Optimality-Preserving Data Reduction for Maximum k-Cut

Michael Kaibel[1]
University of Bonn, Germany

Petra Mutzel
University of Bonn, Germany
Lamarr Institute, Bonn, Germany

Abstract

Preprocessing has become an increasingly important part of solving Maximum Cut to optimality, enabling exact solvers to tackle significantly larger instances. This suggests that exact solvers for the more general Maximum k-Cut problem could also benefit from sophisticated preprocessing. However, to the best of our knowledge, no preprocessing techniques that are effective for $k > 2$ have been published.

In this paper, we introduce structured cut sets, a novel data reduction technique for Maximum k-Cut. We provide criteria under which deleting cut sets is optimality-preserving, yielding a decomposition into connected components that can be solved independently and whose solutions can be combined into an optimal solution for the original graph. Furthermore, we extend several preprocessing techniques from Maximum Cut to Maximum k-Cut. To show that our rules are optimality-preserving, we develop a new proof framework based on the addition of weighted graphs.

We complement our theoretical results by engineering a preprocessing framework for Maximum k-Cut and show its effectiveness in a computational study. The preprocessed instances are typically significantly smaller. Integrating our preprocessing into an exact solver yields significant speed-ups and enables solving more instances to optimality.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization; Theory of computation → Network optimization

Keywords and phrases Data Reduction, Preprocessing, Maximum k-Cut

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.21

Related Version *Full Version*: https://doi.org/10.48550/arXiv.2607.03281 [21]

Supplementary Material *Software (Source Code)*: https://doi.org/10.5281/zenodo.21256337 [22]

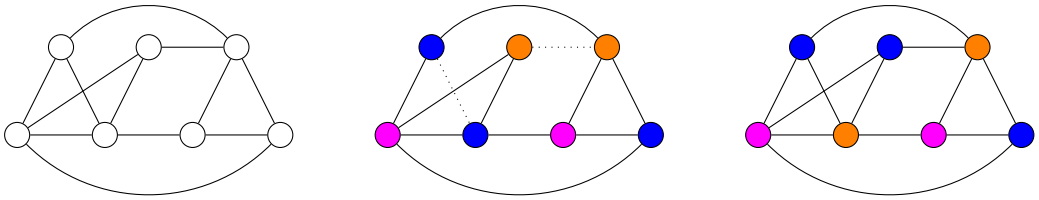
Funding This research was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under grant FOR-5361 – 459420781.

Acknowledgements The authors gratefully acknowledge the access to the Marvin cluster and the support provided by the High Performance Computing & Analytics Lab of the University of Bonn

1 Introduction

Graph partitioning problems are a fundamental class of optimization problems. A classic example is the Maximum k-Cut problem (Max k-Cut for short), which asks for a partition of the nodes of a graph into up to k subsets, maximizing the sum of weights of edges whose endpoints lie in different sets. An example is illustrated in Figure 1. The problem is $\mathcal{NP}$-hard, which can be shown with a reduction from the related k-Colouring problem, one of Karp's 21 $\mathcal{NP}$-complete problems [25].

[1] Corresponding author



(a) An instance. **(b)** A suboptimal solution. **(c)** An optimum solution.

Figure 1 An example of Maximum k-Cut on a unit weight graph with $k = 3$. All edges have weight 1. Partitions are encoded by colour.

Research into Max k-Cut has been motivated by its various applications, e.g., in computing relaxations of frequency assignment problems [10], chip load balancing [17, 38], and image reconstruction [8]. Special attention has been placed on the case $k = 2$, called Maximum Cut (MaxCut for short). Recent research has shown that preprocessing can significantly reduce the size of real world instances for MaxCut by locating substructures for which the behaviour of the optimum solution can be determined ahead of time [13, 29, 3, 26]. This preprocessing enables exact solvers to tackle significantly larger instances, sometimes even solving instances to optimality purely during preprocessing. The only published work on preprocessing for Max k-Cut that we know of is [12]. However, in their computational experiments their technique only had an effect for $k = 2$. The success of preprocessing for MaxCut suggests that Max k-Cut could still benefit from sophisticated preprocessing.

Our Contribution

We extensively study optimality-preserving preprocessing techniques for the Max k-Cut problem. In detail:

- We introduce structured cut sets, a new Max k-Cut data separation rule particularly effective for $k \geqslant 3$. The rule relies on novel criteria under which the problem reduces to independently solving the connected components obtained by deleting a cut set.
- Moreover, we generalize several of the MaxCut preprocessing rules from [3, 13, 29, 26].
- In order to prove that preprocessing rules for Max k-Cut are optimality-preserving, we introduce a new proof framework based on the addition of weighted graphs.
- We propose a preprocessing algorithm based on our theoretical findings, equipped with a framework for reconstructing optimal solutions.
- Our computational study shows that instances preprocessed by our algorithm are typically significantly smaller. When integrated into an exact solver, our preprocessing algorithm leads to substantial speed-ups and more instances solved to optimality.[2]

2 Preliminaries

We consider weighted, undirected graphs $G = (V, E, w)$ with a set of vertices $V, |V| = n$, edges $E \subseteq \binom{V}{2}, |E| = m$ and weights $w : E \to \mathbb{R}$. We denote by $\delta_G(v), N_G(v)$ and $d_G(v)$ the incident edges, neighbourhood and degree of $v \in V$ in G, by $w(S) = \sum_{e \in S} w(e)$ the weight of a set $S \subseteq E$ of edges and by $\delta_G(S) = \{\{s, t\} \mid s \in S, t \notin S\}, N_G(S) = \bigcup_{v \in S} N(v) \backslash S$ the cut set and neighbourhood of $S \subseteq V$. If the graph is clear from context, we omit the subscript.

[2] The code is publicly available under `https://github.com/mkaibel/MaxKCutPreprocessing`

We denote by $G[V'] = (V', \{e \in E \mid e \subseteq V'\}, w)$ the induced subgraph of a vertex set $V' \subseteq V$ and by $G[S] = (\bigcup_{e \in S} e, S, w)$ the induced subgraph of an edge set $S \subseteq E$. For $V' \subseteq V$ we denote by $\partial_G(V') = \{v \in V' \mid \exists w \in V \backslash V' : \{v, w\} \in E\}$ the boundary of V'.

▶ **Definition 1** (k-Partition). *Given a set M we call a function $p : M \to \{1, ..., k\}$ a k-partition of M, which assigns every element $m \in M$ to group $p(m)$.*
For a subset $S \subseteq M$ we denote by $p_{|S} : S \to \{1, ..., k\}$, $p_{|S} : x \mapsto p(x)$ the partition p constrained to the set S. We denote by $S_{p=i} = \{s \in S \mid p(s) = i\}$ the set of all elements in S assigned to group i.

We refer to $\{1, ..., k\}$ as colours and to $p(v)$ as the colour assigned to v by p.

▶ **Definition 2** (Permutations of Partitions). *Let $p : M \to \{1, ..., k\}$ be a k-partition and $\pi : \{1, ..., k\} \to \{1, ..., k\}$ a bijection. We denote by $p^\pi = \pi \circ p$ the permutation of p by π and, for a subset $S \subseteq M$, by $p^{\pi(S)}$ the partition obtained by permuting p with π on S, that is*

$$p^{\pi(S)}(x) = \begin{cases} \pi(p(x)) & \text{if } x \in S \\ p(x) & \text{otherwise} \end{cases}$$

We say two k-partitions p, p' are equivalent and write $p \cong p'$, iff $p^\pi = p'$ for a permutation π.

Partial permutations of partitions will play a major role later, as permuting the partition of a subset $S \subseteq V$ of the nodes only changes how our k-cut behaves on $\delta(S)$.

Now that we have formally defined partitions we can define k-cuts and the considered problem in this work:

▶ **Definition 3** (Maximum k-Cut). *Given a weighted undirected graph $G = (V, E, w)$ and a k-partition $p : V \to \{1, ..., k\}$ of the vertices we denote by*

$$\delta_G(p) = \{\{v, w\} \in E \mid p(v) \neq p(w)\}$$

the k-cut induced by p, that is the set of all edges such that their endpoints have different colours. We denote by $w(p) = w(\delta_G(p)) = \sum_{e \in \delta_G(p)} w(e)$ the weight of the k-cut induced by p. Max k-Cut is the problem of finding a k-partition p that maximizes $w(p)$.

