	25 968 209	998.79	25 970 368	354.79	25 947 936	28.11	25 957 800
web-Stanford	999.97	17 731 195	997.98	17 735 700	999.91	17 679 156	47.62	17 634 819	4.69	**17 799 469**
wiki-Talk	961.05	235 874 406	991.31	235 874 419	996.02	235 852 509	3.85	**235 875 181**	3.36	**235 875 181**
wiki-Vote	0.74	**500 436**	0.75	**500 436**	23.96	**500 436**	0.05	**500 436**	0.06	**500 436**

Table 29: Detailed results of combined algorithms on OSM instances.

	Red + DynWVC1		Red + DynWVC2		Red + HILS		BnR_{dense}		BnR_{full}	
Graph	t_{max}	w_{max}	t_{max}	w_{max}	t_{max}	w_{max}	t_{max}	w_{max}	t_{max}	w_{max}
alabama-AM2	0.11	**174 309**	0.11	**174 309**	0.10	**174 309**	0.40	**174 309**	0.79	**174 309**
alabama-AM3	370.80	185 727	295.20	185 729	4.05	**185 744**	15.79	185 707	80.78	185 707
district-of-columbia-AM1	0.92	**196 475**	0.92	**196 475**	0.37	**196 475**	1.97	**196 475**	4.13	**196 475**
district-of-columbia-AM2	334.12	209 125	982.91	209 056	220.82	**209 132**	20.03	147 450	233.70	147 450
district-of-columbia-AM3	879.25	225 535	789.47	225 031	320.06	**227 534**	553.84	92 784	918.07	92 714
florida-AM2	0.03	**230 595**	0.03	**230 595**	0.03	**230 595**	0.03	**230 595**	0.02	**230 595**
florida-AM3	8.66	237 331	8.57	237 331	8.01	**237 333**	20.52	**237 333**	324.38	226 767
georgia-AM3	2.64	**222 652**	2.62	**222 652**	2.43	**222 652**	4.88	214 918	14.35	214 918
greenland-AM3	712.63	14 007	462.23	14 006	10.34	**14 011**	14.52	13 152	47.25	13 069
hawaii-AM2	0.96	**125 284**	0.96	**125 284**	0.93	**125 284**	3.59	**125 284**	10.89	**125 284**
hawaii-AM3	405.34	140 714	957.61	140 709	329.20	**141 011**	288.58	106 251	1 177.95	129 812
idaho-AM3	40.38	**77 145**	20.79	**77 145**	203.76	**77 145**	866.90	77 010	61.26	76 831
kansas-AM3	13.59	**87 976**	18.43	**87 976**	2.06	**87 976**	11.35	87 925	18.99	87 925
kentucky-AM2	1.13	**97 397**	1.13	**97 397**	1.07	**97 397**	11.35	**97 397**	42.05	**97 397**
kentucky-AM3	766.39	100 479	759.20	100 480	973.22	**100 486**	172.30	91 864	3 346.94	96 634
louisiana-AM3	1.35	**60 024**	1.35	**60 024**	1.33	**60 024**	3.38	**60 024**	20.17	**60 024**
maryland-AM3	2.07	**45 496**	2.07	**45 496**	2.07	**45 496**	3.34	**45 496**	11.08	**45 496**
massachusetts-AM2	0.04	**140 095**	0.04	**140 095**	0.04	**140 095**	0.46	**140 095**	0.48	**140 095**
massachusetts-AM3	10.68	**145 866**	8.38	**145 866**	2.92	**145 866**	12.87	145 617	23.97	145 631
mexico-AM3	5.39	**97 663**	5.34	**97 663**	5.28	**97 663**	14.25	**97 663**	289.14	**97 663**
new-hampshire-AM3	1.51	**116 060**	1.51	**116 060**	1.50	**116 060**	3.25	**116 060**	8.75	**116 060**
north-carolina-AM3	1.76	**49 720**	0.79	**49 720**	0.48	**49 720**	10.45	49 562	11.55	49 562
oregon-AM2	0.04	**165 047**	0.04	**165 047**	0.04	**165 047**	0.04	**165 047**	0.09	**165 047**
oregon-AM3	135.72	175 073	167.56	175 075	5.18	**175 078**	351.99	174 334	474.15	164 941
pennsylvania-AM3	4.35	**143 870**	4.34	**143 870**	4.33	**143 870**	9.98	**143 870**	38.76	**143 870**
rhode-island-AM2	1.03	**184 596**	2.40	**184 596**	0.43	**184 596**	10.70	184 543	16.79	184 543
rhode-island-AM3	993.86	201 667	255.71	201 668	605.61	**201 734**	399.33	162 639	931.05	163 080
utah-AM3	168.07	**98 847**	2.36	**98 847**	2.10	**98 847**	64.04	**98 847**	285.22	**98 847**
vermont-AM3	62.85	63 280	690.58	63 256	2.95	**63 312**	95.81	55 584	443.88	55 577
virginia-AM2	0.25	**295 867**	0.25	**295 867**	0.23	**295 867**	0.93	**295 867**	0.77	**295 867**
virginia-AM3	708.66	308 052	790.21	308 090	19.34	**308 305**	109.20	306 985	786.05	233 572
washington-AM2	0.24	**305 619**	0.24	**305 619**	0.23	**305 619**	2.44	**305 619**	2.20	**305 619**
washington-AM3	59.08	314 097	505.58	314 079	863.47	**314 288**	248.77	271 747	532.25	271 747
west-virginia-AM3	3.06	**47 927**	3.77	**47 927**	2.54	**47 927**	14.38	**47 927**	854.73	**47 927**

Table 30: Detailed results of combined algorithms on SNAP instances.

	Red + DynWVC1		Red + DynWVC2		Red + HILS		$\text{BnR}_{\text{dense}}$		BnR_{full}	
Graph	$t_{\max}$	$w_{\max}$	$t_{\max}$	$w_{\max}$	$t_{\max}$	$w_{\max}$	$t_{\max}$	$w_{\max}$	$t_{\max}$	$w_{\max}$
as-skitter	64.52	123 995 654	85.60	123 995 808	845.70	**123 996 322**	641.38	123 172 824	746.93	123 904 741
ca-AstroPh	0.02	**796 556**	0.02	**796 556**	0.02	**796 556**	0.03	**796 556**	0.03	**796 556**
ca-CondMat	0.01	**1 143 480**	0.01	**1 143 480**	0.01	**1 143 480**	0.02	**1 143 480**	0.02	**1 143 480**
ca-GrQc	0.00	**289 481**	0.00	**289 481**	0.00	**289 481**	0.00	**289 481**	0.00	**289 481**
ca-HepPh	0.02	**579 675**	0.02	**579 675**	0.02	**579 675**	0.02	**579 675**	0.02	**579 675**
ca-HepTh	0.00	**560 662**	0.00	**560 662**	0.00	**560 662**	0.01	**560 662**	0.01	**560 662**
email-Enron	0.03	**2 457 547**	0.03	**2 457 547**	0.03	**2 457 547**	0.04	**2 457 547**	0.03	**2 457 547**
email-EuAll	0.12	**25 330 331**	0.12	**25 330 331**	0.12	**25 330 331**	0.13	**25 330 331**	0.19	**25 330 331**
p2p-Gnutella04	0.01	**667 539**	0.01	**667 539**	0.01	**667 539**	0.01	**667 539**	0.01	**667 539**
p2p-Gnutella05	0.01	**556 559**	0.01	**556 559**	0.01	**556 559**	0.01	**556 559**	0.01	**556 559**
p2p-Gnutella06	0.01	**547 591**	0.01	**547 591**	0.01	**547 591**	0.01	**547 591**	0.01	**547 591**
p2p-Gnutella08	0.00	**435 893**	0.00	**435 893**	0.00	**435 893**	0.00	**435 893**	0.01	**435 893**
p2p-Gnutella09	0.01	**568 472**	0.01	**568 472**	0.01	**568 472**	0.01	**568 472**	0.01	**568 472**
p2p-Gnutella24	0.01	**1 970 329**	0.01	**1 970 329**	0.01	**1 970 329**	0.02	**1 970 329**	0.02	**1 970 329**
p2p-Gnutella25	0.01	**1 697 310**	0.01	**1 697 310**	0.01	**1 697 310**	0.01	**1 697 310**	0.02	**1 697 310**
p2p-Gnutella30	0.01	**2 785 957**	0.01	**2 785 957**	0.01	**2 785 957**	0.02	**2 785 957**	0.03	**2 785 957**
p2p-Gnutella31	0.02	**4 750 671**	0.02	**4 750 671**	0.02	**4 750 671**	0.13	**4 750 671**	0.04	**4 750 671**
roadNet-CA	918.32	111 398 659	866.70	111 398 243	994.57	111 402 080	931.36	106 500 027	774.56	**111 408 830**
roadNet-PA	733.57	61 680 822	639.56	61 680 822	947.93	61 682 180	988.62	58 927 755	32.06	**61 686 106**
roadNet-TX	952.53	78 601 859	771.05	78 601 813	946.32	78 602 984	870.62	75 843 903	33.49	**78 606 965**
soc-Epinions1	0.08	**5 668 401**	0.08	**5 668 401**	0.08	**5 668 401**	0.07	**5 668 401**	0.11	**5 668 401**
soc-LiveJournal1	916.65	283 973 802	996.68	283 973 997	761.51	**283 975 036**	86.66	283 869 420	270.96	283 948 671
soc-Slashdot0811	0.14	**5 650 791**	0.14	**5 650 791**	0.14	**5 650 791**	0.10	**5 650 791**	0.18	**5 650 791**
soc-Slashdot0902	0.17	**5 953 582**	0.17	**5 953 582**	0.17	**5 953 582**	0.13	**5 953 582**	0.21	**5 953 582**
soc-pokec-relationships	1 400.47	43 734 005	1 400.47	43 734 005	2 400.00	**82 845 330**	287.40	82 595 492	1 404.57	75 717 984
web-BerkStan	373.58	43 877 439	612.64	43 877 349	859.76	**43 877 507**	22.58	43 138 612	831.75	43 766 431
web-Google	3.20	56 313 343	3.30	56 313 349	3.01	**56 313 384**	2.08	**56 313 384**	3.16	**56 313 384**
web-NotreDame	147.60	25 995 575	850.00	25 995 615	173.50	**25 995 648**	354.79	25 947 936	28.11	25 957 800
web-Stanford	5.08	17 799 379	5.19	17 799 405	131.24	**17 799 556**	47.62	17 634 819	4.69	17 799 469
wiki-Talk	2.30	**235 875 181**	2.30	**235 875 181**	2.30	**235 875 181**	3.85	**235 875 181**	3.36	**235 875 181**
wiki-Vote	0.04	**500 436**	0.04	**500 436**	0.03	**500 436**	0.05	**500 436**	0.06	**500 436**

H Convergence Plots for Generalized Reductions

We present additional convergence plots for our experimental evaluation of exact and heuristic MWIS algorithms (Section 4.2.4). This includes the heuristic algorithms HILS and DynWVC, their extended variants Red + HILS and Red + DynWVC, as well as our exact algorithms $\text{BnR}_{\text{dense}}$ and BnR_{full}. The results for heuristic algorithms are event-based geometric average values over five runs with different random seeds.

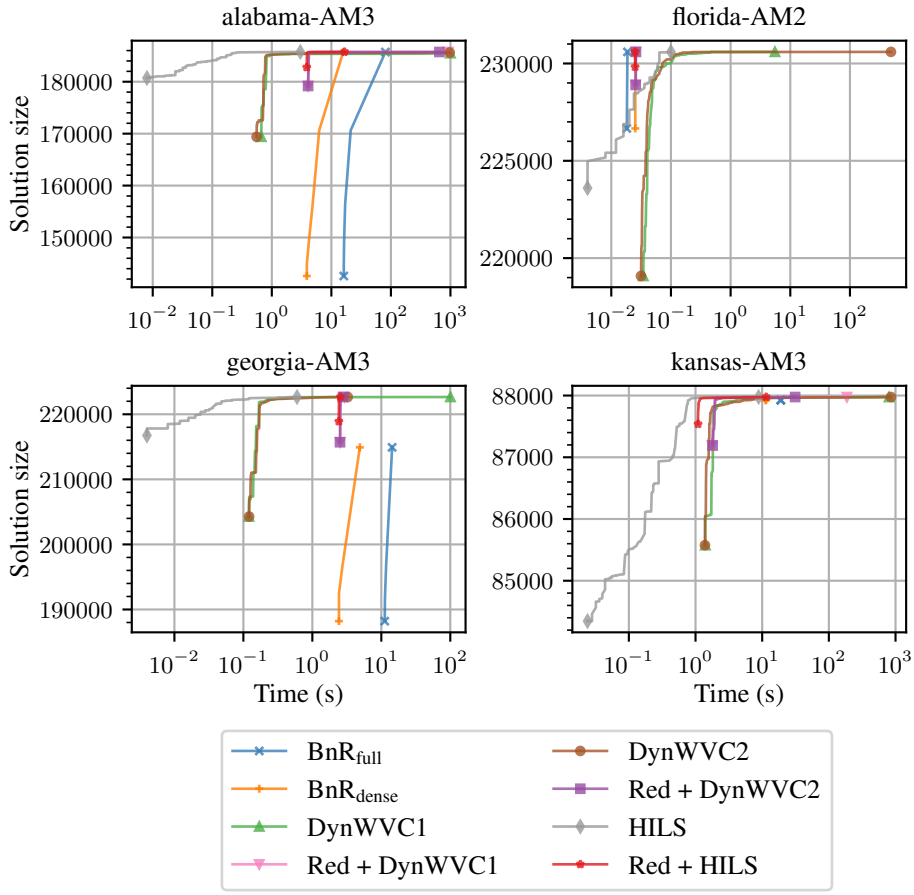


Figure 7: Convergence plots for alabama-AM3 (top left), florida-AM2 (top right), georgia-AM3 (bottom left), and kansas-AM3 (bottom right).

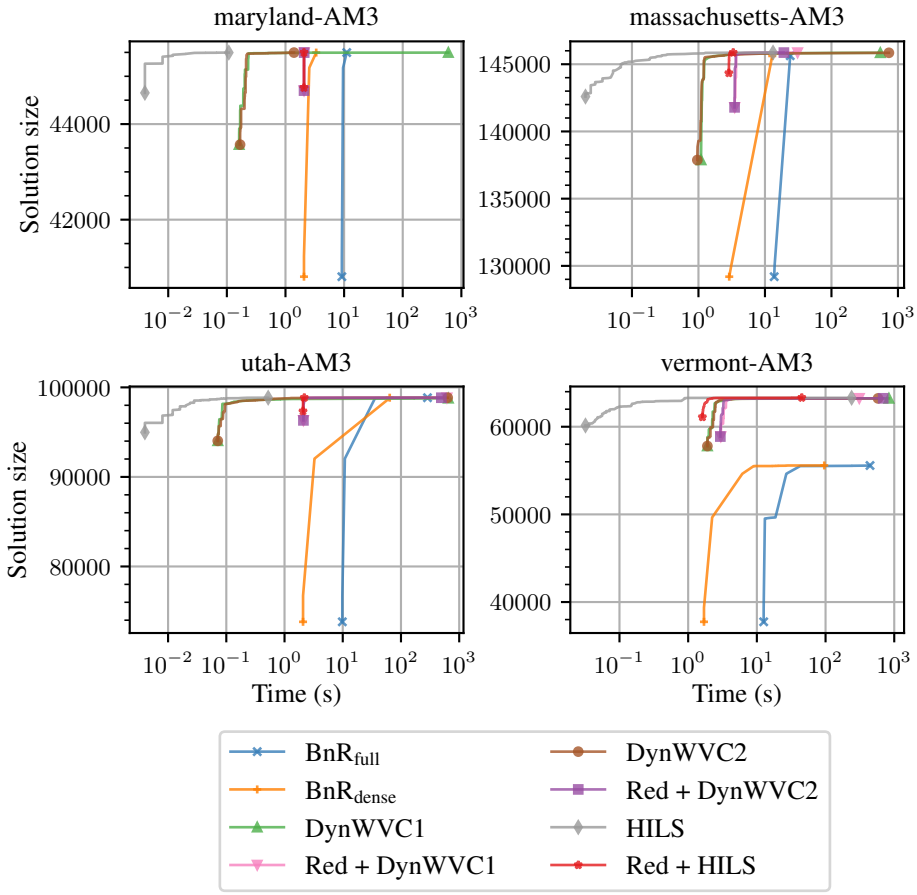


Figure 8: Convergence plots for maryland-AM3 (top left), massachusetts-AM3 (top right), utah-AM3 (bottom left), and vermont-AM3 (bottom right).

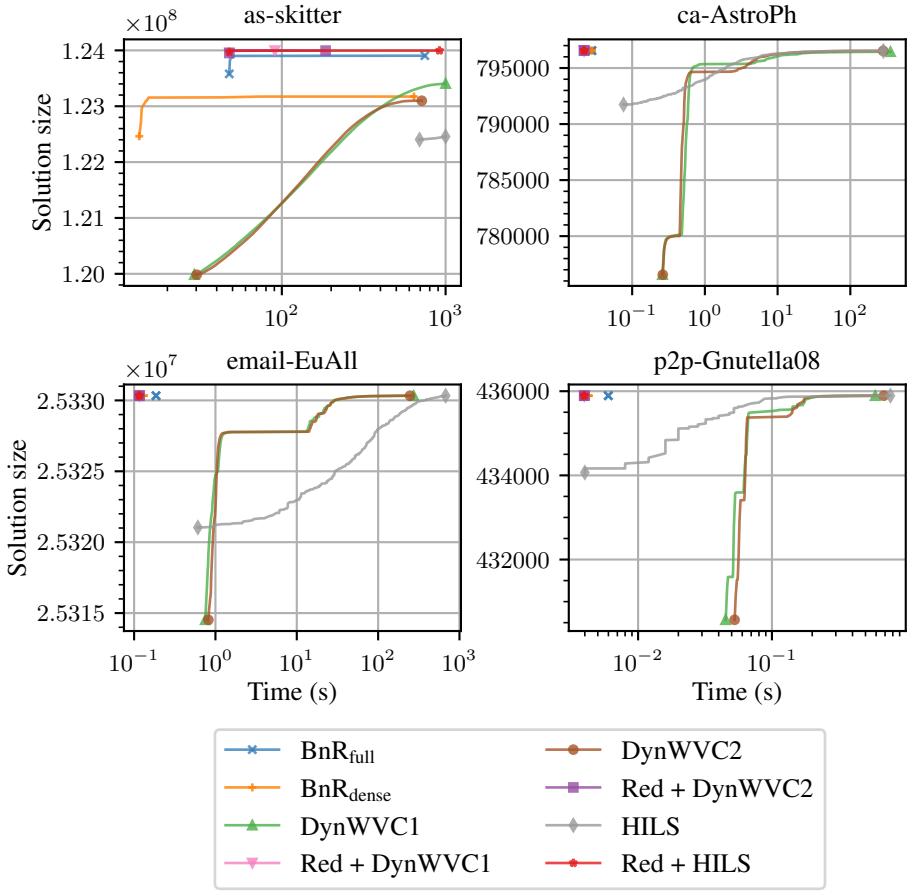


Figure 9: Convergence plots for as-skitter (top left), ca-AstroPh (top right), email-EuAll (bottom left), and p2p-Gnutella08 (bottom right).

