Algorithm Engineering and Integer Programming for the Maximum Cut Problem

Dissertation
zur
Erlangung des Doktorgrades (Dr. rer. nat.)
der
Mathematisch-Naturwissenschaftlichen Fakultät
der
Rheinischen Friedrich-Wilhelms-Universität Bonn

von
Jonas Charfreitag
aus
Tübingen

Bonn, 2024

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn

Gutachterin / Betreuerin:	Prof. Dr. Petra Mutzel
Gutachter:	Prof. Dr. Thorsten Koch
Tag der Promotion:	08.05.2025
Erscheinungsjahr:	2025

Abstract

The maximum cut problem, MaxCut for short, is one of the fundamental problems in combinatorial optimization. It is part of a large family of graph partitioning problems that ask for a division of all vertices of the graph into disjoint sets. In its optimization form MaxCut is about finding a bipartition that maximizes the value of a cut. A cut in the context of a vertex partition is defined as the set of edges connecting vertices of different partitions. The value of a cut is the sum of all weights associated with the edges of the cut.

Although MaxCut allows for a compact description, it has highly relevant modeling power. Many real-world applications for MaxCut have been described in the literature over the years; these range from chip design to scheduling sports leagues and, more recently, the benchmarking of certain quantum algorithms and computers. Depending on the application, good solutions do not suffice and practitioners and researchers are interested in optimal solutions. Unfortunately, MaxCut is NP-hard in general, and it is still unknown whether we can solve NP-hard problems fast, that is, in time polynomial in the input size. However, practical algorithms for solving MaxCut to optimality have been designed in the past, and this thesis aims to improve them and develop new ones. We focus especially on techniques for sparse graphs, as real-world instances often turn out to be sparse.

An important way to speed up algorithms for hard problems is to reduce the search space before even exploring it. The process of performing these reductions without sacrificing optimal solutions is called presolving. We develop new presolving algorithms for MaxCut in three categories. One of these is based on vertex separators that have not been explicitly considered for MaxCut so far.

For exploring and further pruning of the search space for MaxCut we resort to integer programming and the branch-and-cut algorithm, which has yielded good results for exact MaxCut algorithms in the past. We extend previous work and suggest a refined integer program. This model has implications for other modules part of the branch-and-cut algorithm that we present in detail. Examples are the generation of cutting planes and problem-specific branching rules. From these we derive new and concrete algorithms that can be employed in MaxCut solvers based on branch-and-cut.

To evaluate the practical relevance of our new techniques, we perform elaborate experimental studies and compare them against the state of the art. For this we carefully engineered a new solver that contains state-of-the-art techniques and our new ones. The solver is competitive to the fastest MaxCut solver in general and clearly outperforms the state of the art for certain inputs. In detailed ablation studies, we track these improvements down to our new techniques. Especially, our presolving and cutting plane generation offers significant speed-up potential of up to one order of magnitude over the state of the art.

Acknowledgments

First of all, I would like to thank my advisor Petra Mutzel for the opportunity to work in her group, her scientific supervision, and her strong encouragement. I also want to thank the other two members of our original "McSparse gang" for introducing me to what became the center of my research for the last few years, MaxCut: Michael Jünger and especially Sven Mallach for many fruitful discussions and great insight and advice.

I also thank the student assistants with whom I had the pleasure to work on the code behind this project: Mohammed Ghannam, Konstantinos Papadopoulos, Claude Jordan, Michael Kaibel and Franziska Kehe. In the same context, special thanks are due to researchers who go the extra mile and make all parts of their work publicly available. The scientific community in general, and me and my research in particular, significantly benefit from work not locked behind paywalls and without closed source code.

Thanks to my former colleagues at Dortmund and Bonn and especially the current ones Philip Mayer, Jonas Sauer, and Lukas Schürmann for all the scientific and less serious discussions, the poof-reading, and the great atmosphere at work in general.

The final and very important thanks go to my friends and especially my family and girlfriend! Many parts of this journey relied on your unconditional support behind the scenes.

20.12.2024
Jonas Charfreitag

Contents

1. Introduction

Many real-world networks and network-like structures can be modeled as so-called graphs. This includes, but is not limited to, road networks, chemical molecules, social networks, and the World Wide Web. One branch of computer science is concerned with algorithms that help to analyze properties of these graphs.

An Optimization Problem Consider the graph depicted in Figure 1.1. Imagine each vertex of the graph representing one account in a social network. The numbers next to the edges (we call these edge weights) indicate how often two persons were talking to each other recently; the higher the number, the more the two accounts were chatting.

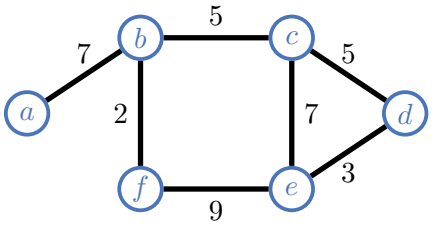


Figure 1.1.: A graph with six vertices (the blue cycles) $\{a, b, c, d, e, f\}$ and seven edges (the black connections).

Now, let us say that we want to test a new feature for the social network on a subgroup of people. Clearly, it might make sense to pick a diverse group of accounts to properly evaluate feedback on the new functionality. One possible measure for the diversity of a set of accounts is to sum over all edge weights of edges leaving the set of selected accounts. If this results in a large number, most of the communication happens between the selected group and the rest, and there is little interaction inside of our selected group.

We can formulate this scenario as what we call a *combinatorial optimization problem* in computer science and mathematics. Every such problem has some kind of input, in our case a graph. Additionally, every problem has some kind of output, in our case, a subset of all vertices. Finally, combinatorial optimization problems are not satisfied with any solution, but ask for a solution that maximizes (or minimizes) some value.

The Maximum Cut Problem Formally describing our example of selecting a group of users for a feature test reveals that we actually want to solve what the literature calls the *Maximum Cut Problem*, or MaxCut for short. MaxCut is the central problem of this thesis and a classical combinatorial optimization problem on graphs, part of the large family of graph partitioning problems. More than 50 years of research on

MaxCut resulted in a multitude of theoretical results and practical algorithms for the problem, many of which were motivated by real-world applications of the problem. Unfortunately, MaxCut belongs to an important class of problems, for which we have not found provably fast algorithms in the general case (or a proof that none exist), yet.

Clearly, a necessary condition for a combinatorial problem to be hard is the existence of many possible solutions. Otherwise, one could always just enumerate all possible solutions and pick one with the highest objective value. When solving MaxCut on a graph, we ask for a subset of all n vertices. This results in 2^n possible solutions. It is important to note, that an exponentially sized solution space does not generally rule out fast (that is, faster than full enumeration) algorithms. Had we asked for a group of people in our example around Figure 1.1, that has as *little* contact as possible to the rest of the network, we would have been in the territory of the so called *Minimum Cut Problem*. For this problem researchers have presented multiple fast algorithms in the past. But even if a problem like MaxCut turns out to be hard in theory, this does not mean that tackling it in practice is unlikely to succeed.

1.1. Integer Programming and Algorithm Engineering

Many problems arising from real-world applications can be modeled as *integer programs* (MaxCut being one of them). Finding optimal solutions for integer programs also belongs to the class of problems for which we do not know whether or not fast algorithms exist. However, integer programming is quite popular in theory and practice. For one, using a common modeling language has the advantage that algorithmic advances for integer programming can directly impact many different problems. In addition, very powerful solvers (many of which are based on the branch-and-cut algorithm) have been developed in academia and industry. They solve many integer programs in reasonable time, despite missing theoretical guarantees on their running time.

In fact, this discrepancy between the real-world performance of algorithms and the theory surrounding them can be observed throughout the field of computer science. The branch of research that deals with this gap between theory and practice is called *algorithm engineering*. It lies at the intersection of theoretical computer science and practical software engineering and involves a cycle of systematic design, analysis, implementation, and experimental evaluation of algorithms, ensuring their efficiency and effectiveness in real-world applications.

1.2. Overview and Contribution

The core contribution of this thesis are new techniques for faster exact solving of MaxCut on sparse graphs and the evaluation of these in extensive experimental studies. We develop novel theoretical results for presolving techniques and branch-and-cut modules. From these we derive new practical algorithms and their impact on a new state-of-the-art solver is tested with a diverse set of established benchmark instances. Speedups of up to one order of magnitude over the already sophisticated state of the art can be observed.

In the following, we summarize the contents of each chapter of this thesis and highlight scientific contributions.

Chapter 2 introduces mathematical basics and notation relevant throughout this thesis and some basic algorithms.

Chapter 3 elaborates on MaxCut, some related problems, and highlights the corresponding applications. We also introduce our new solver as a web-service (McSparse) and the benchmark instances for the next two chapters here. McSparse is the first integer linear programming solver directly available to the general public and has received more than 45 000 instance submissions since its introduction, of which most could be solved to optimality. This clearly shows the public interest in solving MaxCut to optimality.

Chapter 4 is concerned with the theory and practice of presolving techniques for MaxCut. Presolving generally consists of the two components decomposition and data reduction. For MaxCut related data reduction, we suggest a unified view on existing presolving techniques. Concretely, we categorize them into the three main groups: vertex similarity based, edge separator based, and vertex separator based. We introduce one new rule for each of the first two categories. The last category (vertex separator based reduction) has, to our knowledge, not been explicitly considered for MaxCut. Through a new framework for vertex separators, we extend previous work and derive multiple new data reduction rules. We carefully engineered an algorithm that incorporates previous rules and our new ones. In the experiments, we then see that our new algorithm is more effective in reducing graphs than the state-of-the-art approaches. Additionally, our new presolving algorithm turns out to be highly efficient. When paired with a state-of-the-art heuristic, we get better solutions in the same amount of time, and when paired with state-of-the-art exact solvers, we see speedups of up to one order of magnitude.

Chapter 5 introduces an exact algorithm based on branch and cut and the theory behind its components. The branch-and-cut framework allows for the integration of many different modules. We focus on fundamental ones: presolving, binary programming formulations, cutting plane generation, branching rules, and primal heuristics. Our theoretical analysis of a new model, which we call the root-triangulated model, uncovers implications for cutting plane generation and branching rules that allow algorithmic exploitation. Concretely, we show that for two important types of cutting planes, the structure of the model allows more efficient separation, when other branch-and-cut components have fixed certain variables of the model. Additionally, the analysis gives clear motivation for our new branching rule, called degree-dynamic. We also uncover the relation of certain cutting planes and xor-constraints and present new ideas to fully exploit symmetry in a shortest path subroutine for the generation of cutting planes. All ideas are tested by implementing them in a new solver (called SMS) that turns out to be competitive to the state of the art. An ablation study reveals the influence of the different new components we suggested and shows that they result in speedups of up to one order of magnitude.

Finally, Chapter 6 summarizes our results and discusses open problems and future work for research on exact MaxCut solving.

1.3. Relevant Publications

The author of this work was involved in three published peer-reviewed publications relevant to this thesis. They constitute the building blocks for the results presented throughout this work.

- A new publicly available solver, which we will discuss in Section 3.3, and an experimental study were presented in:
  Jonas Charfreitag, Michael Jünger, Sven Mallach, and Petra Mutzel (2022). "McSparse: Exact Solutions of Sparse Maximum Cut and Sparse Unconstrained Binary Quadratic Optimization Problems". In: *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2022, Alexandria, VA, USA, January 9-10, 2022*. Ed. by Cynthia A. Phillips and Bettina Speckmann. SIAM, pp. 54–66. DOI: 10.1137/1.9781611977042.5.

- The main results for Chapter 4 were first developed in:
  Jonas Charfreitag, Christine Dahn, Michael Kaibel, Philip Mayer, Petra Mutzel, and Lukas Schürmann (2024a). "Separator Based Data Reduction for the Maximum Cut Problem". In: *22nd International Symposium on Experimental Algorithms, SEA 2024, July 23-26, 2024, Vienna, Austria*. Ed. by Leo Liberti. Vol. 301. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 4:1–4:21. DOI: 10.4230/LIPICS.SEA.2024.4.

- The basis for Chapter 5 was first introduced in:
  Jonas Charfreitag, Sven Mallach, and Petra Mutzel (2023). "Integer Programming for the Maximum Cut Problem: A Refined Model and Implications for Branching". In: *SIAM Conference on Applied and Computational Discrete Algorithms, ACDA 2023, Seattle, WA, USA, May 31 - June 2, 2023*. Ed. by Jonathan W. Berry, David B. Shmoys, Lenore Cowen, and Uwe Naumann. SIAM, pp. 63–74. DOI: 10.1137/1.9781611977714.6.

2. Preliminaries

In this chapter, we introduce some notation and mathematical basics that are relevant throughout this thesis. In general, we follow the notation of Schrijver (2003) and Korte and Vygen (2018). We also present some basic graph algorithms that will form the building blocks of more involved techniques later.

2.1. Sets and Graphs

For two sets S and T, $S \subseteq T$ denotes that S is a subset of T. To describe a *proper* subset T ($T \subseteq S$ and $T \neq S$), we use $S \subset T$. The symmetric difference of two sets S_1 and S_2 will be denoted by $S_1 \oplus S_2$. A bipartition of a set S is a pair (P_1, P_2), of two disjoint sets $P_1 \cap P_2 = \emptyset$ whose union is S, so $P_1 \cup P_2 = S$. We also define a partial bipartition $P' = (S'_1, S'_2)$ of a set S, with again $S'_1 \cap S'_2 = \emptyset$, but $S'_1 \cup S'_2 \subset S$.

The most important combinatorial structures of this work are *simple graphs*. A simple graph is a tuple (V, E), where V is a finite set containing all *vertices* of the graph and $E \subseteq V \times V$ is the finite set of all *edges* (all connections between vertices from V). We differentiate between directed and undirected simple graphs. For undirected simple graphs, E consists of unordered pairs of two vertices $\{u, v\}$, with $v \in V \wedge u \in V \wedge v \neq u$. For directed simple graphs, the edges in E are ordered pairs (u, v), with $v \in V \wedge u \in V \wedge v \neq u$. The graphs just described are called simple, as, by their definition, there is either one edge connecting two vertices u and v or none. This disallows *parallel edges*. In addition, an edge may never be of the form $\{u, u\}$ or (u, u). Hence, all graphs we consider do not contain *self-loops*. As nonsimple graphs are of no interest in this work, we will use graph and simple graph synonymously. We call two vertices *adjacent* to each other if they share an edge. For an undirected edge $e = \{u, v\}$ or a directed edge $e = (u, v)$ the two vertices u and v are said to be *incident* to the edge e.

A graph $G' = (V', E')$ is called a subgraph of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. If E' contains all edges of E incident to vertices of V', G' is called an induced subgraph. Our notation for an induced subgraph of a vertex set V' is $G[V']$.

To keep the notation compact, for a graph G we often refer to the set containing a single vertex only by the vertex itself, instead of $\{v\}$ we just write v. Sometimes we also shorten the notation for edges from $e = \{u, v\}$ to uv. $G - v$ for a vertex v is $G[V \setminus \{v\}]$.

The neighborhood of a vertex v is denoted by $N(v)$ and the degree of a vertex by $d(v)$, therefore $d(v) = |N(v)|$. For sets $S \subset V$ we write $N(S)$ to capture the union of all neighborhoods of vertices in S excluding all vertices in S. The two-neighborhood N_2 of a vertex u is the set of all vertices adjacent to u or a neighbor of u: $N_2(u) = N(u) \cup \bigcup_{v \in N(u)} N(v) \setminus \{u\}$.

Most graphs we are going to deal with have a weight associated with each edge. In these cases we call them weighted graphs, consisting of the triple $G = (V, E, w)$, where w is a weight function $w : E \to \mathbb{R}$. For compact notation, we often write w_e or w_{uv} for the weight of the edge $e = \{u, v\}$ instead of $w(\{u, v\})$.

An important operation on graphs we will make use of is what we call an *edge contraction*. We define the contraction of an edge $e = \{u, v\}$ as replacing u and v by a new vertex a. For every vertex $b \in N(\{u, v\})$ an edge $\{a, b\}$ is added with weight $w_{ub} + w_{vb}$ if both u and v are adjacent to b and the same weight as the edge between u or v and b otherwise.

Graph properties There are many metrics that capture the overall structure of graphs. We present two fundamental ones for undirected graphs here. Let $G = (V, E)$, $n = |V|$ and $m = |E|$. A relevant graph property is the *average vertex degree*:

$$\frac{\sum_{u \in V} d(v)}{n} = \frac{2m}{n}$$

Another relevant property is the *density* of a simple undirected graph $G = (V, E)$ is the fraction of edges in G compared to all possible edges:

$$\frac{m}{\frac{1}{2} \cdot \binom{n}{2}} = \frac{2m}{n^2 - n}$$

The number of edges of a graph relative to its number of vertices is often used to discriminate between different types of graphs. Graphs with many edges, formally graphs where $|E| \in \Omega(|V|^2)$, are usually called *dense*. Graphs with $|E| \in O(|V|)$ on the other hand, are called *sparse*. Observe that the definition for sparse graphs implies a constant average degree.

2.1.1. Paths, Cycles and Distances

Given a simple graph $G = (V, E)$, a *walk* P is a sequence of vertices and edges of G: $v_0, \{v_0, v_1\}, v_1, \ldots \{v_k, v_{k+1}\}, v_{k+1}$. If all vertices of a walk are pairwise different, we call the walk a *path*. A *chord* of a path is an edge not part of the path, but incident to two vertices of the path. If a path has no chord, we call it chordless. A *cycle* is a walk, but the first and last vertex are the same. If all vertices apart from the start and end vertex of a cycle are pairwise different, we call the cycle simple. Similarly to paths, a cycle in a graph G is *chordless*, if there is no edge connecting two vertices in the cycle, which is not part of the cycle itself.

The weighted length (or just length) of a path P is the sum of all edge weights of edges in P. The hop length of a path P is the number of edges in P. If all edges of a graph have a weight of 1, the weighted length is equal to the hop length.

2.1.2. Classes of Graphs

Many different classes of graphs have been described over the years. Relevant for this work are the following.

- **Complete graphs:** We call a graph $G = (V, E)$ a complete graph if it has a density of one.
- **Bipartite graphs:** Graphs without cycles of odd length are called bipartite. Their vertices can be partitioned into two sets in such a way that no edge connects two vertices from the same set.
- **Planar graphs:** A graph is said to be planar if it admits an embedding into the plane, without any edges crossing.
- **Regular graphs:** A graph in which all edges have the same degree is called regular.
- **Grid graphs:** A square grid (also called a mesh or lattice) graph of dimension d, is a graph with $V = \{1, \ldots, n\}^d$ and vertices with Euclidean distance of 1 are adjacent to each other. If additionally each vertex whose i-th entry is a 1 is connected to the vertex whose i-th entry is n and all other entries are the same, we call the graph a **torus graph**.
- **Trees and forests:** Forests are graphs $G = (V, E)$ without cycles. If additionally $|E| - 1 = |V|$, a forest is called a tree.

2.2. Cuts in Graphs

Of central interest in this work are bipartitions $S = (S_1, S_2)$ of the vertex set of a given weighted graph $G = (V, E, w)$. Depending on the context, it is sometimes more compact to talk about the characteristic vector of a vertex bipartition $z \in \{0, 1\}^{|V|}$ with

$$z_S(u) = \begin{cases} 1, & \text{if } u \in S_2 \\ 0, & \text{otherwise} \end{cases} \tag{2.1}$$

Every vertex bipartition (S_1, S_2) of a graph induces a *cut*:

Definition 2.2.1 (Cut)**.** *For a graph $G = (V, E)$ a cut $\delta \subseteq E$ is a set of edges induced by a vertex bipartition (S_1, S_2). Every edge that is part of the cut is incident to exactly one vertex in S_1 and one vertex in S_2.*

Depending on the context, we either represent a cut $\delta \subseteq E$ by the set of its edges or by its characteristic vector $x \in \{0, 1\}^{|E|}$, where we define the incidence vector to be

$$x_\delta(e) = \begin{cases} 1, & \text{if } e \in \delta \\ 0, & \text{otherwise} \end{cases} \tag{2.2}$$

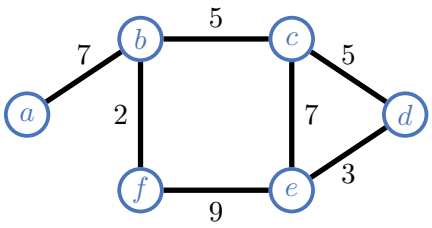


Figure 2.1.: A graph G and a MaxCut solution. An optimal bipartition is given by $(\{a, c, e\}, \{b, d, f\})$. Because of the inherent symmetry of MaxCut, the other optimal bipartition is $(\{b, d, f\}, \{a, c, e\})$. Edges of the optimal cut are dashed. The optimal value is $\Delta(G) = 23$.

Note that every vertex bipartition induces one unique cut, but, because of symmetry, there are always two vertex bipartitions inducing the same cut. Therefore, we sometimes simplify our wording and also call a vertex bipartition a cut when we refer to the cut induced by the vertex bipartition. Directly following from the definition of a cut in a graph G, one may see:

Observation 2.2.2 (Cuts and Cycles)**.** *For every graph $G = (V, E)$ and every cut in this graph:*

- *Every cut in G induces a bipartite (sub-)graph*
- *A cut cannot contain a cycle of odd length*
- *A selection of edges $F \subseteq E$ forms a cut in a graph G, if and only if it contains an even number of edges of each cycle of G.*

Cut Problems The weight of a cut δ is the sum of all edge weights in δ: $\sum_{e \in \delta} w_e$. The combinatorial optimization problem of finding a cut of maximum value is called MaxCut. We define $\Delta(G)$ as the value of a maximum cut. For a given (partial) bipartition $P' = (S'_1, S'_2)$ we write $\Delta(G, P')$ to denote the maximum value of all cuts in G respecting P'. A cut respects a partial bipartition if none of its edges connects vertices from the same set in P'. To define the core problem of this thesis formally:

Definition 2.2.3 (MaxCut Problem)**.**
Input: *A weighted undirected graph $G = (V, E, w)$.*
Objective: *Find a cut of maximum value.*

Figure 2.1 illustrates the MaxCut problem in an example. The problem of finding a cut of minimum value is called MinCut and usually has some additional constraints on the input and solution:

Definition 2.2.4 (MinCut Problem)**.**
Input: *A weighted undirected graph $G = (V, E, w)$, where $w_e \geq 0 \; \forall e \in E$.*
Objective: *Find a nonempty cut δ with minimum value.*

Sometimes we are interested in cuts for which two specific vertices are in different partitions. The literature usually calls those s-t-cuts:

Definition 2.2.5 (s-t-cut)**.** *A cut δ implied by the vertex bipartition (S_1, S_2) is called an s-t-cut for two vertices s and t, if $s \in S_1$ and $t \in S_2$.*

A data structure to compactly represent all minimum s-t-cuts of a graph was introduced by Gomory and Hu (1961) and is generally called the Gomory-Hu tree. Gusfield (1990) introduced an algorithm for the same problem, which is considered easier to implement (Goldberg and Tsioutsiouliklis 2001), but has the same running time.

2.3. Connected Components and Vertex Separator

A *connected component* of a graph G is a maximal subgraph of G in which any pair of vertices can reach each other by a walk. If a graph has exactly one connected component, it is called a *connected* graph. A graph with more than $k \geq 1$ vertices is called *k-connected* if it is connected, and the deletion of an arbitrary set of up to $k-1$ vertices does not change this property. A *k-connected component* of a graph G is a maximal subgraph of G in which every vertex can reach every other vertex via at least k vertex disjoint paths in G. For $k = 2$ k-connected components are also called *biconnected components* or *blocks*. A structure capturing the biconnected components of a graph and their relation is called a *block cut tree*. Each vertex of the tree represents one biconnected component of the graph, and two vertices share an edge if the biconnected components they represent have a vertex of the original graph in common. The concept has been extended to 3-connected (tri-connected) components. Hopcroft and Tarjan (1973) showed how to calculate the tri-connected components in linear time and the tree capturing the relation between the components was later formalized under the name of *SPQR-trees* by Battista and Tamassia (1990).

Vertices whose removal increases the number of connected components of a graph are called *vertex separators* or *vertex-cuts*, *articulation-sets* or *separating-sets* (Schrijver 2003). Vertex separators of size one are called articulation points. Formally:

Definition 2.3.1 (Vertex Separator)**.** *Let $G = (V, E)$ be an undirected graph. A vertex separator is a set of vertices $S \subset V$ for which removing S from G ($G[V \setminus S]$) increases the number of connected components by at least one. A vertex separator is minimal, if and only if no proper subset $S' \subset S$ forms a vertex separator. A vertex separator of minimum size is called a minimum vertex separator.*

The definition for vertex separators can be modified for edge separators, leading to an alternative definition of cuts:

Definition 2.3.2 (Edge Separator)**.** *Let $G = (V, E)$ be an undirected graph. An edge separator is a set of edges $T \subseteq E$ for which removing T from G increases the number of connected components by at least one.*

Algorithm 1: Heuristic Vertex Separator Search

Input: $G = (V, E)$, $k \geq 1$, $s \geq 1$
Output: $U \subset V$

```
1  S ← {v}
2  T ← N(v)
3  U ← ∅
4  while |S| + |T| ≤ s do
5      if |T| ≤ k then
6          U ← U ∪ T
7      u ← arg min_{u∈T} |N(S ∪ {u})|
8      S ← S ∪ {u}
9      T ← N(T ∪ {u})
10 return U
```

The literature offers two types of algorithms for finding vertex separators: One class of algorithms finds vertex separators of a certain size. The other is concerned with finding a minimum vertex separator.

Vertex Separator of Fixed Size Finding vertex separators of size $k \in 1, 2$ is similar to finding the $k + 1$-connected components of a graph. The linear-time algorithm of Tarjan (1972) for calculating the biconnected components of a graph via depth-first search implies a linear time algorithm for finding 1-separators. Similarly, the linear time algorithm for calculating the SPQR-tree of a graph, forms a basis for an algorithm for finding separators of size two. For separators of size three, there is an algorithm based on ear decomposition by Kanevsky and Ramachandran (1991) with runtime in $O(n^2)$.

Minimum Vertex Separators An algorithm for finding the smallest vertex separator of a graph was presented by Even and Tarjan (1975). The algorithm employs MaxFlow as a subroutine and when paired with Dinics MaxFlow algorithm, they show the smallest vertex separator can be found in $O(|V|^{1/2} \cdot |E|^2)$ time. Galil (1980), Esfahanian and Hakimi (1984), and Henzinger et al. (2000) presented improvements of this technique and Kanevsky (1993) extended the algorithm for the enumeration of all vertex separators of minimum size. For randomized algorithms, Li et al. (2021) recently introduced the state-of-the-art Monte Carlo algorithm that essentially runs in the time of a MaxFlow-algorithm.

Finding Small Vertex Separators Heuristically Hüffner et al. (2010) describe a fast heuristic for finding small vertex separators. Algorithm 1 describes our version of their technique. Although this procedure is not guaranteed to find all vertex separators, it has the benefit of not having requirements on the structure of the input graph and finds vertex separators of size $[1, \dots s]$. This algorithm has a $O(n + s^3)$ runtime implemen-

tation. Setting up a simple ID based lookup table, which allows to query if a vertex is in S, T or neither in constant time, requires $O(n)$ time. The while loop is executed at most s times, as S increases by exactly one in each iteration. Inside the while loop, only finding the arg min requires non-constant time. For finding the vertex that minimizes the size of the neighborhood one can make use of the following equation in line 7: $N|(S \cup \{u\})| = |T| - 1 + \sum_{v \in N(u)}((v \notin S) \wedge (v \notin T))$. If u has more than $2 \cdot s$ many neighbors, we do not need to consider it for the arg min, as adding u to S would always result in $|S| + |T| > s$. There are at most s vertices in T, therefore finding the arg min requires $O(s^2)$ time.

2.4. Mathematical Programming

Combinatorial optimization problems can be described in declarative programming style as a *mathematical programs*, or MPs for short. A mathematical program consists of three main parts:

- A set of (decision) variables encoding solutions
- A set of constraints, defining the set of feasible solutions
- An objective function, assigning a value to each solution

If we are interested in finding a solution with maximum value, we call the problem a maximization problem, otherwise a minimization problem. Every minimization problem can be transformed into a maximization problem, by multiplying the objective coefficients of all variables by -1. Therefore, when not explicitly stated otherwise, w.l.o.g. we assume all mathematical programs in the following to be of maximization type.

Depending on the type of variables, constraints, and objective function, the literature differentiates between types of mathematical programs. If the objective function is linear and all constraints are linear, the program is called a *linear program*. *Quadratic programs* allow for linear and quadratic objective functions and constraints. If variables are required to be integer, the program is usually called an *integer program* and a *binary program*, if all variables are required to take on binary values. When variables may be of mixed types, the program is called a MIP (mixed integer program). Sometimes, it allows for a more compact and / or intuitive formulation to express the objective of a binary program in terms of boolean algebra. It is important to note that every boolean expression on binary variables can be translated into an equivalent quadratic expression. For example, a logical expression on two variables $x \wedge y$ for example simply becomes $x \cdot y$ and a logical xor expression on two variables $x \oplus y$ becomes $x + y - 2xy$.

Of most interest for this thesis are binary linear programs. One in maximization form with n variables looks like this:

$$\begin{aligned} \text{maximize } & c^\intercal \cdot x \\ \text{s.t. } & Ax \leq b \\ & x \in \{0,1\}^n \end{aligned}$$

Here c is the cost vector of the objective function, A is the constraint matrix, b is the right-hand side.

Solving Integer Linear Programs Solving integer linear programs is NP-hard in general. But because of their broad applicability, a lot of effort has been put into the design of algorithms for integer linear programs. Many successful ones are based on the branch-and-cut framework. Branch and cut is an extension of branch and bound and consists of two main components. In general branch-and-cut algorithms explore the solution space via a search tree and try to prune as many nodes of the tree as early as possible. The pruning is achieved through *bounding*: By employing *primal heuristics* the algorithm searches for good feasible solutions. The value of the best solution currently is referred to as *primal bound*. At each node of the tree the algorithm then calculates an upper bound, the *dual bound*, on the objective value achievable in the (sub-)solution space the node represents. If the primal bound exceeds or matches the dual bound, the current node can be pruned. Otherwise, the algorithm splits the solution space into two and creates a child node for each sub-problem. The process of splitting the solution space is called *branching*.

This framework suits integer linear programs quite well, as for the calculation of lower bounds the *LP relaxation* of the integer linear program comes in handy. By dropping the integer constraints on variables, the resulting program becomes a linear program and linear programs can be solved in polynomial time. The solution of the linear program is an upper bound on the solution of the integer linear program. The cut in branch and cut refers to a technique to further strengthen the LP relaxation. By generating additional valid inequalities and adding them to the LP relaxation one aims for

2.5. Benchmarking Algorithms and Solvers

When benchmarking algorithms or solvers, one has to carefully choose the metrics used for comparisons. Runtimes are often aggregated into an average, but the average is quite susceptible to outliers. Therefore, a special version of the geometric mean, the shifted geometric mean is often to be preferred. Already in its general form, the geometric mean is less sensitive to large values. Furthermore, shifting all values reduces the impact of small values.

Definition 2.5.1 (Shifted Geometric Mean)**.** *Let $t_1, \ldots, t_k \in \mathbb{R}_{\geq 0}$ be a series of measurements and $s \in \mathbb{R}_{\geq 0}$:*

$$\left(\prod_{i=1}^{n}(t_i + s)\right)^{\frac{1}{n}} - s \tag{2.3}$$

is called the shifted geometric mean, with shift s.