Equivalent to Max k-Cut is the Minimum k-Partition problem, in which we minimize the weight of edges for which both endpoints have the same colour.

2.1 Related Work

As previously mentioned, on general graphs Max k-Cut is $\mathcal{NP}$-hard. Even though Max k-Cut can likely not be solved in polynomial time, various approaches that can solve real world instances to optimality in reasonable time have been proposed over the years. These are usually based on ILP formalisms [6, 1]. Different techniques to solve the ILP models have been proposed. The orbital fixing algorithm by Kaibel et. al [23] efficiently prunes symmetries in the assignment formulation. To obtain better dual bounds, both cutting planes [6, 7], and semidefinite relaxations [14, 2, 37, 30, 32, 31] have been employed. For a recent study on exact solvers, we refer to Rodrigues de Sousa et. al [31].

Several different approaches for preprocessing MaxCut have been developed. Lange et al. [26] introduced criteria that show that certain edges with high weight are cut and certain edges with high negative weight are not cut in an optimum solution. They exploited this knowledge to then contract the endpoints. Rules that enable removing or simplifying

substructures, such as unit weight cliques with a small boundary and induced 3-paths, were developed by Ferizovic et al. [13]. Recently, Charfreitag et al. [3] developed a framework than enables exploiting 2 and 3 separators in preprocessing. To our knowledge the only theoretical work on preprocessing for Max k-Cut is by Fakhimi et al. [12]. They introduce a folding techniques that allows contracting vertices under certain circumstances. However in their computational experiments their preprocessing only had an effect for $k = 2$ and could not reduce any instance for $k > 2$ beyond what trivial preprocessing could already accomplish.

3 Data Reduction for Maximum k-Cut

Data reduction for MaxCut and Max k-Cut is based on two central components: Data transformations and data separations:

▶ **Definition 4** (Data Transformation). *A **data transformation** $\mathcal{A}$ for Max k-Cut transforms a weighted graph $G = (V, E, w)$ into a weighted graph $G' = (V', E', w')$ and provides a reconstruction algorithm $\mathcal{R}_\mathcal{A}$ which maps any k-partition p' of V' to a k-partition p of V. We call a **data transformation** a **data reduction** iff either $|V'| < |V|$ or $|V'| = |V|$ and $|E'| < |E|$.*

▶ **Definition 5** (Data Separation). *A **data separation** $\mathcal{A}$ for Max k-Cut transforms a weighted graph $G = (V, E, w)$ into a number of weighted graphs $G'_1 = (V_1, E_1, w_1), ..., G'_i = (V_i, E_i, w_i)$ and provides a reconstruction algorithm $\mathcal{R}_\mathcal{A}$, which maps k-partitions $p_1, ..., p_i$ of $V_1, ..., V_i$ to a k-partition p of V.*

▶ **Definition 6** (Optimality-Preserving). *We call a data transformation (resp. a separation) **optimality-preserving** if and only if $\mathcal{R}_\mathcal{A}$ maps any optimum Max k-Cut solution p' (resp. $p_1, ..., p_i$) for G' (resp. $G'_1, ..., G'_i$) to an optimum solution p for G.*

Our definitions build on the notion of data transformations introduced in [3]. However, their definitions require that data transformations produce an offset β between the optimum objective values of G and G'. We decided to instead require a reconstruction algorithm, as we want to obtain not just the optimum solution value, but also a corresponding k-partition. Still, all data transformations/separations discussed in this paper also yield a constant offset. This enables computing lower and upper bounds for the original instance from lower and upper bounds of the reduced instance. For an overview of the offsets see Appendix A.1.

Since our goal is to compute optimum solutions for Max k-Cut, all data separations and transformations we consider are optimality-preserving. To reduce the verbosity we will omit specifying that a data separation/reduction rule is optimality-preserving outside of theorems.

3.1 The Data Reduction Framework

Previous research on MaxCut preprocessing concerned itself with reducing the instances and obtaining the optimum solution value [26, 13, 29, 3]. While the preprocessing rules discussed allow for reconstruction of an optimum solution for the original graph, the presented frameworks do not describe how to perform this reconstruction, nor is it clear how new preprocessing rules would be added to the reconstruction.

To this end, we introduce a new preprocessing framework that collects reconstruction algorithms during preprocessing. Our data reduction framework relies on repeatedly applying data transformation/separation rules until we can no longer reduce the graph. For reconstruction, we build a reconstruction rule tree: When applying a data reduction/separation rule to G', we add its reconstruction algorithm as a child to the reconstruction algorithm of

the rule that created G'. Once we cannot reduce anything any more, we solve the remaining graphs using some exact solver. We then apply the reconstruction rules in reverse DFS order, so whenever a reconstruction rule is called we have access to optimum solutions for all graphs that it depends on. A visualization of this can be seen in Figure 2.

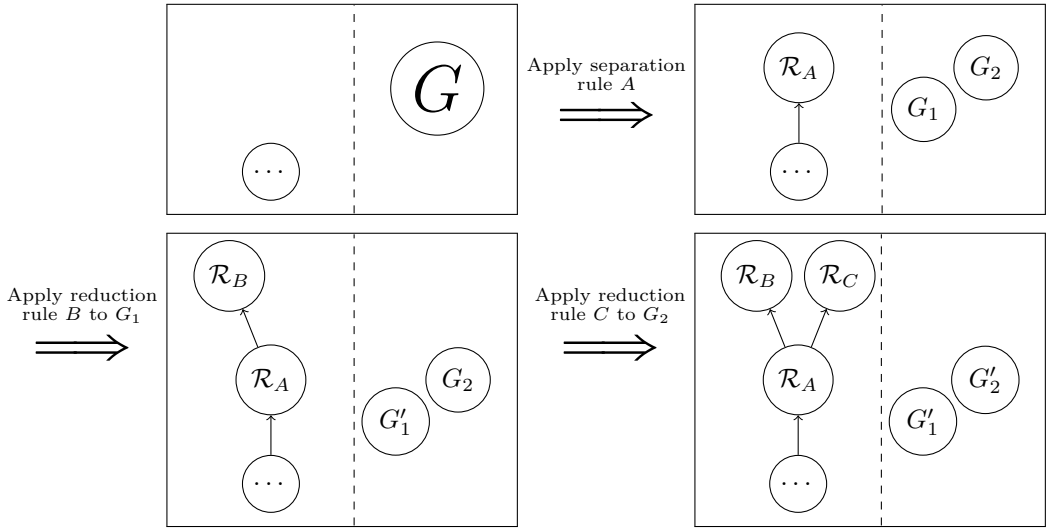


Figure 2 The preprocessing framework building the reconstruction tree (left) while applying data separation and reduction rules to the graphs (right).

3.2 Criteria for Preserving Optimality

As many of the following proofs will use similar techniques, we establish some notation and results that provide a general framework for our later proofs to use.

▶ **Definition 7** (Sum of Weighted Graphs). *Given two weighted graphs $G_1 = (V_1, E_1, w_1)$ and $G_2 = (V_2, E_2, w_2)$, let $E_1 \cap E_2 = F$. We define their sum $G = (V, E, w) = G_1 + G_2$ with $V = V_1 \cup V_2$, $E = E_1 \cup E_2$ and*

$$w(e) = \begin{cases} w_1(e) + w_2(e) & \text{if } e \in F \\ w_1(e) & \text{if } e \in E_1 \backslash F \\ w_2(e) & \text{if } e \in E_2 \backslash F \end{cases}$$

We note that if we consider graphs that can be transformed into each other only by adding/deleting weight 0 edges and degree 0 vertices, this notion of + forms a group. With this, we can now establish an optimality criterion via the addition of weighted graphs.