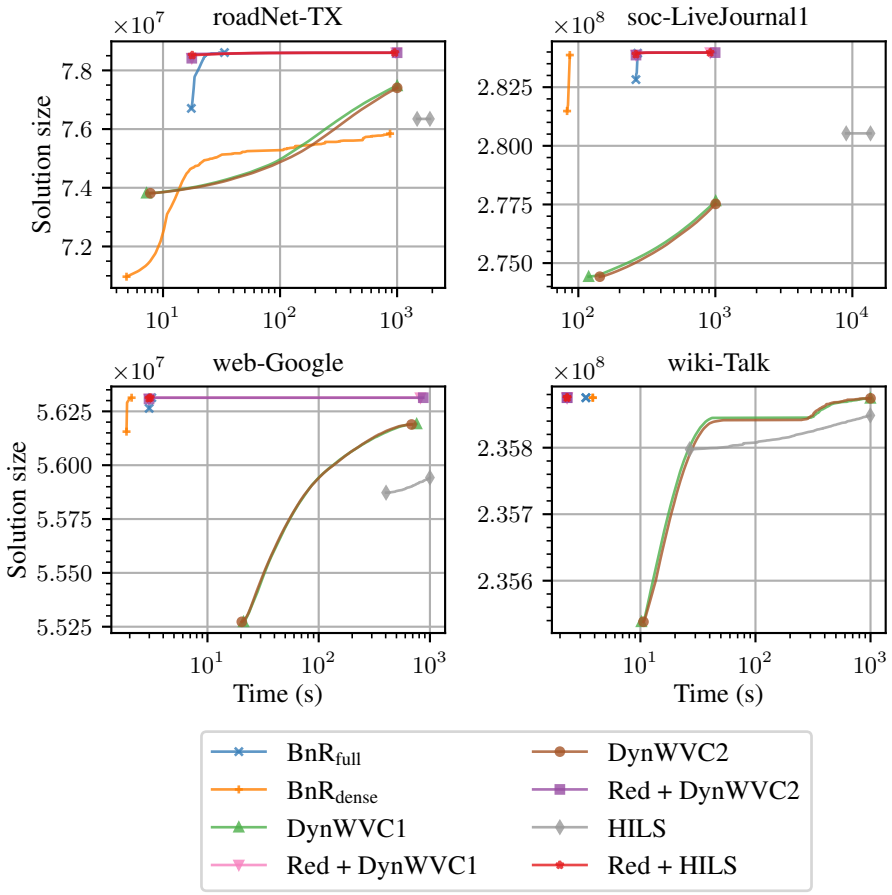


Figure 10: Convergence plots for roadNet-TX (top left), soc-LiveJournal1 (top right), web-Google (bottom left), and wiki-Talk (bottom right).

I Branch-and-Reduce for Comparison Increasing Transformations

In the following, we present detailed results on the irreducible graphs computed for the instances used in our experimental evaluation of exact MWIS algorithms (Section 4.3.4). This includes the algorithms $\text{BnR}_{\text{dense}}$, BnR_{full}, NonIncreasing, Cyclic-Fast, and Cyclic-Strong. Detailed tables show the number of vertices $n(\mathcal{K})$ of the obtained irreducible graphs, the time t_r (in seconds) needed to obtain them, and the total solving time t_t (in seconds). The global best solving time t_t is highlighted in bold. Rows are highlighted in gray if one of our algorithms NonIncreasing, Cyclic-Fast, or Cyclic-Strong is able to obtain an empty graph.

Table 31: Detailed results on FE instances.

Graph	$n(\mathcal{K})$	t_r	t_t	$n(\mathcal{K})$	t_r	t_t	$n(\mathcal{K})$	t_r	t_t	$n(\mathcal{K})$	t_r	t_t	$n(\mathcal{K})$	t_r	t_t
FE instances	$\text{BnR}_{\text{dense}}$			BnR_{full}			NonIncreasing			Cyclic-Fast			Cyclic-Strong		
fe_4elt2	8 580	0.29	-	8 578	0.87	-	562	0.10	-	0	0.12	**0.13**	0	0.16	0.17
fe_body	16 107	0.69	-	15 992	3.40	-	1 162	0.16	-	625	0.44	-	553	0.94	-
fe_ocean	141 283	1.05	-	0	5.94	**5.99**	138 338	8.90	-	138 134	9.61	-	138 049	10.78	-
fe_pwt	34 521	0.46	-	34 521	2.70	-	25 550	0.78	-	20 241	1.80	-	14 107	5.65	-
fe_rotor	98 271	9.80	-	98 271	24.47	-	91 946	4.80	-	91 634	4.82	-	89 647	11.11	-
fe_sphere	15 269	0.21	-	15 269	1.47	-	2 961	0.34	-	147	0.62	**0.83**	0	0.75	0.77
fe_tooth	10 922	1.69	-	10 801	3.79	-	15	0.41	0.46	0	0.30	**0.34**	0	0.28	0.32

Table 32: Detailed results on OSM instances.

Graph	$n(\mathcal{K})$	t_r	t_t	$n(\mathcal{K})$	t_r	t_t	$n(\mathcal{K})$	t_r	t_t	$n(\mathcal{K})$	t_r	t_t	$n(\mathcal{K})$	t_r	t_t
OSM instances	$\text{BnR}_{\text{dense}}$			BnR_{full}			NonIncreasing			Cyclic-Fast			Cyclic-Strong		
alabama-AM2	173	0.06	0.31	173	0.07	0.55	0	0.01	**0.01**	0	0.01	0.01	0	0.01	0.01
alabama-AM3	1 614	12.05	-	1 614	14.37	-	1 288	0.34	-	456	1.45	**3.94**	0	33.11	33.16
district-of-columbia-AM1	800	1.22	-	800	1.28	-	367	0.03	39.81	185	0.41	**0.80**	0	3.65	3.66
district-of-columbia-AM2	6 360	11.86	-	6 360	14.39	-	5 606	0.85	-	1 855	2.51	-	1 484	84.91	-
district-of-columbia-AM3	33 367	63.23	-	33 367	358.14	-	32 320	33.68	-	28 842	66.67	-	25 031	441.44	-
florida-AM2	41	0.01	0.01	41	0.01	0.01	0	0.00	0.00	0	0.00	**0.00**	0	0.00	0.00
florida-AM3	1 069	31.52	45.81	1 069	35.20	-	814	0.13	3.85	661	0.44	**2.93**	267	42.26	45.05
georgia-AM3	861	8.99	892.17	861	10.14	-	796	0.08	25.97	587	0.69	**10.35**	425	12.84	32.53
greenland-AM3	3 942	3.81	-	3 942	24.77	-	3 953	3.94	-	3 339	10.27	-	3 339	54.44	-
hawaii-AM2	428	2.08	4.27	428	2.15	10.22	262	0.07	0.18	0	0.09	**0.09**	0	0.10	0.10
hawaii-AM3	24 436	70.38	-	24 436	743.04	-	24 184	98.22	-	22 997	118.52	-	21 087	632.02	-
idaho-AM3	3 208	3.17	-	3 208	29.91	-	3 204	6.96	-	3 160	8.74	-	2 909	33.77	-
kansas-AM3	1 605	2.46	-	1 605	4.81	-	1 550	0.49	-	903	2.46	**430.93**	860	41.61	489.15
kentucky-AM2	442	2.05	11.85	442	2.19	67.28	183	0.20	0.39	0	0.22	**0.23**	0	0.41	0.42
kentucky-AM3	16 871	109.47	-	16 871	3 344.67	-	16 807	237.86	-	15 947	298.49	-	15 684	705.46	-
louisiana-AM3	382	4.56	6.55	382	5.04	25.22	349	0.03	0.82	0	0.07	**0.07**	0	0.16	0.16
maryland-AM3	187	7.59	8.49	187	8.65	10.73	335	0.03	0.19	0	0.11	**0.11**	0	0.15	0.15
massachusetts-AM2	196	0.04	0.36	196	0.04	0.58	193	0.02	0.07	0	0.06	**0.06**	0	0.07	0.07
massachusetts-AM3	2 008	9.42	-	2 008	12.62	-	1 928	0.36	-	1 636	1.08	-	1 632	31.83	-
mexico-AM3	620	25.29	80.23	620	27.52	991.99	514	0.03	**1.47**	483	0.28	1.50	0	21.03	21.30
new-hampshire-AM3	247	4.99	6.19	247	5.69	15.89	164	0.02	0.17	0	0.07	**0.07**	0	0.09	0.09
north-carolina-AM3	1 178	0.69	-	1 178	1.22	-	1 146	0.25	-	1 144	0.43	-	700	47.38	379.088
oregon-AM2	35	0.04	0.05	35	0.05	0.05	0	0.01	**0.01**	0	0.02	0.02	0	0.01	0.01
oregon-AM3	3 670	9.95	-	3 670	34.95	-	3 584	3.92	-	3 417	6.21	-	2 721	38.72	-
pennsylvania-AM3	315	16.69	20.71	315	19.39	113.87	317	0.03	0.39	0	0.07	**0.07**	0	0.12	0.12
rhode-island-AM2	1 103	0.55	-	1 103	0.68	-	845	0.17	163.07	0	0.53	**0.53**	0	4.57	4.58
rhode-island-AM3	13 031	7.75	-	13 031	193.76	-	12 934	26.54	-	12 653	29.75	-	12 653	59.69	-
utah-AM3	568	8.21	51.91	568	8.97	276.27	396	0.03	0.87	0	0.09	**0.09**	0	0.40	0.41
vermont-AM3	2 630	4.79	-	2 630	9.82	-	2 289	0.97	-	2 069	1.37	-	2 045	55.28	-
virginia-AM2	237	0.13	0.61	237	0.12	0.99	0	0.03	**0.03**	0	0.03	0.03	0	0.03	0.03
virginia-AM3	3 867	34.13	-	3 867	39.74	-	3 738	0.40	-	2 827	1.28	-	2 547	81.67	-
washington-AM2	382	0.24	5.31	382	0.18	8.58	171	0.05	0.37	0	0.06	**0.06**	0	0.07	0.07
washington-AM3	8 030	50.21	-	8 030	67.00	-	7 649	2.19	-	6 895	3.12	-	6 159	73.52	-
west-virginia-AM3	991	10.69	-	991	12.13	-	970	0.08	238.39	890	0.33	**155.49**	881	38.73	241.68

Table 33: Detailed results on mesh instances.

Graph	$n(\mathcal{K})$	t_r	t_t	$n(\mathcal{K})$	t_r	t_t	$n(\mathcal{K})$	t_r	t_t	$n(\mathcal{K})$	t_r	t_t	$n(\mathcal{K})$	t_r	t_t
mesh instances	$\text{BnR}_{\text{dense}}$			BnR_{full}			NonIncreasing			Cyclic-Fast			Cyclic-Strong		
beethoven	1 254	0.02	7.86	427	0.02	0.08	0	0.01	**0.01**	0	0.01	0.01	0	0.01	0.01
blob	5 746	0.08	-	1 464	0.06	0.20	0	0.03	**0.03**	0	0.03	0.04	0	0.03	0.03
buddha	380 315	5.56	-	107 265	26.19	67.85	86	1.83	2.74	0	1.87	**2.26**	0	1.91	2.39
bunny	24 580	0.34	-	3 290	0.56	0.89	0	0.12	**0.14**	0	0.13	0.16	0	0.15	0.18
cow	1 916	0.02	-	513	0.02	0.06	0	0.01	0.01	0	0.01	**0.01**	0	0.01	0.01
dragon	51 885	0.89	-	12 893	1.34	3.83	0	0.18	**0.21**	0	0.19	0.23	0	0.21	0.25
dragonsub	218 779	2.60	-	19 470	4.15	5.66	506	1.03	2.08	0	1.13	**1.36**	0	1.07	1.28
ecat	239 787	4.07	-	26 270	10.09	12.93	274	2.12	3.16	0	2.12	**2.51**	0	2.14	2.56
face	7 588	0.09	-	1 540	0.10	0.21	0	0.03	0.04	0	0.03	**0.03**	0	0.03	0.04
fandisk	2 851	0.05	-	336	0.03	0.07	51	0.02	0.03	0	0.02	**0.02**	0	0.02	0.02
feline	14 817	0.20	-	2 743	0.25	0.47	0	0.08	0.09	0	0.08	**0.09**	0	0.08	0.09
gameguy	13 959	0.17	-	312	0.10	0.12	0	0.06	0.07	0	0.06	**0.07**	0	0.06	0.07
gargoyle	6 512	0.15	-	1 819	0.14	0.36	0	0.03	0.03	0	0.03	**0.03**	0	0.03	0.03
turtle	91 624	1.17	-	16 095	1.92	4.98	186	0.42	0.65	0	0.41	**0.49**	0	0.47	0.56
venus	1 898	0.02	-	175	0.01	0.02	0	0.01	0.01	0	0.01	**0.01**	0	0.01	0.01

Table 34: Detailed results on SNAP instances.

Graph	$n(\mathcal{K})$	t_r	t_t	$n(\mathcal{K})$	t_r	t_t	$n(\mathcal{K})$	t_r	t_t	$n(\mathcal{K})$	t_r	t_t	$n(\mathcal{K})$	t_r	t_t
SNAP instances	$\text{BnR}_{\text{dense}}$			BnR_{full}			NonIncreasing			Cyclic-Fast			Cyclic-Strong		
as-skitter	26 584	25.82	-	8 585	36.69	-	3 426	4.75	-	2 782	5.50	-	2 343	6.80	-
ca-AstroPh	0	0.02	**0.03**	0	0.02	0.03	0	0.02	0.03	0	0.03	0.04	0	0.03	0.03
ca-CondMat	0	0.02	0.03	0	0.01	0.02	0	0.01	**0.02**	0	0.03	0.03	0	0.01	0.02
ca-GrQc	0	0.00	0.00	0	0.00	0.00	0	0.00	**0.00**	0	0.00	0.00	0	0.00	0.00
ca-HepPh	0	0.01	0.02	0	0.01	0.02	0	0.01	**0.01**	0	0.01	0.02	0	0.01	0.01
ca-HepTh	0	0.01	0.01	0	0.00	0.01	0	0.01	**0.01**	0	0.01	0.01	0	0.00	0.00
email-Enron	0	0.02	**0.03**	0	0.02	0.03	0	0.04	0.04	0	0.03	0.03	0	0.03	0.03
email-EuAll	0	0.08	0.17	0	0.09	0.16	0	0.06	**0.08**	0	0.09	0.13	0	0.07	0.10
p2p-Gnutella04	0	0.01	0.01	0	0.01	0.01	0	0.01	**0.01**	0	0.01	0.01	0	0.01	0.01
p2p-Gnutella05	0	0.01	**0.01**	0	0.01	0.01	0	0.01	0.01	0	0.01	0.01	0	0.01	0.01
p2p-Gnutella06	0	0.01	0.01	0	0.01	0.01	0	0.01	**0.01**	0	0.01	0.01	0	0.01	0.01
p2p-Gnutella08	0	0.00	0.00	0	0.00	0.01	0	0.00	0.00	0	0.00	**0.00**	0	0.00	0.00
p2p-Gnutella09	0	0.00	0.01	0	0.01	0.01	0	0.00	0.01	0	0.00	**0.00**	0	0.00	0.01
p2p-Gnutella24	0	0.01	0.02	0	0.02	0.03	0	0.01	**0.01**	0	0.01	0.02	0	0.01	0.01
p2p-Gnutella25	10	0.01	0.02	0	0.01	0.02	0	0.01	**0.01**	0	0.01	0.02	0	0.02	0.02
p2p-Gnutella30	0	0.01	0.02	0	0.02	0.03	0	0.02	0.02	0	0.02	**0.02**	0	0.01	0.02
p2p-Gnutella31	0	0.04	0.07	0	0.04	0.07	0	0.03	**0.03**	0	0.05	0.06	0	0.04	0.05
roadNet-CA	234 433	3.96	-	66 406	20.51	437.62	478	2.14	5.70	0	2.42	**3.57**	0	2.59	3.07
roadNet-PA	133 814	2.43	-	35 442	7.73	23.86	300	1.05	2.24	0	1.19	**1.44**	0	1.14	1.40
roadNet-TX	153 985	2.65	-	40 350	10.49	24.30	882	1.23	3.98	0	1.32	**1.64**	0	1.34	1.65
soc-Epinions1	7	0.05	**0.07**	0	0.06	0.08	0	0.08	0.10	0	0.07	0.08	0	0.07	0.08
soc-LiveJournal1	60 041	236.88	-	29 508	213.74	-	4 319	22.27	-	3 530	24.13	-	1 314	37.77	-
soc-Slashdot0811	0	0.08	0.11	0	0.08	0.11	0	0.07	**0.08**	0	0.07	0.09	0	0.06	0.07
soc-Slashdot0902	0	0.07	**0.09**	0	0.07	0.10	0	0.09	0.11	0	0.08	0.10	0	0.10	0.12
soc-pokec-relationships	926 346	299.11	-	898 779	1 013.39	-	808 542	188.57	-	807 412	217.83	-	807 395	388.57	-
web-BerkStan	36 637	6.58	-	16 661	8.70	-	1 999	6.86	120.05	151	6.46	**6.83**	151	7.89	8.25
web-Google	2 810	1.57	**2.40**	1 254	2.42	3.66	361	1.75	2.95	46	1.88	2.47	46	7.97	9.24
web-NotreDame	13 464	1.03	-	6 052	2.03	-	2 460	0.40	-	2 061	0.56	**1.60**	117	2.44	2.57
web-Stanford	14 153	1.81	-	3 325	2.45	-	112	2.25	2.50	0	1.80	**1.99**	0	2.17	2.38
wiki-Talk	0	1.00	**1.71**	0	1.32	1.96	0	1.26	1.84	0	1.24	1.80	0	1.67	2.28
wiki-Vote	477	0.03	0.12	0	0.02	0.03	0	0.02	**0.02**	0	0.02	0.02	0	0.02	0.02

J State-of-the-Art Comparison for Increasing Transformations

We present detailed results on the solutions computed for the instances used in our experimental evaluation of exact and heuristic MWIS algorithms in Section 4.3.4. This includes the heuristic algorithms DynWVC and HILS, as well as our exact algorithms NonIncreasing, Cyclic-Fast, and Cyclic-Strong. Tables show the best solution weight $w_{\max}$ found by each algorithm and the time $t_{\max}$ (in seconds) required to compute it. Results for heuristic approaches show the best solution weight (and time to compute it) over five runs with different random seeds. The global best solution is highlighted in bold. Rows are highlighted in gray if one of our exact algorithms NonIncreasing, Cyclic-Fast, or Cyclic-Strong is able to solve the corresponding instances.

Table 35: Best solutions on FE instances.

Graph	t_{max}	w_{max}	t_{max}	w_{max}	t_{max}	w_{max}	t_{max}	w_{max}	t_{max}	w_{max}	t_{max}	w_{max}
FE instances	DynWVC1		DynWVC2		HILS		Cyclic-Fast		Cyclic-Strong		Non-Increasing	
fe_4elt2	961.12	427 755	974.87	427 755	759.23	427 646	0.11	**428 029**	0.11	**428 029**	0.18	428 016
fe_body	504.31	1 678 616	499.03	1 678 496	806.46	1 678 708	0.51	**1 680 182**	0.86	1 680 117	0.27	1 680 133
fe_ocean	983.53	7 222 521	379.75	7 220 128	999.57	7 069 279	18.85	6 591 832	19.04	6 591 537	18.85	6 597 698
fe_pwt	814.23	1 176 721	320.05	**1 176 784**	932.43	1 175 754	3.03	1 162 232	5.45	888 959	1.57	1 151 777
fe_rotor	961.76	**2 659 653**	874.68	2 659 473	973.92	2 650 132	13.95	2 531 152	20.55	2 538 117	13.56	2 532 168
fe_sphere	875.87	616 978	872.36	616 978	843.67	616 528	0.63	**617 816**	0.67	**617 816**	0.46	617 585
fe_tooth	353.21	3 031 269	619.96	3 031 385	994.97	3 032 819	0.26	**3 033 298**	0.26	**3 033 298**	0.27	**3 033 298**

Table 36: Best solutions on OSM instances.