For large n or large values of t_i, the product may grow quite fast, which can result in numerical problems for computer programs. Hence, a practical implementation might resort to the following equivalent formula instead:

$$\exp\left(\frac{1}{n}\sum_{i=1}^{n}\log(t_i+s)\right)-s \tag{2.4}$$

Performance Variability and Seeding Solvers in general and especially MIP-solvers tend to perform differently, even when seemingly neutral changes are made to the input or the solver. The concept behind this observable noise is called *performance variability* (Lodi and Tramontani 2013). To allow for a statistically sound analysis of benchmark result, it is important to run the same experiment multiple times with different random seeds. Random seeds or seeds for short serve two main purposes in the context of our experiments. 1) They permute the input instance, by assigning each vertex of the input a new ID. For presolving this can result in rules being applied in different order and therefore lead to different results. For MIP-solving this implicitly shuffles the rows and columns of the corresponding MIP, which may change the tie-breaking order for e.g. the branching heuristic. 2) They define the sequence of pseudo random numbers for all algorithmic components making use of (pseudo) randomness. As a result primal heuristics relying on randomness might perform better / worse with different seeds.

3. MaxCut and Related Problems

Research on MaxCut is strongly motivated by its many applications in, e.g., image processing (Rother et al. 2007), sports tournament schedules (Elf et al. 2003) or VLSI design (Barahona et al. 1988), and its importance for quantum annealing (Jünger et al. 2021). In the first part of this chapter we will introduce the central optimization problem of this thesis, MaxCut, in more detail and discuss its complexity. Next, we investigate the relation between MaxCut and other optimization problems and highlight the corresponding real-world applications. For all the selected problems originally defined on graphs, the transformation to MaxCut results in the same set of vertices and edges and only requires the modification of edge weights. Not increasing the size of the input when transforming to MaxCut highlights the broad scope of algorithms for solving MaxCut in theory and practice. The second to last part of this chapter is concerned with MaxCut solvers. We discuss existing solvers along with our web-solver first introduced in a peer-reviewed publication in 2022[1]. Finally, we present instances (more precisely, graphs) that solvers usually get benchmarked on and that serve as a benchmark set for our computational experiments later.

3.1. MaxCut

Recall Definition 2.2.3: The maximum cut problem (MaxCut) takes an edge weighted graph $G = (V, E, w)$ as input and asks for a vertex bipartition V maximizing the value of its implied cut. The cut is defined as the set of edges connecting vertices from different partitions and its value is the sum over all weights of edges forming the cut.

MaxCut on a weighted input graph $G = (V, E, w)$ can be formulated as a compact mathematical program with one binary variable per vertex. The variable encodes the partition of the corresponding vertex and the vector of variables is the characteristic vector of a bipartition. The objective function sums over all edges and if the two endpoints belong to different partitions the edge contributes with its weight to the objective value.

$$\max \sum_{\{i,j\} \in E} (z_i \oplus z_j) w_{ij} \tag{3.1}$$

$$z \in \{0,1\}^{|V|} \tag{3.2}$$

[1] Jonas Charfreitag, Michael Jünger, Sven Mallach, and Petra Mutzel (2022). "McSparse: Exact Solutions of Sparse Maximum Cut and Sparse Unconstrained Binary Quadratic Optimization Problems". In: *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2022, Alexandria, VA, USA, January 9-10, 2022*. Ed. by Cynthia A. Phillips and Bettina Speckmann. SIAM, pp. 54–66. DOI: 10.1137/1.9781611977042.5.

Transforming the logical expression in the objective function leads to the following unconstrained binary quadratic program for MaxCut (we will cover different binary linear programming formulations for MaxCut in more detail later in Section 5.1):

$$\max \sum_{\{i,j\} \in E} (z_i + z_j - 2z_i z_j) \cdot w_{ij} \tag{3.3}$$

$$z \in \{0,1\}^{|V|} \tag{3.4}$$

MaxCut Complexity

In general, MaxCut is NP-hard (Karp 1972) which means that no one has yet found an algorithm solving MaxCut in polynomial time and we do not even know if such an algorithm exists. Furthermore, MaxCut is APX-hard (Papadimitriou and Yannakakis 1991). This implies that unless there is a polynomial-time algorithm for MaxCut no polynomial-time approximation scheme for MaxCut exists. However, for some special classes of graphs, polynomial-time algorithms have been developed over the years (Hadlock 1975; Liers and Pardella 2012; Barahona 1983; Chimani et al. 2019; Galluccio et al. 2001; Grötschel and Nemhauser 1984; McCormick et al. 2003). For some additional classes of graphs FPT-algorithms are known (parameterized e.g. in crossing number (Chimani et al. 2020), or MaxCut-value above certain bounds (Crowston et al. 2015; Madathil et al. 2020)). We want to discuss three special cases here, for which MaxCut can be solved in polynomial time. The resulting algorithms will be part of our experiments later.

No / Few Positive Weights Clearly, the optimal cut in a graph where all edges have negative weight is empty and has a value of zero. Therefore, in these cases, it only takes one loop over all edges, $O(m)$ time, to find the optimal solution. This can be generalized as follows: If only one edge has positive value, either this edge is part of the maximum cut or the optimal cut is empty. Finding the cut with highest value containing the edge u, v comes down to finding a minimum s-t-cut in a graph with positive edges only.

Perfect Cuts We call a cut in a graph *perfect* if it contains all edges with weight > 0 and no edges with weight < 0. Finding a perfect cut in a graph G, or proving that no such cut exists, takes $O(n+m)$ time. This can be achieved by a simple graph traversal, which greedily partitions every vertex once it is explored. For graphs with positive edge weights only, this is the same as checking whether or not the graph is bipartite.

Perfect Cardinality Cut We say that a cut δ in a graph G with n vertices has perfect cardinality if no cut in any graph with n vertices consists of strictly more edges. Therefore, a perfect cardinality cut always contains $\lfloor \frac{n}{2} \rfloor \cdot \lceil \frac{n}{2} \rceil$ many edges. Finding a perfect cardinality cut, or proving that no such cut exists, takes $O(n+m)$ time and comes down to checking if G contains a $K_{\lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil}$ subgraph. The algorithm is related to ideas presented by Arbib (1988) and we sketch it below.

1. If $m < \lfloor \frac{n}{2} \rfloor \cdot \lceil \frac{n}{2} \rceil$, the graph can not have a $K_{\lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil}$ subgraph.
2. Else: Check for the existence of a $K_{\lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil}$ subgraph by using the following observation about the connected components of the complement graph of G, say $\overline{G}$: There is a $K_{\lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil}$ subgraph in G iff there is a bipartition (V_1, V_2) of V with $|V_1| = \lfloor \frac{n}{2} \rfloor, |V_2| = \lceil \frac{n}{2} \rceil$ such that in $\overline{G}$ there is no edge connecting vertices from V_1 to vertices in V_2. We can check for the existence of such a bipartition, by solving the SubsetSum Problem. The input is a list of the sizes of every connected component in $\overline{G}$. We use the fact that SubsetSum reduces to Knapsack by making one item per given number x_i, which has a weight of x_i and a value of x_i. Then we can use the dynamic programming algorithm for Knapsack by Dantzig (1957), which runs in time $O(nC)$ where C is the capacity of the knapsack and n the number of items. As there are at most n inputs for the SubsetSum problem and our target sum is n this yields an $O(n^2)$ algorithm to check if a graph has a $K_{\lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil}$ subgraph. As this case is only relevant when $m \in \Omega(n^2)$, the whole algorithm runs in $O(n + m)$.

If for a graph G all edges have the same positive weight, finding the maximum cut is the same as finding a cut of maximum cardinality. Therefore, the above algorithm solves MaxCut in linear time for such graphs, if they allow for a perfect cardinality cut.

3.2. Related Problems and Applications

We now turn to a whole family of problems related to MaxCut, which all have their own branch in the literature. All of these problems allow for a simple and straightforward transformation to MaxCut, operating on the same graph structure as the original problem.

One such example is the MaximumBipartiteSubgraph problem (Poljak and Tuza 1993), which is equivalent to MaxCut. Finding the largest (as in number of edges) bipartite subgraph of any graph is the same as finding a maximum cut in the same graph, where all edges have a weight of 1 (compare Observation 2.2.2).

3.2.1. MinUncut and EdgeBipartization

Another problem related to MaxCut is MinUncut: The problem of finding a vertex bipartition minimizing the sum over all weights of uncut edges. A corresponding binary program can be modeled like this:

$$\min \sum_{\{i,j\} \in E} (1 - (z_i \oplus z_j)) w_{ij} \tag{3.5}$$

$$z \in \{0,1\}^{|V|} \tag{3.6}$$

A bipartition maximizing the weight of edges part of the implied cut also minimizes the weight of all uncut edges, as the transformation to MaxCut is straight forward and results in the same binary program.

MinUncut is also known as the optimization version of EdgeBipartization (Guo et al. 2006), which asks for the minimum number of edges one has to delete to form a bipartite graph (the direct relation follows as for MaximumBipartiteSubgraph directly from Observation 2.2.2).

3.2.2. Balanced Subgraph and Signed MaxCut

Signed graphs (introduced by Heider 1946; Harary 1953) are unweighted graphs, where each edge is assigned to one of two categories (has one of two signs): $=$ and $\neq$. A graph is called balanced if it admits a vertex bipartition, where all vertices connected by an $\neq$-edge are in different partitions and all vertices incident to a $=$-edge are in the same partition.

Balanced Subgraph The problem of finding the minimum number of edges that need to be removed from a signed graph to get a balanced graph is called BalancedSubgraph (Hüffner et al. 2010) and this number is also known as the *frustration index* of the graph (Facchetti et al. 2011). The frustration index has been used, to e.g. identify coalitions in social networks (Aref and Neal 2020). A compact formulation of the problem asks for the vertex bipartization minimizing the number of $=$-edges connecting vertices in different partitions and $\neq$-edges incident to vertices in the same partition:

$$\min \sum_{\{i,j\}\in E_=} (z_i \oplus z_j) + \sum_{\{i,j\}\in E_\neq} (1 - (z_i \oplus z_j)) \tag{3.7}$$

$$z \in \{0,1\}^{|V|} \tag{3.8}$$

Transformation to MaxCut Transforming the above objective into a maximization problem results in the following.

$$\max \sum_{\{i,j\}\in E_=} -(z_i \oplus z_j) + \sum_{\{i,j\}\in E_\neq} (z_i \oplus z_j) - |E_\neq| \tag{3.9}$$

$$z \in \{0,1\}^{|V|} \tag{3.10}$$

Therefore, BalancedSubgraph for a signed input graph $G = (V, E)$ is equivalent to MaxCut on a graph $G' = (V, E)$ with the same vertices and edges, but edge weights w_e of 1 for $e \in E_\neq$ and -1 for every $e \in E_=$. The transformation results in a constant objective offset of $-|E_\neq|$.

SignedMaxCut Very similar to BalancedSubgraph is the SignedMaxCut problem (Ferizovic et al. 2020) (they mainly differ in their optimization sense). Here, the edges of an input graph $G = (V, E)$ are assigned to the $+$ set (E_+), or $-$ set (E_-). The corresponding optimization problem is the following.

$$\max \sum_{\{i,j\}\in E_-} (z_i \oplus z_j) + \sum_{\{i,j\}\in E_+} (1 - (z_i \oplus z_j)) \tag{3.11}$$

$$z \in \{0,1\}^{|V|} \tag{3.12}$$

Transformation to MaxCut A simple transformation of the objective results in:

$$\max \sum_{\{i,j\}\in E_-} (z_i \oplus z_j) - \sum_{\{i,j\}\in E_+} (z_i \oplus z_j) + |E_+| \tag{3.13}$$

$$z \in \{0,1\}^{|V|} \tag{3.14}$$

Clearly, the transformation is similar to that for BalancedSubgraph. SignedMaxCut for an input graph $G = (V, E)$ is equivalent to MaxCut on a graph $G' = (V, E)$ with the same vertices and edges, but edge weights w_e of 1 for $e \in E_-$ and -1 for every $e \in E_+$. The transformation results in a constant objective offset of $|E_+|$.

3.2.3. Ising Spin Glasses

Ising spin glasses (Binder and Young 1986) come from statistical physics and their *ground states* are of particular interest. They are defined on a graph $G = (V, E)$, where each vertex gets assigned a *spin*. There are two types of spin, up and down, modeled by one variable s_i per vertex with $s_i \in \{-1, 1\}$. Spins connected by an edge interact with *coupling strength* J_{ij}. The objective function (without an external magnetic field), called Hamiltonian in the physics community, is:

$$\min H(\omega) = -\sum_{\{i,j\}\in E} J_{ij} s_i s_j$$

The explicit negative signs in the objective allow for a straightforward transformation into a maximization problem:

$$\max H(\omega) = \sum_{i,j} J_{ij} s_i s_j$$

Transformation to MaxCut Clearly, if two neighboring vertices have the same spin, their coupling strength contributes positively to the objective and negatively if they have different spins. We therefore can rewrite the problem as follows:

$$\max \sum_{i,j} J_{ij}(1 - (z_i \oplus z_j)) - \sum_{i,j} J_{ij}(z_i \oplus z_j) \tag{3.15}$$

$$z \in \{0,1\}^n \tag{3.16}$$

Or more simplified:

$$\max \sum_{i,j} -2 \cdot J_{ij}(z_i \oplus z_j) + \sum_{i,j} J_{ij} \tag{3.17}$$

$$z \in \{0,1\}^n \tag{3.18}$$

Therefore the problem of finding the ground state of an ising spin glass on a graph $G = (V, E)$ with coupling strengths J_{ij} can be transformed into MaxCut on the same graph with edge weights $w_{ij} = -2 \cdot J_{ij}$ and an offset of $\sum_{i,j} J_{ij}$. Note: When $h \neq 0$ the transformation to MaxCut still works, but the introduction of an additional vertex is necessary. We omit the details here.

3.2.4. QUBO

The quadratic unconstrained binary optimization problem (QUBO) asks for the optimal $\{0,1\}$ assignment to n binary variables with a quadratic objective function of the form $\max x^T Q x$. QUBO models have been applied in many different contexts, as highlighted in the survey by Kochenberger et al. (2014). QUBOs in standard form are sometimes defined as minimization problems and sometimes as a maximization problem. We opt for minimization problems here:

$$\min \sum_{ij} x_i x_j q_{ij} \tag{3.19}$$

$$x \in \{0,1\}^n \tag{3.20}$$

MaxCut as a mathematical program with logical objective only requires an xor. QUBO has an and-only objective:

$$\min \sum_{ij} (x_i \wedge x_j) q_{ij} \tag{3.21}$$

$$x \in \{0,1\}^n \tag{3.22}$$

Transformation to MaxCut Transforming a QUBO defined on n variables to a MaxCut problem results in a graph G with $n+1$ vertices. Each vertex $v_i \in \{v_1, \dots v_{n+1}\}$ of G is in one-to-one correspondence with the variable x_i QUBO, except the special vertex v_{n+1}. We call this special vertex r and fix the partition of its variable to 0. We also assume w.l.o.g. that this vertex is connected to all other vertices u with weight c_{ir}. Let $E_{\bar{r}}$ be the set of all edges of G not incident to r. The corresponding MaxCut objective can be written as follows:

$$\max \sum_{\{i,j\} \in E_{\bar{r}}} (z_i + z_j - 2 z_i z_j) w_{ij} + \sum_{i \in V \setminus r} z_i \cdot w_{ir} \tag{3.23}$$

$$\iff \max \sum_{\{i,j\} \in E_{\bar{r}}} -2 z_i z_j w_{ij} + \sum_{i \in V \setminus r} z_i (w_{ir} + \sum_{\{i,j\} \in E_{\bar{r}}} w_{ij}) \tag{3.24}$$

Assuming w.l.o.g. the matrix Q to only have entries in the upper triangle and slightly transforming the objective of QUBO results in:

$$\min \sum_{ij} x_i x_j q_{ij} \tag{3.25}$$

$$\iff \min \sum_{i<j} x_i x_j q_{ij} + \sum_i x_i q_{ii} \tag{3.26}$$

$$\iff \max \sum_{i<j} -x_i x_j q_{ij} + \sum_i -x_i q_{ii} \tag{3.27}$$

The objective functions 3.24 and 3.27 already look quite similar and can be transformed into each other by simple substitution. Applying a one-to-one mapping of $x_i = z_i$ we directly get $c_{ij} = q_{ij}/2$. With this $c_{ir} = -q_{ii} - \sum_{i<j} 1/2 \cdot q_{ij}$ follows. Note: We transformed QUBO into a maximization problem by multiplying the objective by -1, therefore all (optimal) solutions for the original problem and the MaxCut version differ in their sign. For the transformation in the other direction (MaxCut to QUBO) see (Barahona et al. 1989).

3.3. State-of-the-Art MaxCut Solvers

Motivated by the many applications of MaxCut, a whole family of exact MaxCut solvers has been developed over the years. All MaxCut solvers (sometimes tailored to specific types of graphs) that have been developed in recent years are based on branch and bound. We list them in order of their first public appearance:

- Biq Mac (Rendl et al. 2010) makes use of semidefinite and polyhedral relaxations for bounding. The solver is closed source, but is available as a webservice `https://biqmac.aau.at/`.
- BiqCrunch (Krislock et al. 2017) uses semidefinite programming for the bounding procedure. The solver is open source and available as a webservice `https://biqcrunch.lipn.univ-paris13.fr/`.
- BiqBin (Gusmeroli et al. 2022) incorporates semidefinite programming relaxations and an enhanced version of BiqMac. The solver is open source and available as a webservice `http://biqbin.eu/`.
- MADAM (Hrga and Povh 2021) builds on the ideas of BiqBin. The source code is available online `https://github.com/HrgaT/MADAM`.
- McSparse (Charfreitag et al. 2022) is based on integer linear programming and branch and cut. The solver is closed source, but is available as a webservice `https://mcsparse.uni-bonn.de/`.

- QuBowl (Rehfeldt et al. 2023) is also based on branch and cut. The solver is closed source.

Some of these solvers are featured in the benchmarks of Hans Mittelmann for QUBO instances (`https://plato.asu.edu/ftp/qubo.html`) of the QPLIB (Furini et al. 2019). Of the tested solvers and for the considered instances, QuBowl is currently the fastest.

McSparse: A new MaxCut Solver as a Webservice

Before 2022 no state-of-the-art integer linear programming MaxCut solver was available to the general public and the research community. To fill this gap, we created

`mcsparse.uni-bonn.de`

The website serves as a frontend for the McSparse solver, which was introduced by us (Charfreitag et al. 2022). Users can submit MaxCut instances in `mc` format or QUBO instances in `bq` format (see Appendix A.1 for details on the file formats used). To also offer researchers and practitioners interested in spin glasses a simple interface, we also added a special version of McSparse, called McGroundstate: `http://mcsparse.uni-bonn.de/mcgroundstate`. This solver accepts input in `sg` and `gsg` format (again see Appendix A.1). Table 3.1 summarizes some statistics on the usage of our two solvers. Its broad acceptance in the community is also underlined by its appearance in e.g. (King et al. 2023) and (Tarabunga and Castelnovo 2024).

Solver	unique e-mails	instances submitted	opt solution found
McSparse	24	633	283
McGroundstate	39	44 710	44 222

Table 3.1.: Usage statistics for our online solver (collected in October of 2024). A user submission may contain multiple files and we report the total number of submitted files and the total number of files for which our solver calculated the optimal solution before reaching the timelimit of 30 minutes. The McSparse row reports MaxCut submissions and McGroundstate Ising spin glass submissions.

3.4. MaxCut Benchmark Instances

Depending on their main application, different MaxCut solvers and algorithms have been benchmarked on many different inputs over the years. We collected a representative set of sparse instances that will be the focus of our experiments later. This section describes the origin of our benchmark instances and their characteristics.

Most instances considered in the literature are part of at least one publicly available library. For MaxCut specific instances, we are aware of libraries 1., 2., 4. and 5. of the following list. 3. contains many graphs from a variety of different real-world applications.

1. The library of the 7th DIMACS implementation challenge (2000):
   `http://dimacs.rutgers.edu/archive/Challenges/Seventh/Instances/`
2. The Biq Mac Library (Wiegele 2007):
   `https://biqmac.aau.at/biqmaclib.html`
3. The Network Repository (Rossi and Ahmed 2015):
   `https://networkrepository.com/`
4. MQLib (Dunning et al. 2018):
   `https://github.com/MQLib/MQLib`
5. The MaxCut and BQP Instance Library (Mallach 2021):
   `http://bqp.cs.uni-bonn.de/library/html/index.html`

Instances from these libraries can be divided into two main groups: Real-world instances and instances that were artificially generated. We describe our selection and the structure of the graphs next.

Generated Instances

Often authors not only benchmark their solver on real-world graphs but also consider generated instances. For MaxCut three types of generated instances have been used for benchmarking especially often and are part of the Biq Mac Library. The first type are the so-called Erdős–Rényi random graphs (Erdős and Rényi 1960). As we will see later, they turn out to be hard to solve to proven optimality, even for relatively few vertices and edges. For Erdős–Rényi random graphs every edge exists with equal probability. The graphs are parameterized in their number of vertices and edges. The other two types of graphs are certain torus graphs and so-called ising chains, of interest in the context of statistical physics (Liers 2004; Bonato et al. 2014). Table 3.2 and Table 3.3 summarize the characteristics of these instances. Details of their generation are discussed below.

- t2g and t3g: All 18 torodial square grid instances from the Biq Mac Library. The weights are drawn from a Gaussian distribution, multiplied by 10^5 and rounded to the nearest integer; see (Bonato et al. 2014) for details.
- t2pm and t3pm: 13 torodial square grid instances; 10 2d instances of size 70×70 and 3 3d instances of size $3 \times 3 \times 3$, with all weights $\in \{-1, 1\}$ and an equal number of each, as described by Bonato et al. (2014). All instances were generated by us, as instances of this type are considered regularly (Bonato et al. 2014; Nguyen and Minoux 2021), but not publicly available.
- ising: All 30 ising chain instances from the Biq Mac Library. These instances consist of fully connected graphs, where the number of vertices is in $\{100, 150, \ldots, 300\}$. To calculate the edge weights, all vertices are placed equally spaced on a cycle and the weight of an edge depends linearly on the distance of its incident vertices and a random value drawn from a Gaussian distribution, see (Liers 2004) for details.

- pm1s: All 10 graphs from the Biq Mac Library. Erdös-Renyi random graphs with $|V| = 100$, $|E| = 495$ and integer edge weights drawn at random from $\{-1, 1\}$.
- pw01: All 10 graphs from the Biq Mac Library. Same as pm1s, but edge weights are from $[-10, 10]$.
- w01: All 10 graphs from the Biq Mac Library. Same as pm1s, but edge weights are from $[1, 10]$.

Real-World Instances

Table 3.4 shows all the real-world instances that we added to our benchmarks set. We again explain the four sets of instances in more detail.

- mannino: 4 instances stemming from a frequency assignment problem first introduced in work by Bonato et al. (2014) available from the MaxCut and BQP Instance Library (Mallach 2021).
- easy: 19 instances first considered by Ferizovic et al. (2020). All of them can be solved fast by state-of-the-art solvers, as we will see later. All instances stem either from the Network Repository or the MQLib.
- medium: 7 instances from the same sources and of similar structure as the easy instances (MQLib and Network Repository). We selected these instances to fill the gap between the easy instances and the big instances (see next bullet point); the largest easy instance has about 6k vertices and the smallest big instance close to 300k.
- big: 5 instances from the network repository, also considered by Ferizovic et al. (2020) as part of their *large* instance set.

QUBO Instances

This work focuses on solving MaxCut on sparse graphs. Nevertheless, we include a small and diverse subset of well-known quadratic unconstrained binary optimization problems from the Biq Mac Library (Wiegele 2007). Table 3.5 introduces these QUBO instances. The gka instances were originally suggested by Glover et al. (1998), the bqp instances by Beasley (1998), and the be instances by Billionnet and Elloumi (2007). See Section 3.2.4 for their transformation to MaxCut.

set	instance	$\|V\|$	$\|E\|$	$\underline{d}$	$\overline{d}$	$\underline{w}$	$\overline{w}$
pm1s	pm1s_100_0	100	495	3	15	-1	1
pm1s	pm1s_100_1	100	495	3	20	-1	1
pm1s	pm1s_100_2	100	495	1	18	-1	1
pm1s	pm1s_100_3	100	495	3	18	-1	1
pm1s	pm1s_100_4	100	495	3	17	-1	1
pm1s	pm1s_100_5	100	495	2	17	-1	1
pm1s	pm1s_100_6	100	495	4	22	-1	1
pm1s	pm1s_100_7	100	495	4	18	-1	1
pm1s	pm1s_100_8	100	495	3	17	-1	1
pm1s	pm1s_100_9	100	495	2	23	-1	1
pw01	pw01_100_0	100	495	3	15	1	10
pw01	pw01_100_1	100	495	3	20	1	10
pw01	pw01_100_2	100	495	1	18	1	10
pw01	pw01_100_3	100	495	3	18	1	10
pw01	pw01_100_4	100	495	3	17	1	10
pw01	pw01_100_5	100	495	2	17	1	10
pw01	pw01_100_6	100	495	4	22	1	10
pw01	pw01_100_7	100	495	4	18	1	10
pw01	pw01_100_8	100	495	3	17	1	10
pw01	pw01_100_9	100	495	2	23	1	10
w01	w01_100_0	100	466	3	15	-10	10
w01	w01_100_1	100	468	3	20	-10	10
w01	w01_100_2	100	460	1	17	-10	10
w01	w01_100_3	100	475	3	18	-10	10
w01	w01_100_4	100	464	3	17	-10	10
w01	w01_100_5	100	473	2	17	-10	10
w01	w01_100_6	100	474	4	19	-10	10
w01	w01_100_7	100	476	4	17	-10	10
w01	w01_100_8	100	473	3	17	-10	10
w01	w01_100_9	100	475	2	23	-10	10

Table 3.2.: Our benchmark instances from the Biq Mac Library with Erdös-Renyi structure. The columns $\underline{d}$ and $\overline{d}$ capture the min and max degree, $\underline{w}$ and $\overline{w}$ the min and max edge weight respectively.

set	instance	$\|V\|$	$\|E\|$	$\underline{d}$	$\overline{d}$	$\underline{w}$	$\overline{w}$
t2g	t2g10_5555	100	200	4	4	-294 541	290 339
t2g	t2g10_6666	100	200	4	4	-239 344	238 268
t2g	t2g10_7777	100	200	4	4	-238 936	301 004
t2g	t2g15_5555	225	450	4	4	-294 541	290 339
t2g	t2g15_6666	225	450	4	4	-240 195	268 055
t2g	t2g15_7777	225	450	4	4	-247 819	375 001
t2g	t2g20_5555	400	800	4	4	-294 541	308 059
t2g	t2g20_6666	400	800	4	4	-271 149	315 291
t2g	t2g20_7777	400	800	4	4	-288 410	375 001
t3g	t3g5_5555	125	375	6	6	-294 541	290 339
t3g	t3g5_6666	125	375	6	6	-240 195	268 055
t3g	t3g5_7777	125	375	6	6	-238 936	375 001
t3g	t3g6_5555	216	648	6	6	-294 541	308 059
t3g	t3g6_6666	216	648	6	6	-265 601	271 240
t3g	t3g6_7777	216	648	6	6	-288 410	375 001
t3g	t3g7_5555	343	1 029	6	6	-294 541	308 059
t3g	t3g7_6666	343	1 029	6	6	-271 149	315 291
t3g	t3g7_7777	343	1 029	6	6	-298 103	375 001
t2pm	t2pm_70_*	4 900	9 800	4	4	-1	1
t3pm	t3pm_7_*	343	1 029	6	6	-1	1
ising	ising25-100_5555	100	4 950	99	99	-151 693	239 752
ising	ising25-100_6666	100	4 950	99	99	-212 231	170 713
ising	ising25-100_7777	100	4 950	99	99	-181 596	220 276
ising	ising30-100_5555	100	4 950	99	99	-153 151	242 057
ising	ising30-100_6666	100	4 950	99	99	-214 271	172 355
ising	ising30-100_7777	100	4 950	99	99	-183 342	222 394
ising	ising25-150_5555	150	11 175	149	149	-172 274	167 990
ising	ising25-150_6666	150	11 175	149	149	-219 729	176 376
ising	ising25-150_7777	150	11 175	149	149	-192 807	182 372
ising	ising30-150_5555	150	11 175	149	149	-173 927	169 602
ising	ising30-150_6666	150	11 175	149	149	-221 838	178 068
ising	ising30-150_7777	150	11 175	149	149	-194 657	184 121
ising	ising25-200_5555	200	19 900	199	199	-211 364	239 764
ising	ising25-200_6666	200	19 900	199	199	-231 636	190 865
ising	ising25-200_7777	200	19 900	199	199	-162 372	206 896
ising	ising30-200_5555	200	19 900	199	199	-213 390	242 063
ising	ising30-200_6666	200	19 900	199	199	-233 857	192 695
ising	ising30-200_7777	200	19 900	199	199	-163 929	208 879
ising	ising25-250_5555	250	31 125	249	249	-174 893	277 406
ising	ising25-250_6666	250	31 125	249	249	-193 801	194 985
ising	ising25-250_7777	250	31 125	249	249	-183 293	174 714
ising	ising30-250_5555	250	31 125	249	249	-176 569	280 065
ising	ising30-250_6666	250	31 125	249	249	-195 658	196 854
ising	ising30-250_7777	250	31 125	249	249	-185 050	176 389
ising	ising25-300_5555	300	44 850	299	299	-172 591	201 189
ising	ising25-300_6666	300	44 850	299	299	-244 104	192 991
ising	ising25-300_7777	300	44 850	299	299	-218 021	210 331
ising	ising30-300_5555	300	44 850	299	299	-174 244	203 116
ising	ising30-300_6666	300	44 850	299	299	-246 443	194 841
ising	ising30-300_7777	300	44 850	299	299	-220 110	212 347

Table 3.3.: Our benchmark instances from statistical physics. The columns $\underline{d}$ and $\overline{d}$ capture the min and max degree, $\underline{w}$ and $\overline{w}$ the min and max edge weight respectively.
*The t2pm and t3pm graphs all share the exact same metric, hence we aggregated the rows.

set	instance	$\|V\|$	$\|E\|$	$\underline{d}$	$\overline{d}$	$\underline{w}$	$\overline{w}$
easy	soc-firm-hi-tech	33	91	1	16	1	1
easy	g001207	84	149	1	5	1	100 000
easy	g000981	110	188	2	6	1	100 000
easy	ENZYMES_g295	123	139	1	5	1	1
easy	g000292	212	381	2	4	5	13
easy	g000302	317	476	1	4	5	13
easy	ca-netscience	379	914	1	34	1	1
easy	bio-diseasome	516	1 188	1	50	1	1
easy	rt-twitter-copen	761	1 029	1	37	1	1
easy	g001918	777	1 239	1	4	5	13
easy	imgseg_271031	900	1 027	1	518	93 839 059	285 968 046 836
easy	imgseg_35058	1 274	1 806	1	587	-55 510 850 118	112 271 093 673
easy	bio-yeast	1 458	1 948	1	56	1	1
easy	imgseg_106025	1 565	2 629	1	902	93 981 365	136 834 528 589
easy	ca-CSphd	1 882	1 740	1	46	1	1
easy	ego-facebook	2 888	2 981	1	769	1	1
easy	imgseg_105019	3 548	4 325	1	2 753	109 623 218	236 593 516 427
easy	imgseg_374020	5 735	8 722	1	2 213	-46 639 208 299	407 957 172 555
medium	web-google	1 299	2 773	1	59	1	1
medium	inf-power	4 941	6 594	1	19	1	1
medium	ca-Erdos992	5 094	7 515	1	61	1	1
medium	imgseg_138032	12 736	23 664	1	2 204	-316 609 100	242 466 687 931
medium	g000677	17 127	27 352	1	4	1	126
medium	g001075	27 019	39 407	1	4	1	228 668
medium	imgseg_147062	28 552	65 453	1	925	-1 567 963 186	67 209 950 110
medium	g000087	38 418	71 657	2	4	1	198
medium	road-luxembourg-osm	114 599	119 666	1	6	1	1
big	web-Stanford	281 903	1 992 636	1	38 625	1	1
big	ca-MathSciNet	332 689	820 644	1	496	1	1
big	web-it-2004	509 338	7 178 413	1	469	1	1
big	ca-coauthors-dblp	540 486	15 245 729	1	3 299	1	1
big	ca-IMDB	896 305	3 782 447	1	1 590	1	1
mannino	mannino_k48	48	1 128	47	47	13 146	841 699
mannino	mannino_k487a	487	1 435	0	52	101	33 631
mannino	mannino_k487b	487	5 391	2	109	5	176 030
mannino	mannino_k487c	487	8 511	3	140	5	203 785

Table 3.4.: Our collection of benchmark instances of real-world structure. The columns $\underline{d}$ and $\overline{d}$ capture the min and max degree, $\underline{w}$ and $\overline{w}$ the min and max edge weight respectively.

set	instance	$\|V\|$	$\|E\|$	$\underline{d}$	$\overline{d}$	$\underline{w}$	$\overline{w}$
qubo	gka7a	31	241	10	30	-734	976
qubo	gka2c	51	813	26	50	-892	852
qubo	gka5c	81	721	11	80	-926	504
qubo	gka4d	101	2100	31	100	-858	886
qubo	be120_3_5	121	2248	26	120	-912	1 024
qubo	be250_3	251	3279	12	250	-950	946
qubo	bqp250-3	251	3313	12	250	-1 460	1 368

Table 3.5.: Our collection of QUBO benchmark instances. All values refer to the graph resulting from the transformation of QUBO to MaxCut. The columns $\underline{d}$ and $\overline{d}$ capture the min and max degree, $\underline{w}$ and $\overline{w}$ the min and max edge weight respectively.