▶ **Theorem 8.** *Let $G_1 = (V_1, E_1, w_1)$ and $G_2 = (V_2, E_2, w_2)$ be weighted graphs and $G_1 + G_2 = G = (V, E, w)$. Let p be a k-partition of V such that $p_{|V_1}$ is an optimum solution for G_1 and $p_{|V_2}$ is an optimum solution for G_2. Then p is an optimum solution for G.*

Proof. We note that for any k-partition p' we have $w(p') = w_1(p'_{|V_1}) + w_2(p'_{|V_2})$. Using that $p_{|V_1}, p_{|V_2}$ are optimum solutions for G_1 and G_2, respectively, we get $w(p) = w_1(p_{|V_1}) + w_2(p_{|V_2}) \geqslant w_1(p'_{|V_1}) + w_2(p'_{|V_2}) = w(p')$. ◀

We note that in general the optimum solution p for G is not optimum for G_1 or G_2. However, given optimum solutions p', p'' for G_1, G_2 with $p'_{|V_1 \cap V_2} \cong p''_{|V_2 \cap V_2}$ we can efficiently combine these into an optimum solution for G:

▶ **Lemma 9.** *Given two k-partitions p' for V_1 and p'' for V_2 with $p'_{|V_1 \cap V_2} \cong p''_{|V_1 \cap V_2}$, we can compute a k-partition p of $V_1 \cup V_2$ in $\mathcal{O}(n)$ time such that $p_{|V_1} \cong p'$ and $p_{|V_2} \cong p''$.*

Proof sketch. To compute p from p' and p'', we find a permutation π with $p'_{|V_1 \cap V_2} = p''^{\pi}_{|V_1 \cap V_2}$ and then permute p'' with π. For an algorithm and the full proof see Appendix A.2. ◀

4 Optimality-Preserving Data Reduction

In this section we extend several known data reduction rules from MaxCut to Max k-Cut.

Before we get to the more advanced rules, we briefly discuss a simple rule: We can remove any vertex $v \in V$ with $d(v) < k$ and only positive weight incident edges. In the reconstruction we colour v with a colour not used in $N(v)$, of which there must be at least one.

4.1 Cliques with a small Neighbourhood

Cliques are interesting in the context of Max k-Cut-preprocessing. For a clique C with positive unit weight edges, any k-partition p with $\forall i : \left\lceil \frac{|C|}{k} \right\rceil \geqslant |V_{p=i}| \geqslant \left\lfloor \frac{|C|}{k} \right\rfloor$ is an optimum solution. Let $G = (V, E, w)$ be a graph that is the sum of a unit weight clique $C = (V_C, E_C, w_C)$ and $G' = (V', E', w')$. If the intersection $V_C \cap V'$ is small enough, we can solve G' to optimality and then colour the remaining vertices in V_C such that our partition is also optimum for C. This idea has been employed for MaxCut in [13] and was then generalised in [3]. Their results extend to Max k-Cut very naturally.

▶ **Theorem 10** (Same Neighbourhood Clique (generalises Proposition 4.5 in [3]))**.** *Given a weighted undirected graph $G = (V, E, w)$ and a clique subgraph $C = (V_C, E_C, w)$ of G. If, for some constant $c > 0$, we have*

- $\forall v \in V_C, e \in \delta_G(v) : w(e) = c$*, i.e. C is unit weight*
- $\forall v \in V_C : N(v) \backslash V_C = N(V_C)$*, i.e. all $v \in V_C$ have the same neighbourhood outside V_C*
- *and* $|N(V_C)| \leqslant \left\lceil \frac{|V_C \cup N(V_C)|}{k} \right\rceil$

then deleting all vertices in V_C from G and reducing the weights of all edges $\{v, w\}$ with $v, w \in N(C), v \neq w$ by c (if two vertices are not connected we insert an edge with weight $-c$) is an optimality-preserving data reduction rule for Max k-Cut.

The proof for Max k-Cut is analogous to the proof for MaxCut in [3]. While finding maximal cliques in general is difficult, cliques with a small neighbourhood can be located efficiently as noted in [13].

4.2 Triconnected Components and 2-Vertex-Separators

MaxCut can be solved efficiently on graphs with no $K_{3,3}$-minor by exploiting 2-vertex-separators, that is sets of two vertices such that their removal disconnects the graph [5]. These ideas were adapted for MaxCut preprocessing in [3]. They generalise to Max k-Cut:

▶ **Theorem 11** (Two-Vertex-Separator reduction (Corollary 4.3 in [3]))**.** *Given a weighted undirected graph $G = (V, E, w)$ and an induced subgraph $G[V']$ with $\partial_G(V') = \{u, v\}$. Let p' be the best Max k-Cut solution for $G[V']$ constrained by u and v having the same colour, and p'' the best solution in which they have different colours. Then deleting all vertices in $V' \backslash \{u, v\}$ and setting $w(\{u, v\}) = w_1(p'') - w_1(p')$ is an optimality-preserving data reduction.*

Proof sketch. We briefly sketch the proof by [3], which also works for Max k-Cut. Let G' be the graph remaining after the reduction and p be an ptimum solution for G'. If u and v have the same colour p, then we extend p with p' and otherwise we extend with p''. The change of $w(\{u, v\})$ encodes the trade-off between these. For a full proof using our framework based on graph sums see the Appendix B.1. ◀

To efficiently identify 2-vertex separators suitable for this reduction, we make use of the decomposition of G into its triconnected components [20], and then attempt to remove the leaves in the resulting tree structure as suggested in [3]. While [3] removed such components only if they contained $\leqslant 21$ vertices, we noticed that branch-and-bound could frequently find the optimum solution for larger graphs as well. Due to this we instead opted to run an exact solver with a time limit of 1 second. [3] employed similar techniques with 3-vertex-separators. A brief discussion why this is not possible for Max k-Cut can be found in Appendix B.2.

4.3 Negative Weight Dominating Edges

Another class of data reduction rules are dominating edges introduced by Lange et. al [26]. The key idea is that any edge that is not cut in an optimum solution can be safely contracted without affecting optimality.

▶ **Theorem 12** (Negative Dominating Edges (extends Theorem 1 (4) in [26])). *Let $G = (V, E, w)$ be a weighted graph, $\delta(S)$ be a cut set for some $S \subseteq V$ and $e \in \delta(S)$ with $w(e) < 0$. If*

$$-w(e) \geqslant \sum_{e' \in \delta(S) \setminus \{e\}} |w(e')|$$

then contracting e is an optimality-preserving data reduction.

Proof sketch. We adapt the proof by [26]. Let $e = \{u, v\}$ such that $u \in S$. Let p be an optimum solution. If p cuts e, then we swap the colours $p(u), p(v)$ on S. In the worst case we stop cutting all positive weight edges and start cutting all negative weight edges in $\delta(S)$, except for e. However, due to $-w(e)$ being disproportionally high, this yields a k-partition p' with $w(p') \geqslant w(p)$ that does not cut e. Thus, there is an optimum solution that does not cut e and we can contract u and v. ◀

Unfortunately, most other dominating edge rules in [26, 29] and [3] rely on specific properties of $k = 2$. A weaker version of the triangle rule by [13] and a discussion why the other rules do not generalise can be found in the full version of this paper.

5 Optimality-Preserving Data Separation

We now move on to data separations. For these, we describe conditions under which we can solve several subgraphs independent of each other, instead of solving the whole graph.

5.1 Separating (Bi)connected Components

Separating graphs into their (bi)connected components is a well known data separation technique [18] and a standard preprocessing step for MaxCut [3, 29]. We briefly motivate why separating (bi)connected components is optimality-preserving for Max k-Cut. As different (bi)connected components C_1, C_2 intersect in at most one vertex, optimum solutions p', p'' of C_1, C_2, respectively, have $p'_{|C_1 \cap C_2} \cong p''_{|C_1 \cap C_2}$. Therefore p', p'' can be combined into a k-partition p of V using Lemma 9. By Theorem 8 p is optimum for G.

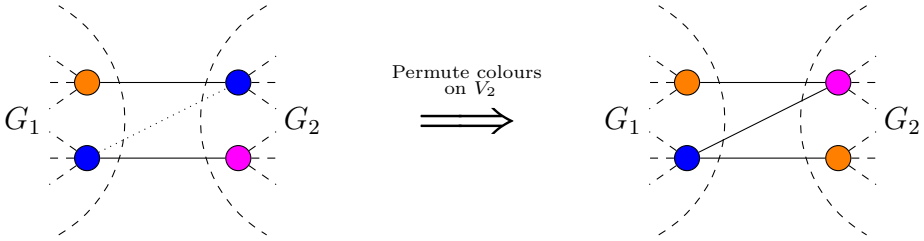


Figure 3 We cut all edges in the cut set by permuting blue → magenta → orange → blue on V_2.