Graph	t_{max}	w_{max}	t_{max}	w_{max}	t_{max}	w_{max}	t_{max}	w_{max}	t_{max}	w_{max}	t_{max}	w_{max}
OSM instances	DynWVC1		DynWVC2		HILS		Cyclic-Fast		Cyclic-Strong		Non-Increasing	
alabama-AM2	0.18	174 252	0.24	174 269	0.03	**174 309**	0.01	**174 309**	0.01	**174 309**	0.01	**174 309**
alabama-AM3	725.34	185 518	199.94	185 655	0.58	**185 744**	1.76	**185 744**	32.42	**185 744**	0.60	**185 744**
district-of-columbia-AM1	23.96	**196 475**	28.42	**196 475**	0.14	**196 475**	0.32	**196 475**	3.52	**196 475**	0.06	**196 475**
district-of-columbia-AM2	159.08	208 989	915.18	208 977	400.69	**209 132**	4.21	**209 132**	84.21	209 131	686.26	174 114
district-of-columbia-AM3	461.10	224 760	313.17	223 955	849.37	**227 613**	904.91	142 454	804.79	156 967	168.55	120 366
florida-AM2	0.18	**230 595**	0.53	**230 595**	0.04	**230 595**	0.00	**230 595**	0.00	**230 595**	0.00	**230 595**
florida-AM3	425.87	237 229	862.04	237 120	3.98	**237 333**	1.57	**237 333**	40.97	**237 333**	2.08	**237 333**
georgia-AM3	0.42	**222 652**	1.31	**222 652**	0.04	**222 652**	0.98	**222 652**	12.97	**222 652**	14.56	**222 652**
greenland-AM3	58.88	14 007	640.46	14 010	1.18	14 011	10.95	14 011	58.24	14 008	5.06	**14 012**
hawaii-AM2	1.89	125 270	1.63	125 270	0.20	**125 284**	0.09	**125 284**	0.10	**125 284**	0.13	**125 284**
hawaii-AM3	406.57	140 656	887.44	140 595	213.32	**141 035**	152.38	116 202	681.39	121 222	155.21	107 879
idaho-AM3	79.67	**77 145**	58.83	**77 145**	0.78	**77 145**	11.95	77 141	40.71	77 144	8.89	77 144
kansas-AM3	333.60	**87 976**	276.26	**87 976**	0.55	**87 976**	2.25	**87 976**	110.41	**87 976**	337.83	**87 976**
kentucky-AM2	3.23	**97 397**	2.92	**97 397**	0.26	**97 397**	0.23	**97 397**	0.44	**97 397**	0.26	**97 397**
kentucky-AM3	951.91	100 476	96.83	100 455	515.99	100 507	354.45	**100 510**	776.69	**100 510**	305.01	100 497
louisiana-AM3	8.63	**60 024**	0.18	60 002	0.01	**60 024**	0.05	**60 024**	0.11	**60 024**	0.15	**60 024**
maryland-AM3	0.79	**45 496**	0.59	**45 496**	0.01	**45 496**	0.11	**45 496**	0.15	**45 496**	0.14	**45 496**
massachusetts-AM2	0.25	**140 095**	0.74	**140 095**	0.01	**140 095**	0.04	**140 095**	0.05	**140 095**	0.03	**140 095**
massachusetts-AM3	980.11	145 852	270.28	145 862	0.77	**145 866**	1.39	**145 866**	31.04	**145 866**	0.76	**145 866**
mexico-AM3	0.71	**97 663**	2.28	**97 663**	0.02	**97 663**	0.96	**97 663**	21.19	**97 663**	0.67	**97 663**
new-hampshire-AM3	0.08	**116 060**	1.63	**116 060**	0.03	**116 060**	0.05	**116 060**	0.08	**116 060**	0.06	**116 060**
north-carolina-AM3	0.58	49 694	114.45	**49 720**	0.03	**49 720**	0.74	**49 720**	45.82	**49 720**	0.47	**49 720**
oregon-AM2	0.62	**165 047**	0.37	**165 047**	0.02	**165 047**	0.01	**165 047**	0.01	**165 047**	0.01	**165 047**
oregon-AM3	174.64	175 059	511.10	175 067	4.65	**175 078**	9.50	**175 078**	39.78	175 077	21.29	**175 078**
pennsylvania-AM3	0.06	**143 870**	0.14	**143 870**	0.02	**143 870**	0.07	**143 870**	0.12	**143 870**	0.16	**143 870**
rhode-island-AM2	7.75	184 537	13.90	184 576	0.24	**184 596**	0.41	**184 596**	4.37	**184 596**	0.27	**184 596**
rhode-island-AM3	230.53	201 470	711.97	201 359	30.15	**201 758**	44.88	167 162	82.02	167 162	45.46	166 103
utah-AM3	215.88	98 802	136.90	**98 847**	0.07	**98 847**	0.09	**98 847**	0.27	**98 847**	0.44	**98 847**
vermont-AM3	28.77	63 234	768.43	63 248	979.14	63 310	145.39	**63 312**	448.54	**63 312**	217.67	**63 312**
virginia-AM2	0.53	295 758	20.50	295 638	0.07	**295 867**	0.02	**295 867**	0.02	**295 867**	0.02	**295 867**
virginia-AM3	754.86	307 782	809.24	307 907	2.52	**308 305**	34.42	**308 305**	200.13	**308 305**	49.42	**308 305**
washington-AM2	1.24	**305 619**	13.35	**305 619**	0.25	**305 619**	0.06	**305 619**	0.07	**305 619**	0.08	**305 619**
washington-AM3	37.94	313 689	383.62	313 844	10.17	**314 288**	3.60	284 684	72.84	288 116	4.56	282 020
west-virginia-AM3	2.75	**47 927**	2.84	**47 927**	0.07	**47 927**	2.88	**47 927**	41.73	**47 927**	2.60	**47 927**

Table 37: Best solutions on mesh instances.

Graph	t_{max}	w_{max}	t_{max}	w_{max}	t_{max}	w_{max}	t_{max}	w_{max}	t_{max}	w_{max}	t_{max}	w_{max}
mesh instances	DynWVC1		DynWVC2		HILS		Cyclic-Fast		Cyclic-Strong		Non-Increasing	
beethoven	8.86	238 726	8.79	238 726	462.31	238 746	0.00	**238 794**	0.00	**238 794**	0.00	**238 794**
blob	39.91	854 843	40.00	854 843	351.91	855 004	0.02	**855 547**	0.02	**855 547**	0.02	**855 547**
buddha	879.42	56 757 052	797.35	56 757 052	999.94	55 490 134	1.75	**57 555 880**	1.77	**57 555 880**	2.24	**57 555 880**
bunny	702.13	3 683 000	695.55	3 683 000	964.60	3 681 696	0.11	**3 686 960**	0.13	**3 686 960**	0.11	**3 686 960**
cow	62.04	269 340	61.40	269 340	935.58	269 464	0.01	**269 543**	0.01	**269 543**	0.01	**269 543**
dragon	970.34	7 943 911	981.51	7 944 042	996.01	7 940 422	0.21	**7 956 530**	0.22	**7 956 530**	0.22	**7 956 530**
dragonsub	323.07	31 762 035	379.11	31 762 035	999.54	31 304 363	1.10	**32 213 898**	1.11	**32 213 898**	1.88	**32 213 898**
ecat	565.03	36 129 804	542.87	36 129 804	999.91	35 512 644	2.19	**36 650 298**	2.29	**36 650 298**	2.44	**36 650 298**
face	87.05	1 218 510	86.38	1 218 510	228.77	1 218 565	0.03	**1 219 418**	0.03	**1 219 418**	0.03	**1 219 418**
fandisk	8.26	462 950	8.42	462 950	232.96	463 090	0.01	**463 288**	0.01	**463 288**	0.01	**463 288**
feline	730.80	2 204 925	734.34	2 204 925	640.98	2 204 911	0.09	**2 207 219**	0.08	**2 207 219**	0.09	**2 207 219**
gameguy	519.12	2 323 941	525.93	2 323 941	736.64	2 322 824	0.05	**2 325 878**	0.05	**2 325 878**	0.05	**2 325 878**
gargoyle	29.25	1 058 496	29.11	1 058 496	724.41	1 058 652	0.03	**1 059 559**	0.03	**1 059 559**	0.03	**1 059 559**
turtle	982.00	14 215 429	976.57	14 213 516	999.68	14 151 616	0.42	**14 263 005**	0.43	**14 263 005**	0.56	**14 263 005**
venus	559.29	305 571	556.38	305 571	130.83	305 724	0.01	**305 749**	0.01	**305 749**	0.01	**305 749**

Table 38: Best solutions on SNAP instances.

Graph	t_{max}	w_{max}	t_{max}	w_{max}	t_{max}	w_{max}	t_{max}	w_{max}	t_{max}	w_{max}	t_{max}	w_{max}
SNAP instances	DynWVC1		DynWVC2		HILS		Cyclic-Fast		Cyclic-Strong		Non-Increasing	
as-skitter	989.05	123 613 404	383.97	123 273 938	999.32	122 658 804	346.69	124 137 148	354.71	**124 137 365**	431.90	124 136 621
ca-AstroPh	32.46	797 475	125.05	797 480	13.47	**797 510**	0.02	**797 510**	0.02	**797 510**	0.02	**797 510**
ca-CondMat	114.85	1 147 814	27.75	1 147 845	50.90	**1 147 950**	0.01	**1 147 950**	0.01	**1 147 950**	0.01	**1 147 950**
ca-GrQc	4.87	**286 489**	1.93	**286 489**	0.34	**286 489**	0.00	**286 489**	0.00	**286 489**	0.00	**286 489**
ca-HepPh	13.21	581 014	17.34	581 028	7.73	**581 039**	0.01	**581 039**	0.01	**581 039**	0.01	**581 039**
ca-HepTh	6.57	561 982	5.30	561 974	4.68	**562 004**	0.00	**562 004**	0.00	**562 004**	0.00	**562 004**
email-Enron	454.49	2 464 887	594.93	2 464 890	71.07	2 464 922	0.02	**2 464 935**	0.03	**2 464 935**	0.02	**2 464 935**
email-EuAll	134.83	**25 286 322**	132.62	**25 286 322**	338.14	**25 286 322**	0.07	**25 286 322**	0.07	**25 286 322**	0.06	**25 286 322**
p2p-Gnutella04	1.46	679 105	2.34	**679 111**	94.12	**679 111**	0.01	**679 111**	0.01	**679 111**	0.01	**679 111**
p2p-Gnutella05	1.15	554 926	3.55	554 931	135.17	**554 943**	0.01	**554 943**	0.01	**554 943**	0.01	**554 943**
p2p-Gnutella06	525.35	548 611	186.97	548 611	1.29	**548 612**	0.01	**548 612**	0.01	**548 612**	0.01	**548 612**
p2p-Gnutella08	0.15	434 575	0.18	**434 577**	0.12	**434 577**	0.00	**434 577**	0.00	**434 577**	0.00	**434 577**
p2p-Gnutella09	0.39	**568 439**	0.28	**568 439**	0.09	**568 439**	0.00	**568 439**	0.00	**568 439**	0.00	**568 439**
p2p-Gnutella24	8.01	**1 984 567**	5.51	**1 984 567**	3.17	**1 984 567**	0.01	**1 984 567**	0.01	**1 984 567**	0.01	**1 984 567**
p2p-Gnutella25	2.66	**1 701 967**	2.20	**1 701 967**	1.17	**1 701 967**	0.01	**1 701 967**	0.01	**1 701 967**	0.01	**1 701 967**
p2p-Gnutella30	8.83	2 787 903	132.71	2 787 899	15.14	**2 787 907**	0.01	**2 787 907**	0.01	**2 787 907**	0.02	**2 787 907**
p2p-Gnutella31	70.88	4 776 960	47.97	4 776 961	115.01	**4 776 986**	0.02	**4 776 986**	0.02	**4 776 986**	0.03	**4 776 986**
roadNet-CA	999.98	109 586 054	999.90	109 582 579	1 000.00	106 584 645	1.94	**111 360 828**	1.86	**111 360 828**	4.09	**111 360 828**
roadNet-PA	511.59	60 990 177	469.18	60 990 177	999.94	60 037 011	0.96	**61 731 589**	1.04	**61 731 589**	1.83	**61 731 589**
roadNet-TX	789.43	77 672 388	694.33	77 672 388	999.97	76 347 666	1.29	**78 599 946**	1.29	**78 599 946**	3.42	**78 599 946**
soc-Epinions1	290.84	5 690 651	272.56	5 690 773	253.10	5 690 874	0.08	**5 690 970**	0.08	**5 690 970**	0.08	**5 690 970**
soc-LiveJournal1	999.99	279 150 686	999.99	279 231 875	1 000.00	255 079 926	51.33	284 036 222	44.19	**284 036 239**	39.36	283 970 295
soc-Slashdot0811	238.18	5 660 385	880.68	5 660 555	446.95	5 660 787	0.09	**5 660 899**	0.08	**5 660 899**	0.08	**5 660 899**
soc-Slashdot0902	270.85	5 971 308	435.90	5 971 476	604.07	5 971 664	0.11	**5 971 849**	0.11	**5 971 849**	0.12	**5 971 849**
soc-pokec-relationships	999.85	**83 223 668**	999.13	83 155 217	1 000.00	82 021 946	254.59	76 075 111	488.31	76 075 700	228.07	76 063 476
web-BerkStan	194.20	43 640 833	164.10	43 637 382	998.73	43 424 373	6.74	**43 907 482**	8.05	**43 907 482**	16.01	**43 907 482**
web-Google	349.08	56 209 005	324.65	56 206 250	995.92	56 008 278	1.72	**56 326 504**	6.44	**56 326 504**	2.17	**56 326 504**
web-NotreDame	949.84	26 010 791	905.72	26 009 287	997.00	26 002 793	1.60	**26 016 941**	2.74	**26 016 941**	1.36	**26 016 941**
web-Stanford	943.85	17 748 798	671.32	17 741 043	999.50	17 709 827	1.68	**17 792 930**	1.86	**17 792 930**	1.71	**17 792 930**
wiki-Talk	951.51	235 836 837	972.93	235 836 913	999.69	235 818 823	1.29	**235 837 346**	1.29	**235 837 346**	1.31	**235 837 346**
wiki-Vote	188.76	500 075	0.32	**500 079**	10.34	**500 079**	0.02	**500 079**	0.02	**500 079**	0.02	**500 079**

K Convergence Plots for Increasing Transformations

We present convergence plots for samples of instances from each instance class (FE, mesh, OSM, SNAP) used in our experimental evaluation of exact and heuristic MWIS algorithms (Section 4.3.4) This includes the heuristic algorithms DynWVC and HILS, as well as the exact algorithms $\text{BnR}_{\text{dense}}$, BnR_{full}, NonIncreasing, Cyclic-Fast, and Cyclic-Strong. The results for heuristic algorithms are event-based geometric average values over five runs with different random seeds.

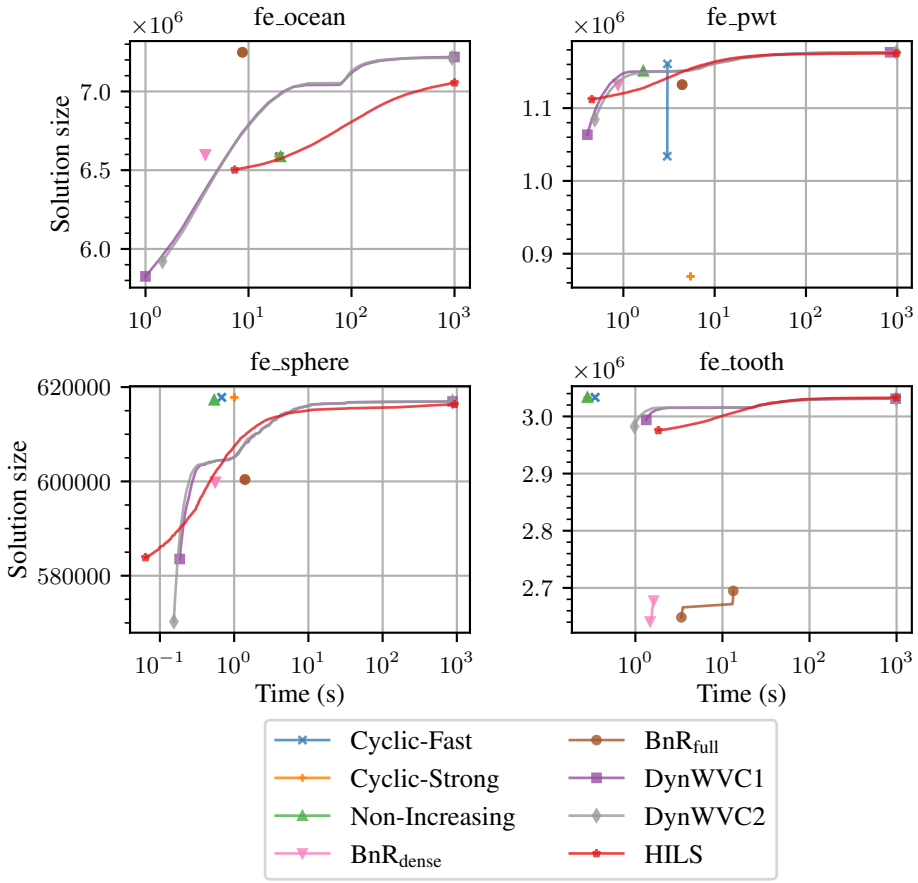


Figure 11: Convergence plots for FE instances fe_ocean (top left), fe_pwt (top right), fe_sphere (bottom left), and fe_tooth (bottom right).

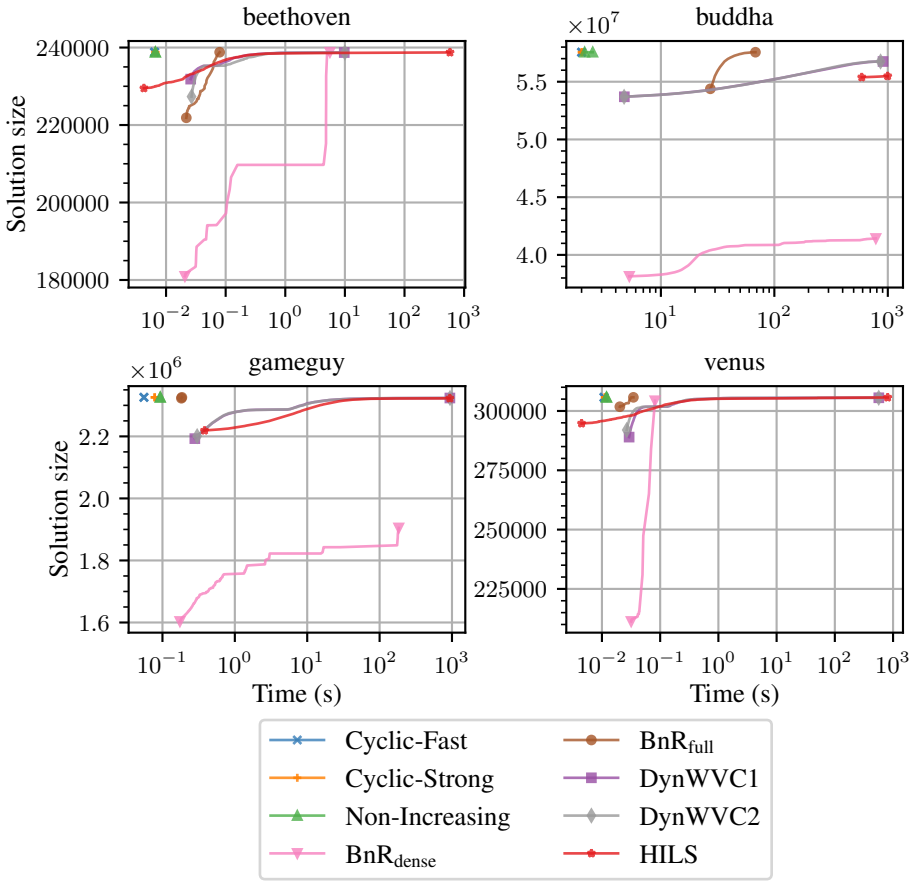


Figure 12: Convergence plots for mesh instances beethoven (top left), buddha (top right), gameguy (bottom left), and venus (bottom right).