4. Presolving

Presolving, also called preprocessing, describes the algorithmic process of simplifying or strengthening a problem description before handing it over to a solver. Numerous experimental studies have shown solvers benefiting significantly from carefully engineered preprocessing, see (Gamrath et al. 2015; Achterberg et al. 2020; Gleixner et al. 2023) for MIP-solving or (Abu-Khzam et al. 2022) and references therein for purely combinatorial approaches. Preprocessing often consists of two different components: Data reduction and decomposition.

Decomposition divides the problem into smaller components that can be solved independently of each other. Data reduction is the process of taking the input of an algorithmic problem and applying so-called *reduction rules* with the aim of reducing the size of the input. These reductions preserve optimal solutions and hopefully speed up the solver.

In this chapter we review presolving techniques for the MaxCut problem from the literature and introduce new ones. Most of the novel concepts in this chapter were first published in a peer-reviewed paper in 2024[1]. We start of with decomposition techniques before moving on to data reduction rules. Afterwards, we design a full presolving framework, that incorporates all presented techniques. Finally, we perform an experimental study to evaluate the framework.

4.1. Decomposition

One relevant aspect of preprocessing is decomposition. If the input graph G for a MaxCut problem consists of multiple connected components, each can be solved independently. Calculating an optimal bipartition on one connected component has no side effects on others. Furthermore, as is characteristic for combinatorial optimization problems on graphs (Hochbaum 1993), the input can also be split into its biconnected components, which again can be solved independently, and the partial solutions can be merged into a solution for the complete input efficiently (Grötschel and Nemhauser 1984). For MaxCut the process of calculating the optimal solution to the original problem takes linear time and only requires the block-cut tree of the input graph. After calculating the optimal solution for each block (biconnected component), start at any block and traverse the tree in a DFS fashion (or BFS etc.) and build the final solution

[1] Jonas Charfreitag, Christine Dahn, Michael Kaibel, Philip Mayer, Petra Mutzel, and Lukas Schürmann (2024a). "Separator Based Data Reduction for the Maximum Cut Problem". In: *22nd International Symposium on Experimental Algorithms, SEA 2024, July 23-26, 2024, Vienna, Austria*. Ed. by Leo Liberti. Vol. 301. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 4:1–4:21. DOI: 10.4230/LIPICS.SEA.2024.4.

by flipping the blocks solution if needed. Decomposition into (bi-)connected components has not only been discussed in theory, but has also been used for practical MaxCut solving in the past; see e.g. (Rehfeldt et al. 2023).

4.2. Data Reduction

In general, data reduction algorithms for MaxCut employ different data reduction rules to shrink the input graph by reducing the number of vertices and / or edges. The overall goal is to make the graph easier to handle for heuristics or exact solvers. The concept of data reduction stems from research on parameterized complexity (Flum and Grohe 2006). Formally, we define:

Definition 4.2.1 (Valid Data Reduction)**.** *Let $G = (V, E, w)$ be a weighted undirected connected graph. A data reduction rule, which transforms G into G' is valid, if*
1) every optimal solution in G', can be mapped to an optimal solutions in G, where the values of the solutions differ by a constant offset β.
2) the rule can be applied in polynomial time and β can be calculated in polynomial time.

Note that the above Definition 4.2.1 does not include the process of finding candidates for a specific data reduction rule. We call a data reduction rule *feasible*, if it is valid and candidates can be found in polynomial time. We will summarize data reduction rules from the literature in the following. They can be grouped into different categories and we will introduce new data reduction rules for each one. The general structure of their proofs is often quite similar and we suggest to view them through the lens of so-called *improving mappings*, introduced by Shekhovtsov (2014). For simplicity, we narrow their scope to MaxCut:

Definition 4.2.2 (Improving Mapping)**.** *Let $G = (V, E, w)$ be a weighted input graph for* MaxCut*. A mapping $f : \{0,1\}^{|V|} \to \{0,1\}^{|V|}$ from* MaxCut *solutions (bipartitions) to* MaxCut *solutions for G is called improving if for every solution s it holds, that $\Delta(G, s) \leq \Delta(G, f(s))$.*

As Lange et al. (2019) describe, there are two types of trivial improving mappings for MaxCut: The identity that maps any solution to itself, with the same objective value, and the optimal mapping, which maps any solution to the optimal solution. The first has no benefit at all; the second is in general NP-hard to compute. The interesting mappings fall between these two and form the basis for data reduction rules; the existence of certain improving mappings allows for the removal of vertices or edges in the input graph.

One example of this is, if we find an improving mapping that always assigns two vertices (say u and v) to the same partition. In this case, we can limit the search space for finding an optimal MaxCut solution for the input graph G, by discarding all solutions with u and v in different partitions. This shrinkage of the search space can be encoded in G by transforming it into G' via the contraction of the two vertices u and v. Also, the existence of such an improving mapping implies, that there is at least one optimal solution in G with u and v in the same partition.

Proposition 4.2.3. *Let $G = (V, E, w)$ be the input of a* MaxCut *problem. If for two vertices $u, v \in V$ we can prove the existence of an improving mapping that always assigns u and v to the same partition, the contraction of u and v is a valid data reduction, with $\beta = 0$.*

Proof. The data reduction is valid, because
1) Every solution in G' can be mapped to a solution in G with the exact same solution value and there is at least one optimal solution in G' with value $\Delta(G)$.
2) The contraction can be performed in polynomial time and β is always 0. □

If an optimal solution with u and v in different partitions is guaranteed to exist, we need an intermediate step to make use of this in a data reduction rule.

Proposition 4.2.4. *Let G be an undirected weighted graph $G = (V, E, w)$ and $\delta' \subseteq E$ some cut in G. $G' = (V, E, w')$ is the graph resulting from G by negating the weights of all edges in δ'. The transformation from G to G' is valid with offset $\beta = \sum_{e \in \delta'} w_e$.*

Proof. G and G' only differ in some edge weights and any cut in G is a cut in G' and vice versa. The symmetric difference between all cuts in G and δ' is a bijection from cuts in G to cuts in G'. This bijection maps any cut with value c in G to one in G' with the exact same value (because of the definition of β). Therefore, $\Delta(G) = \Delta(G') + \beta$ and the transformation is valid. □

By combining Proposition 4.2.4 and Proposition 4.2.3 we get a general reduction technique when an improving mapping with u and v in different partitions exists:

Proposition 4.2.5. *Let $G = (V, E, w)$ be the input of a* MaxCut *problem. If for two vertices $u, v \in V$ we can prove the existence of an improving mapping that always assigns u and v to different partitions, negating all edge weights of edges incident to u and contracting u and v afterward is a valid data reduction with $\beta = \sum_{e \in \delta(u)} w_e$.*

Types of MaxCut Data Reduction We categorize MaxCut valid data reduction rules based on the type of graph structure they are working with. Some rules require two vertices to have similar neighborhoods, some rules consider certain edge separators of the input and some work on vertex separators.

Additionally, we call a data reduction rule *weight stable*, if applying the rule does not introduce new weights. This characteristic can be useful for the interplay between certain rules. Some rules require certain subgraphs to only have edges with the same weight, therefore it might be beneficial to apply these rules first to a graph where all edges have the same weight, before applying other rules that introduce new weights.

4.2.1. Similar Neighborhood Vertices

Vertices with similar neighborhoods give rise to the first category of data reduction rules that we are introducing. The first rule for MaxCut of this type was presented by Rehfeldt et al. (2023) and is discussed next.

Proposition 4.2.6 (Similar Vertices)**.** *Let $G = (V, E, w)$. If for two vertices $u, v \in V$ $N(u)\backslash\{v\} = N(v)\backslash\{u\}$ there exists an $\alpha \geq 1$ or $\alpha \leq -1$ with $w_{ux} = \alpha w_{vx} \forall x \in N(u)\backslash\{v\}$, then, if*

- *$\alpha > 0$ and ($\{u, v\} \notin E$ or $w_{uv} \leq 0$), there is at least one optimal cut with u and v in the same partition*
- *$\alpha < 0$ and ($\{u, v\} \notin E$ or $w_{uv} \geq 0$), there is at least one optimal cut with u and v in different partitions*

Proof. Consider the case for $\alpha > 0$ (for $\alpha < 0$ the proof is symmetric). If $\{u, v\} \in \delta$ for some cut δ in G let w.l.o.g. $\sum_{e \in \delta \cap N(u)\backslash\{u,v\}} w_e \geq \alpha \sum_{e \in \delta \cap N(v)\backslash\{u,v\}} w_e$. The assignment of v to the same partition as u is improving, because $w_{uv} \leq 0$ and $w_{ux} = \alpha w_{vx} \forall x \in N(u) \setminus \{v\}$. □

Next we suggest a new rule, also based on two vertices with a similar neighborhood. The rule is applicable in cases not covered by Proposition 4.2.6 of Rehfeldt et al. (2023).

Proposition 4.2.7 (Twin Vertices)**.** *Let $G = (V, E, w)$. If for an edge $e = \{u, v\} \in E$ all of the following conditions hold*

- $N(u) \cup \{u\} = N(v) \cup \{v\}$
- $0 \leq w_{ux} = w_{vx} \ \forall x \in N(u) \setminus \{v\}$
- $w_{uv} \leq w_{vx} \ \forall x \in N(u) \setminus \{v\}$
- $d(u) \mod 2 = 0$

there is an optimal solution with u and v in the same partition.

Proof. All cases with $w_{uv} \leq 0$ are already covered by Proposition 4.2.6. We show that for the remaining cases there also exists an improving mapping, which always assigns u and v to the same partition. If $\{u, v\} \in \delta$ let w.l.o.g. u be the vertex, whose incident edges contribute more (or equal) to the cuts value $\sum_{e \in \delta \cap N(u)} w_e \geq \sum_{e \in \delta \cap N(v)} w_e$. Because $|N(u) \setminus v| = |N(v) \setminus u|$ is odd, the cut value contribution from u is at least w_{uv} higher than of v. Therefore assigning v to the same partition as u is always (not necessarily strictly) improving. □

For finding candidates for Proposition 4.2.7, Rehfeldt et al. (2023) suggest the same hashing techniques that MIP-solvers employ to find parallel rows (Achterberg et al. 2020). Clearly, the same applies for 4.2.6. To possibly achieve better performance in practice, the cases for Proposition 4.2.7, in which the two vertices u and v are adjacent, can also be covered without hashing, by looping over all edges once. Figure 4.1 shows two graphs on which the rules for vertices with similar neighborhood can be applied.

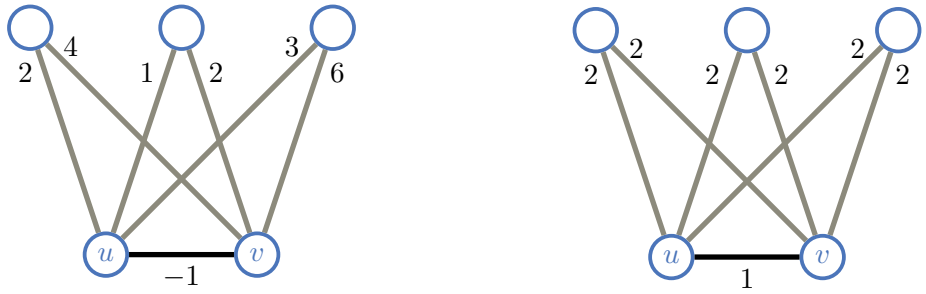


Figure 4.1.: Example structures, where the data reduction rules for similar vertices (left) and twin vertices (right) can be applied. In both cases an optimal solution with u and v in the same partition is guaranteed to exist.

4.2.2. Edge Separator based Data Reduction

Some data reduction rules for MaxCut (at least implicitly) work on edges that form an edge separator (which is the same as a cut) in G. The following rule of Lange et al. (2019) is of this type:

Proposition 4.2.8 (Dominating Edge)**.**
Let $G = (V, E, w)$. If for any edge $e = \{u, v\} \in E$ and a subset $U \subset V$ with $e \in \delta(U)$ and $u \in U$ the inequality

$$|w_e| \geq \sum_{e' \in \delta(U) \setminus \{e\}} |w_{e'}| \tag{4.1}$$

holds, then there exists a cut $\hat{\delta}$ with maximum value with $e \notin \hat{\delta}$ if $w_e \leq 0$ and $e \in \hat{\delta}$ if $w_e \geq 0$.

Proof. Consider the case of $w_e < 0$ and let δ' be any cut with $e \in \delta'$. Assigning all vertices in U to the opposite partition they are currently assigned to worsens the cuts value by at most $\sum_{e' \in \delta(U) \setminus \{e\}} |w_{e'}| - w_e$. If inequality 4.1 holds, this implies an improving mapping. For when $w_e \geq 0$ and $e \notin \delta'$ the construction of an improving mapping follows analogously. □

The fastest way to find candidates for this rule is to only consider the cases where $|U| = 1$. This leads to an $O(m)$ algorithm for finding all candidates consisting of a single vertex. To efficiently consider any set U, Gomory-Hu trees (Gomory and Hu 1961) can be used, although this is considerably slower, as their construction requires $|V| - 1$ many MaxFlow computations.

Lange et al. (2019) also present a data reduction rule for triangles. Their rule has two natural extensions. The first one was found by Rehfeldt et al. (2023) and the second was not covered in the literature before. This results in three data reduction rules for triangles, which we summarize in the following. Figure 4.2 visualizes the two types of rules.

Proposition 4.2.9 (Triangles)**.**
Let the edges $\{a,b\}$, $\{a,c\}$ and $\{b,c\}$ form a triangle in G. Additionally, let $U_1 \subset V$ such that $\{\{a,b\},\{a,c\}\} \subseteq \delta(U_1)$ and $U_2 \subset V$ such that $\{\{a,b\},\{b,c\}\} \subseteq \delta(U_2)$.

1) *If the two inequalities (introduced by Lange et al. 2019)*

$$-w_{ab} - w_{ac} \geq \sum_{e' \in \delta(U_1) \setminus \{\{a,b\},\{a,c\}\}} |w_{e'}|$$

$$-w_{ab} - w_{bc} \geq \sum_{e' \in \delta(U_2) \setminus \{\{a,b\},\{b,c\}\}} |w_{e'}|$$

hold, there exists a cut $\hat{\delta}$ with maximum value with $\{a,b\} \notin \hat{\delta}$.

2) *If the two inequalities (introduced by Rehfeldt et al. 2023)*

$$w_{ab} + w_{ac} \geq \sum_{e' \in \delta(U_1) \setminus \{\{a,b\},\{a,c\}\}} |w_{e'}|$$

$$w_{ab} - w_{bc} \geq \sum_{e' \in \delta(U_2) \setminus \{\{a,b\},\{b,c\}\}} |w_{e'}|$$

hold, there exists a cut $\hat{\delta}$ with maximum value with $\{a,b\} \in \hat{\delta}$.

3) *If the two inequalities (new)*

$$-w_{ab} + w_{ac} \geq \sum_{e' \in \delta(U_1) \setminus \{\{a,b\},\{a,c\}\}} |w_{e'}|$$

$$-w_{ab} + w_{bc} \geq \sum_{e' \in \delta(U_2) \setminus \{\{a,b\},\{b,c\}\}} |w_{e'}|$$

hold, there exists a cut $\hat{\delta}$ with maximum value with $\{a,b\} \notin \hat{\delta}$.

We sketch a proof for 1) (see Lange et al. 2019 for details) and show how 2) and 3) can be deduced from 1). For 2) Rehfeldt et al. (2023) already present a proof, but we suggest a more compact one based on Proposition 4.2.4, which also shows the correctness of our new implication, 3).

Proof. For proving 1): As $\{a,b,c\}$ forms a triangle in G, if $\{a,b\} \in \delta$ then either $\{a,c\} \in \delta$ or $\{b,c\} \in \delta$. If both inequalities hold, there is always an improving mapping which assigns a and b to the same partition.

Now 2) and 3) follow: If we apply the technique from Proposition 4.2.4 to G, by choosing $\delta^* = \delta(u)$ for the transformation, resulting in G' and 1) holds for G', we see that there is maximum cut $\hat{\delta}'$ in G' with $\{a,b\} \notin \hat{\delta}'$ and therefore, because of Proposition 4.2.4, a maximum cut $\hat{\delta}$ in G with $\{a,b\} \in \hat{\delta}$. But for 1) to hold in G', 2) needs to hold in G proving the correctness of 2). Following the same pattern, 3) can be derived from 2). Just choose $\delta^* = \delta(v)$ for the transformation from G' to G'' and 2) holds in G'' iff 3) holds in G. □

Figure 4.2.: Example structures, where the data reduction rules for dominating edges (left) and triangles (right) can be applied. In both cases an optimal solution with a and b in the same partition exists.

4.2.3. Vertex Separator based Data Reduction

Vertex separators (see Definition 2.3.1) have been used for data reduction in the past, by e.g. Polzin and Vahdati Daneshmand (2006) for the SteinerTree problem or by Hüffner et al. (2010) for the BalancedSubgraph problem. For MaxCut on the other hand, they have, to the best of our knowledge, not been exploited in practice so far. Still, they implicitly appear in some algorithmic proofs (Barahona 1983; Chimani et al. 2019) and existing rules from the literature can be generalized through the lens of vertex separators. We harmonize the results from the literature, derive a practical framework for data reduction based on vertex separators and introduce new rules based on the framework. All rules we consider in this section have similar structures, captured by the following definition.

Definition 4.2.10 (Data reduction rule type Separator)**.**
For a vertex separator S in $G = (V, E, w)$ separating $H \subset V$ from the rest of the graph, a rule is of type Separator, if it reduces G to $G' = G[V \setminus H]$, effectively deleting all vertices in H from G. All edges keep their original weights, apart from those in $G'[S]$.

What differentiates the rules of this type from the rules of the previous two sections is that they not only allow for the contraction of two vertices, but remove full sets of vertices (see Figure 4.3 for a visualization). Clearly, a rule of this type is only valid if no information relevant for finding optimal solutions is lost. This fact can be captured in a compact condition.

Theorem 4.2.11. *Let S be a vertex separator in $G = (V, E, w)$ separating $H \subset V$ from the rest of the graph, and $\mathcal{P}_S$ be the set of all possible bipartitions of vertices in S. Then a data reduction rule of type* Separator is valid for G if the system of equations

$$\sum_{e \in \delta_{G[S]}(P)} (w(e) + \gamma_e) + \beta = \Delta(G[H \cup S], P) \quad \forall P \in \mathcal{P}_S$$

has a feasible solution for the variables γ_e and β.

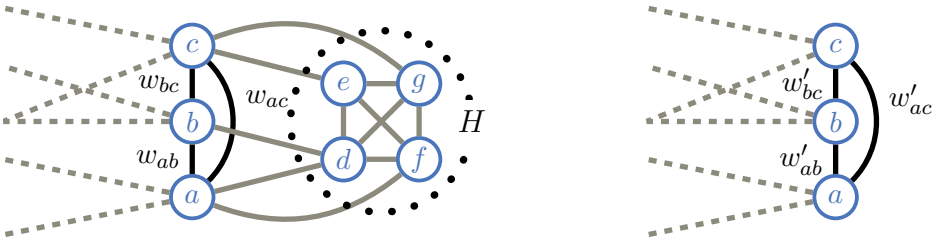


Figure 4.3.: Example of a reduction rule of type vertex separator. Left: Original graph G with vertex separator $S = \{a, b, c\}$ separating H from the rest of the graph. Right: The resulting reduced graph G' with adapted edge weights, i.e., H has been deleted from the remaining graph clearly reducing the size of the instance.

Proof. Let $G' = G[V \setminus H]$ and all edges in $G'[S]$ get their weight updated by adding the corresponding γ_e. This transformation fulfills Definition 4.2.1 and is a valid data reduction as:
1) For any bipartition $(U', \bar{U'})$ in G' with MaxCut value c' we can find a bipartition $(U, \bar{U})$ in G with a cut value of $c \geq c' + \beta$, because by construction of G' combining $(U', \bar{U'})$ with the optimal partitioning of the vertices in $V(G) \setminus V(G')$ we get one for G with a cut value of exactly $c' + \beta$.
2) For any bipartition $P = (U, \bar{U})$ in G with MaxCut value c we can make sure we can map it to a bipartition $(U', \bar{U'})$ in G' with a cut value $c' + \beta \geq c$, as again by construction of G' the bipartition resulting from removing all vertices from P which are not in G' results in a bipartition whose cut value in G' is at least as high as the one of P in G. □

Note that in the following we will assume the graph $G[S]$ to always be fully connected. This is a valid assumption, as one can always add missing edges with a weight of zero.

Small Vertex Separators Separators of size one allow to decompose the graph into independent components (Section 4.1). Other small vertex separators, more specifically those of size two and three, give rise to data reduction rules through Theorem 4.2.11, as we will show next.

Proposition 4.2.12. *Let $G = (V, E, w)$ be a graph and $S = \{a, b\}$ a 2-separator in G separating $H \subset V$ from the rest of the graph. For more compact notation define $\tilde{H} := H \cup S$. Then the data reduction rule of type* Separator *is valid with the following values: $\beta = \Delta(G[\tilde{H}], (\{a, b\}, \emptyset))$ and $\gamma_{ab} = \Delta(G[\tilde{H}], (\{a\}, \{b\})) - \Delta(G[\tilde{H}], (\{a, b\}, \emptyset)) - w_{ab}$.*

Proof. There are two possible bipartitions for vertices in S and $G[S]$ contains one edge $(e = \{a, b\})$. Therefore the equation system of Theorem 4.2.11 gives two equations and

two variables.

$$\beta = \Delta(G[\tilde{H}], (\{a, b\} : \emptyset)) \qquad w_{ab} + \gamma_{ab} + \beta = \Delta(G[\tilde{H}], (\{a\} : \{b\}))$$

Solving the system for β and γ_{ab} yields the stated equations. □

This general concept for vertex separators of size 2 covers some existing rules, like rule 2 and 6 of Ferizovic et al. (2020) as special cases. Their rule 6 operates on vertices with degree two and results in exactly the same reduction as the rule for dominating edges 4.2.8 for vertices with two neighbors. We restate their rule 2, as it is weight stable as opposed to the general rule for 2-separators (4.2.12).

Proposition 4.2.13 (3-paths)**.** *Let $G = (V, E, w)$ be a graph where all edges have a weight of 1 and the edges $\{v_1, v_2\}$, $\{v_2, v_3\}$ and $\{v_3, v_4\}$ form a path of length three and $\{v_1, v_4\} \notin E$ and $d(v_2) = d(v_3) = 2$. Creating G' by deleting v_2 and v_3 from G, and adding the edge $\{v_1, v_4\}$ is a valid* MaxCut *data reduction and $\Delta(G) = \Delta(G') + 2$*

Proof. As $d(v_2) = 2$ the rule for 2-separators is applicable. In the resulting graph $d(v_3)$ is still 2 and the 2-separator rule is applicable again, leading to the described reduction with $\beta = 2$. □

Vertex separators of size 3 also allow for a generalized data reduction rule:

Proposition 4.2.14. *Let $G = (V, E)$ be a graph and $S = \{a, b, c\}$ a 3-separator in G separating H from the rest of the graph. For more compact notation define $\tilde{H} \coloneqq H \cup S$. Then the data reduction rule of type* Separator is valid with the following constant offset β and values for γ:

$$\begin{aligned}
c_0 &:= \Delta(G[\tilde{H}], (\{a, b, c\}, \emptyset)) & \beta &= c_0 \\
c_1 &:= \Delta(G[\tilde{H}], (\{a, b\}, \{c\})) & \gamma_{ac} &= 1/2 \cdot (c_1 + c_3 - c_0 - c_2) - w_{ac} \\
c_2 &:= \Delta(G[\tilde{H}], (\{a, c\}, \{b\})) & \gamma_{ab} &= 1/2 \cdot (c_2 + c_3 - c_0 - c_1) - w_{ab} \\
c_3 &:= \Delta(G[\tilde{H}], (\{b, c\}, \{a\})) & \gamma_{bc} &= 1/2 \cdot (c_1 + c_2 - c_0 - c_3) - w_{bc}
\end{aligned}$$

Proof. There are four possible bipartitions for vertices in S and $G[S]$ contains three edges. Therefore, the equation system of Theorem 4.2.11 gives four equations and four variables. The constants $c_0, \ldots, c_{|\mathcal{P}|}$, introduced for better readability, represent the right-hand sides of the equation system (that is, the optimal MaxCut values for the corresponding partial bipartition). Solving this system for the constant offset β and the edge weights gives the presented equations. □

Vertex Separators of Arbitrary Size For vertex separators larger than three, the above techniques only work in specific cases, since the number of potential bipartitions exceeds the number of possible edges in $G[S]$, resulting in an overdetermined system of equations. For a 4-separator we already get eight partitions (resulting in eight equations), but only

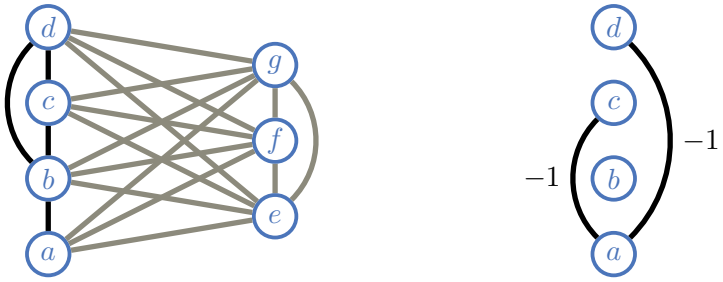


Figure 4.4.: Example of a same neighborhood clique reduction. Left: The original graph (all edges have a weight of one). The vertices $\{e, f, g\}$ form a clique and have all neighbors outside of the clique in common. Right: The graph after applying the data reduction rule.

seven variables (six for edges and β). Nevertheless for graphs with unit weights special rules have been described, for which their proof can be simplified through the lens of our framework and a new and more general one can be deduced. Ferizovic et al. (2020) present a number of data reduction rules based on cliques, namely their rules 1, 3, 4, 5 and 7. We discuss these rules in a moment, but we start with our generalization of their rules 1, 5, and 7. The rule operates on special cliques, as illustrated in Figure 4.4.

Proposition 4.2.15 (Same Neighborhood Clique)**.** *Let $G = (V, E, w)$. If for a clique $C \subset V$ in G, for which $w_e = 1$ for all edges e incident to at least one vertex u in C, $|C| + 1 \geq |N(C)| \geq 1$ and $N(C) = N(u) \setminus C$ $\forall u \in C$, removing vertices in C from the graph and updating weights of edges between vertices in $N(C)$ is a valid data reduction of type* Separator*.* The constant offset is $\beta = \lfloor(|N(C)| + |C|)/2\rfloor \cdot \lceil(|N(C)| + |C|)/2\rceil$. All weights of edges in $G[N(C)]$ get reduced by 1.

Proof. The vertex set $S := N(C)$ forms a vertex separator of G with C on one side. Recall, we assume w.l.o.g. $G[S]$ to be fully connected. Consider the equations from Theorem 4.2.11: Subtracting the weight of edges contributing to the value of the cut from both sides, leaves $\sum_{\delta_{G[S]}(P)}(\gamma_e) + \beta$ as the left-hand side. The resulting right-hand side $\Delta(G[S \cup C], P) - \sum_{\delta_{G[S]}(P)}(w_e)$ can be interpreted as the value of the maximum cut in G': $\Delta(G', P)$, where G' is $G[S \cup C]$, except all edges incident to two vertices in S have a weight of 0. Because of $|S| \leq |C| + 1$, no matter how vertices in $G'[S]$ get partitioned, $\Delta(G', P)$ can always be maximized by partitioning vertices in $S \cup C$ into two partitions, whose sizes differ by at most one. Therefore, for the partial bipartition $P = (S, \emptyset)$, the value of $\Delta(G', P)$ is $\beta = \lfloor(|S| + |C|)/2\rfloor \cdot \lceil(|S| + |C|)/2\rceil$. For general P this value has to be reduced by the number of edges cut in $G'[S]$, as they have a weight of 0 in G': $\beta - |\delta_{G[S]}(P)|$. With $\gamma_e = -1$ this is equal to the left-hand side of the equation $(\beta + \sum_{\delta_{G[S]}(P)}(-1))$. □

Candidates can be found fast exactly as described by Ferizovic et al. (2020) for their rules 5 and 7.

If $G[N(C)]$ is fully connected with weight 1 edges, then $G[C \cup N(C)]$ forms a clique and thus Proposition 4.2.15 leads to the same reduction as rule 1 of Ferizovic et al. (2020). Their rule 1 is always weight stable, therefore we restate it here.