5.2 Structured Cut Sets

Here, we introduce a new separation rule called *structured cut sets*, which makes use of the combinatorial benefits of larger values of k.

By definition, deleting the edges of a cut set $C \subseteq E$ from G decomposes it into at least two disjoint components. In this section we assume w.l.o.g. that deleting the cut set splits G into exactly two connected components, $G_1 = (V_1, E_1, w_1)$ and $G_2 = (V_2, E_2, w_2)$. We also consider only cut sets in which all edges have positive edge weights (called positive cut sets). For a discussion why cut sets containing negative edges are disregarded, see Appendix C.5. Given optimum solutions p', p'' for G_1, G_2, we combine them into a k-partition p of V with $\forall v \in V_1 : p(v) = p'(v)$ and $\forall v \in V_2 : p(v) = p''(v)$. We now want to permute the colours on V_2 such that all edges in C are cut by p. If this is possible, p is optimum for G, as it is optimum for G_1, G_2 and $G[C]$ and $G = G_1 + G_2 + G[C]$. See Figure 3 for an example.

We develop criteria to decide whether such a colour permutation exists, independent of the optimal solutions for G_1 and G_2. This allows us to solve G_1 and G_2 independently, as we can be certain that all edges in C can be cut. As our criteria work by showing that reconstruction is possible, we first introduce our reconstruction algorithm.

5.2.1 Reconstruction for Structured Cut Sets

In order to compute an optimum solution p of G from partitions p' and p'' of G_1 and G_2, we introduce the notion of a complement that preserves the bipartition of a graph:

▶ **Definition 13** (Bipartite Complement Graph). *Given an undirected bipartite graph $G = (L \sqcup R, E)$ such that L, R is a bipartition of its vertices, we define the bipartite complement of G with respect to L, R as*

$$\overline{G[L,R]} = (L \sqcup R, \{\{l, r\} \mid l \in L, r \in R, \{l, r\} \notin E\})$$

Figure 4 A cut set and its colour relation graph for $k = 4$ with the partition encoded via colours.

The central idea of our algorithm is to construct a colour relation graph G_{CR}, which has $2k$ vertices. We have one vertex for every colour $i \in \{1, ..., k\}$ and each side L, R. Two vertices $(i, L), (j, R)$ are connected in G_{CR} iff an edge $\{v, w\} \in C, v \in V_1, w \in V_2$ connects vertices

with $p(v) = i, p(w) = j$. An example can be seen in Figure 4. These edges in G_{CR} encode that we are not allowed to map j to i when permuting the colours on V_2. Correspondingly, edges in the bipartite complement of G_{CR} encode how we are allowed to permute p on V_2. Any perfect matching will give us a valid permutation π. If we permute p with π on V_2, all edges in C will be cut. Correspondingly, given a solution p for G with $p_{|V_1} \cong p', p_{|V_2} \cong p''$ that cuts all edges in C, we can derive a perfect matching in the colour relation graph. We do this by determining a permutation π that maps p'' into $p_{|V_2}$.

From this idea we can derive the following reconstruction algorithm:

Algorithm 1 ReconstructSCS$(G = (V, E, w), C, p', p'', k)$.

1: Find connected components $G_1 = (V_1, E_1, w_1), G_2 = (V_2, E_2, w_2)$ in G without C
2: Initialise $G_{CR} = (\{1, ..., k\} \times \{L, R\}, \{\{(p'(v), L), (p''(w), R)\} \mid \{v, w\} \in C, v \in V_1, w \in V_2\})$
3: Compute $M = \text{MaximumMatching}(G_{CR}[\{(i, L) \mid i \in \{1, ..., k\}\}, \{(i, R) \mid i \in \{1, ..., k\}\}])$
4: **if** $|M| < k$ **then**
5: **return** `RECONSTRUCTION NOT POSSIBLE`
6: Initialise array π of length k such that $\forall \{(i, L), (j, R)\} \in M : \pi[j] = i$
7: Compute k-partition p of V with $\forall v \in V_1 : p(v) = p'(v)$ and $\forall v \in V_2 : p(v) = \pi[p''(v)]$
8: **return** p

▶ **Theorem 14.** *Given G, C, p' and p'', ReconstructSCS finds a k-partition p of V that cuts all $e \in C$ with $p_{|V_1} \cong p'$ and $p_{|V_2} \cong p''$ if one exists, and returns an error otherwise. Its runtime is in $\mathcal{O}(n + |C| + k^{2.5})$.*

The proof follows the previous outline and can be found in full in Appendix C.1. By applying Theorem 8 we obtain the following result:

▶ **Corollary 15.** *Let C be a positive cut set that partitions G into G_1 and G_2 with optimum solutions p' and p'', respectively. If ReconstructSCS returns a partition p, then p is optimum for G.*

We now know how to efficiently reconstruct a cut set, provided that reconstruction is possible. However, there are cut sets for which reconstruction is not always possible. An example for $k = 3$ is a cut set that induces a $K_{2,2}$.

Therefore, before splitting G at a positive cut set, we must make sure that ReconstructSCS finds a partition. We present some efficiently checkable criteria that assure that we can reconstruct a solution or prove that this is not possible. They make no assumptions about p' and p'', and therefore work regardless of the partitions provided for V_1 and V_2.

5.2.2 Criteria for Removing Structured Cut Sets

We first introduce two criteria showing that sufficiently small cut sets can always be removed.

▶ **Theorem 16.** *Let $G = (V, E, w)$ be a weighted graph and C be a positive cut set. If*
- *$G[C]$ has at most k vertices or*
- *$G[C]$ has at most $k - 1$ edges*

then splitting G into G_1, G_2 by deleting C is an optimality-preserving data separation.

Proof sketch. If there are at most k vertices in $G[C]$, then we can permute the colours on G_2 such that no vertices on the left and right side share a colour. If there are less than k edges in the cut set and we permute the colours on G_2 uniformly at random, then in expectation there are $|C| \cdot \frac{1}{k} < 1$ edges in C we do not cut. Therefore there must be a permutation that cuts all edges in C. For an algorithmic proof see Appendix C.3. ◀

Both of these criteria can be checked efficiently. However, as we can see in Figure 5, there are cut sets that we can split that satisfy neither criterion.

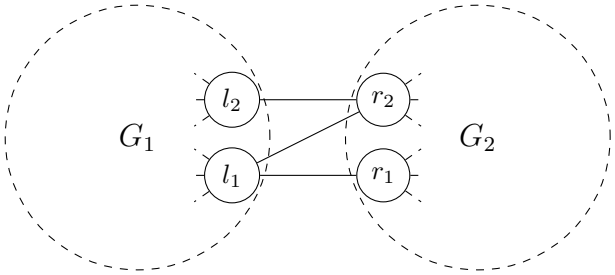


Figure 5 A cut set. For $k = 3$, neither criterion from Theorem 16 holds. However, as l_2 and r_1 can have the same colour, we can always reconstruct.

A naive approach to solve this issue would be to simply enumerate all colourings of $\partial_G(V_1), \partial_G(V_2)$ and check if we can reconstruct. However, this would be very time consuming, even for small k. Instead we present another criterion that can be applied to larger separators and takes their combinatorial structure into account:

▶ **Theorem 17.** *Let $G = (V, E, w)$ be a weighted graph and C be a positive cut set that splits G into $G_1 = (V_1, E_1, w_1)$ and $G_2 = (V_2, E_2, w_2)$. We denote $L = \partial_G(V_1)$ and $R = \partial_G(V_2)$. If $\overline{G[C][L, R]}$ contains a matching M of size $\geqslant 2 \cdot (|V_C| - k) - 1$, then splitting G into G_1 and G_2 by deleting C is an optimality-preserving data separation*

Proof sketch. Given p' and p'', we can derive the colour relation graph from $G[C]$ by repeatedly contracting vertices that have the same colour and side. We then add vertices for the colours that are missing. We call these newly added vertices jokers, as in the complement graph we can match them to any vertex on the other side. An example of this can be seen in Figure 6. If, for an $e \in M$ one of its endpoints is contracted, then we remove it. Therefore, for each contraction, we lose at most 2 edges in M while gaining a joker. The bound is chosen such that we can match all vertices not covered by the remaining edges in M with jokers after any contraction sequence. For the full proof see Appendix C.4. ◀

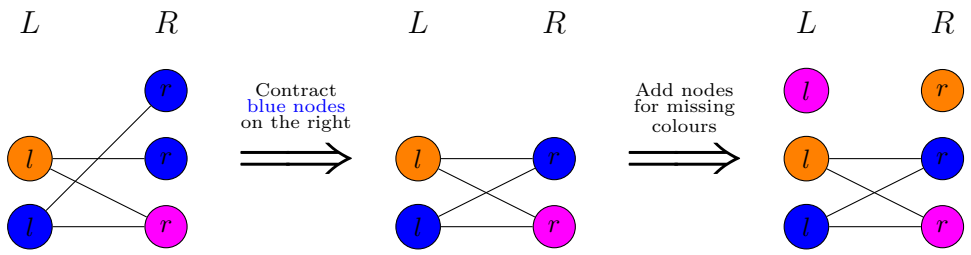


Figure 6 Obtaining G_{CR} by contracting nodes sharing side and colour, then adding nodes for the missing colours.