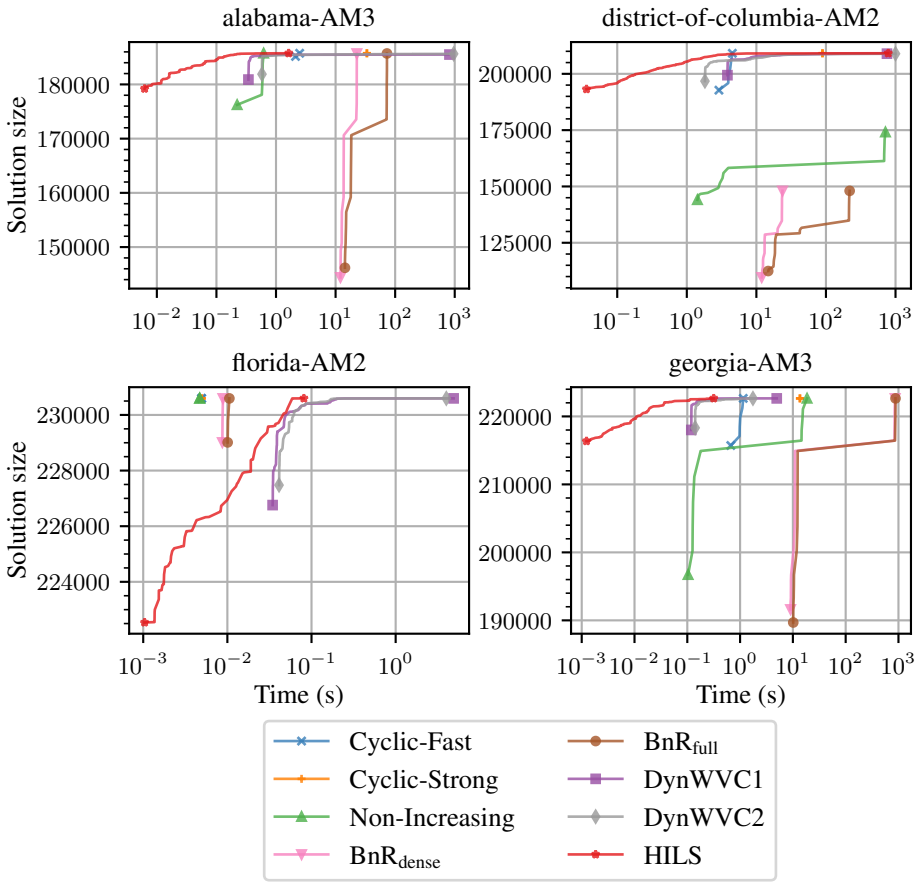


Figure 13: Convergence plots for OSM instances alabama-AM3 (top left), district-of-columbia-AM2 (top right), florida-AM2 (bottom left), and georgia-AM3 (bottom right).

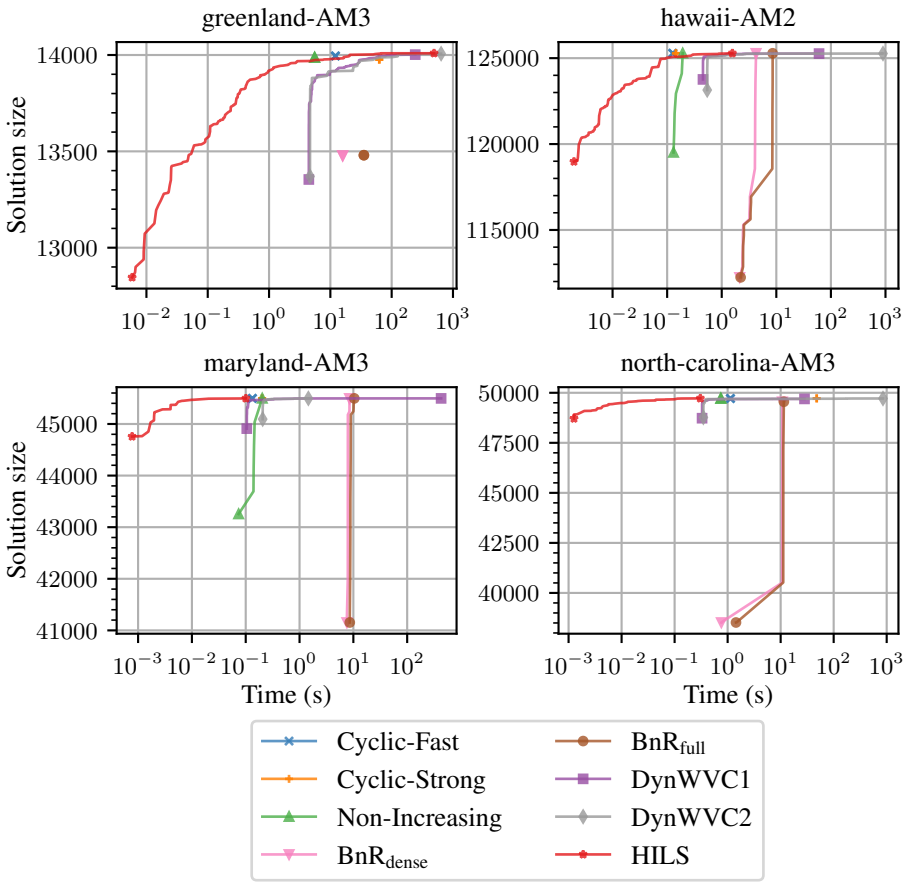


Figure 14: Convergence plots for OSM instances greenland-AM3 (top left), hawaii-AM2 (top right), maryland-AM3 (bottom left), and north-carolina-AM3 (bottom right).

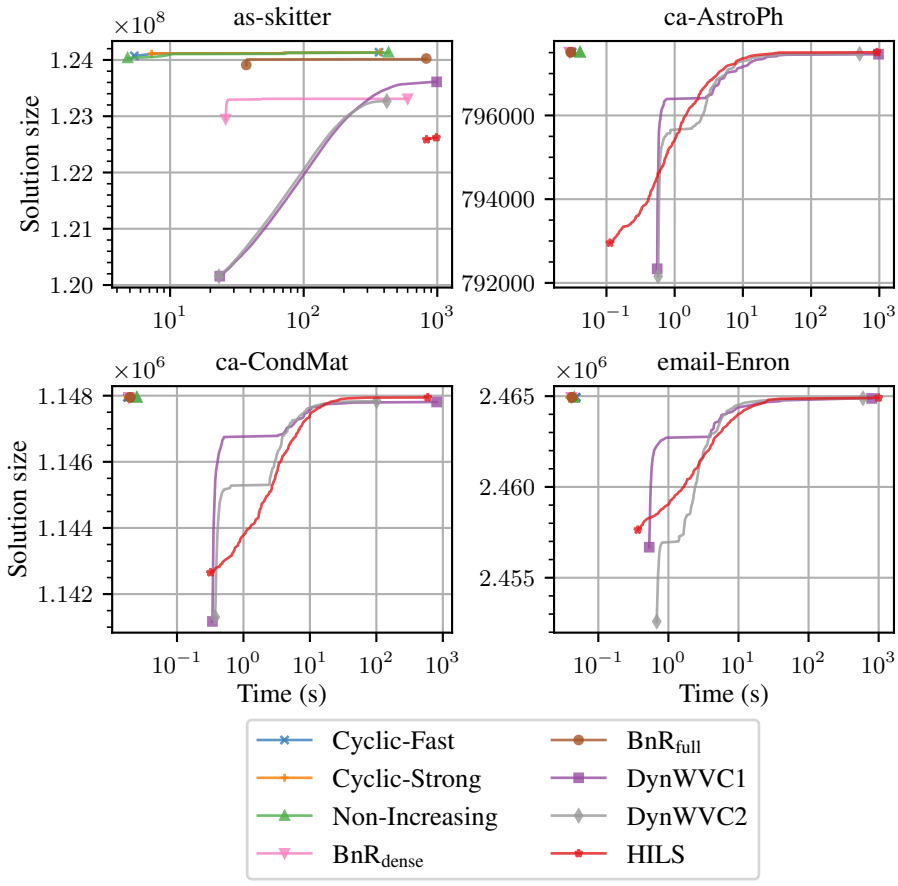


Figure 15: Convergence plots for SNAP instances as-skitter (top left), ca-AstroPh (top right), ca-CondMat (bottom left), and email-Enron (bottom right).

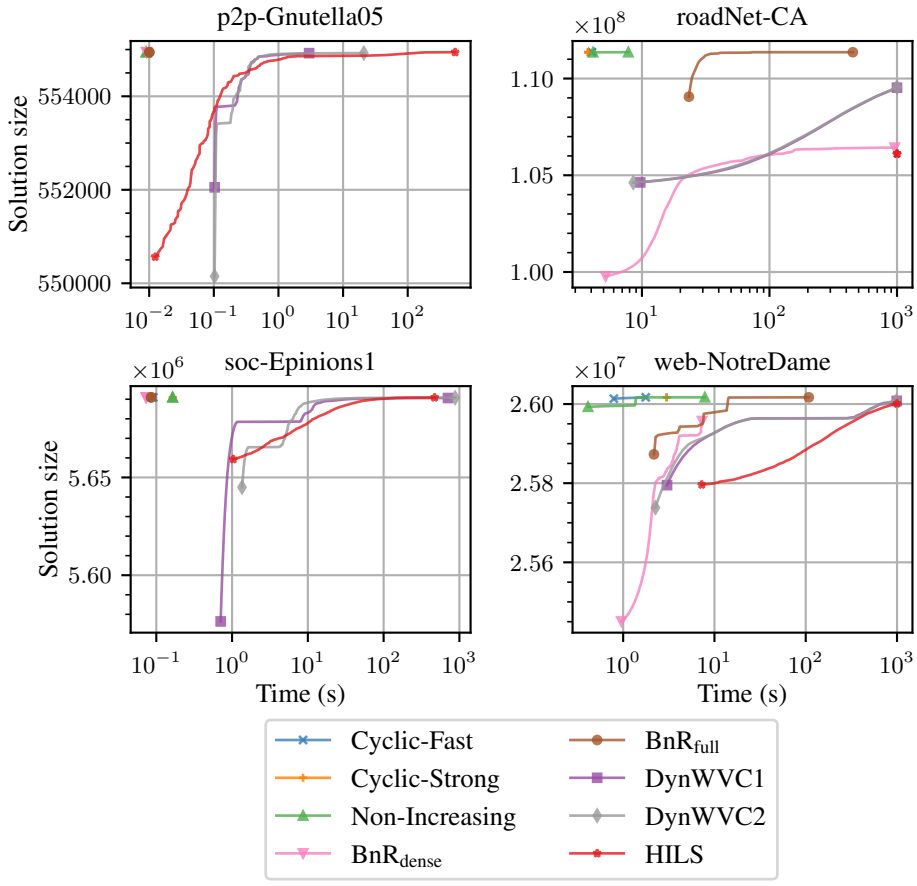


Figure 16: Convergence plots for SNAP instances p2p-Gnutella05 (top left), roadNet-CA (top right), soc-Epinions1 (bottom left), and web-NotreDame (bottom right).

L Reduced Rudy Instances for Maximum Cuts

Finally, we present the results of applying our reduction algorithm presented in Section 5.3 on the rudy instances of the BiqMac Library [Wie18]. For each instance, we present the reduction efficiency $e(G) = 1 - n(\mathcal{K})/n$ for a reduced instance $\mathcal{K}$.

Table 39: Reduction efficiency for rudy instances.

Name	n	m	$e(G)$
g05_100	100	2 475	0.00
g05_60	60	885	0.00
g05_80	80	1 580	0.00
pm1d_100	100	4 901	0.00
pm1d_80	80	3 128	0.00
pm1s_100	100	495	0.00
pm1s_80	79	316	0.01
pw01_100	100	495	0.00
pw05_100	100	2 475	0.00
pw09_100	100	4 455	0.00
w01_100	100	470	0.00
w05_100	100	2 356	0.00
w09_100	100	4 245	0.00

Publications and Supervised Theses

In Conference Proceedings

Sebastian Lamm and Peter Sanders. "Communication-Efficient Massively Distributed Connected Components". In: *to appear IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS)*. 2022

Demian Hespe, Sebastian Lamm, and Christian Schorr. "Targeted Branching for the Maximum Independent Set Problem". In: *Intl. Symp. on Experimental Algorithms (SEA)*. 2021, 17:1–17:21. DOI: 10.4230/LIPIcs.SEA.2021.17

Alexander Gellner, Sebastian Lamm, Christian Schulz, Darren Strash, and Bogdán Zaválnij. "Boosting Data Reduction for the Maximum Weight Independent Set Problem Using Increasing Transformations". In: *Symp. on Algorithm Engineering and Experiments (ALENEX)*. 2021, pages 128–142. DOI: 10.1137/1.9781611976472.10

Demian Hespe, Sebastian Lamm, Christian Schulz, and Darren Strash. "WeGotYouCovered: The Winning Solver from the PACE 2019 Challenge, Vertex Cover Track". In: *SIAM Workshop on Combinatorial Scientific Computing (CSC)*. 2020, pages 1–11. DOI: 10.1137/1.9781611976229.1

Damir Ferizovic, Demian Hespe, Sebastian Lamm, Matthias Mnich, Christian Schulz, and Darren Strash. "Engineering Kernelization for Maximum Cut". In: *Symp. on Algorithm Engineering and Experiments (ALENEX)*. 2020, pages 27–41. DOI: 10.1137/1.9781611976007.3

Sebastian Lamm, Christian Schulz, Darren Strash, Robert Williger, and Huashuo Zhang. "Exactly Solving the Maximum Weight Independent Set Problem on Large Real-World Graphs". In: *Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2019, pages 144–158. DOI: 10.1137/1.9781611975499.12

Daniel Funke, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. "Communication-Free Massively Distributed Graph Generation". In: *IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS)*. 2018, pages 336–347. DOI: 10.1109/IPDPS.2018.00043

Timo Bingmann, Michael Axtmann, Emanuel Jöbstl, Sebastian Lamm, Huyen Chau Nguyen, Alexander Noe, Sebastian Schlag, Matthias Stumpp, Tobias Sturm, and Peter Sanders. "Thrill: High-Performance Algorithmic Distributed Batch Data Processing with C++". In: *IEEE Intl. Conf. on Big Data (BigData)*. 2016, pages 172–183. DOI: 10.1109/BigData.2016.7840603

Jakob Dahlum, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. "Accelerating Local Search for the Maximum Independent Set Problem". In: *Intl. Symp. on Experimental Algorithms (SEA)*. 2016, pages 118–133. DOI: 10.1007/978-3-319-38851-9_9

Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. "Finding Near-Optimal Independent Sets at Scale". In: *Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2016, pages 138–150. DOI: 10.1137/1.9781611974317.12

Sebastian Lamm, Peter Sanders, and Christian Schulz. "Graph Partitioning for Independent Sets". In: *Intl. Symp. on Experimental Algorithms (SEA)*. 2015, pages 68–81. DOI: 10.1007/978-3-319-20086-6_6

Journal Articles

Daniel Funke, Sebastian Lamm, Ulrich Meyer, Manuel Penschuck, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. "Communication-Free Massively Distributed Graph Generation". In: *J. Parallel Distributed Comput.* 131 (2019), pages 200–217. DOI: 10.1016/j.jpdc.2019.03.011

Peter Sanders, Sebastian Lamm, Lorenz Hübschle-Schneider, Emanuel Schrade, and Carsten Dachsbacher. "Efficient Parallel Random Sampling - Vectorized, Cache-Efficient, and Online". In: *ACM Trans. Math. Softw.* 44.3 (2018), 29:1–29:14. DOI: 10.1145/3157734

Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. "Finding Near-Optimal Independent Sets at Scale". In: *J. Heuristics* 23.4 (2017), pages 207–229. DOI: 10.1007/s10732-017-9337-x

Technical Reports

Demian Hespe, Sebastian Lamm, and Christian Schorr. "Targeted Branching for the Maximum Independent Set Problem". In: *Computing Research Repository (CoRR)* abs/2102.01540 (2021). DOI: 10.48550/ARXIV.2102.01540

Faisal N. Abu-Khzam, Sebastian Lamm, Matthias Mnich, Alexander Noe, Christian Schulz, and Darren Strash. "Recent Advances in Practical Data Reduction". In: *Computing Research Repository (CoRR)* abs/2012.12594 (2020). DOI: 10.48550/ARXIV.2012.12594

Manuel Penschuck, Ulrik Brandes, Michael Hamann, Sebastian Lamm, Ulrich Meyer, Ilya Safro, Peter Sanders, and Christian Schulz. "Recent Advances in Scalable Network Generation". In: *Computing Research Repository (CoRR)* abs/2003.00736 (2020). DOI: 10.48550/ARXIV.2003.00736

Alexander Gellner, Sebastian Lamm, Christian Schulz, Darren Strash, and Bogdán Zaválnij. "Boosting Data Reduction for the Maximum Weight Independent Set Problem Using Increasing Transformations". In: *Computing Research Repository (CoRR)* abs/2008.05180 (2020). DOI: 10.48550/ARXIV.2008.05180

Demian Hespe, Sebastian Lamm, Christian Schulz, and Darren Strash. "WeGotYouCovered: The Winning Solver from the PACE 2019 Implementation Challenge, Vertex Cover Track". In: *Computing Research Repository (CoRR)* abs/1908.06795 (2019). DOI: 10.48550/ARXIV.1908.06795

Damir Ferizovic, Demian Hespe, Sebastian Lamm, Matthias Mnich, Christian Schulz, and Darren Strash. "Engineering Kernelization for Maximum Cut". In: *Computing Research Repository (CoRR)* abs/1905.10902 (2019). DOI: 10.48550/ARXIV.1905.10902

Sebastian Lamm, Christian Schulz, Darren Strash, Robert Williger, and Huashuo Zhang. "Exactly Solving the Maximum Weight Independent Set Problem on Large Real-World Graphs". In: *Computing Research Repository (CoRR)* abs/1810.10834 (2018). DOI: 10.48550/ARXIV.1810.10834

Daniel Funke, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. "Communication-Free Massively Distributed Graph Generation". In: *Computing Research Repository (CoRR)* abs/1710.07565 (2017). DOI: 10.48550/ARXIV.1710.07565

Peter Sanders, Sebastian Lamm, Lorenz Hübschle-Schneider, Emanuel Schrade, and Carsten Dachsbacher. "Efficient Random Sampling - Parallel, Vectorized, Cache-Efficient, and Online". In: *Computing Research Repository (CoRR)* abs/1610.05141 (2016). DOI: 10.48550/ARXIV.1610.05141

Timo Bingmann, Michael Axtmann, Emanuel Jöbstl, Sebastian Lamm, Huyen Chau Nguyen, Alexander Noe, Sebastian Schlag, Matthias Stumpp, Tobias Sturm, and Peter Sanders. "Thrill: High-Performance Algorithmic Distributed Batch Data Processing with C++". In: *Computing Research Repository (CoRR)* abs/1608.05634 (2016). DOI: 10.48550/ARXIV.1608.05634

Jakob Dahlum, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. "Accelerating Local Search for the Maximum Independent Set Problem". In: *Computing Research Repository (CoRR)* abs/1602.01659 (2016). DOI: 10.48550/ARXIV.1602.01659

Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. "Finding Near-Optimal Independent Sets at Scale". In: *Computing Research Repository (CoRR)* abs/1509.00764 (2015). DOI: 10.48550/ARXIV.1509.00764

Sebastian Lamm, Peter Sanders, and Christian Schulz. "Graph Partitioning for Independent Sets". In: *Computing Research Repository (CoRR)* abs/1502.01687 (2015). DOI: 10.48550/ARXIV.1502.01687

Theses

Sebastian Lamm. "Communication Efficient Algorithms for Generating Massive Networks". Master's thesis. Karlsruhe Institute of Technology, Germany, 2017

Sebastian Lamm. "Evolutionary Algorithms for Independent Sets". Bachelor's thesis. Karlsruhe Institute of Technology, Germany, 2014

Supervised Theses

Tim Niklas Uhl. "Communication Efficient Triangle Counting". Master's thesis. Karlsruhe Institute of Technology, Germany, 2021

Adrian Feilhauer. "Communication-Free Generation of Graphs with Planted Communities". Master's thesis. Karlsruhe Institute of Technology, Germany, 2020