Proposition 4.2.16 (Cliques)**.** *Let $G = (V, E)$, $U \subseteq V$ form a clique in G and $U' \subseteq U$ be the set of "inner" vertices $a \in U'$, for which $N(a) \subseteq U$. If $|U'| \geq \lfloor |U|/2 \rfloor$, creating G' by removing all vertices in U' from G and all edges between vertices in $U \setminus U'$ is a valid* MaxCut *data reduction with $\beta = \lfloor |U|/2 \rfloor * \lceil |U|/2 \rceil$.*

Rule 5 and 7 of Ferizovic et al. (2020) have the same type of clique as their nucleus as our rule 4.2.15, but only allow the removal of vertices from C until $|C| = |N(C)|$. Still, these rules are useful special cases, as they are weight stable:

Proposition 4.2.17 (Clique+, rule 5 of Ferizovic et al. (2020))**.** *Let $G = (V, E, w)$. If for a clique C in a graph G, $|C| = |N(C)| \geq 1$ and $N(C) = N(u) \setminus C \ \forall u \in C$ deleting a vertex from C is a valid data reduction with $\beta = |C|$.*

Proposition 4.2.18 (Clique++, rule 7 of Ferizovic et al. (2020))**.** *Let $G = (V, E, w)$. If for a clique C in a graph G, $|C| > \max\{|N(C)|, 1\}$ and $N(C) = N(u) \setminus C \ \forall u \in C$ deleting two vertices from C is a valid data reduction with $\beta = |N(C) - 1|$.*

4.2.4. Additional Rules

Ferizovic et al. (2020) introduce two additional rules (their rules 3 and 4), which we have not yet covered. They do not directly fit into our framework, but can enable other rules in our algorithm, and hence we consider them now. The two rules are the reverse of each other and are concerned with the so-called *near cliques*. A near clique in a graph $G = (V, E)$ is a set $C \subseteq V$ of vertices for which the induced subgraph $G[C]$ is only missing one edge. For a near clique C let $C_{\text{int}} \subseteq C$ be the set of vertices that do not have neighbors outside of C. Rule 3 of Ferizovic et al. (2020) states:

Proposition 4.2.19 (Near Clique+)**.** *Let $G = (V, E, w)$ and $w_e = 1$ for all edges. In addition, let $C \subseteq V$ form a near clique for which the missing edge $e = \{u, v\}$ connects two vertices, which only have neighbors in C. Then adding the edge $\{u, v\}$ does not change the* MaxCut *value of G if $|C|$ is odd or $|C_{int}| > 2$.*

This rule can be reversed, leading to rule 4 of Ferizovic et al. (2020):

Proposition 4.2.20 (Near Clique-)**.** *Let $G = (V, E, w)$ and $w_e = 1$ for all edges. In addition, let $C \subseteq V$ form a clique. Then removing an edge $\{u, v\}$ between two internal vertices does not change the* MaxCut *value of G if $|C|$ is odd or $|C_{int}| > 2$.*

In our algorithm, we apply rule 4.2.19 if it enables the application of rule 4.2.16. Because rule 4.2.20 can enable rule proposition 4.2.6, we apply it to cliques for which rule 4.2.16 is not applicable.

Algorithm 2: Data Reduction Algorithm

```
Input: G = (V, E, w)
Output: G'
G' ← copy(G)
redFound ← true                                   // reduction found
while redFound do
    while redFound do
        while redFound do
            redFound ← fastReductions(G')
        redFound ← neighborhoodHashReductions(G')
    redFound ← vertexSeparatorReductions(G')
return G'
```

4.3. Presolving Algorithm

Applying the data reduction rules we described on their own is more or less straight forward, but when they are combined into an algorithm together, some engineering decisions have to be made. We discuss our complete presolving algorithm, which employs the decomposition from Section 4.1 and all state-of-the-art MaxCut data reduction techniques from Section 4.2. Our presolving algorithm consists of the typical two phases: Decomposition and data reduction.

Decomposition In the decomposition phase, the input graph is divided into its connected and biconnected components, which can be solved independently. This step can be implemented in linear time, $O(|V| + |E|)$ to be specific. For each of the resulting (bi)connected components, we first check if they allow for polynomial-time solving. To this end, we apply the algorithms from Section 3.1. If none of these algorithms are applicable, the core data reduction algorithm is invoked.

Data Reduction Sophisticated presolvers operate in rounds and apply data reduction rules of different complexity in each round (Achterberg et al. 2020; Gleixner et al. 2023). For the commercial solver Gurobi for example, this strategy is realized by three nested loops, as Achterberg et al. (2020) report. In the inner most loop only the fastest reductions are performed and if only a few new ones are found, the middle loop performs more expensive checks. In the same manner, expensive reductions in the outermost loop are performed if neither of the two other steps finds sufficiently many reductions. Our presolving algorithm for MaxCut operates in a similar way, but as we observed that our implementation performed well in general, we exhaustively search for applicable rules in each step. We stop the algorithm if no new reductions have been found by any of the rules we implemented. This avoids the need for fine-tuning the algorithm for good definitions of "sufficiently many" reductions. Algorithm 2 sketches our core data reduction algorithm. Each while-loop has one or more rules assigned to it.

Most of the rules that we implemented are part of the innermost loop. For all of them, it is enough to explore the neighborhood or the two-neighborhood of a vertex, which tends to be small for sparse graphs. To make this part of our algorithm especially fast, we keep track of which rules have been checked for which vertex. As soon as the neighborhood of a vertex changes, the relevant check labels get reset. Additionally, we define an order over all rules and maintain all vertices in a priority queue. The labeling and the priority queue allow us to always check the fastest of the fast rules first in an efficient and exhaustive manner. As long as all edges have a weight of 1, we order the rules from fast to slow as follows: Degree zero, degree one, three path, clique, separated clique. Once all weight-stable rules have been applied exhaustively or the graph has edges with weights $\neq 1$, we apply all rules in this order: Degree zero, degree one, degree two, dominating edge, similar neighborhood with edge and twins, triangles, degree three. The degree three rule is used last, because it is the only rule that may add new edges to the graph, which we try to avoid as long as possible.

The second loop hashes vertices based on their neighborhood and applies the reduction for similar non-adjacent vertices (Proposition 4.2.6).

The outermost performs the data reductions based on small vertex separators. Here we originally experimented with the an implementation of the SPQR algorithm for finding vertex separators of size two. The implementation is part of the OGDF (Chimani et al. 2013) and the only publicly available one we are aware of (note: implementing the algorithm from scratch is quite involved (Gutwenger and Mutzel 2000)). After some testing, we settled for using the heuristic Algorithm 1 for finding small vertex separators, because 1) the OGDF implementation is recursive and allocates excessive amount of stack memory, requiring potentially non-portable configurations of the operating system to work, and 2) the heuristic finds separators of size two and three simultaneously. For vertex separators of size three, we are also not aware of any specialized implementations, and implementations of exact techniques would again be either quite involved or way slower (see Section 2.3).

We limit the search to vertex separators for which the graph H of Theorem 4.2.11 has at most a constant number of vertices. This allows for fast solving of the two (for vertex separators of size two) or four (for vertex separators of size three) MaxCut subproblems one has to solve for Proposition 4.2.12 and Proposition 4.2.14 respectively. For easy replicability, we implemented a simple enumeration-based solver that solves MaxCut problems on graphs with 21 or less vertices clearly below 1 second.

Planarity Preserving Some real-world instances for MaxCut are planar graphs. These can be tackled by specialized MaxCut solvers, with polynomial worst-case running time. In general, these algorithms are out of scope for this thesis, but we still want to highlight a property of the above presolving algorithm useful in this context.

Proposition 4.3.1. *Let $G = (V, E, w)$ be a planar input graph to a* MaxCut *problem. Applying our data reduction framework to G results in a planar graph G'.*

Proof. Applying data reduction that deletes vertices and / or edges from a planar graph does never result in a non-planar graph. Therefore, we only need to argue why the rules

that add edges and / or contract vertices do not interfere with planarity. Contracting vertices that have the same neighborhood is planarity preserving. Also, contracting vertices connected by an edge never results in a non-planar graph. Therefore, the rules for vertices with similar neighborhood and all edge separator based rules preserve planarity. The general rules for vertex separators of size 2 and 3 also preserve planarity, because when edges have to be added, the removal of the separated component "makes room" in a planar embedding, which is sufficient for the planar embedding of potentially new edges, connecting vertices of the vertex separator. Finally, no planar graph may contain a $K_{3,3}$ (minor) and therefore the same arguments works for all clique based rules. □

4.4. Experimental Evaluation

In the following sections, we evaluate the different preprocessing techniques and compare their ability to reduce the size of the input graph. We also investigate the impact the presolving has on the solving time of exact and heuristic solvers.

4.4.1. Implementation Details and Setup

We first describe the relevant details of our experimental setup.

Hardware All experiments were carried out on servers equipped with an AMD EPYC 7543P 32-Core CPU[2], 256GB of RAM and have Ubuntu 22.04 installed. Our code is written in C++20 and compiled with GCC 11.4. All code is free and open source software, publicly available at github[3] and archived by Zenodo (Charfreitag et al. 2024b).

Solver To analyze the impact of our new preprocessing algorithm on exact solvers, we use Gurobi (Gurobi Optimization, LLC 2024) and sms (the solver introduced in detail in Chapter 5). We use Gurobi 11.0.3 for the experiments, as Gurobi also served as the baseline for QuBowl (Rehfeldt et al. 2023) and is publicly available for academia (QuBowl itself is not publicly available). To allow Gurobi to solve MaxCut right away, we convert the problem to an unconstrained binary quadratic problem (compare Section 3.2.4). Both solvers were restricted to one thread only and we set the MIP-Gap to 10^{-6}. To make the comparison more interesting, we also give Gurobi a hint on the reducibility of the instances, by setting the presolve parameter to *aggressive*.

For the analysis of the impact on a state-of-the-art MaxCut heuristic, we opted for the heuristic of Burer et al. (2002) implemented by Dunning et al. (2018)[4]. We choose the algorithm of Burer et al. (2002), as it is also the primal heuristic of choice for the state-of-the-art MaxCut solver QuBowl (Rehfeldt et al. 2023).

[2] `https://www.amd.com/en/products/processors/server/epyc/7003-series/amd-epyc-7543p.html`
[3] `https://github.com/CharJon/SMS`
[4] We added some quality of life changes to their publicly available code: `https://github.com/CharJon/MQLib`

	V [%]		E [%]		pr [s]	
set	sota	new	sota	new	sota	new
easy	9.23	**4.53**	12.90	7.35	0.01	0.02
medium	19.37	**8.33**	27.01	13.89	0.07	0.18
big	37.81	**29.52**	53.32	46.55	26.45	39.26
mannino	63.16	**61.33**	68.73	67.03	0.00	0.00
pm1s	99.60	**99.00**	99.94	99.92	0.00	0.00
pw01	98.90	**98.60**	99.78	99.78	0.00	0.00
w01	99.00	**98.60**	99.79	99.79	0.00	0.00
t2g	62.14	**55.43**	72.33	66.75	0.00	0.00
t2pm	**100.00**	**100.00**	100.00	100.00	0.05	0.06
t3g	94.90	**94.76**	98.15	98.06	0.00	0.00
t3pm	**100.00**	**100.00**	100.00	100.00	0.00	0.00
ising	2.84	**2.78**	0.38	0.40	0.01	0.02

Table 4.1.: Average effectiveness of our implementation of the state-of-the-art (sota) preprocessing of Rehfeldt et al. 2023 and our algorithm (new). Percentage of remaining vertices ($|V|$ %) and edges ($|E|$ %), as well as average runtime in seconds (pr [s]).

Seeding To compensate for performance variability (see Section 2.5), we run each experiment five times with different seeds and treat each seed-instance pair as a separate data point.

4.4.2. Effectiveness

We first highlight the effectiveness of our new techniques. To this end, we compare it with our implementation of the state-of-the-art MaxCut presolving of Rehfeldt et al. (2023). The performance of a data reduction framework not only depends on the rules in use, but also on the order of their application, the way candidates are searched for and on whether or not the search is done exhaustively. This is why we opted for our own implementation of the state-of-the-art as a baseline.

Table 4.1 highlights the overall effectiveness of our framework compared with our implementation of the state of the art. As can be seen, our algorithm improves over the state of the art on nearly all instance sets. Especially for the sets containing real-world graphs, our preprocessing often removes significantly more vertices and edges than the state-of-the-art algorithm. For the "easy" set, we see that 13 instances get solved to optimality by presolving alone, including all 5 image segmentation instances. Instances from the "medium" set have on average close to 50 % less vertices, when our preprocessing is applied instead of the state-of-the-art algorithm. On most of the generated instances (see the lower half of Table 4.1) the difference between our algorithm and the state of the art is small. A notable exception here is the t2g set. Originally, all graphs in this set are 4-regular, but we observed edge contractions resulting in vertices with degree three. As a result, our algorithm applied the rule for three separators, which explains the removal of about 8 % more edges.

We also see that our presolving is quite fast in general. Only for instances from the

"big" set, preprocessing takes more than 0.2 s and applying our additional rules does not influence the runtimes most of the time. These positive trends give motivation to further analyze the performance of our presolving algorithm. Specifically, we want to answer the following two questions next.

- Does the higher effectiveness of our framework also help exact or heuristic solvers to be faster / calculate better solutions in less time?
- How much does each of our new techniques contribute to the overall performance?

We will investigate the first question by evaluating the efficiency next and then proceed to perform an ablation study for the second one.

4.4.3. Efficiency

To analyse the the efficiency of our presolving framework we focus on instance sets, for which 1) the presolving has a reasonable impact and 2) the instances are non trivial for state-of-the-art solvers in their original form (not like e.g. instances from the easy set). The relevant instances then are all from the "medium" and "big" sets and mannino_k487b and mannino_k487c (the other two mannino instances can be solved very fast).

Exact Solving

	$\|V\|$ [%]		$\|E\|$ [%]		pr [s]		Gurobi [s]		sms [s]	
	sota	new	sota	new	sota	new	sota	new	sota	new
g001075	11.17	1.21	15.65	2.11	0.06	0.09	12.96	1.69	3.34	**0.20**
g000677	21.81	7.21	28.62	11.28	0.04	0.12	18.33	5.10	4.17	**0.75**
g000087	32.07	18.85	37.10	23.79	0.12	0.16	1370.61	129.46	108.36	**31.12**
ca-Erdos992	14.86	10.46	36.35	33.42	0.05	0.06	670.19	814.79	77.82	**53.14**
imgseg_138032	12.86	3.73	19.21	5.87	0.04	0.05	12.12	2.22	1.08	**0.21**
imgseg_147062	49.15	24.68	56.21	30.03	0.09	0.58	1217.40	202.15	47.96	**17.01**
inf-power	17.20	5.12	24.76	10.24	0.04	0.15	6.20	1.90	1.25	**0.41**
road-lux	4.57	0.41	6.76	0.83	0.23	0.31	14.01	2.28	5.70	**0.62**
web-google	10.76	3.31	18.82	7.34	0.01	0.12	0.27	**0.21**	3.32	1.43
mannino_k48	100.00	100.00	100.00	100.00	0.00	0.00	**0.10**	**0.10**	2.44	2.44
mannino_k487a	37.78	31.83	67.67	61.18	0.00	0.00	**1.11**	1.22	1.55	1.45
mannino_k487b	48.53	47.36	43.52	43.21	0.00	0.00	8.22	8.17	6.81	**5.96**
mannino_k487c	66.32	66.12	63.75	63.72	0.01	0.01	**162.73**	273.42	279.84	257.37

Table 4.2.: Efficiency of our (new) presolving algorithm compared to the efficiency of the state-of-the-art (sota) when paired with the exact solvers Gurobi and sms. Column two and three report the percentage of vertices / edges left after the presolving and pr [s] is the reduction runtime only. The two solver columns show the total runtime (including preprocessing) for solving the respective instance to optimality. All values are averages over 5 seeds per instance.

Table 4.2 summarizes our results on the efficiency of our presolving when paired with exact solvers. The instances from the "big" set are to large for current exact solvers,

	baseline	$\|V\|$ [%]		$\|E\|$ [%]		pr [s]		improvement	
presolving instance		sota	new	sota	new	sota	new	sota	new
ca-IMDB	3 381 673	46.73	40.86	87.17	86.96	9.71	12.21	19 891	**21 486**
ca-MathSciNet	602 191	31.87	22.08	53.66	45.61	1.77	2.36	1 167	**1 768**
ca-coauthors-dblp	8 223 016	70.42	63.90	85.19	80.64	17.30	25.98	2 403	**3 390**
inf-road_central	15 678 818	21.05	3.35	28.83	6.32	123.79	150.91	456 396	**514 978**
web-Stanford	1 591 307	53.12	43.48	61.54	56.29	6.24	37.80	11 476	**12 026**
web-it-2004	4 053 078	3.68	3.48	3.55	3.51	3.79	3.71	**1**	**1**

Table 4.3.: Solution value comparison on the "big" instances. The heuristic of Burer et al. (2002) was run for 3600 s (baseline). When paired with the **sota** or **our** preprocessing, the runtime was reduced by the time the preprocessing took (pr [s]). The improvement column shows the absolute improvement over the baseline. The remaining columns report the percentage of vertices and edges left after the preprocessing. All values are averages over 5 seeds per instance.

hence we do not benchmark on these here. As can be seen, both tested solvers profit significantly, if instances get preprocessed before they get handed over to the solver. For both solvers, some instances get solved to proven optimality up to an order of magnitude faster: Gurobi is more than 10x faster on g000087 and SMS on road-luxembourg-osm and g001075. While SMS nearly always benefits from our additional presolving, this does not hold true for Gurobi on ca-Erdos992 and two of the mannino instances (one possible reason is discussed later in Section 5.5). However, the geometric mean of the speedups is 2.4 for Gurobi and 2.8 for SMS.

Heuristic Solving

To test the impact of our preprocessing on instances too large for the exact solvers (the instances in the "big" set), we employ the heuristic MaxCut algorithm of Burer et al. (2002). Table 4.3 compares the solution values of the MaxCut heuristic, when paired with the state-of-the-art preprocessing versus when paired with our algorithm. The heuristic clearly benefits from the additional preprocessing. For all graphs applying either the state-of-the-art preprocessing our ours first, before handing over the graph to the heuristic, improves the values of the solutions found. For five of the six graphs, the heuristic finds better solutions if our preprocessing is applied instead of the state-of-the-art one. Only for the web-it-2004 graph we observe no difference in the final solution value. This is expected, as also the number of remaining edges is pretty close for the state-of-the-art algorithm and ours. For the other instances even small changes in the number of vertices or edges remaining like for ca-IMDB led to better solutions. We also want to highlight the moderate increase in the running time of the preprocessing for most instances. Only web-Stanford takes significantly more time. We profiled the code and observed that preprocessing spends about 50 % of its time calling the enumeration solver to apply the rule for vertex separators of size three. This part of the algorithm could probably be improved using a more sophisticated algorithm.

4.4.4. Ablation Study

The two previous sections have shown that exact and heuristic solvers benefit from our new preprocessing techniques introduced earlier in this chapter. We now want to investigate the influence of our five new rules in detail. As we are mainly concerned with solving MaxCut to optimality, we will only consider the impact on the exact solver SMS in this section.

There are different possibilities to analyze the benefits that a specific data reduction rule offers. One could, for example, simply count the number of vertices or edges that could be removed by this specific rule, but this does not fully capture the usefulness of a data reduction rule. A rule that helps to enable other rules and allows for some type of chain reaction might be extremely important, even if the rule itself only removes a few vertices or edges. This is why we use our full preprocessing algorithm in our ablation study and only turn off the component for which we want to analyze its impact.

Twins, Cliques and Triangles

We start off by analyzing the impact of the three non-vertex separator related rules.

Our presolving algorithm found very few cases in which the new twin rule (Proposition 4.2.7) was applicable. There was no instance where more than five vertices were removed, and turning it off made little difference in general. We therefore conclude that, for the benchmark set we are working with, the rule does not seem to be helpful for our presolving algorithm. Our new rule for cliques (Proposition 4.2.15), is applicable to the same graph structure as the two rules already described by Ferizovic et al. (2020). In their experiments, they observed few instances that have this specific sub-structure, and we made the same observation. In general, there are few cases where our new rule for cliques helps to improve over the state of the art. Still, there is one notable exception: The web-it-2004 graph. Here, turning off our new rule only slightly increases the size of the resulting graphs (about 1 % more edges), but only due to the special structure solver (Section 3.1). This solver removes significantly more edges when the new rule is turned off. This shows that real-world graphs do contain the clique structure on which our new rule operates, but often they are part of small (bi)connected components.

The new rule for triangles from Proposition 4.2.9 has a noticeable impact on the size of the graph and the solving time of SMS, as shown in Figure 4.5. Making use of all three rules for triangles helps to further reduce most of the 11 graphs (for mannino_k487b and ca-Erdos992 the triangle rules have very little effect and for web-google all remaining graphs were extremely small anyways - they had less than 45 vertices). Especially for the weighted g00 and imgseg instances, the rules tend to remove many edges. Also, including our new rule for triangles ("all") mostly improves over deactivating it individually ("no-new"). Sometimes the benefit seems small, but we clearly observed diminishing returns for all triangle rules in our preliminary experiments. When activating triangle rules, independent of order, every additional rule yields less benefit. As it requires very little code to implement an additional triangle rule, if one is already there, we strongly suggest implementing all of them (including our new one).

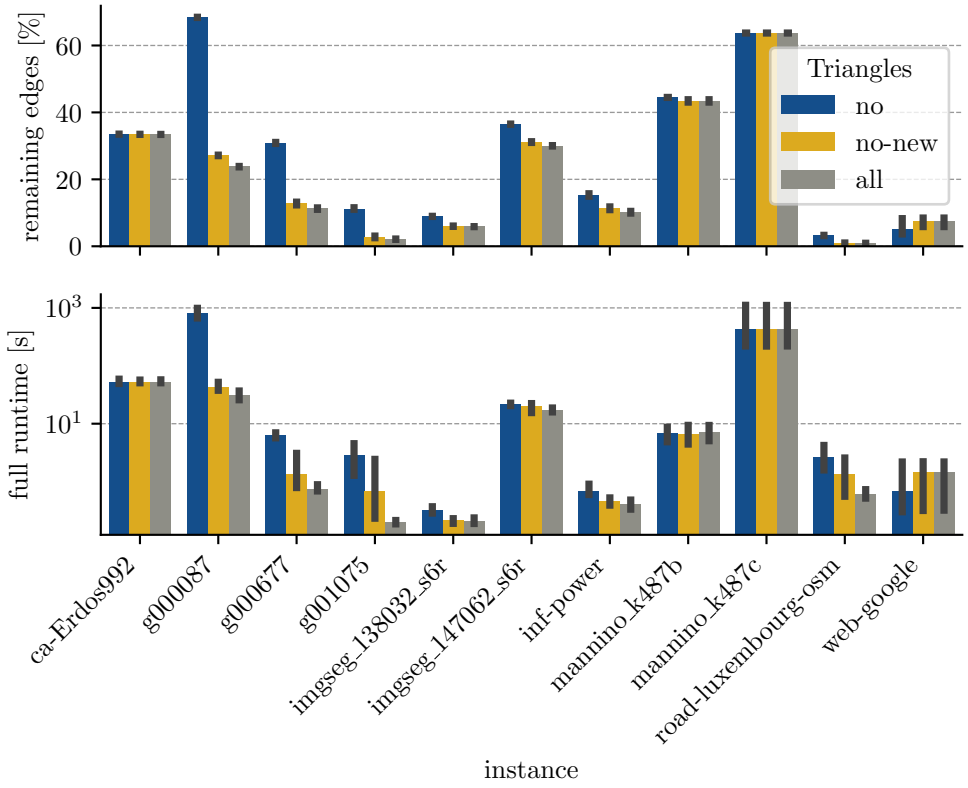


Figure 4.5.: Effectiveness and efficiency of our full algorithm, with certain triangle-related rules turned on and off. The variant with all triangle rules deactivated is "no" and "all" has all rules activated. In "no-new" only our new rule is deactivated. Top: Relative number of remaining edges. Bottom: Total runtime in seconds (data reduction + SMS, logarithmic scaling). Results shown are for five seeds per instance. Bar height is mean and error bars show min and max.

Vertex Separators

We now turn to our rules for vertex separators of size two (Proposition 4.2.12) and three (Proposition 4.2.14). To investigate their impact, we compare three settings of our algorithm in Figure 4.6. Most of the time, the use of 2-separators only slightly reduces the size of the reduced graph. Still, the full runtime (presolving time + SMS) never worsens, and for g001075 even improves by about 5 %. Note: Although the baseline for this experiment does not apply the 2-separator rule, vertices with a degree of two still get reduced, because the rule for dominating edges will always contract one of the two incident edges (Proposition 4.2.8).

Our rule for 3-separators, on the other hand, is not only extremely fast and effective but also attributes to much of the speedups over the state of the art seen earlier (compare

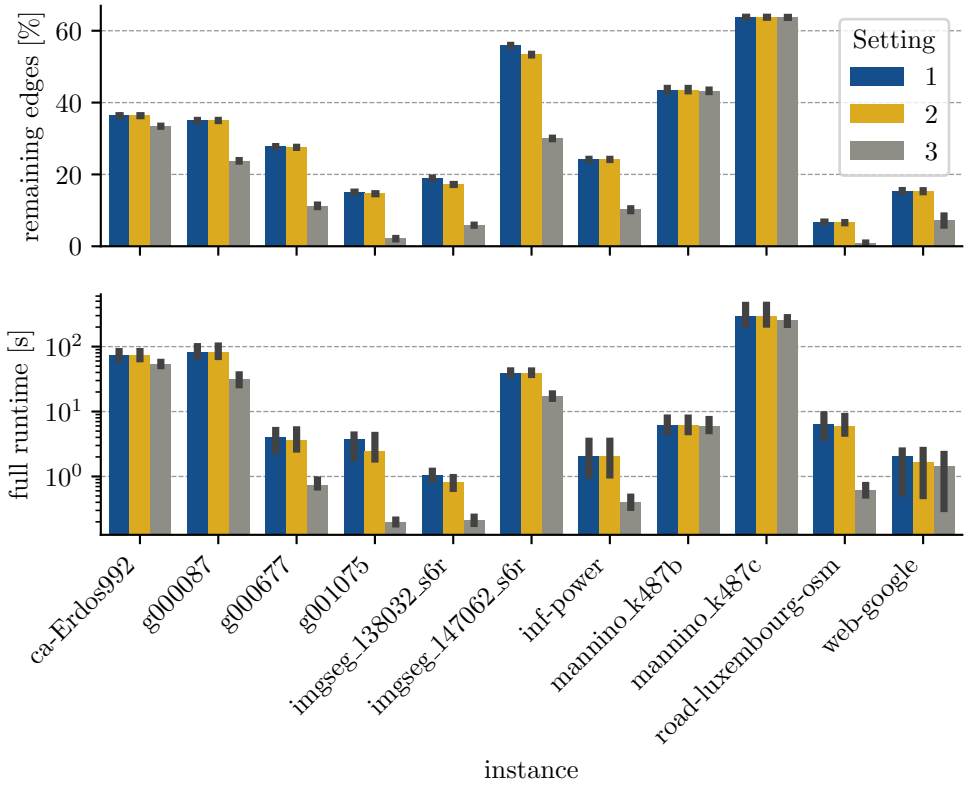


Figure 4.6.: Effectiveness and efficiency of different settings of our presolving: The full algorithm including rules for 2- and 3-separators is setting "3". The version of our algorithm, where only the rule for 3-separators is turned off is "2". For "1", the data reduction making use of 2-separators is disabled as well (apart from vertices with degree 2, as they are always covered by the dominating edge rule). Top: Relative number of remaining edges. Bottom: Total runtime in seconds (data reduction + SMS, logarithmic scaling) for each instance. Results shown are for five seeds per instance. Bar height is mean and error bars show min and max.

Table 4.2). In general, there is not a single instance for which considering separators of size two and three does not result in a speedup for the overall solving time. For g001075 the total solving time decreases by about an order of magnitude (from 2.2 s to 0.19 s). Most reductions based on vertex separators of size three stem from vertices of degree three; on average 88 % of all vertex separator based reductions (not including reductions based on vertices with degree two) are degree-three reductions. Sparse graphs have many vertices with a low degree by definition and our benchmark set has many sparse real-world graphs. This is one reason for the good performance of our degree-three rule in practice.

4.5. Conclusion

The literature already offers many different preprocessing techniques and experimental results on presolving for MaxCut (Lange et al. 2019; Ferizovic et al. 2020; Rehfeldt et al. 2023). In this chapter, we extended the theory around data reduction rules, by introducing five new rules, and investigated their performance. One of our new rules complements the rule of Rehfeldt et al. (2023) for pairs of vertices with a similar neighborhood, and one complements the rules for triangles in the input graph of Lange et al. (2019) and Rehfeldt et al. (2023). Of these two, especially the new rule for triangles helps to shrink the input graph. The other three rules are derived from our suggested framework for making use of vertex separators. The analysis of these rule showed that vertex separators of size three allow for significant reduction of the input. Overall, our data reduction rules not only turned out to be effective in removing vertices and edges from the considered input graphs but are also beneficial to exact and heuristic solvers. The exact solvers became faster by up to one order of magnitude on our benchmark set and for large graphs (with at least 820 000 edges) a state-of-the-art heuristic finds considerably better solutions when paired with our new techniques.

5. Exact Branch-and-Cut Algorithm

Integer programming and branch and cut have been employed for numerous combinatorial optimization problems in the past and are well established for MaxCut (Barahona et al. 1988; Barahona et al. 1989; De Simone et al. 1995; De Simone et al. 1996; Liers 2004; Bonato 2011; Rehfeldt et al. 2023). In general, sophisticated branch-and-cut algorithms employ a multitude of different components. Fundamental ones are (in no particular order): Presolving, which was already discussed in Chapter 4; cutting plane generation, which is the process of strengthening a given MIP model by adding additional constraints; branching and node selection strategies, which decide how to explore the search tree; primal heuristics, which try to generate good primal feasible solutions, which allow early pruning of branch-and-bound nodes.

In this chapter, we review frequently used concepts in the context of solving MaxCut to optimality through branch and cut. Building on these, we establish new theoretical results and derive novel techniques from the deepened insights. To test our new concepts, we engineered the MaxCut solver SMS, for which we implemented existing and all our new techniques. Many of them were first presented in a peer-reviewed paper in 2023[1].

At its core, our solver is based on the free and open-source MIP-solver SCIP (Achterberg 2007; Bolusani et al. 2024). The solver allows for nearly unlimited customization and we implemented many new MaxCut-specific modules. All components of our solver that are not part of SCIP itself are visualized in Figure 5.1. The presolving is performed as discussed in Chapter 4. Special exact and purely combinatorial algorithms for MaxCut we implemented are covered in Section 3.1. The remaining components will be discussed in this chapter as follows: Before employing branch and cut, one has to pick an MIP model, in our case for the MaxCut problem. Different MIP models and their properties are discussed in Section 5.1. Afterwards, Section 5.2 is concerned with cutting planes and Section 5.3 with general and MaxCut specific branching heuristics. Next, we move on to the primal heuristics in Section 5.4. Section 5.5 discusses an important detail for LP relaxation based bounding in the context of MaxCut. This chapter ends with an experimental study (Section 5.6), which investigates the performance of our solver and analyzes the impact of our new proposed concepts and the solvers' components in detail.

[1] Jonas Charfreitag, Sven Mallach, and Petra Mutzel (2023). "Integer Programming for the Maximum Cut Problem: A Refined Model and Implications for Branching". In: *SIAM Conference on Applied and Computational Discrete Algorithms, ACDA 2023, Seattle, WA, USA, May 31 - June 2, 2023*. Ed. by Jonathan W. Berry, David B. Shmoys, Lenore Cowen, and Uwe Naumann. SIAM, pp. 63–74. DOI: 10.1137/1.9781611977714.6.