This criterion can be checked efficiently in time $\mathcal{O}(|C| + k^{2.5})$ using the algorithm by [19]. Further, if the cut set contains at most k vertices, we only require $|M| \geqslant 0$. Therefore, this criterion implies the first criterion from Theorem 16. Note that this criterion is sufficient, but for $k > 3$ it is not necessary. For $k = 4$ an example of a graph for which the criterion does not hold, but that is valid to separate, can be seen in Figure 7.

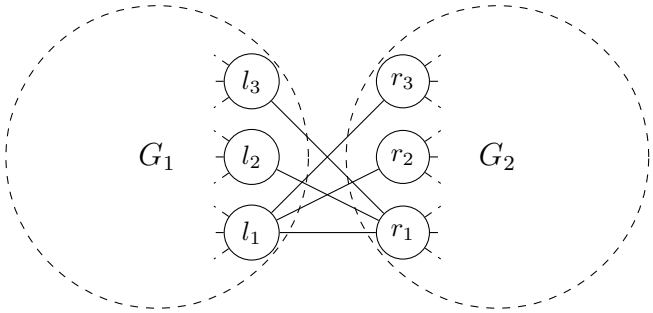


Figure 7 If $k = 4$ we can always reconstruct here by assigning l_1 and r_1 to different groups and then assigning the remaining 2 groups to l_2, l_3, r_2, r_3 if necessary. As all edges contain either l_1 or r_1 it is no problem if e.g. l_2, r_3 are assigned to the same group. Note that this structured cut sets fulfils none of our criteria.

To complement the criteria showing that reconstruction is possible, we also show a simple criterion that shows reconstruction is not guaranteed to be possible:

▶ **Theorem 18.** *Given a weighted undirected graph $G = (V, E, w)$ and a positive cut set C in G that splits G into $G_1 = (V_1, E_1, w_1)$ and $G_2 = (V_2, E_2, w_2)$. If $\max\{|\partial_G(V_1)|, |\partial_G(V_2)|\} \geqslant k$, then there exist k-partitions p', p'' of V_1 and V_2, such that there is no k-partition p of V with $p_{|V_1} \cong p'$ and $p_{|V_2} \cong p''$ p that cuts all edges in C.*

Proof. Let $|\partial_G(V_1)| \geqslant k$. If p' uses all k colours vertices in $\partial_G(V_1)$ and p'' colours all vertices in $\partial_G(V_2)$ same colour, then at least one edge in C can not be cut by permuting p on V_2. ◀

This implies that for $k = 2$ we can only remove cut sets containing one edge, which are already covered by splitting biconnected components.

5.3 Criteria where one Solution is Known

So far we assumed that the solutions for G_1 and G_2 are unknown. If we know the optimum solution for w.l.o.g. G_2, we can contract all vertices in V_2 that share a colour. If any criterion from Theorems 16 or 17 is fulfilled, we can delete all vertices in V_2. Knowing the solution for G_2 also enables a new criterion: Let $v_1, ..., v_r$ be the boundary of the contracted G_2. If $d_{G[C]}(v_i) \leqslant k - i$, we can delete all vertices in V_2. During reconstruction we permute p on V_2 such that v_i gets a colour not used by any vertex in $N_{G[C]}(v_i)$ or $v_1, ..., v_{i-1}$.

Given a positive cut set C we can not split with Theorems 16 or 17. If $|V_2| \leqslant 20$ we solve G_2 to optimality and check the criteria above.

5.4 Finding Structured Cut Sets

As we do not have a nice "if and only if" characterisation of which cut sets we can split, designing an efficient algorithm to find them is difficult. Theorem 18 implies that we can only split structured cut sets with at most $(k-1)^2$ many edges. We therefore propose to find structured cut sets by identifying small cut sets, checking our criteria, and solving one side to optimality if it is sufficiently small. For this we employ a heuristic based on the randomized MinimumCut algorithm FastCut by Karger and Stein [24]. FastCut works by randomly contracting edges to generate large numbers of small cuts, which we take as candidates.

6 Computational Experiments

For our experiments, we implemented the assignment model for Max k-Cut by [6]. To strengthen the model we employ greedy heuristics to separate cycle, general clique, wheel and bicycle wheel inequalities from [6] and $1, -1$ hypermetric inequalities from [7]. The solver serves both as a baseline to compare against and as the solver for the kernel graphs left over by our preprocessor. To provide a warm start for the solver we implemented a local search algorithm with tabu search based on [28]. We also used this heuristic for experiments on very large instances.

For the order of data separation and reduction rules we chose to first employ rules that maintain local positivity and unit weight of edges, to ensure that rules requiring this can still be applied later. This lead us to the following order:

1. Remove low degree vertices and split (bi)connected components
2. Locate and remove structured cut sets using FastCut and Theorems 16 and 17.
3. Remove unit weight cliques with a small neighbourhood (Theorem 10).
4. Contract negative-weight dominating edges.
5. Find two vertex separators yielding at least one small component and apply Theorem 11.
6. Find cut sets where one side is small, solve it and check if it can be removed.

If any rule successfully reduces the graph, we restart at the top. We use queues and flags to keep track of which vertices are promising candidates for the various rules and to make sure we do not run expensive rules too often. Similar techniques are used in [13, 3]. We decided to forego the preprocessing rule by [12], as they reported no reduction for $k > 2$.

6.1 Setup

We implemented everything in `C++ 20`. For a graph class and fundamental algorithms we relied on NetworKit [34]. To locate biconnectors we use the SPQR-tree implementation of [16] in the Open Graph Drawing Framework [4]. We compiled the code using gcc 14.3.0 with the "-O3" flag. All experiments were run on a machine with 2 Intel Xeon "Sapphire Rapids" 48-core CPUs and 1024 GB RAM running AlmaLinux 9.6. To solve ILPs we used the state of the art solver Gurobi 13 [15] and set it to use as many threads as needed. We also employed parallelism to find structured cut sets and in our local search heuristic.

6.2 Instance Selection

We used the instances by [3], which is the most recent study on MaxCut-preprocessing. These are separated into four sets: *easy*, *medium*, *big* and *torus*. The first three sets contain instances from image segmentation and VLSI-chip design problems from [9] and biological, co-author and social networks from [33][3]. The instances are split into *easy*, *medium* and *hard* based on how challanging they are for MaxCut. As the difficulty for Max k-Cut varies strongly depending on k we decided to keep the grouping of [3]. The *torus* instances contain 2 and 3d toroid grids from statistical physics from [39], originally generated by [27]. We also relaxed the CELAR, GRAPH and DUTtest1 sets of frequency assignment problems from [11], referred to as *fap* instances. These were originally provided by the CALMA project and [36]. See the full version of this paper for more detail.

[3] We removed the web-google instance used in [3], as it no longer appears to be on the network repository. We instead added a larger instance called web-google to the big set.

6.3 Running an Experiment

We compare runs with no preprocessing, naive preprocessing (only low degree vertex removal and splitting (bi)connected components), and our preprocessing. We evaluate the k-values $\{3, \ldots, 8, 10, 12\}$. On the easy, medium, fap and torus datasets we employ Gurobi with a 30 minute time limit to solve the instances to optimality. For the big data set we run a local search heuristic with a two hour time limit. Preprocessing is included in the time limit and reported runtime. To account for variances, we run 5 seeds for Gurobi and 3 for the heuristic.