Christian Schorr. "Improved Branching Strategies for Maximum Independent Sets". Bachelor's thesis. Karlsruhe Institute of Technology, Germany, 2020

Alexander Gellner. "Engineering Generalized Reductions for the Maximum Weight Independent Set Problem". Master's thesis. Karlsruhe Institute of Technology, Germany, 2020

Damir Ferizovic. "A Practical Analysis of Kernelization Techniques for the Maximum Cut Problem". Master's thesis. Karlsruhe Institute of Technology, Germany, 2019

Tom George. "Distributed Kernelization for Independent Sets". Bachelor's thesis. Karlsruhe Institute of Technology, Germany, 2018

Bibliography

[Abu+04] Faisal N. Abu-Khzam, Rebecca L. Collins, Michael R. Fellows, Michael A. Langston, W. Henry Suters, and Christopher T. Symons. "Kernelization Algorithms for the Vertex Cover Problem: Theory and Experiments". In: *Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2004, pages 62–69. [see pages 2, 11, 28, 37]

[Abu+07] Faisal N. Abu-Khzam, Michael R. Fellows, Michael A. Langston, and W. Henry Suters. "Crown Structures for Vertex Cover Kernelization". In: *Theory Comput. Syst.* 41.3 (2007), pages 411–430. DOI: 10.1007/s00224-007-1328-0. [see page 130]

[Abu+20] Faisal N. Abu-Khzam, Sebastian Lamm, Matthias Mnich, Alexander Noe, Christian Schulz, and Darren Strash. "Recent Advances in Practical Data Reduction". In: *Computing Research Repository (CoRR)* abs/2012.12594 (2020). DOI: 10.48550/ARXIV.2012.12594. [see pages 1, 2, 11, 25, 26, 89, 90, 202]

[AC19] Maram Alsahafy and Lijun Chang. "Computing Maximum Independent Sets over Large Sparse Graphs". In: *Intl. Conf. on Web Information Systems Engineering (WISE)*. 2019, pages 711–727. DOI: 10.1007/978-3-030-34223-4_45. [see pages 3, 4, 31, 32, 40, 73, 75, 86]

[Age94] Alexander A. Ageev. "On Finding Critical Independent and Vertex Sets". In: *SIAM J. Discret. Math.* 7.2 (1994), pages 293–295. DOI: 10.1137/S0895480191217569. [see pages 96, 97]

[AI16] Takuya Akiba and Yoichi Iwata. "Branch-and-Reduce Exponential/FPT Algorithms in Practice: A Case Study of Vertex Cover". In: *Theor. Comput. Sci.* 609 (2016), pages 211–225. DOI: 10.1016/j.tcs.2015.09.023. [see pages 1–5, 15, 18, 26–28, 33, 37–42, 47–49, 52–54, 57–59, 64–66, 73–75, 79–82, 90, 97, 99, 105, 106, 108, 121]

[AKK11] Ferhat Ay, Manolis Kellis, and Tamer Kahveci. "SubMAP: Aligning Metabolic Pathways with Subnetwork Mappings". In: *J. Comput. Biol.* 18.3 (2011), pages 219–235. DOI: 10.1089/cmb.2010.0280. [see page 90]

[Ale+03] Gabriela Alexe, Peter L. Hammer, Vadim V. Lozin, and Dominique de Werra. "Struction Revisited". In: *Discret. Appl. Math.* 132.1-3 (2003), pages 27–46. DOI: 10.1016/S0166-218X(03)00388-3. [see pages 112, 113]

[AO09] Emely Arráiz and Oswaldo Olivo. "Competitive Simulated Annealing and Tabu Search Algorithms for the Max-Cut Problem". In: *Genetic and Evolutionary Computation Conf. (GECCO)*. 2009, pages 1797–1798. DOI: 10.1145/1569901.1570167. [see page 130]

[APR98] James Abello, Panos M. Pardalos, and Mauricio G. C. Resende. "On Maximum Clique Problems in Very Large Graphs". In: *External Memory Algorithms*. 1998, pages 119–130. DOI: 10.1090/dimacs/050/06. [see page 32]

[ARW12] Diogo V. Andrade, Mauricio G. C. Resende, and Renato F. Werneck. "Fast Local Search for the Maximum Independent Set Problem". In: *J. Heuristics* 18.4 (2012), pages 525–547. DOI: 10.1007/s10732-012-9196-4. [see pages 4, 18, 27, 29, 41, 44, 48, 49, 56–58, 64, 65, 93, 99]

[Bab94] Luitpold Babel. "A Fast Algorithm for the Maximum Weight Clique Problem". In: *Computing* 52.1 (1994), pages 31–38. DOI: 10.1007/BF02243394. [see page 91]

[Bäc96] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice - Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. 1996. ISBN: 9780195099713. [see page 44]

[Bad+18] David A. Bader, Andrea Kappes, Henning Meyerhenke, Peter Sanders, Christian Schulz, and Dorothea Wagner. "Benchmarking for Graph Clustering and Partitioning". In: *Encyclopedia of Social Network Analysis and Mining*. 2018. DOI: 10.1007/978-1-4939-7131-2_23. [see pages 18, 151]

[Bal70] Michel L. Balinski. "On a Selection Problem". In: *Manage. Sci.* 17.3 (1970), pages 230–231. DOI: 10.1287/mnsc.17.3.230. [see page 97]

[Bar+16] Lukas Barth, Benjamin Niedermann, Martin Nöllenburg, and Darren Strash. "Temporal Map Labeling: A New Unified Framework with Experiments". In: *ACM Intl. Conf. on Advances in Geographic Information Systems (SIGSPATIAL)*. 2016, 23:1–23:10. DOI: 10.1145/2996913.2996957. [see pages 2, 19, 90, 147]

[Bar+88] Francisco Barahona, Martin Grötschel, Michael Jünger, and Gerhard Reinelt. "An Application of Combinatorial Optimization to Statistical Physics and Circuit Layout Design". In: *Oper. Res.* 36.3 (1988), pages 493–513. DOI: 10.1287/opre.36.3.493. [see pages v, vii, 3, 129, 130]

[Bar82] Francisco Barahona. "On the Computational Complexity of Ising Spin Glass Models". In: *J. Phys. A: Mathematical and General* 15.10 (1982), pages 3241–3253. DOI: 10.1088/0305-4470/15/10/028. [see pages v, vii, 3, 129]

[Bar96] Francisco Barahona. "Network Design Using Cut Inequalities". In: *SIAM J. Optim.* 6.3 (1996), pages 823–837. DOI: 10.1137/S1052623494279134. [see page 129]

[Bat+14] Mikhail Batsyn, Boris Goldengorin, Evgeny Maslov, and Panos M. Pardalos. "Improvements to MCS Algorithm for the Maximum Clique Problem". In: *J. Comb. Optim.* 27.2 (2014), pages 397–416. DOI: 10.1007/s10878-012-9592-6. [see pages 31, 58, 65]

[Ben+11] Thierry Benoist, Bertrand Estellon, Frédéric Gardi, Romain Megel, and Karim Nouioua. "LocalSolver 1.x: A Black-Box Local-Search Solver for 0-1 Programming". In: *4OR* 9.3 (2011), pages 299–316. DOI: 10.1007/s10288-011-0165-9. [see pages 130, 140]

[BH13] Una Benlic and Jin-Kao Hao. "Breakout Local Search for the Max-Cut Problem". In: *Eng. Appl. Artif. Intell.* 26.3 (2013), pages 1162–1173. DOI: 10.1016/j.engappai.2012.09.001. [see page 130]

[Bin+16a] Timo Bingmann, Michael Axtmann, Emanuel Jöbstl, Sebastian Lamm, Huyen Chau Nguyen, Alexander Noe, Sebastian Schlag, Matthias Stumpp, Tobias Sturm, and Peter Sanders. "Thrill: High-Performance Algorithmic Distributed Batch Data Processing with C++". In: *IEEE Intl. Conf. on Big Data (BigData)*. 2016, pages 172–183. DOI: 10.1109/BigData.2016.7840603. [see page 201]

[Bin+16b] Timo Bingmann, Michael Axtmann, Emanuel Jöbstl, Sebastian Lamm, Huyen Chau Nguyen, Alexander Noe, Sebastian Schlag, Matthias Stumpp, Tobias Sturm, and Peter Sanders. "Thrill: High-Performance Algorithmic Distributed Batch Data Processing with C++". In: *Computing Research Repository (CoRR)* abs/1608.05634 (2016). DOI: 10.48550/ARXIV.1608.05634. [see page 203]

[BJR89] Francisco Barahona, Michael Jünger, and Gerhard Reinelt. "Experiments in Quadratic 0-1 Programming". In: *Math. Program.* 44.1-3 (1989), pages 127–137. DOI: 10.1007/BF01587084. [see page 12]

[BK73] Coenraad Bron and Joep Kerbosch. "Finding All Cliques of an Undirected Graph (Algorithm 457)". In: *Commun. ACM* 16.9 (1973), pages 575–576. DOI: 10.1145/362342.362367. [see page 32]

[BK94] Thomas Bäck and Sami Khuri. "An Evolutionary Heuristic for the Maximum Independent Set Problem". In: *IEEE Conf. on Evolutionary Computation (ICEC)*. 1994, pages 531–535. DOI: 10.1109/ICEC.1994.350004. [see pages 42, 43, 46, 50]

[BMZ02] Samuel Burer, Renato D. C. Monteiro, and Yin Zhang. "Rank-Two Relaxation Heuristics for MAX-CUT and Other Binary Quadratic Programs". In: *SIAM J. Optim.* 12.2 (2002), pages 503–521. DOI: 10.1137/S1052623400382467. [see pages 131, 133]

[Bol+11] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. "Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks". In: *Intl. Conf. on World Wide Web (WWW)*. 2011, pages 587–596. DOI: 10.1145/1963405.1963488. [see pages 18, 149]

[Bou+12] Nicolas Bourgeois, Bruno Escoffier, Vangelis T. Paschos, and Johan M. M. van Rooij. "Fast Algorithms for Max Independent Set". In: *Algorithmica* 62.1-2 (2012), pages 382–415. DOI: 10.1007/s00453-010-9460-7. [see page 74]

[BP01] Roberto Battiti and Marco Protasi. "Reactive Local Search for the Maximum Clique Problem". In: *Algorithmica* 29.4 (2001), pages 610–637. DOI: 10.1007/s004530010074. [see page 33]

[Bré79] Daniel Brélaz. "New Methods to Color Vertices of a Graph". In: *Commun. ACM* 22.4 (1979), pages 251–256. DOI: 10.1145/359094.359101. [see page 99]

[Bri59] Rene De La Briandais. "File Searching Using Variable Length Keys". In: *Western Joint Computer Conf. (IRE-AIEE-ACM)*. 1959, pages 295–298. DOI: 10.1145/1457838.1457895. [see page 139]

[Bro+90] Andries E. Brouwer, James B. Shearer, Neil J. A. Sloane, and Warren D. Smith. "A New Table of Constant Weight Codes". In: *IEEE Trans. Inf. Theory* 36.6 (1990), pages 1334–1380. DOI: 10.1109/18.59932. [see pages 2, 90]

[BT07] Sergiy Butenko and Svyatoslav Trukhanov. "Using Critical Sets to Solve the Maximum Independent Set Problem". In: *Oper. Res. Lett.* 35.4 (2007), pages 519–524. DOI: 10.1016/j.orl.2006.07.004. [see pages 5, 28, 91, 96, 97, 107]

[But+02] Sergiy Butenko, Panos M. Pardalos, Ivan Sergienko, Vladimir Shylo, and Petro Stetsyuk. "Finding Maximum Independent Sets in Graphs Arising from Coding Theory". In: *ACM Symp. on Applied Computing (SAC)*. 2002, pages 542–546. DOI: 10.1145/508791.508897. [see pages 2, 28, 33]

[BV04] Paolo Boldi and Sebastiano Vigna. "The WebGraph Framework I: Compression Techniques". In: *Intl. Conf. on World Wide Web (WWW)*. 2004, pages 595–602. DOI: 10.1145/988672.988752. [see pages 18, 149]

[BY86] Egon Balas and Chang Sung Yu. "Finding a Maximum Clique in an Arbitrary Graph". In: *SIAM J. Comput.* 15.4 (1986), pages 1054–1068. DOI: 10.1137/0215075. [see pages 26, 91]

[BZ03] Pavel A. Borisovsky and Marina S. Zavolovskaya. "Experimental Comparison of Two Evolutionary Algorithms for the Independent Set Problem". In: *Workshops on Applications of Evolutionary Computing (EvoWorkshops)*. 2003, pages 154–164. DOI: 10.1007/3-540-36605-9_15. [see pages 42, 43, 46]

[Cai+13] Shaowei Cai, Kaile Su, Chuan Luo, and Abdul Sattar. "NuMVC: An Efficient Local Search Algorithm for Minimum Vertex Cover". In: *J. Artif. Intell. Res.* 46 (2013), pages 687–716. DOI: 10.1613/jair.3907. [see pages 29–31]

[Cai+18] Shaowei Cai, Wenying Hou, Jinkun Lin, and Yuanjie Li. "Improving Local Search for Minimum Weight Vertex Cover by Dynamic Strategies". In: *Intl. Joint Conf. on Artificial Intelligence (IJCAI)*. 2018, pages 1412–1418. DOI: 10.24963/ijcai.2018/196. [see pages 19, 93, 106, 107, 124, 154]

[Cai+21] Shaowei Cai, Jinkun Lin, Yiyuan Wang, and Darren Strash. "A Semi-Exact Algorithm for Quickly Computing A Maximum Weight Clique in Large Sparse Graphs". In: *J. Artif. Intell. Res.* 72 (2021), pages 39–67. DOI: 10.1613/jair.1.12327. [see page 95]

[Cai15] Shaowei Cai. "Balance between Complexity and Quality: Local Search for Minimum Vertex Cover in Massive Graphs". In: *Intl. Joint Conf. on Artificial Intelligence (IJCAI)*. 2015, pages 747–753. [see pages 29–31, 93, 95]

[CFJ04] Benny Chor, Mike Fellows, and David W. Juedes. "Linear Kernels in Linear Time, or How to Save k Colors in $\mathcal{O}(n^2)$ Steps". In: *Intl. Workshop on Graph-Theoretic Concepts in Computer Science (WG)*. 2004, pages 257–269. DOI: 10.1007/978-3-540-30559-0_22. [see page 35]

[CH19] Yuning Chen and Jin-Kao Hao. "Dynamic Thresholding Search for Minimum Vertex Cover in Massive Sparse Graphs". In: *Eng. Appl. Artif. Intell.* 82 (2019), pages 76–84. DOI: 10.1016/j.engappai.2019.03.015. [see page 31]

[Cha+22] Jonas Charfreitag, Michael Jünger, Sven Mallach, and Petra Mutzel. "McSparse: Exact Solutions of Sparse Maximum Cut and Sparse Unconstrained Binary Quadratic Optimization Problems". In: *Symp. on Algorithm Engineering and Experiments (ALENEX)*. 2022, pages 54–66. DOI: 10.1137/1.9781611977042.5. [see pages 131, 132]

[Cha20] Lijun Chang. "Efficient Maximum Clique Computation and Enumeration over Large Sparse Graphs". In: *VLDB J.* 29.5 (2020), pages 999–1022. DOI: 10.1007/s00778-020-00602-z. [see page 32]

[Che+12] James Cheng, Yiping Ke, Shumo Chu, and Carter Cheng. "Efficient Processing of Distance Queries in Large Graphs: A Vertex Cover Approach". In: *ACM Intl. Conf. on Management of Data (SIGMOD)*. 2012, pages 457–468. DOI: 10.1145/2213836.2213888. [see page 18]

[Chi+07] Charles C. Chiang, Andrew B. Kahng, Subarna Sinha, Xu Xu, and Alexander Zelikovsky. "Fast and Efficient Bright-Field AAPSM Conflict Detection and Correction". In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 26.1 (2007), pages 115–126. DOI: 10.1109/TCAD.2006.882642. [see pages v, vii, 3, 129]

[CJM15] Robert Crowston, Mark Jones, and Matthias Mnich. "Max-Cut Parameterized Above the Edwards-Erdős Bound". In: *Algorithmica* 72.3 (2015), pages 734–757. DOI: 10.1007/s00453-014-9870-z. [see pages 6, 131, 133]

[CKJ01] Jianer Chen, Iyad A. Kanj, and Weijia Jia. "Vertex Cover: Further Observations and Further Improvements". In: *J. Algorithms* 41.2 (2001), pages 280–301. DOI: 10.1006/jagm.2001.1186. [see pages 33–35]

[CKR11] Jerry Chi-Yuan Chou, Jinoh Kim, and Doron Rotem. "Energy-Aware Scheduling in Disk Storage Systems". In: *Intl. Conf. on Distributed Computing Systems (ICDCS)*. 2011, pages 423–433. DOI: 10.1109/ICDCS.2011.40. [see pages 2, 90, 147]

[CKX06] Jianer Chen, Iyad A. Kanj, and Ge Xia. "Improved Parameterized Upper Bounds for Vertex Cover". In: *Intl. Symp. on Mathematical Foundations of Computer Science (MFCS)*. 2006, pages 238–249. DOI: 10.1007/11821069_21. [see page 1]

[CKX10] Jianer Chen, Iyad A. Kanj, and Ge Xia. "Improved Upper Bounds for Vertex Cover". In: *Theor. Comput. Sci.* 411.40-42 (2010), pages 3736–3756. DOI: 10.1016/j.tcs.2010.06.026. [see page 74]

[CL16] Shaowei Cai and Jinkun Lin. "Fast Solving Maximum Weight Clique Problem in Massive Graphs". In: *Intl. Joint Conf. on Artificial Intelligence (IJCAI)*. 2016, pages 568–574. [see page 95]

[CLL17] Shaowei Cai, Jinkun Lin, and Chuan Luo. "Finding a Small Vertex Cover in Massive Sparse Graphs: Construct, Local Search, and Preprocess". In: *J. Artif. Intell. Res.* 59 (2017), pages 463–494. DOI: 10.1613/jair.5443. [see page 30]

[CLZ17] Lijun Chang, Wei Li, and Wenjie Zhang. "Computing a Near-Maximum Independent Set in Linear Time by Reducing-Peeling". In: *ACM Intl. Conf. on Management of Data (SIGMOD)*. 2017, pages 1181–1196. DOI: 10.1145/3035918.3035939. [see pages 3, 4, 30, 31, 64, 66, 73, 86, 92, 94, 98, 147]

[CM07] Carlos Cotta and Pablo Moscato. "Memetic Algorithms". In: *Handbook of Approximation Algorithms and Metaheuristics*. 2007. DOI: 10.1201/9781420010749.ch27. [see page 14]

[CP90] Randy Carraghan and Panos M. Pardalos. "An Exact Algorithm for the Maximum Clique Problem". In: *Oper. Res. Lett.* 9.6 (1990), pages 375–382. DOI: 10.1016/0167-6377(90)90057-C. [see page 75]

[Cro+13] Robert Crowston, Gregory Z. Gutin, Mark Jones, and Gabriele Muciaccia. "Maximum Balanced Subgraph Problem Parameterized Above Lower Bound". In: *Theor. Comput. Sci.* 513 (2013), pages 53–64. DOI: 10.1016/j.tcs.2013.10.026. [see pages 6, 131, 133, 135, 136]

[CSS11] Shaowei Cai, Kaile Su, and Abdul Sattar. "Local Search with Edge Weighting and Configuration Checking Heuristics for Minimum Vertex Cover". In: *Artif. Intell.* 175.9-10 (2011), pages 1672–1696. DOI: 10.1016/j.artint.2011.03.003. [see page 93]

[Dah+16a] Jakob Dahlum, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. "Accelerating Local Search for the Maximum Independent Set Problem". In: *Intl. Symp. on Experimental Algorithms (SEA)*. 2016, pages 118–133. DOI: 10.1007/978-3-319-38851-9_9. [see pages 2, 7, 25, 29, 30, 66, 73, 93, 94, 202]