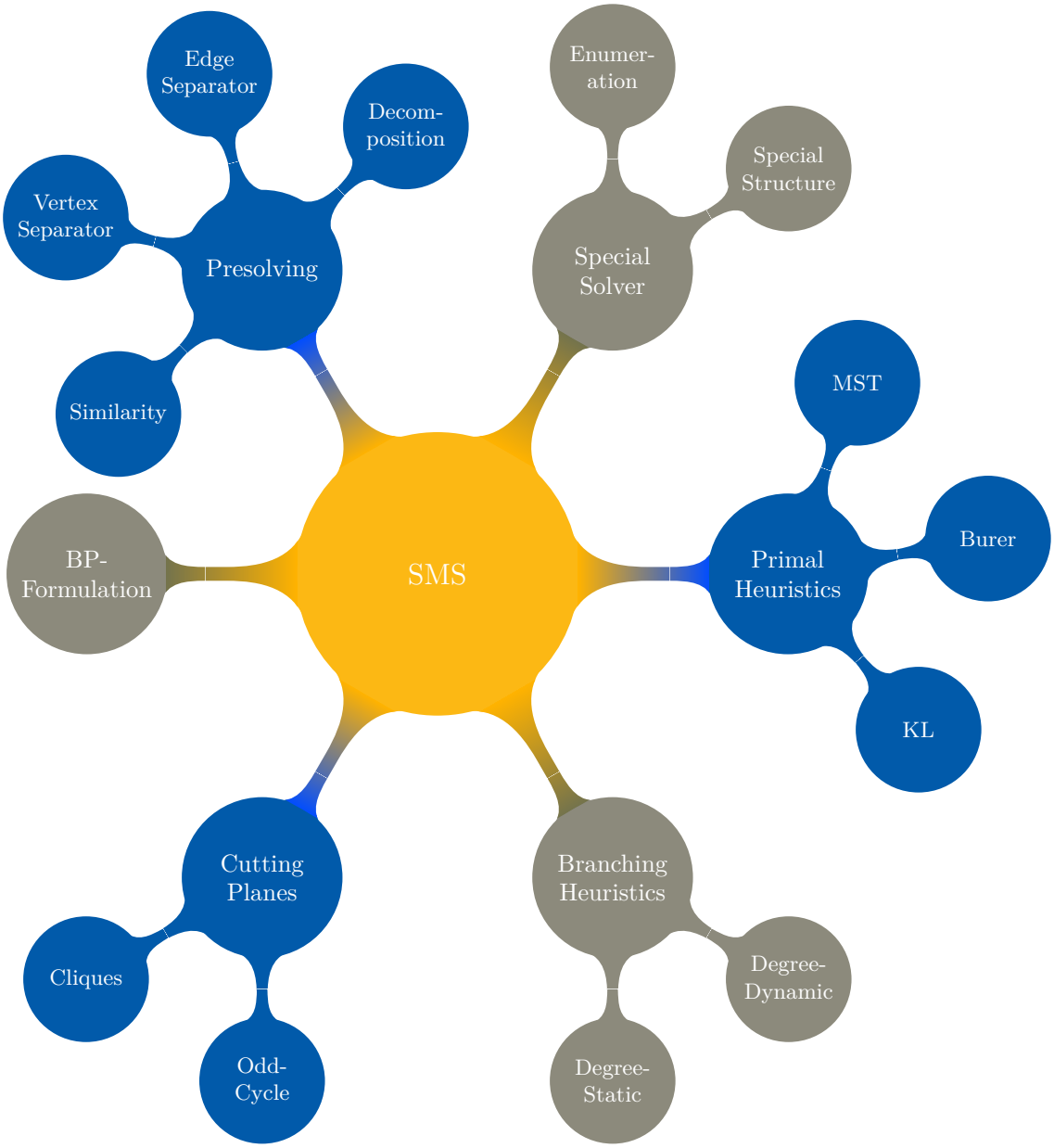


Figure 5.1.: The different components of our integer programming based solver for MaxCut: SMS.

5.1. ILP Formulations

In Chapter 3 basics of modeling MaxCut as a mathematical program have been discussed. Here we introduce and compare different relevant models in more detail. All models will be stated for an undirected weighted input graph $G = (V, E, w)$.

Edge Based Models In their seminal work, Barahona and Mahjoub (1986) introduced the following ILP for MaxCut, which we are going to call the edge based model, or `(E)` for short.

$$\text{maximize} \sum_{e \in E} w_e x_e \tag{5.1a}$$

$$\text{s.t.} \sum_{e \in S} x_e - \sum_{e \in C \setminus S} x_e \leq |S| - 1 \qquad \forall \text{cycles } C \subseteq E \text{ and } \forall S \subseteq C, |S| \text{ odd} \tag{5.1b}$$

$$x_e \in \{0, 1\} \qquad \forall e \in E \tag{5.1c}$$

The `(E)` model has exactly $|E|$ many variables in total, one per edge. These edge variables encode whether an edge is part of the cut and the solution vector x of all these variables forms a characteristic vector of a cut. Although the number of variables is linear in the input, the number of constraints can grow exponentially with the size of the input graph. An arbitrary graph might contain exponentially many cycles in the input size, and for each cycle there are exponentially many constraints of type (5.1b). Therefore, explicitly representing all of these is computationally expensive. Fortunately, it is also unnecessary. Despite the exponential number of constraints, the LP relaxation can be solved in polynomial time. A famous result of Grötschel et al. (1981) shows that if at least one violated constraint can be found in polynomial time, the LP relaxation can be solved in polynomial time. Barahona and Mahjoub (1986) describe a separation routine to find violated inequalities in polynomial time of type (5.1b), which we will discuss in detail in Section 5.2.1. The integrality gap of `(E)` depends on the input. For planar graphs e.g. the relaxation has integer optimal solutions.

A simple way to reduce the potentially exponential number of constraints of `(E)` is to extend G to a complete graph, by adding all missing edges with a weight of 0. The resulting model has $O(n^2)$ variables and $O(n^3)$ constraints. This model is easy to formulate, but because of the cubic number of constraints, it scales quite badly in practice (Frangioni et al. 2005). This might be the reason why we are not aware of any empirical work on this augmented model. However, the literature offers some alternative compact models based on (5.1b). Lancia and Serafini (2011) extend `(E)` and encode the separation routine of Barahona and Mahjoub (1986) into their model. This results in $O(n \cdot m)$ constraints, which is already quadratic for sparse graphs. We do not consider this model any further, as a quadratic number of constraints seems unappealing for practical applications.

Mallach (2024) recently suggested a family of binary programming formulations for MaxCut based on spanning trees and (5.1b). These models work with a sufficient selection of subsets of constraints of type (5.1b). Therefore, the value of their LP relaxation is upper bounded by the one of the `(E)` model and we do not go into more detail here.

Models Based on Vertices and Edges A compact IP formulation of linear size can be derived from the concepts of Simone (1990), explicitly described by, e.g., Deza and Laurent (1997) and later reconsidered by Nguyen and Minoux (2021). We will call this model the vertex-edge model, or `(VE)` for short.

$$\text{maximize} \sum_{\{i,j\}\in E} w_{ij}x_{ij} \tag{5.2a}$$

$$x_{ij} + z_i + z_j \leq 2 \qquad \forall\{i,j\} \in E \tag{5.2b}$$

$$x_{ij} - z_i - z_j \leq 0 \qquad \forall\{i,j\} \in E \tag{5.2c}$$

$$-x_{ij} + z_i - z_j \leq 0 \qquad \forall\{i,j\} \in E \tag{5.2d}$$

$$-x_{ij} - z_i + z_j \leq 0 \qquad \forall\{i,j\} \in E \tag{5.2e}$$

$$z_i \in \{0,1\} \qquad \forall i \in V \tag{5.2f}$$

$$x_{ij} \in \{0,1\} \qquad \forall\{i,j\} \in E \tag{5.2g}$$

The z_i variables encode a vertex bipartition: In every integer solution, every vertex i is either assigned to subset zero ($x_i = 0$) or to subset one ($x_i = 1$). The x_{ij} variables are the same as for the `(E)` model and encode the cut induced by the vertex bipartition of the z variables. What might make the model appealing is that it is compact by design; it only requires $|V|+|E|$ many variables and has $4|E|$ constraints. Its integrality gap on the other side is as bad as possible: By setting all z_i to 0.5, all constraints are satisfied, independent of the values of the x variables. As a result, all x_{ij} with positive coefficient w_{ij} will have a value of 1 and all x_{ij} with negative coefficient w_{ij} a value of 0. The objective value then becomes the trivial MaxCut upper bound: $\sum_{e\in\{e:w_e>0\}} w_e$. Note: For edges with positive weight, one can drop (5.2d) and (5.2e) and for edges with negative weight (5.2b) and (5.2c). In addition, the integrality requirement on the edge variables can be dropped. Every solution with integral z variables will also have integral x variables (assuming that there are no zero-weight edges).

The Root-Triangulated Model The `(VE)` model involves at least one symmetry: As each cut in a graph is implied by two different vertex bipartitions, there are at least two different solution vectors z in `(VE)` resulting in the same optimal solution values. This type of symmetry can lead to the exploration of many unnecessary branch-and-bound nodes. However, this symmetry can be broken quite easily. Frangioni et al. (2005) introduced a model for MaxCut in the context of Lagrangian relaxation that can also be employed as a binary program for branch-and-cut solvers. The underlying idea is that one vertex in `(VE)` can be assigned freely to any of the two partitions. We pick an arbitrary vertex $r \in V$, the *root*, and assign it to subset zero by setting $z_r = 0$. This

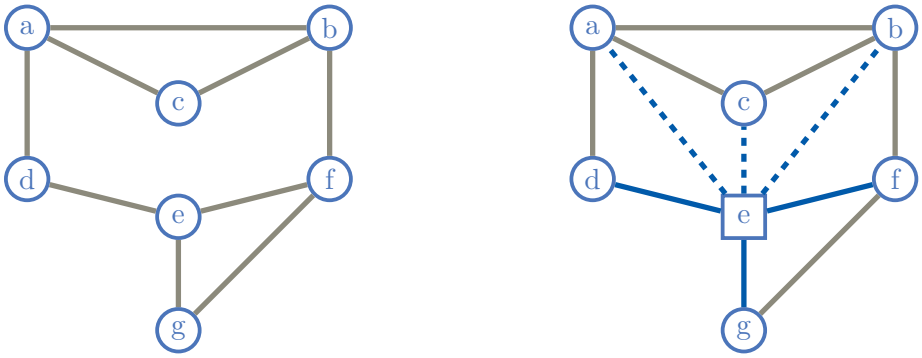


Figure 5.2.: Left: The input graph. Right: The root triangulated version of the graph. The vertex e became the root and dashed edges got added to the graph.

leads to the root triangulated model for MaxCut, `(RT)` for short:

$$\text{maximize} \sum_{\{i,j\}\in E} w_{ij}x_{ij} \tag{5.3a}$$

$$x_{ij} + x_{ri} + x_{rj} \leq 2 \qquad \forall\{i,j\} \in E : r \notin \{i,j\} \tag{5.3b}$$

$$x_{ij} - x_{ri} - x_{rj} \leq 0 \qquad \forall\{i,j\} \in E : r \notin \{i,j\} \tag{5.3c}$$

$$-x_{ij} + x_{ri} - x_{rj} \leq 0 \qquad \forall\{i,j\} \in E : r \notin \{i,j\} \tag{5.3d}$$

$$-x_{ij} - x_{ri} + x_{rj} \leq 0 \qquad \forall\{i,j\} \in E : r \notin \{i,j\} \tag{5.3e}$$

$$x_{ri} \in \{0,1\} \qquad \forall i \in V \setminus \{r\} \tag{5.3f}$$

This model has one variable per edge and one variable per vertex $u \neq r$ that is not neighboring r in the original graph, so $|E| + |V| - d(r) - 1$ in total. Intuitively, the model is the result of adding edges with a weight of zero to all non-neighbors of r. This is visualized in Figure 5.2. The model has strictly fewer variables and constraints compared to the `(VE)` model, while at least maintaining the same integrality gap (in practice it is often better). The model only restricts the variables corresponding to edges incident to the root to binary values (5.3f) since binary values for these variables already imply a feasible solution. The model can be viewed as a vertex-edge based, as the x_{ri} variables encode a bipartition the same way as the z_i variables in the `(VE)` model do and every edge of the input has a corresponding variable. Additionally, the model is also related to the purely edge based spanning tree models of Mallach (2024): All edges incident to the root in the root-triangulated input graph form a spanning tree.

We will base the solver for our experiments on the root-triangulated model for three main reasons. First of all, its compact form makes it well suited for sparse graphs, the focus of this work. Second, as the variables form a superset of the variables of the edge based model. Therefore, all techniques developed for the edge based model can be applied to the root triangulated model as well. Finally, we will show special properties of the model in the following sections, from which we derive new and fine-tuned algorithms that might help to improve the performance of branch-and-cut MaxCut solvers.

5.2. Cutting Planes

We will further discuss two types of MaxCut-valid inequalities, originally presented by Barahona and Mahjoub (1986). Both help to tighten the LP relaxation of our `(RT)` model when added as cutting planes.

Independent of the concrete inequality, Barahona and Mahjoub (1986) describe a technique for deriving facet-defining inequalities from other facet-defining inequalities:

Proposition 5.2.1. *Given any* MaxCut*-valid facet inequality defined on variables corresponding to edges from some edge set T of the form:*

$$\sum_{e \in T} a_e \cdot x_e \leq k \tag{5.4}$$

A new facet-defining inequality can be derived by picking some cut $\delta(S)$ and transforming (5.4) *into:*

$$\sum_{e \notin \delta(S)} a_e \cdot x_e + \sum_{e \in \delta(S)} a_e \cdot (1 - x_e) \leq k \tag{5.5}$$

In practice, we sometimes want to find inequalities that have maximum violation. For this, the following observation of Liers (2004, p. 24) is helpful.

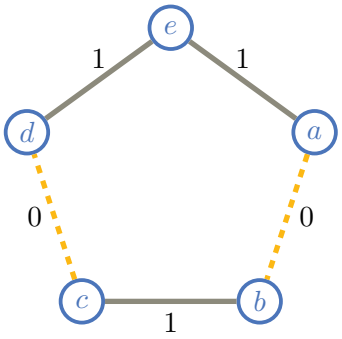
Proposition 5.2.2 (Switching)**.** *Finding the cut $\delta(S)$ for which the transformation from inequality* (5.4) *to inequality* (5.5) *results in the most violated inequality is equivalent to finding a maximum cut in a graph $G[T]$, with edge weights $a_e(1 - 2x_e)$.*

Proof. To find the most violated inequality, we want to find the cut $\delta(S)$ that maximizes the left side. Let $G' = G[T]$ be the graph induced by all edges in T. All variables of the original constraint contribute $1 - x_e$ if they are in $\delta(S)$ or x_e if they are not in $\delta(S)$. Transforming this cut / uncut problem into MaxCut results in a graph, where all edges have weights $a_e(1 - 2x_e)$ and the objective has a constant offset of $\sum_{e \in T} x_e$. Any cut in G' with objective value $> k - \sum_{e \in T} x_e$ violates the inequality. □

5.2.1. Odd-Cycle Inequalities

One central class of valid inequalities, first described by Barahona and Mahjoub (1986) and at the core of the MIP model (5.1), is often referred to as odd-cycle inequalities. The basic intuition behind these inequalities is the fact that for any cut in a graph G and any cycle C in G, the number of edges part of the cut and part of the cycle has to be even. More formally, for any cycle C and any set $F \subseteq C$ of odd cardinality ($|F| \mod 2 = 1$), the following inequality has to hold:

$$\sum_{e \in C \cap F} x_e - \sum_{e \in C \setminus F} x_e \leq |C| - 1 \tag{5.6}$$



$$\sum_{e \in C \cap F} 1 - x_e + \sum_{e \in C \setminus F} x_e \not\geq 1$$
$$C = \{\{a,b\},\{b,c\},\{c,d\},\{d,e\},\{e,f\}\}$$
$$F = \{\{a,e\},\{d,e\},\{b,c\}\}$$

Figure 5.3.: Left: A graph forming a cycle. All solid edges have a weight of 1, the dashed edges have a weight of 0. Right: A violated odd-cycle inequality formed from the cycle to the left. The resulting left hand side of the inequality is zero.

Slightly modifying the inequality gives the alternative form:

$$\sum_{e \in C \cap F} 1 - x_e + \sum_{e \in C \setminus F} x_e \geq 1 \tag{5.7}$$

As can be seen, the "odd" in odd-cycle inequalities does not refer to the length of the cycle but to the cardinality of the set of edges F. Although this may lead to confusion, we stick to the established naming scheme to avoid divergence from the literature. As a side note: Transforming odd-cycle inequalities by Proposition 5.2.1 gives a new odd-cycle inequality for the same cycle, with a different set F. Figure 5.3 visualizes the inequalities in an example.

Relation to xor-Constraints

Inspired by the analysis of Zhang and Siegel (2012) we want to highlight the direct relationship between MaxCut specific odd-cycle inequalities and general parity constraints. This has, to the best of our knowledge, not been discussed in the MaxCut related literature so far. In fact, odd-cycle inequalities are just the linear relaxation of the corresponding xor (exclusive-or) or even-parity constraint.

Theorem 5.2.3. *Let $G = (V, E)$, $C \subseteq E$ be some cycle in G, and $x_e \in \{0,1\}$ for all $e \in C$. Then $\bigoplus_{e \in C} x_e = \bot$ if and only if $\sum_{e \in S}(1 - x_e) + \sum_{e \in C \setminus S} x_e \geq 1$ for all $S \subseteq C$, $|S|$ odd.*

Proof. Suppose that $\bigoplus_{e \in C} x_e = \bot$ and let $T = \{e \in C : x_e = 1\}$. Since $|T|$ is even, for any $S \subseteq C$, $|S|$ odd, it follows that either $\exists e \in S : e \notin T$, or $\exists e \in T : e \notin S$, i.e., $e \in C \setminus S$. Since in the first case, $x_e = 0$, and in the second case, $x_e = 1$, both clearly satisfy $\sum_{e \in S}(1 - x_e) + \sum_{e \in C \setminus S} x_e \geq 1$.

On the other hand, if $\sum_{e \in S}(1 - x_e) + \sum_{e \in C \setminus S} x_e \geq 1$ for all $S \subseteq C$, $|S|$ odd, then $|\{e \in C : x_e = 1\}|$ must be even and thus $\bigoplus_{e \in C} x_e = \bot$. □

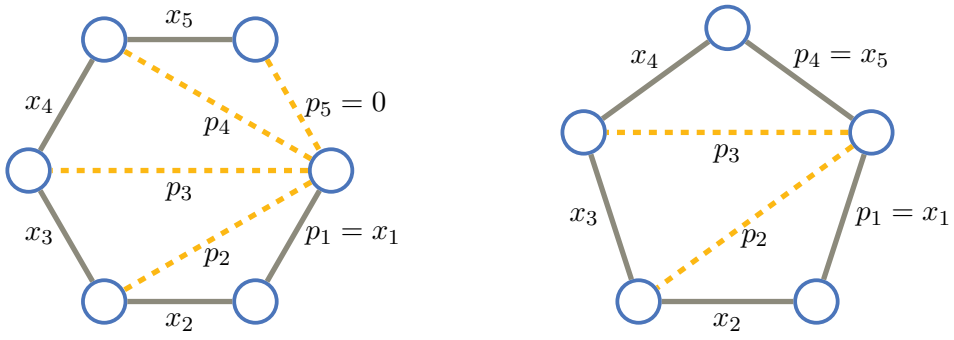


Figure 5.4.: Illustration of the asymmetric formulation for an XOR-constraint with five binary variables x_1, x_2, x_3, x_4, x_5 (left) and its simplification (right) resulting in a "pointed triangulation" (Nguyen and Minoux 2021).

Throughout the branch-and-cut algorithm, the variables will be fixed to a value of 0 or 1. As the boolean XOR operation is associative, we can state the following observation.

Observation 5.2.4. *If, for any cycle C, all but one (x_e) or all but two edge-variables (x_e, x_f) are fixed (binary), all odd-cycle inequalities related to C collapse into a single equation ($x_e = 0$ or $x_e = 1$, respectively $x_e = 1 - x_f$ or $x_e = x_f$).*

For an XOR expression on n binary variables, let $p_k := \bigoplus_{i=1}^{k} x_i$, for $k \in \{1, \dots, n\}$ capture the "intermediate" parity of the expression, if only the first k variables are considered. Clearly $\bigoplus_{i=1}^{n} x_i = x_n \oplus p_{n-1}$. Also, p_n captures the parity of the full expression, so constraints of the odd-cycle type require $p_n = \bot$. With this we can see that applying a *pointed triangulation*, described by Nguyen and Minoux (2021), to a chordless cycle leads to the same constraints, as described by Carr and Konjevod (2005) as part of their "asymmetric formulation" for the parity polytope. This previously unreported relation is illustrated in Figure 5.4.

Due to the direct correspondence between xor-constraints and odd-cycle inequalities, a result of Taghavi and Siegel (2008, Theorem 1) implies that only one odd-cycle inequality associated with the same cycle may be violated at a time. We formalize this result in the context of odd-cycle constraints in the following.

Lemma 5.2.5. *Let $C \subseteq E$ be a cycle in a graph $G = (V, E)$ and let $\bar{x} \in \mathbb{R}^E$ with $0 \leq \bar{x} \leq 1$. If $S \subseteq C$ is an odd-cardinality subset that admits a odd-cycle inequality violated by $\bar{x}$, then the odd-cycle inequalities w.r.t. any other odd-cardinality subset $S' \subseteq C$, $S' \neq S$, are* strictly *satisfied by $\bar{x}$.*

Proof. W.l.o.g., let $S \subseteq C$ with $|S|$ odd be a largest subset of C that admits a violated odd-cycle inequality for some given $\bar{x} \in \mathbb{R}^E$. Rewrite this inequality as $\sum_{e \in S}(1 - \bar{x}_e) + \sum_{e \in C \setminus S} \bar{x}_e < 1$.

Algorithm 3: Maximize LHS for Odd-Cycle Inequality

Input: Cycle C and values in LP relaxation x
Output: $F \subseteq C$, with $|F|$ odd

```
F, f ← (∅, none)
for e ∈ C do
    if x(e) > 0.5 then
        F ← F ∪ {e}
    if f = none or |0.5 − x(e)| < |0.5 − x(f)| then
        f ← e
if |F| is even then
    if f ∈ F then
        F ← F \ {f}
    else
        F ← F ∪ {f}
return F
```

Now, any subset $S' \subseteq C$, $S' \neq S$, (with $|S| = |S'|$) differs from S in at least one variable. If $C = S$, then S is clearly the only odd subset of this cardinality. So, let $S \subsetneq C$, and choose $e_1 \in S$ and $e_2 \in C \setminus S$ arbitrarily. By a simple transformation of the violated constraint, we obtain $1 < (1 - \bar{x}_{e_2}) + \bar{x}_{e_1}$. It follows that assigning any arbitrary $e_1 \in S$ to $C \setminus S'$ and any arbitrary $e_2 \in C \setminus S$ to S' leads to a strictly satisfied odd-cycle constraint.

Similarly, if $|S'| < |S|$, then S' needs to differ from S in at least two elements. Now choosing $e_1, e_2 \in S$, again simple transformations show that $1 < x_{e_1} + x_{e_2}$, i.e., an odd-cycle inequality associated to S' must be strictly satisfied. □

Taghavi and Siegel (2008) show how to find a violated inequality in linear time. Their procedure is practically the same as that suggested by Liers (2004, p. 20) for finding the inequality most violated for a given cycle. Note: The fast algorithm also follows from Proposition 5.2.1. Finding the most violated odd-cycle inequality is the same as solving a MaxCut problem on a cycle. Algorithm 3 shows a one-pass version of the algorithms of Taghavi and Siegel (2008) and Liers (2004).

Interplay with the root-triangulated model

The MIP-formulation we use is the `(RT)` model, which always has one variable for each edge of the input (and potentially some additional variables). Therefore, we can add odd-cycle constraints to tighten the LP relaxation. We now present structural results on these inequalities in the `(RT)` model.

Lemma 5.2.6. *No solution of the LP relaxation of the root-triangulated model violates an odd-cycle inequality whose corresponding cycle contains the root vertex.*

Proof. Nguyen and Minoux (2021) show, that whenever a cycle is triangulated in a vertex v, the triangle inequalities corresponding to the triangulation imply every odd-cycle inequality through v. In the `(RT)` model, every cycle is triangulated in the root r. Therefore, all odd-cycle inequalities of cycles that go through r are implied by the (triangle) constraints of the model. □

During the branch-and-cut algorithm, some vertices might be assigned to a fixed partition. This happens, for example, for each branching decision. Insights into the effects of those fixings might help speed up the overall algorithm. In fact, in the root triangulated model, fixings impact the odd-cycle inequalities, as first discussed by Nguyen and Minoux (2021). As it turns out, their result can be generalized to the following.

Lemma 5.2.7. *In the root-triangulated model, whenever a variable x_{ru} corresponding to a vertex gets fixed (e.g. via branching), no solution of the LP relaxation of the root-triangulated model violates an odd-cycle inequality whose corresponding cycle contains the vertex u.*

Proof. Consider some chordless cycle $C \subseteq V$, with edge set T, with $u \in C$. Clearly $|C| \geq 3$. Let a and b be the neighbors of u in (C, T). Differentiate two cases: $x_{ru} = 0$ and $x_{ru} = 1$. Start with $x_{ru} = 0$: The triangle inequalities on $\{x_{ra}, x_{ru}, x_{ua}\}$ collapse in this case to $x_{ra} = x_{ru}$. And those on $\{x_{rb}, x_{ru}, x_{ub}\}$ to $x_{rb} = x_{ub}$ (see Observation 5.2.4). If there was a violated cycle through u of the form $\sum_{e \in C \cap F} x_e - \sum_{e \in C \setminus F} x_e \leq |C| - 1$ we could replace the two edges incident to u via the equations and get a violated odd-cycle inequality through r. But the odd-cycle inequalities are already implied by the model; see Lemma 5.2.6. The proof for $x_{ru} = 1$ works in the same way. □

Separation

Barahona and Mahjoub (1986) showed, that odd-cycle constraints can be separated in polynomial time. Their procedure operates on an auxiliary graph: For a given input graph $G = (V, E)$ and a solution $\tilde{x} \in \{0, 1\}^{|E|}$ to the LP relaxation create a new weighted graph H, which contains two copies u_l and u_r for each vertex $u \in V$. We call u_l and u_r twins. The set of edges of H consists of four copies of each edge $e = \{u, v\} \in E$: $\{u_l, v_l\}$ and $\{u_r, v_r\}$, each with weight $\tilde{x}_e$ and $\{u_l, v_r\}$ and $\{u_r, v_l\}$, each with weight $1 - \tilde{x}_e$. This auxiliary graph H allows efficient separation of violated inequalities of type (5.7). To find violated inequalities of type (5.7), start once at every vertex u_l and find a shortest path p to u_r. If p is shorter than 1, we found a violated inequality. First, observe that any path in H from some vertex u_l to its twin u_r corresponds to a closed walk / cycle in G. In addition, the number of edges in p that connect the vertices on the left and right sides is always odd (due to the construction of H). Another observation that follows from Hs special structure is the following:

Observation 5.2.8. *Due to the inherent symmetry of the auxiliary graph H, we always have $dist(u_l, v_l) = dist(u_r, v_r)$ and $dist(u_l, v_r) = dist(u_r, v_l)$. As a consequence, we get $dist(u_l, u_r) = dist(u_l, v_l) + dist(u_l, v_r) = dist(u_r, v_l) + dist(u_r, v_r)$ for every v on a shortest path from u_l to u_r.*

Fast shortest path The shortest path in $H = (V, E)$ can be computed with the algorithm of Dijkstra (1959) and a binary heap (Williams 1964) in $O((n+m) \cdot \log(n))$ time; for sparse graphs with $m \in O(n)$ this is in $O(n \log(n))$. This is quite efficient, but for even better performance in practice, further speedup techniques have been employed to modify the vanilla algorithm:

The first is to stop the search if all upcoming paths would be longer than 1. Let $\text{dist}(v)$ be the current distance label of every vertex v. If, while executing Dijkstras algorithm, the smallest distance label of every unprocessed vertex v is $\text{dist}(v) \geq 1$, no new paths of length < 1 will be found and the algorithm can be aborted. This potentially reduces the number of vertices that the algorithm explores.

Second, Jünger and Mallach (2019) suggest to take advantage of the symmetry of the auxiliary graph H. If a vertex v_l and its twin v_r have been assigned their final distance label and $\text{dist}(v_l) + \text{dist}(v_r) \geq 1$, outgoing edges of v_l and v_r can be ignored for finding shortest path, as no path of length < 1 can pass through them (the proof follows directly from Observation 5.2.8).

The symmetry of H allows for a generalization of the above ideas, which has (to the best of our knowledge) not been reported so far. First note, when exploring H via Dijkstras algorithm from u_l we implicitly also explore the graph from u_r, the target (see Observation 5.2.8). This *bi-directional search* can lead to significant speedups in practice (Pohl 1969). Because of this implicit bi-directional Dijkstra, the algorithm may stop if it encounters a vertex with distance ≥ 0.5, instead of one with distance ≥ 1. An explicit bi-directional search, as suggested by Nguyen and Minoux (2021) adds unnecessary overhead.

Also note, there is no need to construct the separation graph H explicitly. Neighborhood queries for the Dijkstras algorithm can be performed on the original graph, and edge weights from the current LP relaxation can be stored separately (potentially reducing the algorithm's memory footprint and improving its cache locality).

Blocklist To further speed up the algorithm, we also suggest the use of a blocklist. Vertices on the blocklist get ignored when searching for shortest paths. One straightforward application for this list is Lemma 5.2.7: Every vertex whose partition is already fixed (via e.g. branching) can be put on the blocklist, as it will never be part of a violated odd-cycle inequality. The blocklist also helps to speed up the algorithm when calls to the separation algorithm result in no violated cycle (when no u_l-u_r path of length < 1 exists). This happens especially often, when no violated odd-cycle constraints exist anymore. There are two types of vertices that can be put on the blocklist in this case: The starting vertex u: If no path of length < 1 from u_l to u_r exists, there is no need to consider u_l or u_r for other shortest paths. Furthermore, all vertices reachable from u_l in H, by only traversing edges with weight 0 can be put on the blocklist as well: let v_l be a vertex reachable by a zero-weight path from u_l. Assume there was a path of length < 1 from v_l to its twin v_r. If v_l is reachable by a zero-weight path from u_l, v_r must be reachable by a zero-weight path from u_r (Observation 5.2.8). Therefore, a path of length < 1, should have been found from u_l (the same argument holds with the roles of

v_r and v_l exchanged), and hence not only u, but also v can not appear on any shortest path of length < 1.

Post-processing The shortest path found by the above procedure may not be chordless. Jünger and Mallach (2019) designed techniques to extract chordless cycles from results of the separation algorithm. They found them to be beneficial for the overall performance of their branch-and-cut algorithm. Therefore, we also extract short chordless cycles from a closed walk, employing a two-step approach, introduced by Rehfeldt et al. (2023):

First, we extract all simple cycles from a closed walk using the technique of Jünger and Mallach (2019). Next we check for each simple cycle if it contains chords and whether or not these chords allow to construct shorter violated odd-cycle constraints. This not only has the benefit of extracting chordless cycles, but also might yield multiple violated odd-cycle inequalities per separated shortest path.