Given a combination of instance and k, we determine a winner: The winner is the configuration that found the optimum solution on the largest number of seeded runs. In case of a tie, we first break by the average dual bound on the solutions not solved to optimality and then by runtime on the solved instances.

6.4 Computational Results

Figure 8 shows the number of optimum solutions found for the instances from [3] and the fap instance set. We observe significant improvements for the challenging fap data set, with the exception for $k = 5$. For $k = 7$ we solve three additional instances and for $k = 8$ we solve five more instances. In both cases the additional instances solved are from the real world CELAR data set, while our performance on the synthetic GRAPH and DUTtest1 instances is the same as both naive and no preprocessing. The improvements for larger values of k are due to the increased effectiveness of our structured cut sets. With increasing k, many of the subgraphs split by our preprocessing are easy to solve, since they can be k-coloured easily. On the torus instances, the number of solved instances decreases with increasing k, as the size of the search space grows significantly, while the grid structure means very few cutting planes are applicable. Most easy and medium instances can be routinely solved, both with and without preprocessing. Since solution times for the easy instances are below one second, even with the naive preprocessing, their discussion is only in the full version of this paper.

The number of graphs to be solved independently differs significantly between the different instance sets and values of k, with values ranging from 1 to multiple hundreds for the large instances. However, for almost all instances and k-values one of the remaining graphs was significantly larger (usually $> 50\%$ of the remaining vertices and edges) and accounted for almost all of the runtime. For more detail see the full version of this paper.

We now take a closer look into the medium dataset. Table 1 shows the number of wins, the runtimes, and the sizes of the remaining graphs after preprocessing. We observe significant gains over naive preprocessing. Across all k, the average graph kernels remaining after our preprocessing are significantly smaller than those from naive preprocessing. This leads to our preprocessing achieving more wins and lower average runtime for most k. This is especially visible for $k \geqslant 8$ where Gurobi with our preprocessing beats pure Gurobi by an order of magnitude and naive preprocessing by a factor of 6. The most challenging instances in the medium data set are network instances. These frequently contain small, dense subgraphs that are only sparsely connected to the remaining graph. Our preprocessing splits these off or removes them with the clique, structured cut sets and biconnector rules, making the kernels much easier to solve.

On the fap dataset, preprocessing removes significantly less than on the previous data sets (see Table 2). This is primarily caused by fap instances being denser than the previous network instances. Still, our preprocessing is able to separate or remove dense subgraphs that are only sparsely connected to the remaining graph. Due to this, we observe more wins and faster runtimes on average, in addition to solving more instances to optimality.

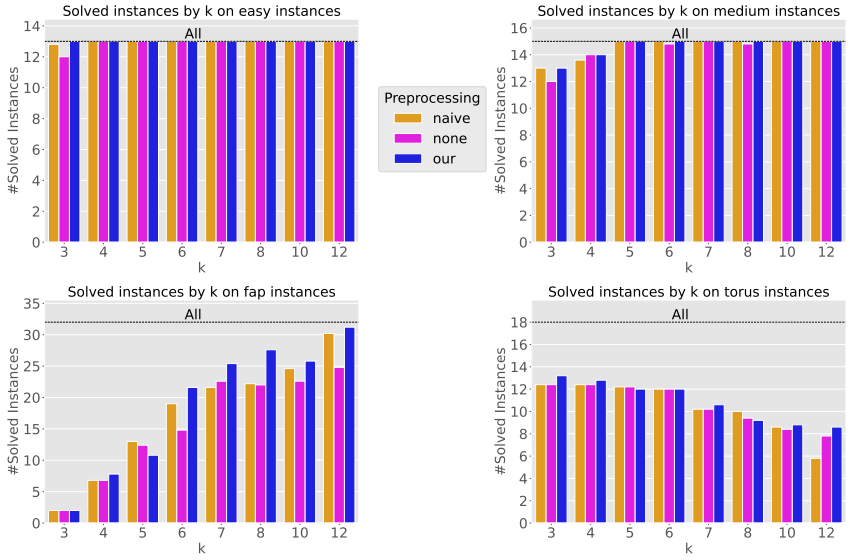


Figure 8 Number of optimum solutions found within 30 minutes on **easy**, **medium**, **fap** and **torus** instances. If the optimum was found only for some seeds the instance is counted fractionally.

Table 1 Number of wins, average runtime and average percentage of vertices and edges remaining after **naive** and **our** preprocessing algorithm on the **medium** data set.

	Gurobi		**Naive**				**Our**			
k	wins	t[s]	$\|V\|$[%]	$\|E\|$[%]	wins	t[s]	$\|V\|$[%]	$\|E\|$[%]	wins	t[s]
3	2	486	32.55	36.74	4	441	**16.75**	**22.60**	**11**	**330**
4	5	**282**	19.77	21.73	6	364	**8.34**	**12.35**	**8**	366
5	2	175	14.47	15.36	8	150	**4.98**	**7.31**	**9**	**122**
6	1	143	11.62	11.53	9	47	**3.36**	**6.05**	**11**	**40**
7	1	135	10.18	9.21	**9**	61	**2.29**	**4.20**	**9**	**20**
8	1	134	9.02	7.34	8	67	**1.55**	**3.07**	**13**	**10**
10	1	109	8.14	5.66	**10**	67	**0.91**	**1.82**	**10**	**10**
12	1	119	7.40	3.91	**11**	67	**0.21**	**0.27**	10	**10**

Table 2 Number of wins, average runtime and average percentage of vertices and edges remaining after **naive** and **our** preprocessing algorithm on the **fap** data set.

	Gurobi		**Naive**				**Our**			
k	wins	t[s]	$\|V\|$[%]	$\|E\|$[%]	wins	t[s]	$\|V\|$[%]	$\|E\|$[%]	wins	t[s]
3	11	1,722	99.09	99.74	13	1,722	**93.93**	**96.70**	**19**	**1,717**
4	**13**	1,548	95.20	98.05	11	1,535	**91.95**	**95.73**	**13**	**1,502**
5	9	1,326	87.60	93.02	**14**	**1,238**	**82.52**	**89.34**	13	1,361
6	5	1,020	63.98	76.38	10	760	**56.73**	**68.66**	**18**	**627**
7	3	658	59.05	72.03	10	595	**46.02**	**55.94**	**19**	**467**
8	2	698	48.02	62.83	13	562	**34.72**	**43.75**	**17**	**331**
10	3	635	35.09	50.01	12	480	**17.79**	**25.92**	**17**	**402**
12	0	475	27.05	41.25	12	126	**11.34**	**17.59**	**25**	**66**

The torus instances differ in their structure significantly from the other data sets as their grid structure and the large number of negative edge weights makes almost all our preprocessing rules not applicable. In Table 3 we see that naive preprocessing is able to remove almost nothing, even for higher values of k. Our preprocessing is able to remove significantly more, as contracting negative-weight dominating edges breaks up the grid like structure, enabling other preprocessing rules.

Table 3 Number of wins, average runtime and average percentage of vertices and edges remaining after **naive** and **our** preprocessing algorithm on the **torus** data set.

	Gurobi		Naive				Our			
k	wins	t[s]	$\|V\|$[%]	$\|E\|$[%]	wins	t[s]	$\|V\|$[%]	$\|E\|$[%]	wins	t[s]
3	1	626	100.00	100.00	3	625	**91.40**	**95.30**	**15**	**588**
4	2	638	100.00	100.00	5	638	**91.38**	**95.26**	**12**	**618**
5	5	671	97.27	94.75	6	669	**84.41**	**86.60**	**10**	**653**
6	**8**	748	97.27	94.75	4	**727**	**81.60**	**83.52**	**8**	740
7	3	875	96.25	92.75	5	873	**77.14**	**77.67**	**10**	**861**
8	5	953	96.25	92.75	4	**900**	**74.92**	**75.39**	**9**	904
10	7	1,045	96.25	92.75	3	1,014	**72.93**	**73.17**	**8**	**944**
12	6	1,208	96.25	92.75	3	1,315	**72.62**	**72.71**	**9**	**1,029**

On the big data set we observe more mixed results. As seen in Table 4, for most instances our preprocessing does not achieve significantly greater reductions than naive preprocessing, while taking much longer. A notable exception is the web-it-2004 instance, on which our preprocessing removes almost 90% of nodes and 65% of edges for all k, while naive preprocessing can not remove more than 12% of nodes and 2% of edges, even for $k = 10$. Unfortunately our preprocessing usually does not lead to better heuristic values. A significant amount of time is spend to ensure everything is optimality-preserving. This conflicts with finding the best solution within the time limit, as the objective value on the kernels has the greatest impact on the objective for the whole graph.