[Dah+16b] Jakob Dahlum, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. "Accelerating Local Search for the Maximum Independent Set Problem". In: *Computing Research Repository (CoRR)* abs/1602.01659 (2016). DOI: 10.48550/ARXIV.1602.01659. [see pages 7, 25, 203]

[Del+09] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. "Engineering Route Planning Algorithms". In: *Algorithmics of Large and Complex Networks*. 2009, pages 117–139. DOI: 10.1007/978-3-642-02094-0_7. [see pages 18, 153]

[DF99] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. 1999. DOI: 10.1007/978-1-4612-0515-9. [see page 11]

[DFH19] M. Ayaz Dzulfikar, Johannes Klaus Fichte, and Markus Hecher. "The PACE 2019 Parameterized Algorithms and Computational Experiments Challenge: The Fourth Iteration (Invited Paper)". In: *Intl. Symp. on Parameterized and Exact Computation (IPEC)*. 2019, 25:1–25:23. DOI: 10.4230/LIPIcs.IPEC.2019.25. [see pages 18, 67, 147, 157–160]

[DGJ09] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. 2009. ISBN: 9780821843833. [see pages 18, 153]

[DGS18] Iain Dunning, Swati Gupta, and John Silberholz. "What Works Best When? A Systematic Evaluation of Heuristics for Max-Cut and QUBO". In: *INFORMS J. Comput.* 30.3 (2018), pages 608–624. DOI: 10.1287/ijoc.2017.0798. [see pages 19, 129, 131, 133, 140, 143, 161]

[DH11] Timothy A. Davis and Yifan Hu. "The University of Florida Sparse Matrix Collection". In: *ACM Trans. Math. Softw.* 38.1 (2011), 1:1–1:25. DOI: 10.1145/2049662.2049663. [see pages 18, 156]

[DM02] Elizabeth D. Dolan and Jorge J. Moré. "Benchmarking Optimization Software with Performance Profiles". In: *Math. Program.* 91.2 (2002), pages 201–213. DOI: 10.1007/s101070100263. [see page 22]

[Don+21] Yuanyuan Dong, Andrew V. Goldberg, Alexander Noe, Nikos Parotsidis, Mauricio G. C. Resende, and Quico Spaen. "New Instances for Maximum Weight Independent Set from a Vehicle Routing Application". In: *Computing Research Repository (CoRR)* (2021). DOI: 10.48550/ARXIV.2105.12623. [see page 90]

[Don+22] Yuanyuan Dong, Andrew V. Goldberg, Alexander Noe, Nikos Parotsidis, Mauricio G. C. Resende, and Quico Spaen. "A Metaheuristic Algorithm for Large Maximum Weight Independent Set Problems". In: *Computing Research Repository (CoRR)* abs/2203.15805 (2022). DOI: 10.48550/ARXIV.2203.15805. [see pages v, vii, 2, 90, 94, 147]

[dSHK13] Samuel de Sousa, Yll Haxhimusa, and Walter G. Kropatsch. "Estimation of Distribution Algorithm for the Max-Cut Problem". In: *Intl. Workshop on Graph-Based Representations in Pattern Recognition (GbRPR)*. 2013, pages 244–253. DOI: 10.1007/978-3-642-38221-5_26. [see pages 129, 143]

[dSR18] Marcelo de Souza and Marcus Ritt. "Automatic Grammar-Based Design of Heuristic Algorithms for Unconstrained Binary Quadratic Programming". In: *European Conf. on Evolutionary Computation in Combinatorial Optimization (EvoCOP)*. 2018, pages 67–84. DOI: 10.1007/978-3-319-77449-7_5. [see page 133]

[Ebl+12] John D. Eblen, Charles A. Phillips, Gary L. Rogers, and Michael A. Langston. "The Maximum Clique Enumeration Problem: Algorithms, Applications, and Implementations". In: *BMC Bioinform.* 13.S-10 (2012), S5. DOI: 10.1186/1471-2105-13-S10-S5. [see page 32]

[Edw73] Christopher S. Edwards. "Some Extremal Properties of Bipartite Subgraphs". In: *Canad. J. Math.* 25.3 (1973), pages 475–485. DOI: 10.4153/CJM-1973-048-x. [see page 131]

[Edw75] Christopher S. Edwards. "An Improved Lower Bound for the Number of Edges in a Largest Bipartite Subgraph". In: *Czechoslovak Symp. on Graph Theory*. 1975, pages 167–181. [see page 131]

[EHd84] Christian Ebenegger, Peter L. Hammer, and Dominique de Werra. "Pseudo-Boolean Functions and Stability of Graphs". In: *Algebraic and Combinatorial Methods in Operations Research*. Volume 95. 1984, pages 83–97. DOI: 10.1016/S0304-0208(08)72955-4. [see pages 5, 91, 92, 96, 112–114]

[EK20] Duncan Eddy and Mykel J. Kochenderfer. "A Maximum Independent Set Method for Scheduling Earth Observing Satellite Constellations". In: *Computing Research Repository (CoRR)* abs/2008.08446 (2020). DOI: 10.48550/ARXIV.2008.08446. [see page 147]

[EM18] Michael Etscheid and Matthias Mnich. "Linear Kernels and Linear-Time Algorithms for Finding Large Cuts". In: *Algorithmica* 80.9 (2018), pages 2574–2615. DOI: 10.1007/s00453-017-0388-z. [see pages 6, 131, 133, 140]

[Fan+14] Zhiwen Fang, Chu-Min Li, Kan Qiao, Xu Feng, and Ke Xu. "Solving Maximum Weight Clique Using Maximum Satisfiability Reasoning". In: *European Conf. on Artificial Intelligenc (ECAI)*. 2014, pages 303–308. DOI: 10.3233/978-1-61499-419-0-303. [see page 95]

[Fan+15] Yi Fan, Chengqian Li, Zongjie Ma, Ljiljana Brankovic, Vladimir Estivill-Castro, and Abdul Sattar. "Exploiting Reduction Rules and Data Structures: Local Search for Minimum Vertex Cover in Massive Graphs". In: *Computing Research Repository (CoRR)* abs/1509.05870 (2015). DOI: 10.48550/ARXIV.1509.05870. [see pages 29–31]

[Fan+17] Yi Fan, Nan Li, Chengqian Li, Zongjie Ma, Longin Jan Latecki, and Kaile Su. "Restart and Random Walk in Local Search for Maximum Vertex Weight Cliques with Evaluations in Clustering Aggregation". In: *Intl. Joint Conf. on Artificial Intelligence (IJCAI)*. 2017, pages 622–630. DOI: 10.24963/ijcai.2017/87. [see page 95]

[Far+17] Luérbio Faria, Sulamita Klein, Ignasi Sau, and Rubens Sucupira. "Improved Kernels for Signed Max Cut Parameterized Above Lower Bound on (r, l)-Graphs". In: *Discret. Math. Theor. Comput. Sci.* 19.1 (2017). DOI: 10.23638/DMTCS-19-1-14. [see pages 6, 131, 133]

[Fei20] Adrian Feilhauer. "Communication-Free Generation of Graphs with Planted Communities". Master's thesis. Karlsruhe Institute of Technology, Germany, 2020. [see page 204]

[Fer+19] Damir Ferizovic, Demian Hespe, Sebastian Lamm, Matthias Mnich, Christian Schulz, and Darren Strash. "Engineering Kernelization for Maximum Cut". In: *Computing Research Repository (CoRR)* abs/1905.10902 (2019). DOI: 10.48550/ARXIV.1905.10902. [see pages 7, 129, 203]

[Fer+20] Damir Ferizovic, Demian Hespe, Sebastian Lamm, Matthias Mnich, Christian Schulz, and Darren Strash. "Engineering Kernelization for Maximum Cut". In: *Symp. on Algorithm Engineering and Experiments (ALENEX)*. 2020, pages 27–41. DOI: 10.1137/1.9781611976007.3. [see pages 2, 7, 129, 201]

[Fer19] Damir Ferizovic. "A Practical Analysis of Kernelization Techniques for the Maximum Cut Problem". Master's thesis. Karlsruhe Institute of Technology, Germany, 2019. [see pages 131, 133, 137, 138, 204]

[FGK09] Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch. "A Measure & Conquer Approach for the Analysis of Exact Algorithms". In: *J. ACM* 56.5 (2009), 25:1–25:32. DOI: 10.1145/1552285.1552286. [see pages 2, 4, 11, 28, 33, 34, 38, 40, 74, 92]

[FHL19] Robert Fischbach, Tilman Horst, and Jens Lienig. "Assembly-Related Chip/Package Co-Design of Heterogeneous Systems Manufactured by Micro-Transfer Printing". In: *Design, Automation and Test in Europe Conf. and Exhibition (DATE)*. 2019, pages 956–959. DOI: 10.23919/DATE.2019.8714884. [see pages 2, 147]

[FNS16] Stefan Funke, André Nusser, and Sabine Storandt. "On k-Path Covers and their Applications". In: *VLDB J.* 25.1 (2016), pages 103–123. DOI: 10.1007/s00778-015-0392-3. [see page 18]

[Fom+19] Fedor V. Fomin, Daniel Lokshtanov, Saket Saurabh, and Meirav Zehavi. *Kernelization: Theory of Parameterized Preprocessing*. 2019. ISBN: 9781107057760. [see page 7]

[Fou22] OpenStreetMap Foundation. *OpenStreetMap*. 2022. URL: https://www.openstreetmap.org. [see page 19]

[FR89] Thomas A. Feo and Mauricio G. C. Resende. "A Probabilistic Heuristic for a Computationally Difficult Set Covering Problem". In: *Oper. Res. Lett.* 8.2 (1989), pages 67–71. DOI: 10.1016/0167-6377(89)90002-3. [see page 94]

[FR95] Thomas A. Feo and Mauricio G. C. Resende. "Greedy Randomized Adaptive Search Procedures". In: *J. Glob. Optim.* 6.2 (1995), pages 109–133. DOI: 10.1007/BF01096763. [see page 94]

[Fre60] Edward Fredkin. "Trie Memory". In: *Commun. ACM* 3.9 (1960), pages 490–499. DOI: 10.1145/367390.367400. [see page 139]

[Fun+17] Daniel Funke, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. "Communication-Free Massively Distributed Graph Generation". In: *Computing Research Repository (CoRR)* abs/1710.07565 (2017). DOI: 10.48550/ARXIV.1710.07565. [see page 203]

[Fun+18] Daniel Funke, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. "Communication-Free Massively Distributed Graph Generation". In: *IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS)*. 2018, pages 336–347. DOI: 10.1109/IPDPS.2018.00043. [see page 201]

[Fun+19] Daniel Funke, Sebastian Lamm, Ulrich Meyer, Manuel Penschuck, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. "Communication-Free Massively Distributed Graph Generation". In: *J. Parallel Distributed Comput.* 131 (2019), pages 200–217. DOI: 10.1016/j.jpdc.2019.03.011. [see pages 19, 202]

[Gao+17] Wanru Gao, Tobias Friedrich, Timo Kötzing, and Frank Neumann. "Scaling up Local Search for Minimum Vertex Cover in Large Graphs by Parallel Kernelization". In: *Australasian Conf. on Artificial Intelligence (AI)*. 2017, pages 131–143. DOI: 10.1007/978-3-319-63004-5_11. [see pages 30, 86]

[Gao+18] Jian Gao, Jiejiang Chen, Minghao Yin, Rong Chen, and Yiyuan Wang. "An Exact Algorithm for Maximum k-Plexes in Massive Graphs". In: *Intl. Joint Conf. on Artificial Intelligence (IJCAI)*. 2018, pages 1449–1455. DOI: 10.24963/ijcai.2018/201. [see page 73]

[Gar+14] Frédéric Gardi, Thierry Benoist, Julien Darlay, Bertrand Estellon, and Romain Megel. *Mathematical Programming Solver Based on Local Search*. 2014, page 112. DOI: 10.1002/9781118966464. [see page 130]

[Gel+20] Alexander Gellner, Sebastian Lamm, Christian Schulz, Darren Strash, and Bogdán Zaválnij. "Boosting Data Reduction for the Maximum Weight Independent Set Problem Using Increasing Transformations". In: *Computing Research Repository (CoRR)* abs/2008.05180 (2020). DOI: 10.48550/ARXIV.2008.05180. [see pages 7, 89, 202]

[Gel+21] Alexander Gellner, Sebastian Lamm, Christian Schulz, Darren Strash, and Bogdán Zaválnij. "Boosting Data Reduction for the Maximum Weight Independent Set Problem Using Increasing Transformations". In: *Symp. on Algorithm Engineering and Experiments (ALENEX)*. 2021, pages 128–142. DOI: 10.1137/1.9781611976472.10. [see pages 7, 89, 92, 201]

[Gel20] Alexander Gellner. "Engineering Generalized Reductions for the Maximum Weight Independent Set Problem". Master's thesis. Karlsruhe Institute of Technology, Germany, 2020. [see page 204]

[Geo18] Tom George. "Distributed Kernelization for Independent Sets". Bachelor's thesis. Karlsruhe Institute of Technology, Germany, 2018. [see page 204]

[Geo73] Alan George. "Nested Dissection of a Regular Finite Element Mesh". In: *SIAM Journal on Numerical Analysis* 10.2 (1973), pages 345–363. DOI: 10.1137/0710032. [see page 77]

[GG21] Jiaqi Gu and Ping Guo. "PEAVC: An Improved Minimum Vertex Cover Solver for Massive Sparse Graphs". In: *Eng. Appl. Artif. Intell.* 104 (2021). DOI: 10.1016/j.engappai.2021.104344. [see page 31]

[GJ79] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. 1979. ISBN: 0716710455. [see pages 11, 12, 90]

[GJS74] M. R. Garey, David S. Johnson, and Larry J. Stockmeyer. "Some Simplified NP-Complete Problems". In: *ACM Symp. on Theory of Computing (STOC)*. 1974, pages 47–63. DOI: 10.1145/800119.803884. [see page 129]

[GLC04] Andrea Grosso, Marco Locatelli, and Federico Della Croce. "Combining Swaps and Node Weights in an Adaptive Greedy Approach for the Maximum Clique Problem". In: *J. Heuristics* 10.2 (2004), pages 135–152. DOI: 10.1023/B:HEUR.0000026264.51747.7f. [see page 33]

[GLH10] Fred W. Glover, Zhipeng Lü, and Jin-Kao Hao. "Diversification-Driven Tabu Search for Unconstrained Binary Quadratic Problems". In: *4OR* 8.3 (2010), pages 239–253. DOI: 10.1007/s10288-009-0115-y. [see page 133]

[Glo89] Fred W. Glover. "Tabu Search - Part I". In: *INFORMS J. Comput.* 1.3 (1989), pages 190–206. DOI: 10.1287/ijoc.1.3.190. [see page 13]

[GLP08] Andrea Grosso, Marco Locatelli, and Wayne J. Pullan. "Simple Ingredients Leading to Very Efficient Heuristics for the Maximum Clique Problem". In: *J. Heuristics* 14.6 (2008), pages 587–612. DOI: 10.1007/s10732-007-9055-x. [see pages 18, 29, 33, 49, 56]

[GNN13] Andreas Gemsa, Benjamin Niedermann, and Martin Nöllenburg. "Trajectory-Based Dynamic Map Labeling". In: *Intl. Symp. on Algorithms and Computation (ISAAC)*. 2013, pages 413–423. DOI: 10.1007/978-3-642-45030-3_39. [see pages v, vii]

[GNR16] Andreas Gemsa, Martin Nöllenburg, and Ignaz Rutter. "Evaluation of Labeling Strategies for Rotating Maps". In: *ACM J. Exp. Algorithmics* 21.1 (2016), 1.4:1–1.4:21. DOI: 10.1145/2851493. [see pages 2, 19, 90, 147]

[Gol89] David E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning*. 1989. ISBN: 0201157675. [see page 13]

[Got+19] Lars Gottesbüren, Michael Hamann, Tim Niklas Uhl, and Dorothea Wagner. "Faster and Better Nested Dissection Orders for Customizable Contraction Hierarchies". In: *Algorithms* 12.9 (2019), page 196. DOI: 10.3390/a12090196. [see page 82]

[GS16] Nicholas I. M. Gould and Jennifer A. Scott. "A Note on Performance Profiles for Benchmarking Software". In: *ACM Trans. Math. Softw.* 43.2 (2016), 15:1–15:5. DOI: 10.1145/2950048. [see page 22]

[GT88] Andrew V. Goldberg and Robert E. Tarjan. "A New Approach to the Maximum-Flow Problem". In: *J. ACM* 35.4 (1988), pages 921–940. DOI: 10.1145/48014.61051. [see page 76]

[Gu+21] Jiewei Gu, Weiguo Zheng, Yuzheng Cai, and Peng Peng. "Towards Computing a Near-Maximum Weighted Independent Set on Massive Graphs". In: *ACM SIGKDD Conf. on Knowledge Discovery and Data Mining (KDD)*. 2021, pages 467–477. DOI: 10.1145/3447548.3467232. [see page 94]

[Gur21] LLC Gurobi Optimization. *Gurobi Optimizer Reference Manual*. 2021. URL: https://www.gurobi.com. [see pages 19, 73, 132]

[Gus+20] Nicolò Gusmeroli, Timotej Hrga, Borut Lužar, Janez Povh, Melanie Siebenhofer, and Angelika Wiegele. "BiqBin: A Parallel Branch-and-Bound Solver for Binary Quadratic Problems with Linear Constraints". In: *Computing Research Repository (CoRR)* abs/2009.06240 (2020). DOI: 10.48550/ARXIV.2009.06240. [see pages 131, 132]

[GWA00] Eleanor J. Gardiner, Peter Willett, and Peter J. Artymiuk. "Graph-Theoretic Techniques for Macromolecular Docking". In: *J. Chem. Inf. Comput. Sci.* 40.2 (2000), pages 273–279. DOI: 10.1021/ci990262o. [see pages v, vii, 2, 26]

[Har59] Frank Harary. "On the Measurement of Structural Balance". In: *Behavioral Sci.* 4.4 (1959), pages 316–323. DOI: 10.1002/bs.3830040405. [see pages v, vii, 3, 129]

[Hes+19] Demian Hespe, Sebastian Lamm, Christian Schulz, and Darren Strash. "WeGotYouCovered: The Winning Solver from the PACE 2019 Implementation Challenge, Vertex Cover Track". In: *Computing Research Repository (CoRR)* abs/1908.06795 (2019). DOI: 10.48550/ARXIV.1908.06795. [see pages 7, 25, 203]

[Hes+20] Demian Hespe, Sebastian Lamm, Christian Schulz, and Darren Strash. "WeGotYouCovered: The Winning Solver from the PACE 2019 Challenge, Vertex Cover Track". In: *SIAM Workshop on Combinatorial Scientific Computing (CSC)*. 2020, pages 1–11. DOI: 10.1137/1.9781611976229.1. [see pages 25, 26, 28, 29, 201]

[HK73] John E. Hopcroft and Richard M. Karp. "An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs". In: *SIAM J. Comput.* 2.4 (1973), pages 225–231. DOI: 10.1137/0202019. [see pages 37, 45, 76, 96]

[HLH97] Bernardo A. Huberman, Rajan M. Lukose, and Tad Hogg. "An Economics Approach to Hard Computational Problems". In: *Science* 275.5296 (1997), pages 51–54. DOI: 10.1126/science.275.5296.51. [see page 15]

[HLS21a] Demian Hespe, Sebastian Lamm, and Christian Schorr. "Targeted Branching for the Maximum Independent Set Problem". In: *Intl. Symp. on Experimental Algorithms (SEA)*. 2021, 17:1–17:21. DOI: 10.4230/LIPIcs.SEA.2021.17. [see pages 2, 7, 25, 26, 28, 201]

[HLS21b] Demian Hespe, Sebastian Lamm, and Christian Schorr. "Targeted Branching for the Maximum Independent Set Problem". In: *Computing Research Repository (CoRR)* abs/2102.01540 (2021). DOI: 10.48550/ARXIV.2102.01540. [see pages 7, 25, 202]