An additional technique to ensure the separation of chordless cycles is to explicitly enumerate short chordless cycles of length, e.g. three and four, as already suggested by Barahona et al. (1988). This can be done quite fast for sparse real-world structured graphs, hence we cache all chordless cycles of length 3 (triangles) and length 4 (often referred to as 4-holes) and separate all violated odd-cycle constraints for these cycles fast via Algorithm 3.

5.2.2. Clique Inequalities

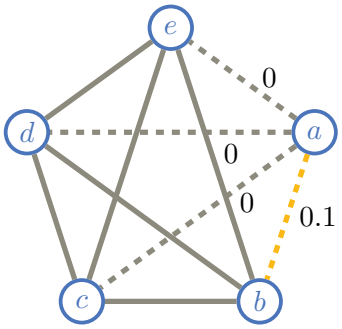
Barahona and Mahjoub (1986) not only introduced the odd-cycle constraints but also facet-inducing inequalities derived from cliques. Let $C \subseteq V$ form a clique in G with $|C| \geq 3$ and $|C| \mod 2 = 1$. Furthermore, let T be the edge set of $G[C]$. Then

$$\sum_{e \in T} x_e \leq \left\lceil \frac{|C|}{2} \right\rceil \cdot \left\lfloor \frac{|C|}{2} \right\rfloor \tag{5.8}$$

is a facet-inducing and valid inequality for MaxCut. Note: For cliques of size 3, that is triangles, the inequalities for cliques (5.8) and for cycles (5.6) are identical. The intuition behind these inequalities is straightforward. No cut in a clique may include more edges than the perfect cardinality cut has (compare Section 3.1). With the transformations of Proposition 5.2.1 every clique in the input of size k gives rise to 2^{k-1} different facet-defining inequalities.

Separation

Unfortunately, there is no known polynomial-time algorithm to separate clique inequalities (5.8). Even finding a clique of maximum cardinality is NP-hard (Karp 1972). Still, clique inequalities have been shown to improve branch-and-cut algorithms for MaxCut in the past, therefore we employ a heuristic to separate violated clique inequalities. For a given clique, the violation check can be performed by solving the MaxCut problem resulting from Proposition 5.2.1. We solve this MaxCut problem to optimality for small



$$\sum_{e \in E} x_e \not\leq 6$$

$$\sum_{e \in E \setminus \delta(a)} x_e - \sum_{e \in \delta(a)} x_e \not\leq 2$$

Figure 5.5.: Left: A graph $G = (V, E)$ forming a clique with values of the current LP relaxation assigned to its edges. All solid edges have a weight of 1, the grey dashed edges have a weight of 0 and the dashed yellow edge (bottom right) has a weight of 0.1. Right: Two violated clique inequalities formed from vertices and edges of the left clique.

cliques and employ a heuristic (see Section 5.4.1) for bigger cliques. The procedure in more detail looks like this:

1. Enumerate and cache maximal cliques of size at least 5 using the algorithm of Eppstein et al. (2013). This algorithm has a running time in $O(dn3^{d/3})$, where d is the degeneracy of the graph (which is typically small for sparse graphs).

2. For each clique C in the cache, employ one of the following two subroutines, depending on the size of C.

   a) For small cliques, enumerate all subcliques of odd cardinality and check for violation of inequalities of type (5.8).

   b) For large cliques, checking all subcliques would result in many checks. Therefore, we only check for violation of the clique inequality of all vertices if the large clique is of odd size. Or all $|C|$ subcliques of size $|C| - 1$, if the clique has even size. For the violation check, we employ the Kernighan-Lin heuristic (Section 5.4.1) on the MaxCut problem resulting from Equation (5.5).

Note: Although we showed that for every cycle there might only be one violated odd-cycle inequality (5.6) for the current lp-solution (Lemma 5.2.5), clique inequalities do not have the same property. For a given clique, there might be multiple violated inequalities, see Figure 5.5 for an example. However, we do not separate all the inequalities per clique. Once step a) from above has found a violated clique on, say, vertices $S \subset C$, it only searches for more violated inequalities in $C \setminus S$.

Interplay with the root-triangulated model

When variables corresponding to vertices get fixed to zero or one, they are combinatorically merged into the root. By construction of the `RT` binary program, the root is

connected to every vertex and, therefore, part of every maximal clique. Thus, merging vertices into the root makes their corresponding variables obsolete in the clique-inequalities, effectively shrinking the size of a cached clique. Our implementation of the separation procedure takes this into account.

5.2.3. Cut Selection and Cutpool

Modern MIP-solvers like SCIP (Bolusani et al. 2024) do not add all violated cutting planes found in one separation round to the current LP relaxation, but employ filter and caching mechanisms. To filter cuts, the so-called *cut selection* algorithms are used, and cuts are cached by a data structure called *cutpool*. The details are beyond our scope here, and we refer to the work of Wesselmann and Suhl (2012) and Dey and Molinaro (2018) for more information on this topic. However, we sketch the relevant basics of the default behavior of SCIP, which we use for our algorithm.

The cutpool, first introduced by Padberg and Rinaldi (1991), works as a cache for cutting planes. In its modern version (e.g. in SCIP) each cut has an associated age that increases each time its violation is checked and it is found to be satisfied. If the age of a cut reaches some predefined limit, it is removed from the global cutpool.

The cut selection tries to select a good subset of all known cuts. A good selection is small, to keep lp solving times low, but also contains many diverse cuts, to cut off as much as possible from the current polyhedron. To balance these two objectives, cut selection assigns a score to each cut, groups all cuts based on their score, and greedily picks cuts that are not similar (parallel) to other selected cuts. The scoring is a weighted sum over the cuts efficacy (the Euclidean distance between the LP optimal solution and the cut hyperplane), its parallelity to the objective function, and its integer support (the relative number of integer variables with non-zero coefficient).

5.3. Branching

If the LP relaxation of a MIP model does not result in an integer feasible solution, the branch-and-cut algorithm tries to add cutting planes to improve the relaxation. In the best case, this leads to an integer-feasible solution. If not, the branching module of branch and cut comes into play. *Branching rules* (also called *branching heuristics* or *branching strategy*) decide which variable to use for branching. Branching is the process of splitting the search space by creating (usually) two subproblems. For binary programs, this (usually) comes down to creating one subproblem where a certain variable x_i is fixed to 0 and one, where x_i is fixed to 1. The overall goal is to branch in a way that results in the smallest number of branch-and-bound nodes. Finding an optimal strategy here is an open problem. There are two types of branching rules: General-purpose branching rules, applicable on any MIP model, and problem-specific ones, which consider e.g. the combinatorial structure of the underlying problem, the MIP is modeling. Often, they calculate a score for each variable with a fractional value and then choose the best variable to branch on based on the score.

5.3.1. General Branching Rules

Many general purpose branching rules have been developed over the years, see e.g. (Morrison et al. 2016). Two of the first sophisticated rules are pseudocost-branching (Bénichou et al. 1971) and strong-branching. The latter was first suggested in the context of the traveling salesman problem (Applegate et al. 1995; Applegate et al. 2006). They form the basis for the state-of-the-art rule reliability pseudocost branching by Achterberg et al. (2005). All these rules try to pick the variable that has the most local impact. The impact is usually measured in terms of change of the objective value of the LP relaxation of the two created children. Pseudocost branching calculates its score based on the historical impact a variable had throughout the branch-and-cut search. Whenever a variable is branched, the pseudocost is updated. In the implementation of SCIP (Bolusani et al. 2024) ties are broken via what is called *root difference* in the code: The absolute difference of the current value of a variable compared to the value the same variable had after solving the root relaxation. Pseudocost branching has very little overhead, but has no information to work with for the first couple of branching decisions. Strong-branching does not suffer from this limitation. It performs (often expensive) lookaheads to find the best variable to branch on. To calculate the branching score for all or a set of variables, the two subproblems for branching on the variable are created and their LP relaxation gets solved. The rule then picks the best candidate. Reliability pseudocost branching combines pseudocost branching and strong branching. It works on the pseudocosts, but calculates them via strong-branching in case they turn out to be unreliable. Unreliable here means either not known yet, or they differ considerably (for some definition of considerably) throughout the branch-and-cut tree.

5.3.2. MaxCut Specific Branching Rules

The results on fixing implications in the `RT` model from the previous chapter (especially Lemma 5.2.7) show promising impact for branching on variables corresponding to vertices. To the best of our knowledge, there are no theoretical results on the impact of branching on variables corresponding to edges. Therefore, our algorithm only branches on the vertex variables.

This is in line with the problem-specific branching rule of Nguyen and Minoux (2021). They suggest branching on the variable of a vertex that has more neighbors with binary (fixed) vertex variables than the average vertex degree of the graph. If multiple such variables with a fractional value exist, they select one that has been chosen the least in previous branching decisions. If no such variable exists, they leave the branching decision to the default branching strategy of the closed-source solver CPLEX. This approach is motivated by their (weaker) version of Lemma 5.2.7 i.e., shall promote propagations and reduce the set of edges with possibly violated odd-cycle inequalities. However, their branching rule might only have an effect rather late in the branch-and-cut tree, as it may require at least as many branching decisions as the average vertex degree in the graph to obtain a candidate variable for it. Moreover, if all vertices have the same degree, the rule cannot determine a fractional branching candidate: It only considers a vertex for

Algorithm 4: Degree-Dynamic Branching Rule

Input: $X = [x_1, x_2, \ldots, x_k]$ (variables with fractional value in LP relaxation)
D (dynamic degrees of all vertices with variables in X)
Output: A branching candidate $x_i \in X$

```
1 i ← arg max_{i∈{1,...,k}} D[i]
2 return X[i]
```

branching if the variables of all its neighbors are binary, but as Observation 5.2.4 states, in this case the variable of the respective vertex would itself already be binary in an optimal solution for the current relaxation.

We suggest two related, but different approaches, which take our new results on fixing into account and aim to overcome shortcomings. It might seem intuitive that branching on variables corresponding to vertices with a high degree potentially has the most impact on the overall problem structure. Hence, we suggest simply prioritizing vertices for branching based on their degree in the original graph; the higher the degree, the higher the priority. This rule is easy to implement and can serve as a baseline for all degree-based branching rules. We are going to call it degree-static. Also, if a high degree corresponds to many chordless cycles containing a specific vertex, Lemma 5.2.7 gives a theoretical basis for this rule. However, if the input graph is regular, this rule degenerates into branching on random variables. It also does not consider the structure of the subproblems corresponding to the branch-and-bound nodes: Previous branching decisions might have led to fixed variables associated with either many or few neighbors of a vertex with a high degree, which is expected to have an impact on the effectiveness of their selection for the local branching decision to be made.

To improve degree-static, we suggest the new branching rule degree-dynamic described in Algorithm 4. Similarly to degree-static, it prioritizes variables corresponding to vertices with high degree. But instead of considering the degree in the original graph, all vertices whose corresponding variables have integer values in the current LP relaxation are ignored for the degree calculation. We call this alternative degree the *dynamic-degree*. In this way the structure of the current subproblem is taken into account and the rule does not necessarily degenerate to random branching on k-regular graphs. Unlike the rule of Nguyen and Minoux (2021), this new rule generally prioritizes vertices with many fractional neighbors over vertices with many fixed neighbors.

5.4. Primal Heuristics

In the context of MIP-solving heuristics calculating feasible solutions for MIPs are usually called primal heuristics. They are essential for the speed of any branch-and-cut solver, because the faster good solutions can be found, the quicker potential cut-offs happen (Berthold 2013).

There are many categories of primal heuristics, differing in their strategy, the type of input they operate on, and their running time. The tree heuristics involved in MaxCut

solvers based on branch and cut can be put into two categories: Heuristics from the first category take any MaxCut solution as their starting point and try to construct solutions with a better objective value. We will call heuristics of this type *improvement heuristics*. Note: If no solution is available, it's always possible to construct the trivial MaxCut solution, where all vertices belong to the same partition, or a random bipartition. The second category takes the values of the current LP relaxation into account. Based on these, the heuristics try to make informed decisions when constructing solutions step by step.

5.4.1. Kernighan-Lin

The Kernighan-Lin heuristic (Kernighan and Lin 1970) is one example of an improvement heuristic and related to the famous heuristic of the same authors for the traveling salesperson problem (Lin and Kernighan 1973). In its most general form, the heuristic tries to find good solutions for the restricted k-way partitioning problem on graphs. But the authors also sketch how to modify the algorithm in order to deal with unrestricted bipartition problems, e.g. MaxCut.

We describe the basic concepts of our version of the Kernighan-Lin heuristic (which slightly differs from the version Bonato (2011) employed for exact MaxCut-solving, as we will explain below) and sketch its running time (the original analysis did not explicitly differentiate between sparse and complete graphs). In general, the algorithm operates in two phases to improve the objective value of any starting solution:

Phase 1 greedily explores sets of k-moves and picks the best one, until it fails to find a k-move improving the solution. A k-move for a given vertex bipartition moves k many vertices to their respective opposite partition. As considering all k-moves is way to expensive, the inner loop greedily selects the best (not necessarily improving) one or two move until every vertex has been switched to its opposite partition exactly once. Afterwards the algorithm reevaluates the history of swaps considered and picks the best subset of swaps, which then forms the k-move that is performed (if the solution value improves). Algorithm 5 shows phase 1 in more detail. We deviate from the implementation of this heuristic by Bonato (2011, p. 29), by not only considering the best 1-move (line 8), but also the best 2-move, involving the best 1-move vertex. This extends the overall search space of the heuristic and comes at close to no cost.

Runtime: The number of outer loops depends on the starting bipartition, but is empirically low for our input graphs. Before starting the inner loop the improvement values need to be calculated, which takes $O(n+m)$ time. Each inner loop has to calculate the arg max (line 8) first, which can be realized with a max-heap data structure. Every vertex gets extracted once from the heap and there will be up $O(m)$ update calls to the heap. For evaluating the 2-move and updating the improvement values, one needs to loop once over the neighborhood of the corresponding vertex. Overall, the heap operations dominate the worst-case complexity and we get $O(n \cdot \log(n) + m \cdot \log(n))$ when employing binary heaps.

Phase 2 tries to further improve the solution of the previous phase. Kernighan and Lin observed in their experiments, that if their heuristic did not find the optimal solution,

Algorithm 5: Kernighan-Lin MaxCut Heuristic

Input: $G = (V, E, w)$ (weighted input graph)
(S_1, S_2) (initial biparition)
Output: (S'_1, S'_2) (locally optimal bipartition)

$(S'_1, S'_2) \leftarrow (S_1, S_2)$
gain $\leftarrow 1$
while *gain* > 0 **do**
  $V' \leftarrow V$ `// vertices not moved yet`
  $D \leftarrow [\,]$
  $I \leftarrow$ `improvement_values(`G`,` (S'_1, S'_2)`)` `//` $O(n+m)$
  while $V' \neq \emptyset$ **do**
    $a \leftarrow \arg\max_{u \in V'}(I[u])$ `// best 1-move`
    $g_a \leftarrow I[a]$
    $b \leftarrow \arg\max_{v \in N(a) \cap V'}(I[a] + I[v] \pm 2w_{au})$ `// best 2-move`
    $g_{ab} \leftarrow I[a] + I[b] \pm 2w_{au}$
    $V' \leftarrow V' \setminus \{a\}$
    $I \leftarrow$ `update_improvements(`G`,` I`,` a`)` `//` $O(d(a))$
    if $g_a \geq g_{ab}$ **then**
      D.append(move=(a), value=g_a)
    else
      $V' \leftarrow V' \setminus b$
      $I \leftarrow$ `update_improvements(`G`,` I`,` b`)` `//` $O(d(b))$
      D.append(move=(a, b), value=g_{ab})
  $p \leftarrow \arg\max_{k \in \{1,\ldots,|D|\}} \sum_{i=1}^{k} D[i]$.value
  gain $\leftarrow \sum_{i=1}^{p} D[i]$.value
  if *gain* > 0 **then**
    $(S'_1, S'_2) \leftarrow$ `perform_k_move(`(S'_1, S'_2)`,` $D[1, \ldots, k]$`)`
return (S'_1, S'_2)

the output of phase 1 was about a $n/2$-swap away from the global optimal solution. To guide the heuristic to solutions that require a large swap, they consider the two graphs induced by each of the two partitions of the best found bipartition individually, say H_1 and H_2. The algorithm calls phase 1 with H_1 and H_2 as its input. There are two possibilities to merge the two resulting partial bipartitions and the algorithm chooses the one that results in a better MaxCut-value. We do not go into more details here, as our solver does not incorporate phase 2. Because the other two heuristics we talk about next, generate diverse solutions for phase 1, we did not see any benefit in phase 2 in our setting.

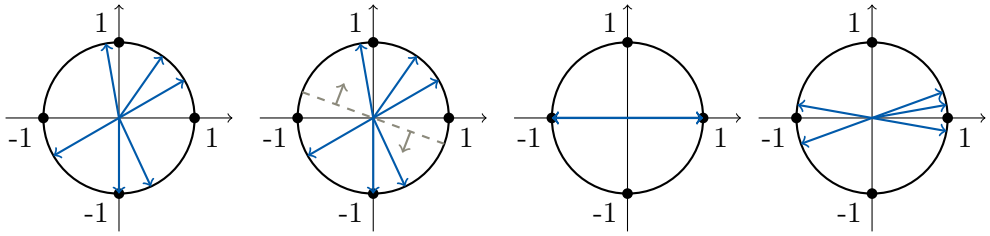


Figure 5.6.: Different steps of the Burer et al. (2002) heuristic. (a): A possible solution after solving unconstrained (5.9) via gradient descent. (b): Step 2 searches for the best diameter split, for which the corresponding bipartition maximizes the cut value. (c): The bipartition from the previous step is translated back into an angle representation. (d): All angles of the current solution are slightly perturbed.

5.4.2. Burer

The improvement heuristic presented by Burer et al. (2002) is inspired by the approximation algorithm of Goemans and Williamson (1995). Their algorithm achieves the best known approximation ratio for MaxCut on graphs with only non-negative weights of 0.879. However, it is generally considered impractical (Burer et al. 2002), as it requires to solve potentially large semidefinite programs. The basic idea of the heuristic of Burer et al. (2002) is to consider a "rank-2" relaxation of the semidefinite program. Each vertex of the input graph is associated with a point on the unit circle, represented by an angle φ. The corresponding mathematical program for MaxCut looks like this:

$$\begin{aligned} \text{minimize} \quad & \sum_{\{i,j\} \in E} \cos\left(\varphi_i - \varphi_j\right) \cdot w_{ij} \\ & \varphi_i \in \{0, \pi\} \qquad \text{for all } i \in V \end{aligned} \tag{5.9}$$

A slightly simplified version of the overall algorithm, with the non-convex program (5.9) at its core, is summarized in Algorithm 6. An iteration of the for loop is shown in Figure 5.6. The two most fundamental steps are performed in lines 4 and 5.

In line 4, the mathematical program (5.9) is relaxed by allowing $\varphi \in \mathbb{R}^{|\mathbb{V}|}$. The resulting unconstrained non-convex optimization problem is then solved by gradient descent, starting from the current vector of angles φ^i. The authors suggest employing a backtracking Armijo line search (Armijo 1966) as the gradient descent routine.

In line 5, the vector of angles is transformed into the characteristic vector of a cut. The candidate vectors z that are considered have entries z_j of the form

$$z_j = \begin{cases} 0 & \text{if } \varphi_j \in [\alpha, \alpha + \pi) \\ 1 & \text{else} \end{cases}$$

for some $\alpha \in [0, \pi)$. Visually, the bipartition is the result of drawing a diameter into the unit circle with an angle of α and assigning vertices based on the resulting left-right split (compare Figure 5.6b). The vertex bipartition resulting in the best cut value is chosen.

Note: For good practical performance Burer et al. (2002) actually suggest to not perform a fixed number of iterations of the for loop (input parameter k), but looping until the value of the best solution found did not improve for a fixed number of times. When this limit is met, they further suggest restarting the algorithm (and sample a new starting solution φ^0).

Algorithm 6: Burer et al. MaxCut Heuristic

Input: $G = (V, E, w)$ (weighted input graph)
k (iteration limit)
Output: z^* (characteristic vector of best bipartition found)

1 $z^* \leftarrow \{0\}^{|V|}$
2 Sample $\varphi^0 \sim [0, 2\pi)^{|V|}$ `// random start vector`
3 **for** $i \leftarrow 0 \dots k$ **do**
    `// find local minimum of unconstrained version of` (5.9) `via gradient descent`
4   $\varphi^i \leftarrow$ `gradient_descent`(start $= \varphi^i$)
    `// find bipartition with maximal cut value that results from a diameter split`
5   $z^i \leftarrow$ `best_bipartition`(φ^i)
6   **if** $\Delta(z^i) > \Delta(z^*)$ **then**
7     $z^* \leftarrow z^i$
8   $\varphi^{i+1} \leftarrow z^i * \pi$ `// transform to` $\{0, \pi\}$ `vector`
9   $\varphi^{i+1} \leftarrow$ slightly perturb all angles of the vector by adding small random $p_j \in [-\pi, \pi]$ values to each entry
10 **return** z^*

Runtime: Burer et al. (2002) do not analyze the theoretical computational complexity of their algorithm. The runtime of each iteration of the loop is dominated by the gradient descent (which is highly non-trivial to analyze for worst-case performance). All other steps allow for linear-time implementations. Finding the best bipartition in line 5 can be realized in $O(m + n \cdot \log(n))$ time. First, sort all vertices based on their current angle in φ^i. Next, treat all angles in φ^i between 0 and π as candidates for α that result in the best cut value ($\alpha \geq \pi$ can be skipped, due to symmetry). When going through these candidates for α in order, only one vertex switches the subset in the corresponding vertex bipartition. In this way, the MaxCut value of the last bipartition can be updated efficiently, resulting in an overall runtime of $O(n \cdot \log(n))$ for sorting and $O(m + n)$ for calculating all $O(n)$ MaxCut values. The remaining lines take at most $O(n)$ time (if the MaxCut values for z^i and z^* are cached in previous steps). We will see in our computational experiments that the heuristics is usually quite fast in practice.

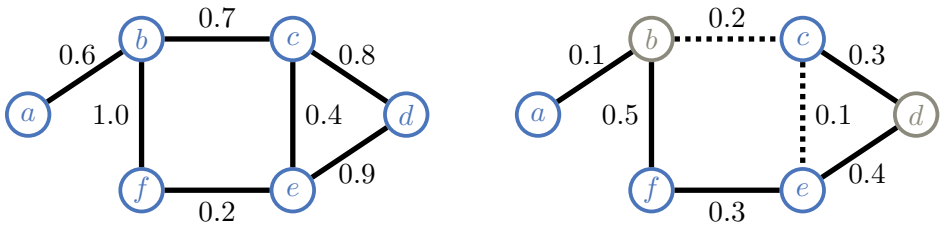


Figure 5.7.: Visualization of the MST heuristic by Barahona et al. (1989). Left: A graph G and the values of all edge variables in the current LP relaxation. Right: The transformed graph H and its maximum spanning tree (non-dashed edges). The spanning tree implies the vertex bipartition $(\{b, d\}, \{a, c, e, f\})$.

5.4.3. MST-Heuristic

A heuristic that makes use of the values of variables corresponding to edges in an LP relaxation was suggested by Barahona et al. (1989). It is based on the observation that a selection of edges of a spanning tree fully defines a bipartition of all vertices / a cut. The heuristic consists of three steps:

1. Construct a graph H with the same set of vertices and edges as the input G, but where the weights of an edge $\{u, v\}$ depend on the value of its corresponding variable x_{uv} in the LP relaxation. Concretely $w_{uv} = |x_{uv} - 0.5|$.
2. Calculate a maximum spanning tree on the resulting graph H.
3. Convert the spanning tree into a bipartition: Pick any vertex, say u, and assign u to the first partition. Perform a graph exploration through all edges of the maximum spanning tree (via e.g. DFS or BFS) starting at u. When traversing an edge from vertex a to vertex b assign b to the same partition as a if the lp value x_{ab} of the corresponding edge is < 0.5 otherwise to the opposite partition of b.

Figure 5.7 shows an example. The underlying intuition is that the closer the lp value of a variable is to zero, the more likely it might be to not be part of an optimal cut, and the closer the lp value of a variable is to one, the more likely it might be that the corresponding edge is part of an optimal cut.

Runtime The MST-heuristic has a guaranteed polynomial running time. When employing Kruskals Algorithm (Kruskal 1956) for finding a maximum spanning tree, the algorithm requires $O((m + n) \cdot log(m))$ time.

5.5. Objective Integrality and Scaling

Clearly, if for an integer program all coefficients of the objective have integer values, all primal solutions also have integer values. In this case, the dual bound of the LP relaxation can always be rounded down, potentially leading to earlier pruning of nodes during the branch-and-bound algorithm.

To further abuse this observation, IP-solvers test if the coefficients of the objective function can be scaled down. If all objective coefficients are divisible by k, scaling the objective by $1/k$ improves the upper bound from the LP relaxation. For example, consider an IP where all values of the objective function are divisible by 5 and for which the LP relaxation gives a value of 7.5. If the best primal solution found has a value of 5 the branch-and-cut algorithm needs to proceed. If we scale the objective by 0.2, the LP relaxation will have a value of 1.5 which can be rounded down to 1.0. The primal bound will be 1 in the scaled down version of the problem, leading to a cutoff of the current node. In our algorithm, we simply search for the greatest common divisor (if all edges of the input graph have integer weights) to find the best value to scale down.

Liers (2004, p. 72) shows that in k-regular graphs, if all edge weights are from $\{-t, t\}$ with $t \in \mathbb{N}$ and either k or t is even, all possible cuts have even values. As a result, all objective coefficients can be divided by two in these cases, again giving better LP relaxation bounds via rounding down. The result allows for a generalization, with a less strict requirement for the edge weights that we present next.

Proposition 5.5.1. *Let $G = (V, E, w)$ be a weighted undirected graph, with $w \in \mathbb{Z}^{|E|}$. If all vertices in $G' = G[E_{odd}]$, where $E_{odd} \subseteq E$ contains all edges with odd weight, have even degree, then every cut in G has even value.*

Proof. We show that every cut in G' and every cut in $G \setminus G'$ has even value. This already implies that all cuts in G have even value, as the cut value of any vertex-bipartition in G, is the sum of the cut value of the same bipartition in G' and $G \setminus G'$. If both values are even, the result follows immediately.

All cuts in $G \setminus G'$ have even value, as by definition of E_{odd}, all edges in $G \setminus G'$ have even weight and the sum over even values is always even.

For G' we show, that every cut in G' has even cardinality. As a direct consequence, every cut in G' has even value, because the sum over even many odd numbers is even. Let δ be any cut in G' with odd cardinality and (S_1, S_2) be a bipartition inducing δ. Clearly, $\sum_{u \in S_1} \text{degree}(u)$ is even (all degrees in G' are even). The difference between $|\delta|$ and $\sum_{u \in S_1} \text{degree}(u)$ has to be even as well, as all edges incident to vertices in S_1 but not in δ have been counted twice in the sum over all degrees. If these two values are always even, a cut δ with $|\delta| \mod 2 = 1$ can not exist in G'. □

Proposition 5.5.1 is especially relevant for e.g. some of the torus graphs from our benchmark set, where all vertices have degree 4 or 6 and edge weights are in $\{-1, 1\}$.

5.6. Experimental Evaluation

We now put the improvements and concepts from the previous sections to the test, and explore the actual performance of our solver and benchmark the impact of its different components (compare Figure 5.1). To this end, we first describe some implementation details and the hardware that we used for our evaluation. We then highlight the general performance of our solver compared to similar software. Finally, we performed an ablation study to analyze the different components in detail.

5.6.1. Implementation Details and Setup

Our implementation integrates multiple open-source libraries. For graph algorithms, we build on NetworKIT (Staudt et al. 2016), for a simple and reproducible experiment pipeline, we employ simexpal (Angriman et al. 2019). The MIP solver our project is based on is SCIP 9.1. (Bolusani et al. 2024) combined with the commercial solver CPLEX 22.1. (Cplex 2022) as the lp-solver. For the primal heuristic of Burer et al. (2002) our project incorporates the code developed by Dunning et al. (2018). All of these libraries (apart from CPLEX) are published under free and open-source licenses. For non-scientific libraries in use, we refer to the source code of our project.

ILP Model Our branch-and-cut algorithm is based on our root-triangulated model (5.3) from Section 5.1. We chose this model because of the promising novel results from the previous sections on the interplay of the `(RT)` model and odd-cycle and clique inequalities, and branching rules.

Branch-and-Cut Algorithm Our solver is based on SCIP (Bolusani et al. 2024), which is highly configurable. We turn off many general purpose modules of SCIP, specifically: General purpose primal heuristics, branching rules, cutting planes, restarts, and general presolving. All of theses would either require modifications to our new modules or turned out to be non-beneficial for solving times anyways (as we will see in a moment). The integrality of all solutions is checked on the original graph (see Section 5.5). In case our presolving introduces fractional weights, even if all optimal solutions have integer values, we inform SCIP accordingly. Before handing over the input to our solver, we employ the presolving of Chapter 4. For primal heuristics, we call the spanning tree heuristic after each separation round. We employ the heuristic of Burer et al. (2002) once at the root node with a timelimit of 2 seconds. Our implementation of the improvement heuristic of Kernighan and Lin (1970) is called after each branch-and-bound node. For branching, we use our new rule degree-dynamic (Algorithm 4) by default. Our solver runs on one thread only, as parallelizing MIP-solvers is non-trivial and out of scope for our investigation. Details of our separation procedure to augment the initial `(RT)` model with cutting planes are visualized in Figure 5.8.

Hardware All experiments were carried out on servers equipped with an AMD EPYC 7543P 32-Core CPU[2], 256GB of RAM and have Ubuntu 22.04 installed. Our code is written in C++20 and compiled with GCC 11.4. All code is free and open source software, publicly available at github[3] and archived by Zenodo (Charfreitag et al. 2024b).

Seeding In all of our following experiments, we run each one at least five times with different seeds, to compensate for performance variability (see Section 2.5). We treat each seed-instance pair as a separate data point.

[2] `https://www.amd.com/en/products/processors/server/epyc/7003-series/amd-epyc-7543p.html`
[3] `https://github.com/CharJon/SMS`

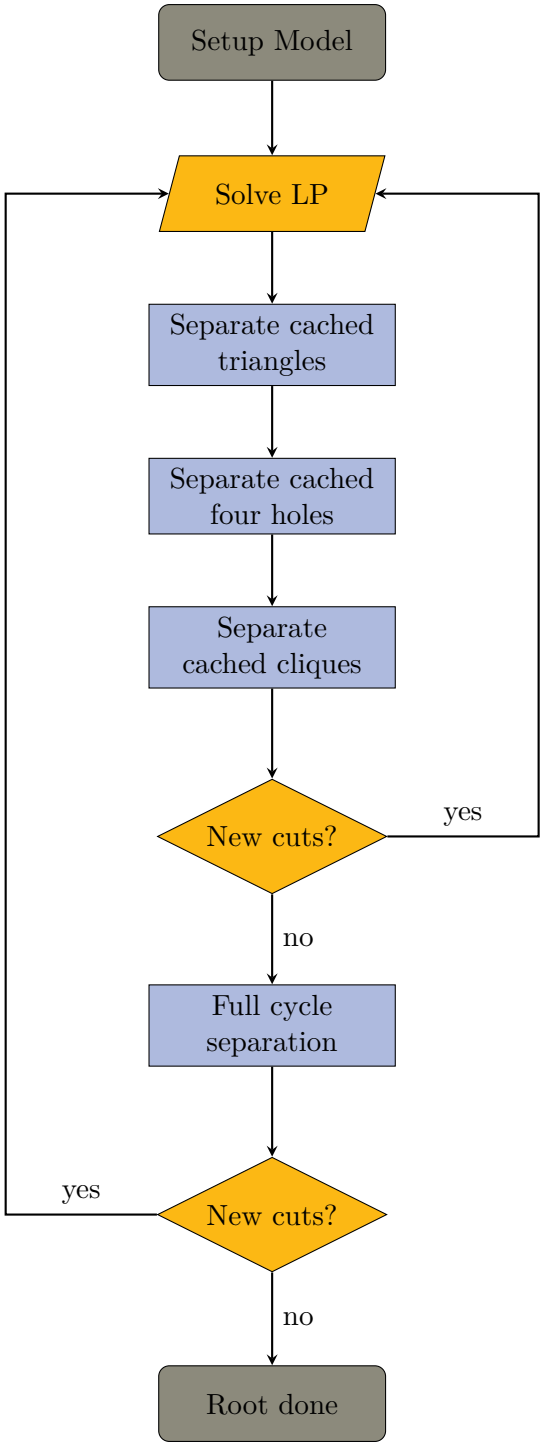


Figure 5.8.: The core of our cutting plane separation at the root node.