Table 4 Best value found by our local search heuristic within 2 hours, percentage of vertices and edges remaining, improvement over the straight heuristic (bvi) and time spend preprocessing (pr) of **naive** and **our** preprocessing on the **big** data set. For the full table see the full version of this paper.

		Heuristic	Naive				Our			
k	**instance**	best value	$\|V\|$[%]	$\|E\|$[%]	bvi	pr[s]	$\|V\|$[%]	$\|E\|$[%]	bvi	pr[s]
4	ca-IMDB	3.687.347	**39.58**	**79.82**	50.525	3	**39.58**	**79.82**	**50.589**	63
	ca-coauthors-dblp	12.070.341	93.80	99.58	**3.618**	12	**84.78**	**97.13**	1.480	1.008
	inf-road-central	16.930.983	**0.00**	**0.00**	**2.430**	17	**0.00**	**0.00**	**2.430**	17
	web-Stanford	2.213.473	65.49	90.45	**4.314**	2	**63.48**	**89.19**	3.096	178
	web-google	4.679.030	56.32	84.80	**22.832**	8	**53.47**	**82.05**	15.212	1.447
	web-it-2004	5.727.019	91.44	98.96	**5.208**	4	**11.04**	**35.92**	**5.208**	44
7	ca-IMDB	3.760.031	**26.13**	**64.50**	15.666	3	**26.13**	**64.50**	**15.719**	49
	ca-coauthors-dblp	13.612.054	86.50	98.60	**4.809**	11	**82.01**	**97.24**	773	2.286
	inf-road-central	16.933.413	**0.00**	**0.00**	**0**	17	**0.00**	**0.00**	**0**	17
	web-Stanford	2.282.032	33.52	69.91	1.436	1	**33.08**	**69.48**	**1.468**	118
	web-google	4.969.060	36.46	67.09	8.982	6	**34.98**	**65.17**	**9.412**	1.128
	web-it-2004	6.434.136	90.01	98.66	3.165	4	**10.65**	**35.82**	**3.166**	41
10	ca-IMDB	3.777.961	**16.29**	**46.38**	**3.865**	2	**16.29**	**46.38**	3.858	37
	ca-coauthors-dblp	14.192.268	79.89	97.19	**4.717**	10	**77.07**	**96.21**	2.338	1.793
	inf-road-central	16.933.413	**0.00**	**0.00**	**0**	16	**0.00**	**0.00**	**0**	16
	web-Stanford	2.298.360	23.20	59.40	**912**	2	**22.99**	**59.13**	907	63
	web-google	5.051.920	22.09	47.17	4.502	4	**21.26**	**45.79**	**4.768**	561
	web-it-2004	6.711.894	88.01	98.02	2.149	4	**10.26**	**35.64**	**2.150**	42

6.5 Ablation Study

To evaluate the impact of our different preprocessing rules, we benchmarked how the performance changes when a rule is disabled. Table 5 shows the results. Across all values of k, disabling the dominating edge rules leads to the largest kernels on the torus set. However, disabling the biconnector, SCS or SCSS rule has a stronger impact on the number of wins.

This suggests that these rules remove data that the solver would usually struggle with. On the medium instances, disabling the clique and dominating edge rule causes the greatest loss in the number of removed edges. This is different on the fap instances, where disabling the biconnector and the SCSS rules lead to the lowest number of removed edges. However, on the medium and fap set, none of clique, biconnector or SCS cause significantly fewer wins when removed. This suggests that these rules have comparable impacts on our ability to solve instances to optimality.

Table 5 Percentage of edges remaining and number of wins comparing full preprocessing except dominating edges (**NoDom**), cliques (**noClq**), 2 vertex separators (**noBicon**), structured cut sets (**noSCS**) and structured cut sets with solving the small side (**noSCSS**). Naive and full preprocessing for reference, the worst values are highlighted. For the full table see the full version of this paper.

		All	**Naive**	**NoDom**		**NoClq**		**noBicon**		**NoSCS**		**NoSCSS**	
k	dataset	$\|E\|$[%]	$\|E\|$[%]	$\|E\|$[%]	wins	$\|E\|$[%]	wins	$\|E\|$[%]	wins	$\|E\|$[%]	wins	$\|E\|$[%]	wins
4	easy	5.77	5.77	**5.77**	**7**	**5.77**	10	**5.77**	9	**5.77**	9	**5.77**	10
	medium	12.35	21.73	14.28	5	**16.52**	**4**	13.91	7	12.32	5	12.35	6
	torus	95.26	100.00	**100.00**	6	95.26	3	95.26	4	95.26	**2**	95.26	3
	fap	95.73	98.05	95.73	11	96.22	10	**96.60**	**6**	95.09	7	95.82	14
7	easy	0.00	0.00	**0.00**	**11**	**0.00**	12	**0.00**	**11**	**0.00**	12	**0.00**	12
	medium	4.20	9.21	**6.60**	10	5.48	7	4.71	11	4.20	**6**	4.20	9
	torus	77.67	92.75	**87.10**	8	77.67	2	78.42	4	77.35	3	79.95	**1**
	fap	55.94	72.03	55.94	**5**	59.64	6	55.91	6	56.37	7	**67.07**	12
10	easy	0.00	0.00	**0.00**	12	**0.00**	**11**	**0.00**	**11**	**0.00**	12	**0.00**	12
	medium	1.82	5.66	**4.31**	10	2.13	**9**	2.30	10	1.88	**9**	1.82	11
	torus	73.17	92.75	**84.18**	9	73.17	**2**	74.12	**2**	72.68	3	76.63	**2**
	fap	25.92	50.01	25.92	6	37.08	**3**	25.92	4	23.03	4	**40.12**	18

7 Conclusion and Outlook

We introduced new optimality-preserving data reduction and data separation rules for Max k-Cut. In particular, we proposed structured cut sets, a new data separation rule that is especially effective for $k \geqslant 3$. Our experimental results show that structured cut sets enables the removal of many cut sets, particularly for larger k. Combined with generalisations of MaxCut-preprocessing rules, this allows us to solve significantly more instances than using an exact solver with naive preprocessing only. This holds for instances from different real-world applications and across different values of k.

Future work includes developing additional criteria for cut sets whose removal preserves optimality. It would also be interesting to investigate whether structured cut sets can be used for problems related to Max k-Cut. Furthermore, it would be interesting to engineer a preprocessing framework especially suited for running heuristics on very large instances.

References

1 Zacharie Ales and Arnaud Knippel. An extended edge-representative formulation for the k-partitioning problem. *Electronic Notes in Discrete Mathematics*, 52:333–342, 2016. doi:10.1016/j.endm.2016.03.044.

2 Miguel F Anjos, Bissan Ghaddar, Lena Hupp, Frauke Liers, and Angelika Wiegele. Solving k-way graph partitioning problems to optimality: The impact of semidefinite relaxations and the bundle method. In *Facets of combinatorial optimization: Festschrift for Martin Grötschel*, pages 355–386. Springer, 2013. doi:10.1007/978-3-642-38189-8_15.

3 Jonas Charfreitag, Christine Dahn, Michael Kaibel, Philip Mayer, Petra Mutzel, and Lukas Schürmann. Separator Based Data Reduction for the Maximum Cut Problem. In *22nd International Symposium on Experimental Algorithms (SEA 2024)*, pages 4–1. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.SEA.2024.4.

4 Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W Klau, Karsten Klein, and Petra Mutzel. The Open Graph Drawing Framework (OGDF). *Handbook of graph drawing and visualization*, 2011:543–569, 2014.

5 Markus Chimani, Martina Juhnke-Kubitzke, Alexander Nover, and Tim Römer. Cut polytopes of minor-free graphs. *arXiv preprint arXiv:1903.01817*, 2019. doi:10.48550/arXiv.1903.01817.