[HLW02] Frank Harary, Meng-Hiot Lim, and Donald C. Wunsch. "Signed Graphs for Portfolio Analysis in Risk Management". In: *IMA J. Mgmt. Math.* 13.3 (2002), pages 201–210. DOI: 10.1093/imaman/13.3.201. [see pages 3, 129]

[HMdW85a] Peter L. Hammer, Nadimpalli V. R. Mahadev, and Dominique de Werra. "Stability in CAN-Free Graphs". In: *J. Comb. Theory, Ser. B* 38.1 (1985), pages 23–30. DOI: 10.1016/0095-8956(85)90089-9. [see page 113]

[HMdW85b] Peter L. Hammer, Nadimpalli V. R. Mahadev, and Dominique de Werra. "The Struction of a Graph: Application to CN-Free Graphs". In: *Comb.* 5.2 (1985), pages 141–147. DOI: 10.1007/BF02579377. [see page 113]

[HMU04] Pierre Hansen, Nenad Mladenovic, and Dragan Urosevic. "Variable Neighborhood Search for the Maximum Clique". In: *Discret. Appl. Math.* 145.1 (2004), pages 117–125. DOI: 10.1016/j.dam.2003.09.012. [see page 33]

[HP21] Timotej Hrga and Janez Povh. "MADAM: A Parallel Exact Solver for Max-Cut Based on Semidefinite Programming and ADMM". In: *Comput. Optim. Appl.* 80.2 (2021), pages 347–375. DOI: 10.1007/s10589-021-00310-6. [see pages 131, 132]

[Hrg+19] Timotej Hrga, Borut Luzar, Janez Povh, and Angelika Wiegele. "BiqBin: Moving Boundaries for NP-hard Problems by HPC". In: *Intl. Conf. on High Performance Computing (HPC)*. 2019, pages 327–339. DOI: 10.1007/978-3-030-55347-0_28. [see pages 131, 132]

[HSS19] Demian Hespe, Christian Schulz, and Darren Strash. "Scalable Kernelization for Maximum Independent Sets". In: *ACM J. Exp. Algorithmics* 24.1 (2019), 1.16:1–1.16:22. DOI: 10.1145/3355502. [see pages 28, 29, 31, 38, 73, 86, 98, 130, 148]

[HT73] John E. Hopcroft and Robert E. Tarjan. "Efficient Algorithms for Graph Manipulation [H] (Algorithm 447)". In: *Commun. ACM* 16.6 (1973), pages 372–378. DOI: 10.1145/362248.362272. [see page 75]

[HT94] Kathy W. Hoke and M. F. Troyon. "The Struction Algorithm for the Maximum Stable Set Problem Revisited". In: *Discret. Math.* 131.1-3 (1994), pages 105–113. DOI: 10.1016/0012-365X(94)90377-8. [see page 113]

[HXC21] Sen Huang, Mingyu Xiao, and Xiaoyu Chen. "Exact Algorithms for Maximum Weighted Independent Set on Sparse Graphs (Extended Abstract)". In: *Intl. Conf. on Computing and Combinatorics (COCOON)*. 2021, pages 617–628. DOI: 10.1007/978-3-030-89543-3_51. [see pages 2, 92, 148]

[IOY14] Yoichi Iwata, Keigo Oka, and Yuichi Yoshida. "Linear-Time FPT Algorithms via Network Flow". In: *Symp. on Discrete Algorithms (SODA)*. 2014, pages 1749–1761. DOI: 10.1137/1.9781611973402.127. [see pages 28, 37, 92]

[JH15] Yan Jin and Jin-Kao Hao. "General Swap-Based Multiple Neighborhood Tabu search for the Maximum Independent Set Problem". In: *Eng. Appl. Artif. Intell.* 37 (2015), pages 20–33. DOI: 10.1016/j.engappai.2014.08.007. [see pages 29, 30, 93]

[Jia+18] Hua Jiang, Chu-Min Li, Yanli Liu, and Felip Manyà. "A Two-Stage MaxSAT Reasoning Approach for the Maximum Weight Clique Problem". In: *AAAI Conf. on Artificial Intelligence (AAAI)*. 2018, pages 1338–1346. [see page 95]

[JLM17] Hua Jiang, Chu-Min Li, and Felip Manyà. "An Exact Algorithm for the Maximum Weight Clique Problem in Large Graphs". In: *AAAI Conf. on Artificial Intelligence (AAAI)*. 2017, pages 830–838. [see page 95]

[Jon06] Kenneth A. De Jong. *Evolutionary Computation - A Unified Approach*. 2006. ISBN: 9781450371278. [see page 44]

[JT96] David S. Johnson and Michael A. Trick. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*. 1996. ISBN: 9780821866092. [see pages 18, 95, 152]

[Kar72] Richard M. Karp. "Reducibility Among Combinatorial Problems". In: *Symp. on the Complexity of Computer Computations*. 1972, pages 85–103. DOI: 10.1007/978-1-4684-2001-2_9. [see pages 1, 11, 26]

[KHN05] Kengo Katayama, Akihiro Hamamoto, and Hiroyuki Narihisa. "An Effective Local Search for the Maximum Clique Problem". In: *Inf. Process. Lett.* 95.5 (2005), pages 503–511. DOI: 10.1016/j.ipl.2005.05.010. [see page 33]

[Kie+10] Tim Kieritz, Dennis Luxen, Peter Sanders, and Christian Vetter. "Distributed Time-Dependent Contraction Hierarchies". In: *Intl. Symp. on Experimental Algorithms (SEA)*. 2010, pages 83–93. DOI: 10.1007/978-3-642-13193-6_8. [see pages 2, 26]

[KLR09] Joachim Kneis, Alexander Langer, and Peter Rossmanith. "A Fine-Grained Analysis of a Simple Independent Set Algorithm". In: *IARCS Annual Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. 2009, pages 287–298. DOI: 10.4230/LIPIcs.FSTTCS.2009.2326. [see pages 38, 40, 41]

[Klu+19] Fabian Klute, Guangping Li, Raphael Löffler, Martin Nöllenburg, and Manuela Schmidt. "Exploring Semi-Automatic Map Labeling". In: *ACM Intl. Conf. on Advances in Geographic Information Systems (SIGSPATIAL)*. 2019, pages 13–22. DOI: 10.1145/3347146.3359359. [see pages v, vii, 2, 26, 147]

[KM06] Kartik Krishnan and John E. Mitchell. "A Semidefinite Programming Based Polyhedral Cut and Price Approach for the Maxcut Problem". In: *Comput. Optim. Appl.* 33.1 (2006), pages 51–71. DOI: 10.1007/s10589-005-5958-3. [see page 131]

[KMR14] Nathan Krislock, Jérôme Malick, and Frédéric Roupin. "Improved Semidefinite Bounding Procedure for Solving Max-Cut Problems to Optimality". In: *Math. Program.* 143.1-2 (2014), pages 61–86. DOI: 10.1145/3005345. [see page 131]

[KMR17] Nathan Krislock, Jérôme Malick, and Frédéric Roupin. "BiqCrunch: A Semidefinite Branch-and-Bound Method for Solving Binary Quadratic Problems". In: *ACM Trans. Math. Softw.* 43.4 (2017), 32:1–32:23. DOI: 10.1145/3005345. [see pages 131, 132]

[Koc+13] Gary A. Kochenberger, Jin-Kao Hao, Zhipeng Lü, Haibo Wang, and Fred W. Glover. "Solving Large Scale Max Cut Problems via Tabu Search". In: *J. Heuristics* 19.4 (2013), pages 565–571. DOI: 10.1007/s10732-011-9189-8. [see pages 130, 131]

[Kum08] Deniss Kumlander. "On Importance of a Special Sorting in the Maximum-Weight Clique Algorithm Based on Colour Classes". In: *Intl. Conf. on Modelling, Computation and Optimization in Information Systems and Management Sciences (MCO)*. 2008, pages 165–174. DOI: 10.1007/978-3-540-87477-5_18. [see page 94]

[Kun13] Jérôme Kunegis. "KONECT: The Koblenz Network Collection". In: *Intl. Conf. on World Wide Web (WWW) (Companion Volume)*. 2013, pages 1343–1350. DOI: 10.1145/2487788.2488173. [see pages 149, 151]

[Lam+15] Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. "Finding Near-Optimal Independent Sets at Scale". In: *Computing Research Repository (CoRR)* abs/1509.00764 (2015). DOI: 10.48550/ARXIV.1509.00764. [see pages 7, 25, 203]

[Lam+16] Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. "Finding Near-Optimal Independent Sets at Scale". In: *Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2016, pages 138–150. DOI: 10.1137/1.9781611974317.12. [see pages 7, 25, 94, 202]

[Lam+17] Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. "Finding Near-Optimal Independent Sets at Scale". In: *J. Heuristics* 23.4 (2017), pages 207–229. DOI: 10.1007/s10732-017-9337-x. [see pages 7, 18, 25, 59, 62, 93, 130, 202]

[Lam+18] Sebastian Lamm, Christian Schulz, Darren Strash, Robert Williger, and Huashuo Zhang. "Exactly Solving the Maximum Weight Independent Set Problem on Large Real-World Graphs". In: *Computing Research Repository (CoRR)* abs/1810.10834 (2018). DOI: 10.48550/ARXIV.1810.10834. [see pages 7, 89, 203]

[Lam+19] Sebastian Lamm, Christian Schulz, Darren Strash, Robert Williger, and Huashuo Zhang. "Exactly Solving the Maximum Weight Independent Set Problem on Large Real-World Graphs". In: *Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2019, pages 144–158. DOI: 10.1137/1.9781611975499.12. [see pages 7, 89, 92, 94, 112, 117, 201]

[Lam14] Sebastian Lamm. "Evolutionary Algorithms for Independent Sets". Bachelor's thesis. Karlsruhe Institute of Technology, Germany, 2014. [see page 203]

[Lam17] Sebastian Lamm. "Communication Efficient Algorithms for Generating Massive Networks". Master's thesis. Karlsruhe Institute of Technology, Germany, 2017. [see page 203]

[Lar07] Craig E. Larson. "A Note On Critical Independence Reductions". In: *Bulletin of the Institute of Combinatorics and its Applications*. 2007, pages 34–46. [see page 92]

[LAS19] Jan-Hendrik Lange, Bjoern Andres, and Paul Swoboda. "Combinatorial Persistency Criteria for Multicut and Max-Cut". In: *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*. 2019, pages 6093–6102. DOI: 10.1109/CVPR.2019.00625. [see pages 131, 132]

[LCH17] Yuanjie Li, Shaowei Cai, and Wenying Hou. "An Efficient Local Search Algorithm for Minimum Weighted Vertex Cover on Massive Graphs". In: *Intl. Conf. on Simulated Evolution and Learning (SEAL)*. 2017, pages 145–157. DOI: 10.1007/978-3-319-68759-9_13. [see pages 19, 93]

[LFX13] Chu-Min Li, Zhiwen Fang, and Ke Xu. "Combining MaxSAT Reasoning and Incremental Upper Bound for the Maximum Clique Problem". In: *IEEE Intl. Conf. on Tools with Artificial Intelligence (ICTAI)*. 2013, pages 939–946. DOI: 10.1109/ICTAI.2013.143. [see pages 31, 75]

[Li+18] Chu-Min Li, Yanli Liu, Hua Jiang, Felip Manyà, and Yu Li. "A New Upper Bound for the Maximum Weight Clique Problem". In: *Eur. J. Oper. Res.* 270.1 (2018), pages 66–77. DOI: 10.1016/j.ejor.2018.03.020. [see page 95]

[Li+20] Ruizhi Li, Shuli Hu, Shaowei Cai, Jian Gao, Yiyuan Wang, and Minghao Yin. "NuMWVC: A Novel Local Search for Minimum Weighted Vertex Cover Problem". In: *J. Oper. Res. Soc.* 71.9 (2020), pages 1498–1509. DOI: 10.1080/01605682.2019.1621218. [see pages 2, 93]

[Liu+15] Yu Liu, Jiaheng Lu, Hua Yang, Xiaokui Xiao, and Zhewei Wei. "Towards Maximum Independent Sets on Massive Graphs". In: *Proc. VLDB Endow.* 8.13 (2015), pages 2122–2133. DOI: 10.14778/2831360.2831366. [see pages 30, 41]

[LJM17] Chu-Min Li, Hua Jiang, and Felip Manyà. "On Minimization of the Number of Branches in Branch-and-Bound Algorithms for the Maximum Clique Problem". In: *Comput. Oper. Res.* 84 (2017), pages 1–15. DOI: 10.1016/j.cor.2017.02.017. [see pages 4, 31, 65–67, 75]

[LJX15] Chu-Min Li, Hua Jiang, and Ruchu Xu. "Incremental MaxSAT Reasoning to Reduce Branches in a Branch-and-Bound Algorithm for MaxClique". In: *Intl. Conf. on Learning and Intelligent Optimization (LION)*. 2015, pages 268–274. DOI: 10.1007/978-3-319-19084-6_26. [see page 75]

[LK14] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. 2014. URL: http://snap.stanford.edu/data. [see pages 18, 19, 150]

[LM99] Manuel Laguna and Rafael Marti. "GRASP and Path Relinking for 2-Layer Straight Line Crossing Minimization". In: *INFORMS J. Comput.* 11.1 (1999), pages 44–52. DOI: 10.1287/ijoc.11.1.44. [see page 94]

[Loz00] Vadim V. Lozin. "Conic Reduction of Graphs for the Stable Set Problem". In: *Discret. Math.* 222.1-3 (2000), pages 199–211. DOI: 10.1016/S0012-365X(99)00408-2. [see page 127]

[LP09] László Lovász and Michael D. Plummer. *Matching Theory*. Volume 367. 2009. ISBN: 9780080872322. [see page 127]

[LQ10] Chu-Min Li and Zhe Quan. "An Efficient Branch-and-Bound Algorithm Based on MaxSAT for the Maximum Clique Problem". In: *AAAI Conf. on Artificial Intelligence (AAAI)*. 2010. [see pages 31, 73, 75, 95]

[LR21] Victor Luncasu and Madalina Raschip. "A Graph-Based Approach for the DNA Word Design Problem". In: *IEEE ACM Trans. Comput. Biol. Bioinform.* 18.6 (2021), pages 2747–2752. DOI: 10.1109/TCBB.2020.3008346. [see page 147]

[LS22] Sebastian Lamm and Peter Sanders. "Communication-Efficient Massively Distributed Connected Components". In: *to appear IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS)*. 2022. [see page 201]

[LSS15a] Sebastian Lamm, Peter Sanders, and Christian Schulz. "Graph Partitioning for Independent Sets". In: *Intl. Symp. on Experimental Algorithms (SEA)*. 2015, pages 68–81. DOI: 10.1007/978-3-319-20086-6_6. [see pages 2, 3, 25, 49, 202]

[LSS15b] Sebastian Lamm, Peter Sanders, and Christian Schulz. "Graph Partitioning for Independent Sets". In: *Computing Research Repository (CoRR)* abs/1502.01687 (2015). DOI: 10.48550/ARXIV.1502.01687. [see pages 3, 203]

[Luo+19] Chuan Luo, Holger H. Hoos, Shaowei Cai, Qingwei Lin, Hongyu Zhang, and Dongmei Zhang. "Local Search with Efficient Automatic Configuration for Minimum Vertex Cover". In: *Intl. Joint Conf. on Artificial Intelligence (IJCAI)*. 2019, pages 1297–1304. DOI: 10.24963/ijcai.2019/180. [see page 31]

[Ma+16] Zongjie Ma, Yi Fan, Kaile Su, Chengqian Li, and Abdul Sattar. "Local Search with Noisy Strategy for Minimum Vertex Cover in Massive Graphs". In: *Pacific Rim Intl. Conf. on Artificial Intelligence (PRICAI)*. 2016, pages 283–294. DOI: 10.1007/978-3-319-42911-3_24. [see pages 29, 30]

[Mas+10] Franco Mascia, Elisa Cilia, Mauro Brunato, and Andrea Passerini. "Predicting Structural and Functional Sites in Proteins by Searching for Maximum-Weight Cliques". In: *AAAI Conf. on Artificial Intelligence (AAAI)*. 2010. [see pages 2, 90]

[McC+17] Ciaran McCreesh, Patrick Prosser, Kyle A. Simpson, and James Trimble. "On Maximum Weight Clique Algorithms, and How They Are Evaluated". In: *Intl. Conf. on Principles and Practice of Constraint Programming (CP)*. 2017, pages 206–225. DOI: 10.1007/978-3-319-66158-2_14. [see page 19]

[McG12] Catherine C. McGeoch. *A Guide to Experimental Algorithmics*. 2012. ISBN: 0521173019. [see page 22]

[MG95] Brad L. Miller and David E. Goldberg. "Genetic Algorithms, Tournament Selection, and the Effects of Noise". In: *Complex Syst.* 9.3 (1995), pages 193–212 . [see page 45]

[ML12] Tianyang Ma and Longin J. Latecki. "Maximum Weight Cliques with Mutex Constraints for Video Object Segmentation". In: *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*. 2012, pages 670–677. DOI: 10.1109/CVPR.2012.6247735. [see page 90]

[MMM09] Saeed Mehrabi, Abbas Mehrabi, and Ali D. Mehrabi. "A New Hybrid Genetic Algorithm for Maximum Independent Set Problem". In: *Intl. Conf. on Software and Data Technologies (ICSOFT)*. 2009, pages 314–317. [see pages 43, 47]

[MR99] Meena Mahajan and Venkatesh Raman. "Parameterizing above Guaranteed Values: MaxSat and MaxCut". In: *J. Algorithms* 31.2 (1999), pages 335–354 . [see page 131]

[MS08] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. 2008. DOI: 10.1007/978-3-540-77978-0. [see pages 13, 14]

[MSZ18] Jayakrishnan Madathil, Saket Saurabh, and Meirav Zehavi. "Max-Cut Above Spanning Tree is Fixed-Parameter Tractable". In: *Intl. Computer Science Symp. in Russia (CSR)*. 2018, pages 244–256. DOI: 10.1007/978-3-319-90530-3_21. [see pages 6, 131, 133]

[NJ75] George L. Nemhauser and Leslie E. Trotter Jr. "Vertex Packings: Structural Properties and Algorithms". In: *Math. Program.* 8.1 (1975), pages 232–248. DOI: 10.1007/BF01580444. [see pages 28, 33, 37]

[NKÖ97] Kari J. Nurmela, Markku K. Kaikkonen, and Patric R. J. Östergård. "New Constant Weight Codes from Linear Permutation Groups". In: *IEEE Trans. Inf. Theory* 43.5 (1997), pages 1623–1630. DOI: 10.1109/18.623163. [see pages 2, 90]

[NPS18] Bruno C. S. Nogueira, Rian G. S. Pinheiro, and Anand Subramanian. "A Hybrid Iterated Local Search Heuristic for the Maximum Weight Independent Set Problem". In: *Optim. Lett.* 12.3 (2018), pages 567–583. DOI: 10.1007/s11590-017-1128-7. [see pages 93, 94, 96, 101, 107, 121]

[Öst01] Patric R. J. Östergård. "A New Algorithm for the Maximum-Weight Clique Problem". In: *Nord. J. Comput.* 8.4 (2001), pages 424–436 . [see page 94]

[Öst02] Patric R. J. Östergård. "A Fast Algorithm for the Maximum Clique Problem". In: *Discret. Appl. Math.* 120.1-3 (2002), pages 197–207. DOI: 10.1016/S0166-218X(01)00290-6. [see pages 91, 94]

[Pen+20] Manuel Penschuck, Ulrik Brandes, Michael Hamann, Sebastian Lamm, Ulrich Meyer, Ilya Safro, Peter Sanders, and Christian Schulz. "Recent Advances in Scalable Network Generation". In: *Computing Research Repository (CoRR)* abs/2003.00736 (2020). DOI: 10.48550/ARXIV.2003.00736. [see page 202]