Instance	SMS		QuBowl		Gurobi	
	bab nodes	time [s]	bab nodes	time [s]	bab nodes	time [s]
pm1s_100_3	**273**	16.5	741	48.0	10 210	97.5
pw01_100_0	**70**	7.5	179	8.5	2 244	15.7
mannino_k487b	2	4.7	15	4.3	850	7.4
mannino_k487c	63	232.4	not avail.	>3 600.0	921	172.4
bio-diseasome	2	0.3	**1**	0.6	462	1.7
ca-netscience	**1**	0.3	**1**	0.0	26	0.4
g000981	**1**	0.0	**1**	0.0	**1**	0.1
imgseg_138032	**1**	0.2	**1**	3.9	741	34.3
gka7a	**1**	0.0	**1**	0.0	**1**	0.1
gka2c	**1**	0.3	**1**	0.3	441	0.2
gka5c	**1**	0.1	**1**	0.1	**1**	0.2
gka4d	38	43.4	**9**	43.7	853	52.2
be120_3_5	40	54.2	**15**	46.6	1 434	108.6
be250_3	79	166.3	**47**	150.7	>11 996	>3600.0
bqp250-3	**13**	56.6	17	84.1	10 819	3422.5

Table 5.1.: Comparison of our new solver (SMS) with the solvers QuBowl (Rehfeldt et al. 2023) and Gurobi 11 (Gurobi Optimization, LLC 2024). Results are shifted geometric means (shift is one) on our hardware, apart from those for QuBowl; they are copied from the corresponding paper (therefore runtimes are not directly comparable).

5.6.2. General Performance

To first present the general performance of our project, Table 5.1 shows a comparison with other solvers: The general purpose solver Gurobi and the MaxCut specific solver QuBowl of Rehfeldt et al. (2023). All MaxCut instances (upper half of the table) and QUBO instances (lower half of the table) are introduced in Section 3.4. We chose these instances because Rehfeldt et al. (2023) report detailed results of their state-of-the-art closed-source solver on them, and they form a diverse subset of instances from our benchmark set. The runtimes for QuBowl are not directly comparable, as this solver was benchmarked on different and potentially slower hardware (single threaded about 22 % according to benchmarks[4]). In general, QuBowl and SMS perform similarly for easy instances. When hardware is taken into account, QuBowl is generally faster for QUBO instances. For the MaxCut instances, the picture is more diverse: On pm1s_100_3 and pw01_100_0 SMS creates significantly fewer branch-and-bound nodes and this results at least for pm1s_100_3 in a runtime difference much larger than hardware differences can explain. The same holds true for mannino_k487c. Here, the run-time difference is even larger (one order of magnitude), which is clearly above the threshold for differences based on the hardware. The speedup by a factor of 30 for imgseg is due to better and possibly faster presolving, discussed in Chapter 4. In the following, we

[4] `https://www.cpubenchmark.net/singleThread.html#server-thread`

analyze which components contribute to the overall good performance in detail. For this, we first examine the influence of our new custom branching rule, which could be responsible for the smaller number of branch-and-bound nodes. Up to now, the influence of primal heuristics has not been discussed in detail for state-of-the-art solvers. We fill this gap by analyzing their influence in the context of our solver. Lastly, we investigate the influence of our cutting plane separation strategies, which helps with the fast solution times for instances like mannino_k487c.

5.6.3. Cutting Planes

The `(RT)` model (5.3), on which our solver is based, is compact and in theory does not require the separation of cutting planes. Nevertheless, we opt for their addition, as Rehfeldt et al. (2023) did for their state-of-the-art solver QuBowl and others in previous work (Barahona et al. 1988; De Simone et al. 1996; Liers 2004; Bonato 2011). In preliminary experiments, we observed that not adding cutting planes at all makes the solver slower by orders of magnitude. Although we separate the same types of constraints, there are still two main differences between our solver and recent projects:

1. We not only calculate and cache triangles, but also chordless 4 cycles, as already suggested by Barahona et al. (1988). For the fast generation of facet-inducing sparse inequalities, we then loop over the cached short chordless cycle in each separation round.

2. We consider cliques of arbitrary size for the separation of clique-inequalities. Other recent solvers only consider small cliques, e.g. QuBowl (Rehfeldt et al. 2023) employs an upper limit of 9.

Caching short chordless cycles (1. from above) is a small modification (recall that using the exact separation routine for odd-cycle constraints finds the same cycles in theory), but it might have a strong impact on overall performance, especially on grid graphs. We therefore further analyze whether this caching is helpful by comparing our solver in default settings with the solver when the module for 4-holes is turned off. For most of the instances from the medium, mannino, torus, and er set, the default setting is slightly faster (around 10 % for the shifted geometric mean over five seeds). Only for the pw01 set and the mannino_k487c instance the solver becomes slightly slower (slowdown of around 10 % for the shifted geometric mean over five seeds). However, a significant benefit can be observed for the t2pm set; see Figure 5.9. Here, explicitly considering chordless cycles of length four cuts solving times in half on average. This is not surprising, as these instances consist of 2D-grids and therefore the shortest chordless cycles in these instances have length four and are fundamental to their structure. It seems like only running the general separation of odd-cycle cycle inequalities misses some of these facet-inducing cutting planes. We conclude that the 4-hole module is helpful most of the time and especially useful if the input is of grid structure.

Considering cliques of arbitrary size (difference 2. from above) might be especially interesting in the context of the mannino instances. Table 5.1 showed a speedup of at

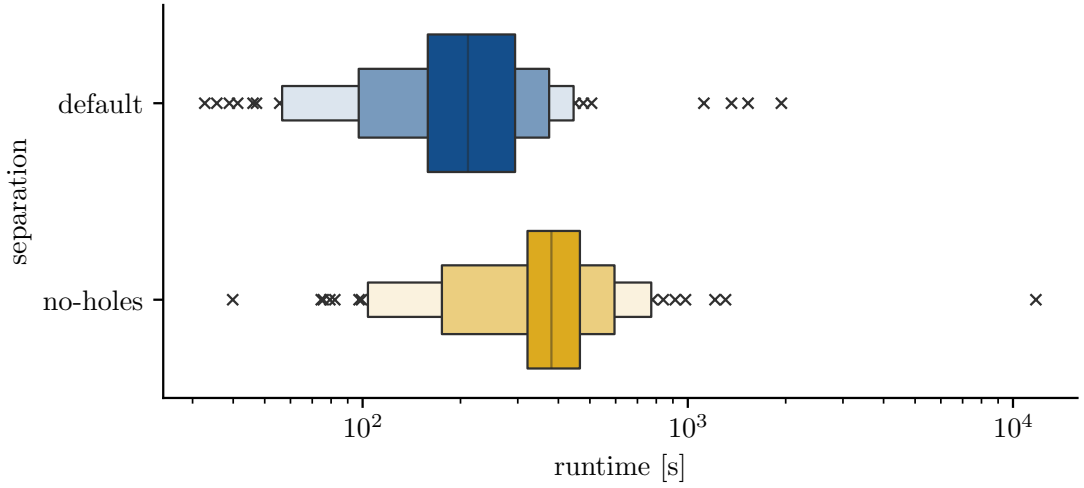


Figure 5.9.: Runtime of our solver for the 10 instances from the t2pm set. With and without the explicit separation of chordless cycles of length 4. All results are aggregated over ten seeds per instance. The width of each box indicates the 50th, 75th, etc. percentile.

least one order of magnitude for our solver on mannino_k487c when compared with QuBowl. As this instance has the most cliques of all graphs in our benchmark set, we investigate how big of an influence the clique-based cutting planes have. When limiting the size of the subcliques to consider to 11, the runtime increases significantly. None of the five seeded runs we tested was able to solve the instance to optimality in less than 3600 seconds. This clearly shows the importance of a well-tuned separation of clique-based inequalities for these types of real-world graphs.

5.6.4. Primal Heuristics

The compactness of the (RT) model (5.3) makes general-purpose primal heuristics applicable; any solution with integer values only is primal feasible. Hence, MIP solvers like SCIP can use their default heuristics to find good primal solutions. Therefore, it is not directly clear whether or not the problem-specific heuristics introduced in Section 5.4 offer any benefit to the overall runtime of our solver. This is what we want to investigate in this section. The state-of-the-art solver QuBowl of Rehfeldt et al. (2023) includes versions of the same heuristics discussed in Section 5.4, but the corresponding experimental study does not investigate the influence of the primal heuristics in detail.

When presolving already solves the instance to optimality, shrinks it to trivial size, or the LP relaxation at the root node is integer, primal heuristics are of less benefit. Thus, we benchmark the solver on instances regularly requiring multiple branch-and-bound nodes (excluding instances from our easy and ising set) and compare two settings: 1) Our solver in default settings: All of SCIPs primal heuristics turned off and only our

		runtime [s]			bab nodes		
set	parameter	min	max	sgm	min	max	sgm
mannino	default	2.28	356.50	10.93	1	297	5.58
	scip-heur-only	1.70	336.09	9.70	1	241	6.06
medium	default	0.17	49.64	2.86	1	69	2.79
	scip-heur-only	0.21	116.17	4.51	1	67	2.81
pm1s	default	4.33	18.23	9.43	27	363	103.54
	scip-heur-only	2.46	21.36	9.25	31	404	116.14
pw01	default	4.34	11.61	7.24	13	195	62.37
	scip-heur-only	2.81	11.21	6.34	14	239	70.02
w01	default	1.09	10.54	4.10	1	155	20.10
	scip-heur-only	1.22	11.50	2.81	1	256	21.95
t2g	default	0.01	2.13	0.17	1	3	1.09
	scip-heur-only	0.02	0.18	0.09	1	3	1.03
t3g	default	0.12	13.60	1.20	1	79	2.50
	scip-heur-only	0.13	22.56	1.17	1	142	2.61
t2pm	default	32.71	1 939.75	202.64	1	1 126	12.44
	scip-heur-only	71.40	1 612.67	342.49	1	833	15.64
t3pm	default	2.76	22.45	8.40	1	107	13.63
	scip-heur-only	6.40	31.11	14.21	7	150	36.52

Table 5.2.: Comparison of the performance of our solver in default settings and our solver when only SCIPs general purpose primal heuristics are active. The sgm colum refers to the shifted geometric mean (shift is 1) and all results are aggregated per instance set over 5 seeds per instance.

implementation of the spanning tree heuristic of Barahona (1983), the improvement heuristic of Kernighan and Lin (1970) and the heuristic of Burer et al. (2002) from the library of Dunning et al. (2018). 2) Our solver, but all default general purpose primal heuristics part of SCIP are turned on and our MaxCut specific heuristics are turned off.

Table 5.2 shows the results. For the Erdős–Rényi graphs (pm1s, pw01, w01) the runtime slightly increases when we only employ our primal heuristics. However, the number of branch-and-bound nodes for these instances is smaller in our default setting. This might indicate that our setting of running the heuristic of Burer et al. (2002) at the root node for exactly 2 s is not fine-tuned for these types instances (that get solved in between 1 to 22 seconds range). For the two torus sets t2pm and t3pm our MaxCut specific heuristics turn out extremely helpful. The shifted geometric mean of the runtimes decreases by close to 50 %. For real-world instances in the medium set, we also see a significant reduction in running times.

5.6.5. Branching

To evaluate the performance of our new branching rules degree-static and degree-dynamic (Algorithm 4), we aim to answer the following question: How do these two perform, compared to general-purpose branching rules (SCIPs default branching rule and SCIPs implementation of pseudocost branching) and the other MaxCut specific one of Nguyen and Minoux (2021)?

Clearly, the comparison only makes sense for instances where the solver creates multiple branch-and-bound nodes, and thus we again leave out the easy and ising set.

For the rule of Nguyen and Minoux (2021), we can draw a compact conclusion: For their implementation, CPLEX 12.7 performs branching decisions until there exists a variable that corresponds to a vertex that has more neighbors with integer value (in the current LP relaxation) than the average vertex degree. By default, CPLEX is assumed to perform some kind of reliability pseudocost branching (the code of this commercial solver is closed source). If we leave the branching to SCIPs version of this rule, our solver produces branch-and-bound trees much smaller than in the study of Nguyen and Minoux (2021) and the condition for their branching rule is never met. Hence, we only compare our rules against SCIPs general-purpose branching in the following.

Table 5.3 shows the runtime and the number of branch-and-bound nodes for our solver, when either one of our or SCIPs general-purpose branching rules is enabled on the instance sets requiring a relevant number of branch-and-bound nodes. As is to be expected, SCIPs reliability pseudocost branching always creates the smallest number of branch-and-bound nodes. But the extra time invested into the strong-branching lookahead does not always pay off; often it is better to just stick to pure pseudocosts. The MaxCut specific rules are always the better choice (considering the shifted geometric mean of the runtimes). For the three Erdös-Renyi instance sets (pm1s, pw01, w01), there is little difference between degree-dynamic and degree-static. But for the most difficult set of torus instances (t2pm), we see our new rule degree-dynamic outperforming the degree-static rule. The solving times for these instances fluctuated the most, so we present them in more detail in Figure 5.10. Pure pseudocost branching performed extremely poorly, hence we did not include it in this plot.

As all vertices have a degree of four in these 2D torus graphs, degree-static degenerates to random branching. The fact that it often outperforms reliability pseudocost branching in runtime shows that most variables (with a fractional value in the current lp solution) are good branching candidates, and expensive lookaheads are rarely worth their runtime. Still, degree-static produces outliers that require significantly more branch-and-bound nodes. By employing degree-dynamic instead of degree-static the number of outliers can be reduced. The mean runtime drops significantly (33 %) and even the shifted geometric mean of runtimes decreases by about 6 % (compare Table 5.3).

The degree-dynamic branching rule leads to fewer ties than degree-static, but for some graphs, there are still many equal branching candidates. We also experimented with tie breaking based on SCIPs pseudocost branching, but this led to worse performance overall.

		runtime [s]				bab nodes			
set	parameter	min	max	sgm	mean	min	max	sgm	mean
mannino	deg-dynamic	2.28	356.50	10.93	62.28	1	297	5.58	25.70
	deg-static	2.29	339.31	10.88	61.55	1	415	5.28	31.85
	pscost	2.29	1 732.56	14.43	189.62	1	1 650	10.96	137.45
	relpscost	2.29	615.85	12.51	98.16	1	212	3.14	15.50
medium	deg-dynamic	0.17	49.64	2.86	9.34	1	69	2.79	7.40
	deg-static	0.17	49.43	2.86	9.31	1	91	2.87	8.69
	pscost	0.17	49.87	2.85	9.27	1	93	2.83	8.11
	relpscost	0.16	49.56	2.86	9.28	1	20	2.40	4.16
pm1s	deg-dynamic	4.33	18.23	9.43	10.34	27	363	103.54	137.92
	deg-static	4.41	20.27	9.70	10.72	29	413	112.07	153.72
	pscost	4.81	39.69	14.90	17.57	39	921	217.70	325.16
	relpscost	9.37	32.20	20.00	21.16	7	351	78.01	132.88
pw01	deg-dynamic	4.34	11.61	7.24	7.46	13	195	62.37	75.12
	deg-static	4.35	11.90	7.15	7.35	13	189	61.22	73.16
	pscost	4.45	22.36	10.23	11.04	15	505	116.31	152.20
	relpscost	6.78	21.66	15.22	15.92	3	151	33.53	50.04
w01	deg-dynamic	1.09	10.54	4.10	4.47	1	155	20.10	34.92
	deg-static	1.09	10.36	4.08	4.45	1	183	20.02	35.88
	pscost	1.09	19.39	4.72	5.53	1	439	30.13	63.68
	relpscost	1.09	18.44	7.16	8.35	1	119	7.10	16.36
t2g	deg-dynamic	0.01	2.13	0.17	0.25	1	3	1.09	1.13
	deg-static	0.01	2.14	0.17	0.25	1	3	1.09	1.13
	pscost	0.01	2.14	0.17	0.25	1	3	1.09	1.13
	relpscost	0.01	2.13	0.17	0.25	1	1	1.00	1.00
t3g	deg-dynamic	0.12	13.60	1.20	2.24	1	79	2.50	9.00
	deg-static	0.12	13.65	1.20	2.22	1	75	2.61	9.53
	pscost	0.12	14.30	1.22	2.35	1	73	2.54	8.82
	relpscost	0.12	31.22	1.49	4.34	1	51	2.00	5.33
t2pm	deg-dynamic	32.71	1 939.75	202.64	268.12	1	1 126	12.44	57.47
	deg-static	33.13	11 536.18	216.57	401.08	1	10 776	14.47	190.76
	pscost	33.12	17 019.63	433.58	1 365.85	1	11 962	74.27	741.70
	relpscost	33.12	1 078.78	252.06	313.36	1	228	6.67	15.82
t3pm	deg-dynamic	2.76	22.45	8.40	9.98	1	107	13.63	29.93
	deg-static	2.76	22.39	8.48	10.09	1	107	14.42	32.33
	pscost	2.78	20.96	9.06	10.88	1	87	15.55	33.93
	relpscost	2.79	33.92	13.82	18.89	1	37	6.32	11.00

Table 5.3.: Comparison of runtime and number of branch and bound nodes of our solver when paired with different branching rules. Aggregated results for each benchmark sets are for five random seeds per instance and sgm refers to the shifted geometric mean (with a shift of 1).

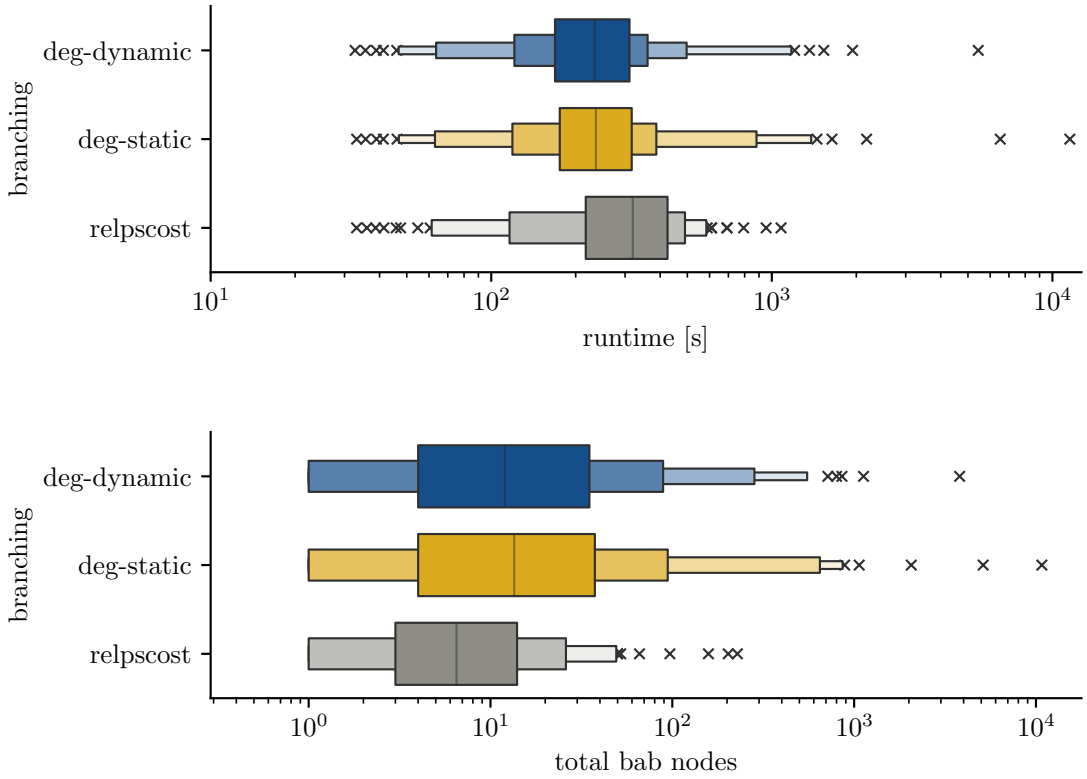


Figure 5.10.: Comparison of runtime and number of branch and bound nodes of our solver when paired with different branching rules on the ten instances from the t2pm set, each solved with 10 different seeds. Top: Runtime of our solver in seconds, logarithmic scale. Bottom: Number of branch-and-bound nodes, logarithmic scale. The width of each box indicates the 50th, 75th, etc. percentile.

5.7. Conclusion

We presented novel theoretical results for integer programming for MaxCut and introduced a new exact MaxCut solver that allowed us to benchmark the impact of our new techniques in practice. In general, the solvers performance is competitive to the state of the art, giving empirical evidence that our new concepts are helpful.

The refined integer programming model for MaxCut, `(RT)`, is compact and pairs well with existing techniques for branch and cut, and we based our solver on this model.

Conveniently, established algorithms for the generation of cutting planes can be applied to the `(RT)` model and our theoretical results reveal tuning potential for them. Fixed variables (by e.g. branching decisions) in this model can be ignored for the separation of odd-cycle inequalities, resulting in less work.

Additionally, the model also has implications for branching rules themselves. In gen-

eral, our model allows for branching on variables corresponding to vertices or variables corresponding to edges, but our results give theoretical evidence that branching on variables corresponding to vertices can be beneficial. For graphs that can be solved fast (in less than 1 minute), our degree-dynamic and degree-static rules perform very similarly and outperform two state-of-the-art branching heuristics (pseudocost branching and reliability pseudocost branching). The 2D torus instances from statistical physics in our benchmark set turn out to be the most difficult instances when it comes to calculating optimal solutions. For them, our new rule degree-dynamic performs best on average as it has fewer outliers than our degree-static rule.

In our analysis of the impact of primal heuristics on our solver, we saw that the solver clearly benefits from fine-tuned MaxCut specific heuristics.

6. Conclusion and Future Work

In this chapter, we draw a brief conclusion of the results presented in this thesis and discuss open problems and possibly fruitful future directions for research.

Conclusion The diverse real-world applications of MaxCut and its special combinatorial structure have led to the development of many different theoretical algorithms and practical solvers in the last decades.

Often real-world input has structure that can be exploited to reduce its size without sacrificing optimal solutions. This process is called presolving, consists of two techniques (data reduction and decomposition) and is generally considered one of the most crucial components for fast integer programming solvers (Achterberg et al. 2020). We suggested categorizing data reduction for MaxCut into three main groups: Reductions based on edge separators, vertex separators, and vertex similarity. For each of these categories, we presented new reduction rules. Those for vertex separators come from a new and unified framework. The most potent rules exploit vertex separators of sizes two and three.

One type of exact MaxCut algorithm from the literature is based on integer programming and branch-and-cut. Almost 40 years after the seminal ILP-based techniques for the MaxCut problem were introduced by Barahona and Mahjoub (1986), they still form the foundation for modern solvers and allow for new theory and engineering. We discussed a refined model and its implications for cutting plane generation and branching decisions. These implications gave intuition for new MaxCut specific branching rules. By revisiting the separation algorithms for well-known cutting planes, we also uncovered advanced engineering techniques.

Each of our new techniques showed promising performance when integrated into our solver. For real-world instances our presolving allows for speed-ups of up to one order of magnitude when compared against the state of the art. For a real-world instance, on which our presolving only slightly outperformed earlier approaches, our fine-tuned separation of certain clique inequalities allowed for a significant reduction in solving time. Our analysis of MaxCut specific primal heuristics from the literature gave new and explicit empirical evidence on their good performance; their inclusion in the solver resulted in speedups between a factor of two and three. And finally, our MaxCut specific branching rules always outperformed the general purpose once.

Outlook and Future Work Our MaxCut solver was well suited to test our new techniques; its performance is comparable to the state-of-the-art solver QuBowl of (Rehfeldt et al. 2023), allowing evaluation of the different components in a reasonable setting. To further improve its practical performance, the parallelization techniques Rehfeldt et al. (2023) might be promising.

We implemented special algorithms that allow us to solve MaxCut for certain classes of graphs to optimality in polynomial time. The literature offers some more algorithms of this type, e.g. for planar graphs, and adding them to the solver would clearly expand its capabilities.

Also, MaxCut presolving might benefit from revisiting presolving for quadratic unconstrained binary optimization and unconstrained pseudo-boolean optimization (and vice versa). Although these problems allow for structure-preserving transformations into each other, it is not directly obvious whether all techniques from one branch of the literature have been explored in the others.

Finally, we want to stress the importance of real-world data and good instance libraries for algorithm engineering. By offering our McSparse solver as a web-service, we hope to collect many instances interesting for practitioners. A good next step would be to analyze these instances, extract a diverse set, and make them publicly available as a part of some (possibly new) instance library. Ideally, this library should be enriched with structural insight into the instances contained. Including nontrivial graph properties, such as biconnectivity or planarity. In addition, adding presolved versions of instances to the library might be helpful to allow other researchers to experiment with hard instances, without the need to reimplement sophisticated presolving algorithms.

Bibliography

Abu-Khzam, Faisal N., Sebastian Lamm, Matthias Mnich, Alexander Noe, Christian Schulz, and Darren Strash (2022). "Recent Advances in Practical Data Reduction". In: *Algorithms for Big Data: DFG Priority Program 1736*. Ed. by Hannah Bast, Claudius Korzen, Ulrich Meyer, and Manuel Penschuck. Cham: Springer Nature Switzerland, pp. 97–133. ISBN: 978-3-031-21534-6. DOI: 10.1007/978-3-031-21534-6_6.

Achterberg, Tobias (2007). "Constraint Integer Programming". PhD thesis. Berlin Institute of Technology. ISBN: 978-3-89963-892-9. URL: http://opus.kobv.de/tuberlin/volltexte/2007/1611/.

Achterberg, Tobias, Robert E. Bixby, Zonghao Gu, Edward Rothberg, and Dieter Weninger (2020). "Presolve Reductions in Mixed Integer Programming". In: *INFORMS J. Comput.* 32.2, pp. 473–506. DOI: 10.1287/ijoc.2018.0857. URL: https://doi.org/10.1287/ijoc.2018.0857.

Achterberg, Tobias, Thorsten Koch, and Alexander Martin (2005). "Branching rules revisited". In: *Oper. Res. Lett.* 33.1, pp. 42–54. DOI: 10.1016/J.ORL.2004.04.002. URL: https://doi.org/10.1016/j.orl.2004.04.002.

Angriman, Eugenio, Alexander van der Grinten, Moritz von Looz, Henning Meyerhenke, Martin Nöllenburg, Maria Predari, and Charilaos Tzovas (2019). "Guidelines for Experimental Algorithmics: A Case Study in Network Analysis". In: *Algorithms* 12.7, p. 127. DOI: 10.3390/A12070127. URL: https://doi.org/10.3390/a12070127.

Applegate, David, Robert Bixby, Vašek Chvátal, and William Cook (1995). *Finding cuts in the TSP (A preliminary report)*. Tech. rep. DIMACS. URL: http://archive.dimacs.rutgers.edu/TechnicalReports/abstracts/1995/95-05.html.

Applegate, David L., Robert E. Bixby, Vašek Chvatál, and William J. Cook (2006). *The Traveling Salesman Problem: A Computational Study*. Princeton University Press. ISBN: 9780691129938. URL: http://www.jstor.org/stable/j.ctt7s8xg (visited on 12/16/2024).

Arbib, Claudio (1988). "A Polynomial Characterization of Some Graph Partitioning Problems". In: *Inf. Process. Lett.* 26.5, pp. 223–230. DOI: 10.1016/0020-0190(88)90144-5. URL: https://doi.org/10.1016/0020-0190(88)90144-5.

Aref, Samin and Zachary Neal (2020). "Detecting coalitions by optimally partitioning signed networks of political collaboration". In: *Scientific reports* 10.1, p. 1506. DOI: 10.1038/s41598-020-58471-z.

Armijo, Larry (1966). "Minimization of functions having Lipschitz continuous first partial derivatives". In: *Pacific Journal of Mathematics* 16.1, pp. 1–3. DOI: 10.2140/pjm.1966.16.1.

Barahona, Francisco (1983). "The Max-Cut Problem on Graphs Not Contractible to K5". In: *Oper. Res. Lett.* 2.3, pp. 107–111. ISSN: 0167-6377. DOI: 10.1016/0167-6377(83)90016-0.

Barahona, Francisco, Martin Grötschel, Michael Jünger, and Gerhard Reinelt (1988). "An Application of Combinatorial Optimization to Statistical Physics and Circuit Layout Design". In: *Oper. Res.* 36.3, pp. 493–513. DOI: 10.1287/opre.36.3.493.

Barahona, Francisco, Michael Jünger, and Gerhard Reinelt (1989). "Experiments in quadratic 0-1 programming". In: *Math. Program.* 44.1-3, pp. 127–137. DOI: 10.1007/BF01587084.

Barahona, Francisco and Ali Ridha Mahjoub (1986). "On the cut polytope". In: *Math. Program.* 36.2, pp. 157–173. DOI: 10.1007/BF02592023. URL: https://doi.org/10.1007/BF02592023.

Battista, Giuseppe Di and Roberto Tamassia (1990). "On-Line Graph Algorithms with SPQR-Trees". In: *Automata, Languages and Programming, 17th International Colloquium, ICALP90, Warwick University, England, UK, July 16-20, 1990, Proceedings*. Ed. by Mike Paterson. Vol. 443. Lecture Notes in Computer Science. Springer, pp. 598–611. DOI: 10.1007/BFB0032061. URL: https://doi.org/10.1007/BFb0032061.

Beasley, John E (1998). *Heuristic algorithms for the unconstrained binary quadratic programming problem*. Tech. rep. The Management School, Imperial College, London, England. URL: http://people.brunel.ac.uk/~mastjjb/jeb/bqp.pdf.

Bénichou, Michel, Jean-Michel Gauthier, Paul Girodet, Gerard Hentges, Gerard Ribière, and O. Vincent (1971). "Experiments in mixed-integer linear programming". In: *Math. Program.* 1.1, pp. 76–94. DOI: 10.1007/BF01584074. URL: https://doi.org/10.1007/BF01584074.

Berthold, Timo (2013). "Measuring the impact of primal heuristics". In: *Operations Research Letters* 41.6, pp. 611–614. DOI: 10.1016/J.ORL.2013.08.007. URL: https://doi.org/10.1016/j.orl.2013.08.007.

Billionnet, Alain and Sourour Elloumi (2007). "Using a Mixed Integer Quadratic Programming Solver for the Unconstrained Quadratic 0-1 Problem". In: *Math. Program.* 109.1, pp. 55–68. DOI: 10.1007/S10107-005-0637-9. URL: https://doi.org/10.1007/s10107-005-0637-9.

Binder, Kurt and A Peter Young (1986). "Spin glasses: Experimental facts, theoretical concepts, and open questions". In: *Reviews of Modern Physics* 58.4, p. 801. DOI: 10.1103/RevModPhys.58.801.

Bolusani, Suresh, Mathieu Besançon, Ksenia Bestuzheva, Antonia Chmiela, João Dionísio, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Mohammed Ghannam, Ambros Gleixner, Christoph Graczyk, Katrin Halbig, Ivo Hedtke, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Dominik Kamp, Thorsten Koch, Kevin Kofler, Jurgen Lentz, Julian Manns, Gioni Mexi, Erik Mühmer, Marc E. Pfetsch, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Mark Turner, Stefan Vigerske, Dieter Weninger, and Lixing Xu (2024). *The SCIP Optimization Suite 9.0*. DOI: 10.48550/arXiv.2402.17702.

Bonato, Thorsten (2011). "Contraction-based separation and lifting for solving the max-cut problem". PhD thesis. Universität Heidelberg. DOI: https://doi.org/10.11588/heidok.00012289.

Bonato, Thorsten, Michael Jünger, Gerhard Reinelt, and Giovanni Rinaldi (2014). "Lifting and separation procedures for the cut polytope". In: *Math. Program.* 146.1-2, pp. 351–378. DOI: 10.1007/S10107-013-0688-2. URL: https://doi.org/10.1007/s10107-013-0688-2.

Burer, Samuel, Renato D. C. Monteiro, and Yin Zhang (2002). "Rank-Two Relaxation Heuristics for MAX-CUT and Other Binary Quadratic Programs". In: *SIAM J. Optim.* 12.2, pp. 503–521. DOI: 10.1137/S1052623400382467. URL: https://doi.org/10.1137/S1052623400382467.

Carr, Robert D. and Goran Konjevod (2005). "Polyhedral Combinatorics". In: *Tutorials on Emerging Methodologies and Applications in Operations Research: Presented at Informs 2004, Denver, CO*. Ed. by Harvey J. Greenberg. New York, NY: Springer New York, pp. 2-1–2-46. ISBN: 978-0-387-22827-3. DOI: 10.1007/0-387-22827-6_2. URL: https://doi.org/10.1007/0-387-22827-6_2.

Charfreitag, Jonas, Christine Dahn, Michael Kaibel, Philip Mayer, Petra Mutzel, and Lukas Schürmann (2024a). "Separator Based Data Reduction for the Maximum Cut Problem". In: *22nd International Symposium on Experimental Algorithms, SEA 2024, July 23-26, 2024, Vienna, Austria*. Ed. by Leo Liberti. Vol. 301. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 4:1–4:21. DOI: 10.4230/LIPICS.SEA.2024.4.

Charfreitag, Jonas, Mohammed Ghannam, Konstantinos Papadopoulos, Claude Jordan, Michael Kaibel, and Franziska Kehe (2024b). *SMS: Sparse MaxCut Solver*. DOI: 10.5281/zenodo.14164842.

Charfreitag, Jonas, Michael Jünger, Sven Mallach, and Petra Mutzel (2022). "McSparse: Exact Solutions of Sparse Maximum Cut and Sparse Unconstrained Binary Quadratic Optimization Problems". In: *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2022, Alexandria, VA, USA, January 9-10, 2022*. Ed. by Cynthia A. Phillips and Bettina Speckmann. SIAM, pp. 54–66. DOI: 10.1137/1.9781611977042.5.

Charfreitag, Jonas, Sven Mallach, and Petra Mutzel (2023). "Integer Programming for the Maximum Cut Problem: A Refined Model and Implications for Branching". In: *SIAM Conference on Applied and Computational Discrete Algorithms, ACDA 2023, Seattle, WA, USA, May 31 - June 2, 2023*. Ed. by Jonathan W. Berry, David B. Shmoys, Lenore Cowen, and Uwe Naumann. SIAM, pp. 63–74. DOI: 10.1137/1.9781611977714.6.

Chimani, Markus, Christine Dahn, Martina Juhnke-Kubitzke, Nils M. Kriege, Petra Mutzel, and Alexander Nover (2020). "Maximum Cut Parameterized by Crossing Number". In: *J. Graph Algorithms Appl.* 24.3, pp. 155–170. DOI: 10.7155/JGAA.00523. URL: https://doi.org/10.7155/jgaa.00523.

Chimani, Markus, Carsten Gutwenger, Michael Jünger, Gunnar W Klau, Karsten Klein, and Petra Mutzel (2013). "The Open Graph Drawing Framework (OGDF)." In: *Handbook of graph drawing and visualization* 2011, pp. 543–569.

Chimani, Markus, Martina Juhnke-Kubitzke, Alexander Nover, and Tim Römer (2019). "Cut Polytopes of Minor-free Graphs". In: *CoRR* abs/1903.01817. DOI: 10.48550/arXiv.1903.01817. arXiv: 1903.01817.

Cplex, IBM ILOG (2022). *IBM ILOG CPLEX Optimization Studio 22.1.0*. URL: http://www.cplex.com.

Crowston, Robert, Mark Jones, and Matthias Mnich (2015). "Max-Cut Parameterized Above the Edwards-Erdős Bound". In: *Algorithmica* 72.3, pp. 734–757. DOI: 10.1007/S00453-014-9870-Z. URL: https://doi.org/10.1007/s00453-014-9870-z.

Dantzig, George B. (1957). "Discrete-Variable Extremum Problems". In: *Operations Research* 5.2, pp. 266–277. ISSN: 0030364X, 15265463. DOI: 10.1287/opre.5.2.266. (Visited on 04/09/2024).

De Simone, Caterina, Martin Diehl, Michael Jünger, Petra Mutzel, Gerhard Reinelt, and Giovanni Rinaldi (1995). "Exact ground states of Ising spin glasses: New experimental results with a branch-and-cut algorithm". In: *Journal of Statistical Physics* 80, pp. 487–496. DOI: 10.1007/BF02178370.

– (1996). "Exact ground states of two-dimensional ±J Ising spin glasses". In: *Journal of Statistical Physics* 84, pp. 1363–1371. DOI: 10.1007/BF02174135.

Dey, Santanu S. and Marco Molinaro (2018). "Theoretical challenges towards cutting-plane selection". In: *Math. Program.* 170.1, pp. 237–266. DOI: 10.1007/S10107-018-1302-4. URL: https://doi.org/10.1007/s10107-018-1302-4.

Deza, Michel Marie and Monique Laurent (1997). *Geometry of cuts and metrics*. Vol. 15. Algorithms and combinatorics. Springer. ISBN: 978-3-540-61611-5. DOI: 10.1007/978-3-642-04295-9.

Dijkstra, Edsger W. (1959). "A note on two problems in connexion with graphs". In: *Numerische Mathematik* 1, pp. 269–271. DOI: 10.1007/BF01386390. URL: https://doi.org/10.1007/BF01386390.

Dunning, Iain, Swati Gupta, and John Silberholz (2018). "What Works Best When? A Systematic Evaluation of Heuristics for Max-Cut and QUBO". In: *INFORMS Journal on Computing* 30.3, pp. 608–624. DOI: 10.1287/ijoc.2017.0798.

Elf, Matthias, Michael Jünger, and Giovanni Rinaldi (2003). "Minimizing breaks by maximizing cuts". In: *Oper. Res. Lett.* 31.3, pp. 343–349. DOI: 10.1016/S0167-6377(03)00025-7. URL: https://doi.org/10.1016/S0167-6377(03)00025-7.

Eppstein, David, Maarten Löffler, and Darren Strash (2013). "Listing All Maximal Cliques in Large Sparse Real-World Graphs". In: *ACM J. Exp. Algorithmics* 18. DOI: 10.1145/2543629. URL: https://doi.org/10.1145/2543629.

Erdős, Paul and Alfréd Rényi (1960). "On the evolution of random graphs". In: *Publ. Math. Inst. Hung. Acad. Sci* 5.1, pp. 17–60.

Esfahanian, Abdol-Hossein and S. Louis Hakimi (1984). "On computing the connectivities of graphs and digraphs". In: *Networks* 14.2, pp. 355–366. DOI: 10.1002/net.3230140211. URL: https://doi.org/10.1002/net.3230140211.

Even, Shimon and Robert Endre Tarjan (1975). "Network Flow and Testing Graph Connectivity". In: *SIAM J. Comput.* 4.4, pp. 507–518. DOI: 10.1137/0204043. URL: https://doi.org/10.1137/0204043.

Facchetti, Giuseppe, Giovanni Iacono, and Claudio Altafini (2011). "Computing global structural balance in large-scale signed social networks". In: *Proceedings of the National Academy of Sciences* 108.52, pp. 20953–20958. DOI: 10.1073/pnas.1109521108. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.1109521108. URL: https://www.pnas.org/doi/abs/10.1073/pnas.1109521108.

Ferizovic, Damir, Demian Hespe, Sebastian Lamm, Matthias Mnich, Christian Schulz, and Darren Strash (2020). "Engineering Kernelization for Maximum Cut". In: *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2020, Salt Lake City, UT, USA, January 5-6, 2020*. Ed. by Guy E. Blelloch and Irene Finocchi. SIAM, pp. 27–41. DOI: 10.1137/1.9781611976007.3.

Flum, Jörg and Martin Grohe (2006). *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer. ISBN: 978-3-540-29952-3. DOI: 10.1007/3-540-29953-X.

Frangioni, Antonio, Andrea Lodi, and Giovanni Rinaldi (2005). "New approaches for optimizing over the semimetric polytope". In: *Math. Program.* 104.2-3, pp. 375–388. DOI: 10.1007/S10107-005-0620-5. URL: https://doi.org/10.1007/s10107-005-0620-5.

Furini, Fabio, Emiliano Traversi, Pietro Belotti, Antonio Frangioni, Ambros M. Gleixner, Nick Gould, Leo Liberti, Andrea Lodi, Ruth Misener, Hans D. Mittelmann, Nikolaos V. Sahinidis, Stefan Vigerske, and Angelika Wiegele (2019). "QPLIB: a library of quadratic programming instances". In: *Math. Program. Comput.* 11.2, pp. 237–265. DOI: 10.1007/S12532-018-0147-4. URL: https://doi.org/10.1007/s12532-018-0147-4.

Galil, Zvi (1980). "Finding the Vertex Connectivity of Graphs". In: *SIAM J. Comput.* 9.1, pp. 197–199. DOI: 10.1137/0209016. URL: https://doi.org/10.1137/0209016.

Galluccio, Anna, Martin Loebl, and Jan Vondrák (2001). "Optimization via enumeration: a new algorithm for the Max Cut Problem". In: *Math. Program.* 90.2, pp. 273–290. DOI: 10.1007/PL00011425. URL: https://doi.org/10.1007/PL00011425.

Gamrath, Gerald, Thorsten Koch, Alexander Martin, Matthias Miltenberger, and Dieter Weninger (2015). "Progress in presolving for mixed integer programming". In: *Math. Program. Comput.* 7.4, pp. 367–398. DOI: 10.1007/S12532-015-0083-5. URL: https://doi.org/10.1007/s12532-015-0083-5.

Gleixner, Ambros M., Leona Gottwald, and Alexander Hoen (2023). "PaPILO: A Parallel Presolving Library for Integer and Linear Optimization with Multiprecision Support". In: *INFORMS J. Comput.* 35.6, pp. 1329–1341. DOI: 10.1287/IJOC.2022.0171. URL: https://doi.org/10.1287/ijoc.2022.0171.

Glover, Fred, Gary A Kochenberger, and Bahram Alidaee (1998). "Adaptive memory tabu search for binary quadratic programs". In: *Management Science* 44.3, pp. 336–345. DOI: 10.1287/mnsc.44.3.336.

Goemans, Michel X. and David P. Williamson (1995). "Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming". In: *J. ACM* 42.6, pp. 1115–1145. DOI: 10.1145/227683.227684.

Goldberg, Andrew V. and Kostas Tsioutsiouliklis (2001). "Cut Tree Algorithms: An Experimental Study". In: *J. Algorithms* 38.1, pp. 51–83. DOI: 10.1006/JAGM.2000.1136. URL: https://doi.org/10.1006/jagm.2000.1136.

Gomory, R. E. and T. C. Hu (1961). "Multi-Terminal Network Flows". In: *Journal of the Society for Industrial and Applied Mathematics* 9.4, pp. 551–570. DOI: 10.1137/0109047. eprint: https://doi.org/10.1137/0109047.

Grötschel, Martin, László Lovász, and Alexander Schrijver (1981). "The ellipsoid method and its consequences in combinatorial optimization". In: *Comb.* 1.2, pp. 169–197. DOI: 10.1007/BF02579273. URL: https://doi.org/10.1007/BF02579273.

Grötschel, Martin and George L. Nemhauser (1984). "A polynomial algorithm for the max-cut problem on graphs without long odd cycles". In: *Math. Program.* 29.1, pp. 28–40. DOI: 10.1007/BF02591727. URL: https://doi.org/10.1007/BF02591727.

Guo, Jiong, Jens Gramm, Falk Hüffner, Rolf Niedermeier, and Sebastian Wernicke (2006). "Compression-based fixed-parameter algorithms for feedback vertex set and edge bipartization". In: *J. Comput. Syst. Sci.* 72.8, pp. 1386–1396. DOI: 10.1016/J.JCSS.2006.02.001. URL: https://doi.org/10.1016/j.jcss.2006.02.001.

Gurobi Optimization, LLC (2024). *Gurobi Optimizer Reference Manual*. URL: https://www.gurobi.com.

Gusfield, Dan (1990). "Very Simple Methods for All Pairs Network Flow Analysis". In: *SIAM J. Comput.* 19.1, pp. 143–155. DOI: 10.1137/0219009. URL: https://doi.org/10.1137/0219009.

Gusmeroli, Nicolò, Timotej Hrga, Borut Luzar, Janez Povh, Melanie Siebenhofer, and Angelika Wiegele (2022). "BiqBin: A Parallel Branch-and-bound Solver for Binary Quadratic Problems with Linear Constraints". In: *ACM Trans. Math. Softw.* 48.2, 15:1–15:31. DOI: 10.1145/3514039. URL: https://doi.org/10.1145/3514039.

Gutwenger, C. and P. Mutzel (2000). "A linear time implementation of SPQR-trees". In: *Graph Drawing, 8th International Symposium, GD 2000, Proceedings*. Ed. by J. Marks. Vol. 1984. Lecture Notes in Computer Science. Springer Verlag, pp. 77–90.

Hadlock, F. (1975). "Finding a Maximum Cut of a Planar Graph in Polynomial Time". In: *SIAM J. Comput.* 4.3, pp. 221–225. DOI: 10.1137/0204019.

Harary, Frank (1953). "On the notion of balance of a signed graph." In: *Michigan Mathematical Journal* 2.2, pp. 143–146.

Heider, Fritz (1946). "Attitudes and cognitive organization". In: *The Journal of psychology* 21.1, pp. 107–112.

Henzinger, Monika Rauch, Satish Rao, and Harold N. Gabow (2000). "Computing Vertex Connectivity: New Bounds from Old Techniques". In: *J. Algorithms* 34.2, pp. 222–250. DOI: 10.1006/jagm.1999.1055. URL: https://doi.org/10.1006/jagm.1999.1055.

Hochbaum, Dorit S. (1993). "Why Should Biconnected Components be Identified First". In: *Discret. Appl. Math.* 42.2, pp. 203–210. DOI: 10.1016/0166-218X(93)90046-Q.

Hopcroft, John E. and Robert Endre Tarjan (1973). "Dividing a Graph into Triconnected Components". In: *SIAM J. Comput.* 2.3, pp. 135–158. DOI: 10.1137/0202012. URL: https://doi.org/10.1137/0202012.

Hrga, Timotej and Janez Povh (2021). "MADAM: a parallel exact solver for max-cut based on semidefinite programming and ADMM". In: *Comput. Optim. Appl.* 80.2,

pp. 347–375. DOI: 10.1007/S10589-021-00310-6. URL: https://doi.org/10.1007/s10589-021-00310-6.

Hüffner, Falk, Nadja Betzler, and Rolf Niedermeier (2010). "Separator-based data reduction for signed graph balancing". In: *J. Comb. Optim.* 20.4, pp. 335–360. DOI: 10.1007/s10878-009-9212-2. URL: https://doi.org/10.1007/s10878-009-9212-2.

Jünger, Michael, Elisabeth Lobe, Petra Mutzel, Gerhard Reinelt, Franz Rendl, Giovanni Rinaldi, and Tobias Stollenwerk (2021). "Quantum Annealing versus Digital Computing: An Experimental Comparison". In: *ACM J. Exp. Algorithmics* 26, 1.9:1–1.9:30. DOI: 10.1145/3459606.

Jünger, Michael and Sven Mallach (2019). "Odd-Cycle Separation for Maximum Cut and Binary Quadratic Optimization". In: *27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany*. Ed. by Michael A. Bender, Ola Svensson, and Grzegorz Herman. Vol. 144. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 63:1–63:13. DOI: 10.4230/LIPICS.ESA.2019.63. URL: https://doi.org/10.4230/LIPIcs.ESA.2019.63.

Kanevsky, Arkady (1993). "Finding all minimum-size separating vertex sets in a graph". In: *Networks* 23.6, pp. 533–541. DOI: 10.1002/net.3230230604.

Kanevsky, Arkady and Vijaya Ramachandran (1991). "Improved Algorithms for Graph Four-Connectivity". In: *J. Comput. Syst. Sci.* 42.3, pp. 288–306. DOI: 10.1016/0022-0000(91)90004-O.

Karp, Richard M. (1972). "Reducibility Among Combinatorial Problems". In: *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*. Ed. by Raymond E. Miller and James W. Thatcher. The IBM Research Symposia Series. Plenum Press, New York, pp. 85–103. DOI: 10.1007/978-1-4684-2001-2_9. URL: https://doi.org/10.1007/978-1-4684-2001-2_9.

Kernighan, Brian W. and Shen Lin (1970). "An efficient heuristic procedure for partitioning graphs". In: *The Bell system technical journal* 49.2, pp. 291–307. DOI: 10.1002/J.1538-7305.1970.TB01770.X. URL: https://doi.org/10.1002/j.1538-7305.1970.tb01770.x.

King, Andrew D, Jack Raymond, Trevor Lanting, Richard Harris, Alex Zucca, Fabio Altomare, Andrew J Berkley, Kelly Boothby, Sara Ejtemaee, Colin Enderud, et al. (2023). "Quantum critical dynamics in a 5,000-qubit programmable spin glass". In: *Nature* 617.7959, pp. 61–66. DOI: 10.1038/s41586-023-05867-2.

Kochenberger, Gary A., Jin-Kao Hao, Fred W. Glover, Mark W. Lewis, Zhipeng Lü, Haibo Wang, and Yang Wang (2014). "The unconstrained binary quadratic programming problem: a survey". In: *J. Comb. Optim.* 28.1, pp. 58–81. DOI: 10.1007/S10878-014-9734-0. URL: https://doi.org/10.1007/s10878-014-9734-0.

Korte, Bernhard H and Jens Vygen (2018). *Combinatorial Optimization*. Sixth Edition. Vol. 21. Springer. ISBN: 9783662560396. DOI: https://doi.org/10.1007/978-3-662-56039-6.

Krislock, Nathan, Jérôme Malick, and Frédéric Roupin (2017). "BiqCrunch: A Semidefinite Branch-and-Bound Method for Solving Binary Quadratic Problems". In: *ACM*

Trans. Math. Softw. 43.4, 32:1–32:23. DOI: 10.1145/3005345. URL: https://doi.org/10.1145/3005345.

Kruskal, Joseph B. (1956). "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem". In: *Proceedings of the American Mathematical Society* 7.1, pp. 48–50. ISSN: 00029939, 10886826. URL: http://www.jstor.org/stable/2033241 (visited on 06/27/2024).

Lancia, Giuseppe and Paolo Serafini (2011). "An effective compact formulation of the max cut problem on sparse graphs". In: *Electron. Notes Discret. Math.* 37, pp. 111–116. DOI: 10.1016/J.ENDM.2011.05.020. URL: https://doi.org/10.1016/j.endm.2011.05.020.

Lange, Jan-Hendrik, Bjoern Andres, and Paul Swoboda (2019). "Combinatorial Persistency Criteria for Multicut and Max-Cut". In: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*. Computer Vision Foundation / IEEE, pp. 6093–6102. DOI: 10.1109/CVPR.2019.00625.

Li, Jason, Danupon Nanongkai, Debmalya Panigrahi, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai (2021). "Vertex connectivity in poly-logarithmic max-flows". In: *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*. Ed. by Samir Khuller and Virginia Vassilevska Williams. ACM, pp. 317–329. DOI: 10.1145/3406325.3451088. URL: https://doi.org/10.1145/3406325.3451088.

Liers, Frauke (2004). "Contributions to determining exact ground-states of Ising spin-glasses and to their physics". PhD thesis. Universität zu Köln. URL: http://kups.ub.uni-koeln.de/id/eprint/1308.

Liers, Frauke and G. Pardella (2012). "Partitioning planar graphs: a fast combinatorial approach for max-cut". In: *Comput. Optim. Appl.* 51.1, pp. 323–344. DOI: 10.1007/s10589-010-9335-5.

Lin, Shen and Brian W. Kernighan (1973). "An Effective Heuristic Algorithm for the Traveling-Salesman Problem". In: *Oper. Res.* 21.2, pp. 498–516. DOI: 10.1287/OPRE.21.2.498. URL: https://doi.org/10.1287/opre.21.2.498.

Lodi, Andrea and Andrea Tramontani (2013). "Performance variability in mixed-integer programming". In: *Theory driven by influential applications*. INFORMS, pp. 1–12. DOI: 10.1287/educ.2013.0112.

Madathil, Jayakrishnan, Saket Saurabh, and Meirav Zehavi (2020). "Fixed-Parameter Tractable Algorithm and Polynomial Kernel for Max-Cut Above Spanning Tree". In: *Theory Comput. Syst.* 64.1, pp. 62–100. DOI: 10.1007/S00224-018-09909-5. URL: https://doi.org/10.1007/s00224-018-09909-5.

Mallach, Sven (2021). *MaxCut and BQP Instance Library*. http://bqp.cs.uni-bonn.de/library/html/index.html.

– (2024). "A Family of Spanning-Tree Formulations for the Maximum Cut Problem". In: *Combinatorial Optimization - 8th International Symposium, ISCO 2024, La Laguna, Tenerife, Spain, May 22-24, 2024, Revised Selected Papers*. Ed. by Amitabh Basu, Ali Ridha Mahjoub, and Juan José Salazar González. Vol. 14594. Lecture Notes in Computer Science. Springer, pp. 43–55. DOI: 10.1007/978-3-031-60924-4_4. URL: https://doi.org/10.1007/978-3-031-60924-4_4.

McCormick, S. Thomas, M. R. Rao, and Giovanni Rinaldi (2003). "Easy and difficult objective functions for max cut". In: *Math. Program.* 94.2-3, pp. 459–466. DOI: 10.1007/s10107-002-0328-8.

Morrison, David R., Sheldon H. Jacobson, Jason J. Sauppe, and Edward C. Sewell (2016). "Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning". In: *Discret. Optim.* 19, pp. 79–102. DOI: 10.1016/J.DISOPT.2016.01.005. URL: https://doi.org/10.1016/j.disopt.2016.01.005.

Nguyen, Viet Hung and Michel Minoux (2021). "Linear size MIP formulation of Max-Cut: new properties, links with cycle inequalities and computational results". In: *Optim. Lett.* 15.4, pp. 1041–1060. DOI: 10.1007/S11590-020-01667-Z. URL: https://doi.org/10.1007/s11590-020-01667-z.

Padberg, Manfred and Giovanni Rinaldi (1991). "A Branch-and-Cut Algorithm for the Resolution of Large-Scale Symmetric Traveling Salesman Problems". In: *SIAM Rev.* 33.1, pp. 60–100. DOI: 10.1137/1033004. URL: https://doi.org/10.1137/1033004.

Papadimitriou, Christos H. and Mihalis Yannakakis (1991). "Optimization, Approximation, and Complexity Classes". In: *J. Comput. Syst. Sci.* 43.3, pp. 425–440. DOI: 10.1016/0022-0000(91)90023-X.

Pataki, Gabor and Stefan H. Schmieta (2000). *The DIMACS library of mixed semidefinite-quadratic-linear programs.* http://dimacs.rutgers.edu/archive/Challenges/Seventh/Instances/.

Pohl, Ira (1969). "Bi-directional and heuristic search in path problems". PhD thesis. Stanford University, USA. URL: https://searchworks.stanford.edu/view/2197829.

Poljak, Svatopluk and Zsolt Tuza (1993). "Maximum cuts and largest bipartite subgraphs". In: *Combinatorial Optimization, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, 1992/93.* Ed. by William J. Cook, László Lovász, and Paul D. Seymour. Vol. 20. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. DIMACS/AMS, pp. 181–244. DOI: 10.1090/DIMACS/020/04. URL: https://doi.org/10.1090/dimacs/020/04.

Polzin, Tobias and Siavash Vahdati Daneshmand (2006). "Practical Partitioning-Based Methods for the Steiner Problem". In: *Experimental Algorithms, 5th International Workshop, WEA 2006, Cala Galdana, Menorca, Spain, May 24-27, 2006, Proceedings.* Ed. by Carme Àlvarez and Maria J. Serna. Vol. 4007. Lecture Notes in Computer Science. Springer, pp. 241–252. DOI: 10.1007/11764298_22. URL: https://doi.org/10.1007/11764298_22.

Rehfeldt, Daniel, Thorsten Koch, and Yuji Shinano (2023). "Faster exact solution of sparse MaxCut and QUBO problems". In: *Mathematical Programming Computation*, pp. 1–26. DOI: 10.1007/s12532-023-00236-6.

Rendl, Franz, Giovanni Rinaldi, and Angelika Wiegele (2010). "Solving Max-Cut to optimality by intersecting semidefinite and polyhedral relaxations". In: *Math. Program.* 121.2, pp. 307–335. DOI: 10.1007/S10107-008-0235-8. URL: https://doi.org/10.1007/s10107-008-0235-8.

Rossi, Ryan A. and Nesreen K. Ahmed (2015). "The Network Data Repository with Interactive Graph Analytics and Visualization". In: *Proceedings of the Twenty-Ninth*

AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA. Ed. by Blai Bonet and Sven Koenig. AAAI Press, pp. 4292–4293. DOI: 10.1609/aaai.v29i1.9277.

Rother, Carsten, Vladimir Kolmogorov, Victor S. Lempitsky, and Martin Szummer (2007). "Optimizing Binary MRFs via Extended Roof Duality". In: *2007 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2007), 18-23 June 2007, Minneapolis, Minnesota, USA*. IEEE Computer Society. DOI: 10.1109/CVPR.2007.383203.

Schrijver, Alexander (2003). *Combinatorial optimization: polyhedra and efficiency*. Vol. 24. Springer. ISBN: 978-3-540-44389-6.

Shekhovtsov, Alexander (2014). "Maximum Persistency in Energy Minimization". In: *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*. IEEE Computer Society, pp. 1162–1169. DOI: 10.1109/CVPR.2014.152. URL: https://doi.org/10.1109/CVPR.2014.152.

Simone, Caterina De (1990). "The cut polytope and the Boolean quadric polytope". In: *Discret. Math.* 79.1, pp. 71–75. DOI: 10.1016/0012-365X(90)90056-N. URL: https://doi.org/10.1016/0012-365X(90)90056-N.

Staudt, Christian L., Aleksejs Sazonovs, and Henning Meyerhenke (2016). "NetworKit: A tool suite for large-scale complex network analysis". In: *Netw. Sci.* 4.4, pp. 508–530. DOI: 10.1017/NWS.2016.20. URL: https://doi.org/10.1017/nws.2016.20.

Taghavi, Mohammad H. and Paul H. Siegel (2008). "Adaptive Methods for Linear Programming Decoding". In: *IEEE Trans. Inf. Theory* 54.12, pp. 5396–5410. DOI: 10.1109/TIT.2008.2006384. URL: https://doi.org/10.1109/TIT.2008.2006384.

Tarabunga, Poetri Sonya and Claudio Castelnovo (May 2024). "Magic in generalized Rokhsar-Kivelson wavefunctions". In: *Quantum* 8, p. 1347. ISSN: 2521-327X. DOI: 10.22331/q-2024-05-14-1347. URL: https://doi.org/10.22331/q-2024-05-14-1347.

Tarjan, Robert Endre (1972). "Depth-First Search and Linear Graph Algorithms". In: *SIAM J. Comput.* 1.2, pp. 146–160. DOI: 10.1137/0201010. URL: https://doi.org/10.1137/0201010.

Wesselmann, Franz and Uwe Suhl (2012). *Implementing cutting plane management and selection techniques*. Tech. rep. University of Paderborn.

Wiegele, Angelika (2007). *Biq Mac Library – A collection of Max-Cut and quadratic 0-1 programming instances of medium size*. http://biqmac.aau.at/biqmaclib.html.

Williams, J. W. J. (1964). "Algorithm 232: Heapsort". In: *Communications of the ACM* 7.6, pp. 347–348. DOI: 10.1145%2F512274.512284.

Zhang, Xiaojie and Paul H. Siegel (2012). "Adaptive Cut Generation Algorithm for Improved Linear Programming Decoding of Binary Linear Codes". In: *IEEE Trans. Inf. Theory* 58.10, pp. 6581–6594. DOI: 10.1109/TIT.2012.2204955. URL: https://doi.org/10.1109/TIT.2012.2204955.

A. Appendix

A.1. File Formats

All files used for our MaxCut experiments were in ".mc" format. The McSparse web solver also accepts graphs in ".mc" format, as well as QUBO instances in ".bq" format. For spinglass submissions, McGroundstate uses the ".sg" and ".gsg" formats. We sketch the basics of the file formats here to allow for easier use of our solvers.

All file formats are based on pure files and allow for an arbitrary number of comments at the beginning of each file. The comment lines must be prefixed with "#".

mc The ".mc" file format describes a one-indexed simple undirected graph. The first non-comment line has the number of vertices (n) and the number of edges (m), separated by a single space character. After this line there are m additional lines, each describing one (unique) edge. An edge is a space-separated list of the form "vertex-ID vertex-ID weight", where the vertex-IDs are from $\{1, \dots n\}$.

bq The ".bq" file format describes a square matrix. The first non-comment line has the dimension (n) and the number of non-zero entries (m), separated by a single space character. After this line there are m additional lines, each describing one (unique) non-zero entry. A non-zero entry is of the form "row column entry", where row and column are from $\{1, \dots n\}$.

sg The ".sg" file format describes a spin glass. The first non-comment line has the number of spines (n) and the number of interactions (m), separated by a single space character. After this line, there are m additional lines, each describing one (unique) interaction. An interaction is a space-separated list of the form "spin-ID spin-ID interaction-strength", where the spin-IDs are from $\{1, \dots n\}$.

gsg The ".gsg" file format describes a square 2D or cubic 3D spin glass of size l. It inherits all properties from the ".sg" format, but additionally assumes all spins to be aligned in a 2D or 3D grid. Let $(d_1, d_2, d_3) \in \{1, l\}^3$ be the position of a spin in a 3D spin glass. The ID of each vertex must be $d_1 + (d_2 - 1) \cdot l + (d_3 - 1) \cdot l^2$.