[PHD19] Elijah Pelofske, Georg Hahn, and Hristo N. Djidjev. "Solving Large Minimum Vertex Cover Problems on a Quantum Annealer". In: *ACM Intl. Conf. on Computing Frontiers (CF)*. 2019, pages 76–84. DOI: 10.1145/3310273.3321562. [see page 28]

[Pri05] Elena Prieto. "The Method of Extremal Structure on the k-Maximum Cut Problem". In: *Computing: The Australasian Theory Symp. (CATS)*. 2005, pages 119–126. [see pages 6, 131]

[PT19] Patrick Prosser and James Trimble. *Peaty: An Exact Solver for the Vertex Cover Problem*. 2019. URL: https://dx.doi.org/10.5281/zenodo.3082356. [see page 73]

[Pul06] Wayne J. Pullan. "Phased Local Search for the Maximum Clique Problem". In: *J. Comb. Optim.* 12.3 (2006), pages 303–323. DOI: 10.1007/s10878-006-9635-y. [see page 33]

[Pul09] Wayne J. Pullan. "Optimisation of Unweighted/Weighted Maximum Independent Sets and Minimum Vertex Covers". In: *Discret. Optim.* 6.2 (2009), pages 214–219. DOI: 10.1016/j.disopt.2008.12.001. [see pages 29, 93]

[Put+15] Deepak Puthal, Surya Nepal, Cécile Paris, Rajiv Ranjan, and Jinjun Chen. "Efficient Algorithms for Social Network Coverage and Reach". In: *IEEE Intl. Cong. on Big Data (BigData Congress)*. 2015, pages 467–474. DOI: 10.1109/BigDataCongress.2015.75. [see pages 2, 26, 147]

[PV06] Anders S. Pedersen and Preben D. Vestergaard. "Bounds on the Number of Vertex Independent Sets in a Graph". In: *Taiwanese Journal of Mathematics* 10.6 (2006), pages 1575–1587. DOI: 10.11650/twjm/1500404576. [see page 120]

[PvdG21] Rick Plachetta and Alexander van der Grinten. "SAT-and-Reduce for Vertex Cover: Accelerating Branch-and-Reduce by SAT Solving". In: *Symp. on Algorithm Engineering and Experiments (ALENEX)*. 2021, pages 169–180. DOI: 10.1137/1.9781611976472.13. [see pages 29, 73]

[RA15] Ryan A. Rossi and Nesreen K. Ahmed. "The Network Data Repository with Interactive Graph Analytics and Visualization". In: *AAAI Conf. on Artificial Intelligence (AAAI)*. 2015, pages 4292–4293. URL: http://networkrepository.com. [see pages 18, 19, 95, 142, 151, 161, 162]

[Reb+11] Steffen Rebennack, Marcus Oswald, Dirk O. Theis, Hanna Seitz, Gerhard Reinelt, and Panos M. Pardalos. "A Branch and Cut Solver for the Maximum Stable Set Problem". In: *J. Comb. Optim.* 21.4 (2011), pages 434–457. DOI: 10.1007/s10878-009-9264-3. [see page 93]

[Rhy70] John M. W. Rhys. "A Selection Problem of Shared Fixed Costs and Network Flows". In: *Manage. Sci.* 17.3 (1970), pages 200–207. DOI: 10.1287/mnsc.17.3.200. [see page 97]

[Rin18] Giovanni Rinaldi. *Rudy*. 2018. URL: http://biqmac.aau.at/library/tar_files/mac_all.tar.gz. [see page 19]

[RKS22] Daniel Rehfeldt, Thorsten Koch, and Yuji Shinano. "Faster Exact Solution of Sparse MaxCut and QUBO Problems". In: *Computing Research Repository (CoRR)* abs/2202.02305 (2022). DOI: 10.48550/ARXIV.2202.02305. [see pages 131, 132]

[RR14] Mauricio G. C. Resende and Celso C. Ribeiro. "GRASP: Greedy Randomized Adaptive Search Procedures". In: *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. 2014, pages 287–312. DOI: 10.1007/978-1-4614-6940-7_11. [see page 94]

[RRW10] Franz Rendl, Giovanni Rinaldi, and Angelika Wiegele. "Solving Max-Cut to Optimality by Intersecting Semidefinite and Polyhedral Relaxations". In: *Math. Program.* 121.2 (2010), pages 307–335. DOI: 10.1007/s10107-008-0235-8. [see pages 130–132, 140]

[San+08] Pedro V. Sander, Diego Nehab, Eden Chlamtac, and Hugues Hoppe. "Efficient Traversal of Mesh Edges using Adjacency Primitives". In: *ACM Trans. Graph.* 27.5 (2008), page 144. DOI: 10.1145/1409060.1409097. [see pages 18, 19, 26, 155]

[San+16] Peter Sanders, Sebastian Lamm, Lorenz Hübschle-Schneider, Emanuel Schrade, and Carsten Dachsbacher. "Efficient Random Sampling - Parallel, Vectorized, Cache-Efficient, and Online". In: *Computing Research Repository (CoRR)* abs/1610.05141 (2016). DOI: 10.48550/ARXIV.1610.05141. [see page 203]

[San+18] Peter Sanders, Sebastian Lamm, Lorenz Hübschle-Schneider, Emanuel Schrade, and Carsten Dachsbacher. "Efficient Parallel Random Sampling - Vectorized, Cache-Efficient, and Online". In: *ACM Trans. Math. Softw.* 44.3 (2018), 29:1–29:14. DOI: 10.1145/3157734. [see page 202]

[San09] Peter Sanders. "Algorithm Engineering - An Attempt at a Definition". In: *Efficient Algorithms*. 2009, pages 321–340. DOI: 10.1007/978-3-642-03456-5_22. [see page 16]

[San10] Peter Sanders. "Algorithm Engineering". In: *Inform. Spektrum* 33.5 (2010), pages 475–478. DOI: 10.1007/s00287-010-0464-0. [see page 16]

[Sch20a] Sebastian Schlag. "High-Quality Hypergraph Partitioning". PhD thesis. Karlsruhe Institute of Technology, Germany, 2020. [see pages 16, 17, 22]

[Sch20b] Christian Schorr. "Improved Branching Strategies for Maximum Independent Sets". Bachelor's thesis. Karlsruhe Institute of Technology, Germany, 2020. [see page 204]

[SHG20] Matthias F. Stallmann, Yang Ho, and Timothy D. Goodrich. "Graph Profiling for Vertex Cover: Targeted Reductions in a Branch and Reduce Solver". In: *Computing Research Repository (CoRR)* abs/2003.06639 (2020). DOI: 10.48550/ARXIV.2003.06639. [see pages 28, 29, 86, 148]

[Shi+17] Satoshi Shimizu, Kazuaki Yamaguchi, Toshiki Saitoh, and Sumio Masuda. "Fast Maximum Weight Clique Extraction Algorithm: Optimal Tables for Branch-and-Bound". In: *Discret. Appl. Math.* 223 (2017), pages 120–134. DOI: 10.1016/j.dam.2017.01.026. [see page 95]

[Ski20] Steven Skiena. *The Algorithm Design Manual*. 2020. DOI: 10.1007/978-3-030-54256-6. [see page 11]

[SLP16] Pablo San Segundo, Alvaro Lopez, and Panos M. Pardalos. "A New Exact Maximum Clique Algorithm for Large and Massive Sparse Graphs". In: *Comput. Oper. Res.* 66 (2016), pages 81–94. DOI: 10.1016/j.cor.2015.07.013. [see page 32]

[SR18] Thomas Stützle and Rubén Ruiz. "Iterated Local Search". In: *Handbook of Heuristics*. 2018, pages 579–605. DOI: 10.1007/978-3-319-07124-4_8. [see page 13]

[SRJ11] Pablo San Segundo, Diego Rodriguez-Losada, and Agustin Jiménez. "An Exact Bit-Parallel Algorithm for the Maximum Clique Problem". In: *Comput. Oper. Res.* 38.2 (2011), pages 571–581. DOI: 10.1016/j.cor.2010.07.019. [see pages 31, 42]

[SS12] Peter Sanders and Christian Schulz. "Distributed Evolutionary Graph Partitioning". In: *Meeting on Algorithm Engineering and Experiments (ALENEX)*. 2012, pages 16–29. DOI: 10.1137/1.9781611972924.2. [see pages 20, 21]

[SS13a] Peter Sanders and Christian Schulz. "KaHIP v0.53 - Karlsruhe High Quality Partitioning - User Guide". In: *Computing Research Repository (CoRR)* abs/1311.1714 (2013). DOI: 10.48550/ARXIV.1311.1714. [see pages 29, 42, 44, 48]

[SS13b] Peter Sanders and Christian Schulz. "Think Locally, Act Globally: Highly Balanced Graph Partitioning". In: *Intl. Symp. on Experimental Algorithms (SEA)*. 2013, pages 164–175. DOI: 10.1007/978-3-642-38527-8_16. [see page 82]

[SS16] Peter Sanders and Christian Schulz. "Scalable Generation of Scale-Free Graphs". In: *Inf. Process. Lett.* 116.7 (2016), pages 489–491. DOI: 10.1016/j.ipl.2016.02.004. [see page 19]

[ST14] Pablo San Segundo and Cristóbal Tapia. "Relaxed Approximate Coloring in Exact Maximum Clique Search". In: *Comput. Oper. Res.* 44 (2014), pages 185–192. DOI: 10.1016/j.cor.2013.10.018. [see pages 31, 42, 75]

[Str16] Darren Strash. "On the Power of Simple Reductions for the Maximum Independent Set Problem". In: *Intl. Conf. on Computing and Combinatorics (COCOON)*. 2016, pages 345–356. DOI: 10.1007/978-3-319-42634-1_28. [see pages 18, 28, 33, 57, 65, 86, 98]

[SW11] Peter Sanders and Dorothea Wagner. "Algorithm Engineering". In: *it Inf. Technol.* 53.6 (2011), pages 263–265. DOI: 10.1524/itit.2011.9072. [see page 16]

[SW13] Peter Sanders and Dorothea Wagner. "Algorithm Engineering". In: *Inform. Spektrum* 36.2 (2013), pages 187–190. DOI: 10.1007/s00287-013-0684-1. [see page 16]

[SWC04] Alan J. Soper, Chris Walshaw, and Mark Cross. "A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph-Partitioning". In: *J. Glob. Optim.* 29.2 (2004), pages 225–241. DOI: 10.1023/B:JOGO.0000042115.44455.f3. [see pages 18, 19, 155]

[SZ18] Sándor Szabó and Bogdán Zaválnij. "A Different Approach to Maximum Clique Search". In: *Intl. Symp. on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. 2018, pages 310–316. DOI: 10.1109/SYNASC.2018.00055. [see page 73]

[TK09] Etsuji Tomita and Toshikatsu Kameda. "An Efficient Branch-and-Bound Algorithm for Finding a Maximum Clique with Computational Experiments". In: *J. Glob. Optim.* 44.2 (2009), page 311. DOI: 10.1007/s10898-008-9362-2. [see page 32]

[Tom+13] Etsuji Tomita, Yoichi Sutani, Takanori Higashi, and Mitsuo Wakatsuki. "A Simple and Faster Branch-and-Bound Algorithm for Finding a Maximum Clique with Computational Experiments". In: *IEICE Trans. Inf. Syst.* 96-D.6 (2013), pages 1286–1298. DOI: 10.1587/transinf.E96.D.1286. [see pages 31, 42, 75]

[TT77] Robert E. Tarjan and Anthony E. Trojanowski. "Finding a Maximum Independent Set". In: *SIAM J. Comput.* 6.3 (1977), pages 537–546. DOI: 10.1137/0206038. [see page 28]

[Tuk57] John W. Tukey. "On the Comparative Anatomy of Transformations". In: *The Annals of Mathematical Statistics* 28.3 (1957), pages 602–632. DOI: 10.1214/aoms/1177706875. [see page 22]

[Tuk77] John W. Tukey. *Exploratory Data Analysis*. 1977. ISBN: 9780201076165. [see page 22]

[Uhl21] Tim Niklas Uhl. "Communication Efficient Triangle Counting". Master's thesis. Karlsruhe Institute of Technology, Germany, 2021. [see page 204]

[VBB15] Anurag Verma, Austin Buchanan, and Sergiy Butenko. "Solving the Maximum Clique and Vertex Coloring Problems on Very Large Sparse Networks". In: *INFORMS J. Comput.* 27.1 (2015), pages 164–177. DOI: 10.1287/ijoc.2014.0618. [see page 32]

[vLA87] Peter J. M. van Laarhoven and Emile H. L. Aarts. *Simulated Annealing: Theory and Applications*. Volume 37. 1987. DOI: 10.1007/978-94-015-7744-1. [see page 13]

[vNT93] Nguyen van Ngoc and Zsolt Tuza. "Linear-Time Approximation Algorithms for the Max Cut Problem". In: *Combinatorics, Probability and Computing* 2 (1993), pages 201–210. DOI: 10.1017/S0963548300000596. [see page 131]

[Wan+12] Yang Wang, Zhipeng Lü, Fred W. Glover, and Jin-Kao Hao. "Path Relinking for Unconstrained Binary Quadratic Programming". In: *Eur. J. Oper. Res.* 223.3 (2012), pages 595–604. DOI: 10.1016/j.ejor.2012.07.012. [see page 133]

[Wan+13] Yang Wang, Zhipeng Lü, Fred W. Glover, and Jin-Kao Hao. "Probabilistic GRASP-Tabu Search Algorithms for the UBQP Problem". In: *Comput. Oper. Res.* 40.12 (2013), pages 3100–3107. DOI: 10.1016/j.cor.2011.12.006. [see page 130]

[Wan+19] Luzhi Wang, Chu-Min Li, Junping Zhou, Bo Jin, and Minghao Yin. "An Exact Algorithm for Minimum Weight Vertex Cover Problem in Large Graphs". In: *Computing Research Repository (CoRR)* abs/1903.05948 (2019). DOI: 10.48550/ARXIV.1903.05948. [see pages 2, 92]

[Wan+20] Yiyuan Wang, Shaowei Cai, Jiejiang Chen, and Minghao Yin. "SCCWalk: An Efficient Local Search Algorithm and its Improvements for Maximum Weight Clique Problem". In: *Artif. Intell.* 280 (2020), page 103230. DOI: 10.1016/j.artint.2019.103230. [see page 95]

[War+05] Deepak Warrier, Wilbert E. Wilhelm, Jeffrey S. Warren, and Illya V. Hicks. "A Branch-and-Price Approach for the Maximum Weight Independent Set Problem". In: *Networks* 46.4 (2005), pages 198–209. DOI: 10.1002/net.20088. [see page 93]

[War03] Deepak Warrier. "A Branch, Price, and Cut Approach to Solving the Maximum Weighted Independent Set Problem". PhD thesis. Texas A&M University, United States, 2003. [see page 93]

[WCY16] Yiyuan Wang, Shaowei Cai, and Minghao Yin. "Two Efficient Local Search Algorithms for Maximum Weight Clique Problem". In: *AAAI Conf. on Artificial Intelligence (AAAI)*. 2016, pages 805–811. [see page 95]

[WH06] Jeffrey S. Warren and Illya V. Hicks. "Combinatorial Branch-and-Bound for the Maximum Weight Independent Set Problem". 2006. URL: https://www.caam.rice.edu/~ivhicks/jeff.rev.pdf. [see pages 91, 98, 99, 107]

[WH13] Qinghua Wu and Jin-Kao Hao. "An Adaptive Multistart Tabu Search Approach to Solve the Maximum Clique Problem". In: *J. Comb. Optim.* 26.1 (2013), pages 86–108. DOI: 10.1007/s10878-011-9437-8. [see pages 29, 30, 95]

[WH15a] Qinghua Wu and Jin-Kao Hao. "A Review on Algorithms for Maximum Clique Problems". In: *Eur. J. Oper. Res.* 242.3 (2015), pages 693–709. DOI: 10.1016/j.ejor.2014.09.064. [see pages 11, 32]

[WH15b] Qinghua Wu and Jin-Kao Hao. "Solving the Winner Determination Problem via a Weighted Maximum Clique Heuristic". In: *Expert Syst. Appl.* 42.1 (2015), pages 355–365. DOI: 10.1016/j.eswa.2014.07.027. [see pages 2, 90]

[WH22] Yang Wang and Jin-Kao Hao. "Metaheuristic Algorithms". In: *The Quadratic Unconstrained Binary Optimization Problem: Theory, Algorithms, and Applications*. 2022. Chapter 9, pages 209–225. ISBN: 9783031045196. [see page 133]

[WHG12] Qinghua Wu, Jin-Kao Hao, and Fred W. Glover. "Multi-Neighborhood Tabu Search for the Maximum Weight Clique Problem". In: *Ann. Oper. Res.* 196.1 (2012), pages 611–634. DOI: 10.1007/s10479-012-1124-3. [see pages 29, 30, 95]

[Wie18] Angelika Wiegele. *BiqMac Library*. 2018. URL: http://biqmac.aau.at/biqmaclib.html. [see pages 19, 132, 161, 200]

[XGA13] Jingen Xiang, Cong Guo, and Ashraf Aboulnaga. "Scalable Maximum Clique Computation Using MapReduce". In: *IEEE Intl. Conf. on Data Engineering (ICDE)*. 2013, pages 74–85. DOI: 10.1109/ICDE.2013.6544815. [see page 31]

[Xia+17] Mingyu Xiao, Weibo Lin, Yuanshun Dai, and Yifeng Zeng. "A Fast Algorithm to Compute Maximum k-Plexes in Social Network Analysis". In: *AAAI Conf. on Artificial Intelligence (AAAI)*. 2017, pages 919–925. [see pages 2, 11, 28]

[Xia+21] Mingyu Xiao, Sen Huang, Yi Zhou, and Bolin Ding. "Efficient Reductions and a Fast Algorithm of Maximum Weighted Independent Set". In: *Intl. Conf. on World Wide Web (WWW)*. 2021, pages 3930–3940. DOI: 10.1145/3442381.3450130. [see page 92]

[XKK16] Hong Xu, T. K. Satish Kumar, and Sven Koenig. "A New Solver for the Minimum Weighted Vertex Cover Problem". In: *Intl. Conf. on Integration of AI and OR Techniques in Constraint Programming (CPAIOR)*. 2016, pages 392–405. DOI: 10.1007/978-3-319-33954-2_28. [see page 93]

[XN13] Mingyu Xiao and Hiroshi Nagamochi. "Confining Sets and Avoiding Bottleneck Cases: A Simple Maximum Independent Set Algorithm in Degree-3 Graphs". In: *Theor. Comput. Sci.* 469 (2013), pages 92–104. DOI: 10.1016/j.tcs.2012.09.022. [see pages 28, 33, 35–38, 78–80]

[XN17] Mingyu Xiao and Hiroshi Nagamochi. "Exact Algorithms for Maximum Independent Set". In: *Inf. Comput.* 255 (2017), pages 126–146. DOI: 10.1016/j.ic.2017.06.001. [see pages 4, 15, 28, 38, 74, 79, 80]

[Zav19] Bogdán Zaválnij. *zbogdan/pace-2019 a*. 2019. URL: https://doi.org/10.5281/zenodo.3228802. [see page 73]

[Zhe+20] Weiguo Zheng, Jiewei Gu, Peng Peng, and Jeffrey Xu Yu. "Efficient Weighted Independent Set Computation over Large Graphs". In: *IEEE Intl. Conf. on Data Engineering (ICDE)*. 2020, pages 1970–1973. DOI: 10.1109/ICDE48307.2020.00216. [see pages 2, 64, 92, 147]

[ZHG17] Yi Zhou, Jin-Kao Hao, and Adrien Goëffon. "PUSH: A Generalized Operator for the Maximum Vertex Weight Clique Problem". In: *Eur. J. Oper. Res.* 257.1 (2017), pages 41–54. DOI: 10.1016/j.ejor.2016.07.056. [see page 95]

[Zuc07] David Zuckerman. "Linear Degree Extractors and the Inapproximability of Max Clique and Chromatic Number". In: *Theory Comput.* 3.1 (2007), pages 103–128. DOI: 10.4086/toc.2007.v003a006. [see page 11]