6 Sunil Chopra and Mendu R Rao. The partition problem. *Mathematical programming*, 59(1):87–115, 1993. doi:10.1007/BF01581239.

7 Sunil Chopra and Mendu R Rao. Facets of the k-partition polytope. *Discrete applied mathematics*, 61(1):27–48, 1995. doi:10.1016/0166-218X(93)E0175-X.

8 Geir Dahl and Truls Flatberg. An integer programming approach to image segmentation and reconstruction problems. *Geometric Modelling, Numerical Simulation, and Optimization: Applied Mathematics at SINTEF*, pages 475–496, 2007. doi:10.1007/978-3-540-68783-2_14.

9 Iain Dunning, Swati Gupta, and John Silberholz. What works best when? A systematic evaluation of heuristics for Max-Cut and QUBO. *INFORMS Journal on Computing*, 30(3):608–624, 2018. doi:10.1287/ijoc.2017.0798.

10 Andreas Eisenblätter. *Frequency assignment in GSM networks: Models, heuristics and lower bounds*. Cuvillier Verlag, 2002. URL: https://nbn-resolving.org/urn:nbn:de:0297-zib-10132.

11 Andreas Eisenblätter and Arie Koster. FAP web, 2000. URL: https://fap.zib.de/.

12 Ramin Fakhimi, Hamidreza Validi, Illya V Hicks, Tamás Terlaky, and Luis F Zuluaga. A folding preprocess for the max k-cut problem. *Optimization Letters*, 19(5):899–917, 2025. doi:10.1007/s11590-025-02199-0.

13 Damir Ferizovic, Demian Hespe, Sebastian Lamm, Matthias Mnich, Christian Schulz, and Darren Strash. Engineering kernelization for maximum cut. In *2020 Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 27–41. SIAM, 2020. doi:10.1137/1.9781611976007.3.

14 Bissan Ghaddar, Miguel F Anjos, and Frauke Liers. A branch-and-cut algorithm based on semidefinite programming for the minimum k-partition problem. *Annals of Operations Research*, 188(1):155–174, 2011. doi:10.1007/s10479-008-0481-4.

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

16 Carsten Gutwenger and Petra Mutzel. A linear time implementation of SPQR-trees. In *International Symposium on Graph Drawing*, pages 77–90. Springer, 2000. doi:10.1007/3-540-44541-2_8.

17 Bruce Hendrickson and Robert Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM Journal on Scientific Computing*, 16(2):452–469, 1995. doi:10.1137/0916028.

18 Dorit S Hochbaum. Why should biconnected components be identified first. *Discrete applied mathematics*, 42(2-3):203–210, 1993. doi:10.1016/0166-218X(93)90046-Q.

19 John E Hopcroft and Richard M Karp. An nˆ5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–231, 1973. doi:10.1137/0202019.

20 John E Hopcroft and Robert Endre Tarjan. Dividing a graph into triconnected components. *SIAM Journal on computing*, 2(3):135–158, 1973. doi:10.1137/0202012.

21 Michael Kaibel and Petra Mutzel. Optimality-Preserving Data Reduction for Maximum k-Cut (Full Version), 2026. doi:10.48550/arXiv.2607.03281.

22 Michael Kaibel and Petra Mutzel. Source Code: Optimality-Preserving Data Reduction for Maximum k-Cut (Version 2). https://doi.org/10.5281/zenodo.21256337, July 2026. doi:10.5281/zenodo.21256337.

23 Volker Kaibel, Matthias Peinhardt, and Marc E Pfetsch. Orbitopal fixing. *Discrete Optimization*, 8(4):595–610, 2011. doi:10.1016/j.disopt.2011.07.001.

24 David R Karger and Clifford Stein. A new approach to the minimum cut problem. *Journal of the ACM (JACM)*, 43(4):601–640, 1996. doi:10.1145/234533.234534.

25 Richard M. Karp. Reducibility Among Combinatorial Problems. In Raymond E. Miller and James W. Thatcher, editors, *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*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972. doi:10.1007/978-1-4684-2001-2_9.

26 Jan-Hendrik Lange, Bjoern Andres, and Paul Swoboda. Combinatorial persistency criteria for multicut and max-cut. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6093–6102, 2019. doi:10.1109/CVPR.2019.00625.

27 Frauke Liers, Michael Jünger, Gerhard Reinelt, and Giovanni Rinaldi. Computing exact ground states of hard Ising spin glass problems by branch-and-cut. *New optimization algorithms in physics*, pages 47–69, 2004. doi:10.1002/3527603794.ch4.

28 Fuda Ma and Jin-Kao Hao. A multiple search operator heuristic for the max-k-cut problem. *Annals of Operations Research*, 248(1):365–403, 2017. doi:10.1007/s10479-016-2234-0.

29 Daniel Rehfeldt, Thorsten Koch, and Yuji Shinano. Faster exact solution of sparse MaxCut and QUBO problems. *Mathematical Programming Computation*, 15(3):445–470, 2023. doi:10.1007/s12532-023-00236-6.

30 Vilmar Jefté Rodrigues de Sousa, Miguel F Anjos, and Sébastien Le Digabel. Computational study of valid inequalities for the maximum k-cut problem. *Annals of Operations Research*, 265(1):5–27, 2018. doi:10.1007/s10479-017-2448-9.

31 Vilmar Jefté Rodrigues de Sousa, Miguel F Anjos, and Sébastien Le Digabel. Computational study of a branching algorithm for the maximum k-cut problem. *Discrete Optimization*, 44:100656, 2022. doi:10.1016/j.disopt.2021.100656.

32 Vilmar Jefté Rodrigues de Sousa, MiguelF Anjos, and Sébastien Le Digabel. Improving the linear relaxation of maximum k-cut with semidefinite-based constraints. *EURO Journal on Computational Optimization*, 7(2):123–151, 2019. doi:10.1007/s13675-019-00110-y.

33 Ryan A. Rossi and Nesreen K. Ahmed. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*, 2015. URL: https://networkrepository.com.

34 Christian L Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. NetworKit: A tool suite for large-scale complex network analysis. *Network Science*, 4(4):508–530, 2016. doi:10.1017/nws.2016.20.

35 Gottfried Tinhofer. A probabilistic analysis of some greedy cardinality matching algorithms. *Annals of Operations Research*, 1(3):239–254, 1984. doi:10.1007/BF01874391.

36 H. P. van Benthem. GRAPH Generating Radio Link Frequency Assignment Problems Heuristically. Master's thesis, Delft University of Technology, 1995.

37 Edwin R van Dam and Renata Sotirov. New bounds for the max-k-cut and chromatic number of a graph. *Linear Algebra and its Applications*, 488:216–234, 2016. doi:10.1016/j.laa.2015.09.043.

38 Chris Walshaw, Mark Cross, and Martin G Everett. Parallel dynamic graph partitioning for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 47(2):102–108, 1997. doi:10.1006/jpdc.1997.1407.

39 Angelika Wiegele. Biq Mac Library – a collection of Max-Cut and quadratic 0-1 programming instances of medium size. http://biqmac.aau.at/biqmaclib.html, 2009.

A Preprocessing Framework

A.1 Offsets

The constant offsets of the discussed rules are as follows:

- **Low degree**: $\sum_{e\in\delta(v)} w(e)$
- **Cliques with a small neighbourhood**: Let c be the unit edge weight of C and $F = N(V_C) \cup V_C$. Then the offset is

$$\frac{c}{2} \cdot \left(|F|^2 - (k - (|F| \mod k)) \cdot \left\lfloor \frac{|F|}{k} \right\rfloor^2 - (|F| \mod k) \cdot \left\lceil \frac{|F|}{k} \right\rceil^2 \right)$$

- **2-Vertex Separators**: $w(p')$
- **Negative Dominating Edges**: 0
- **Negative Dominating Triangles** (see the full version of this paper): 0
- **(Bi)connected Components**: 0
- **Structured Cut Sets**: $\sum_{e\in C} w(e)$
  - If the optimum solution p for w.l.o.g. G_2 is known, then $w(p) + \sum_{e\in C} w(e)$.

A.2 Combining Partitions

Proof of Lemma 9. To compute p from p' and p'' we use the following algorithm:

Algorithm 2 CombinePartitions($p' : V_1 \to \{1, ..., k\}, p'' : V_2 \to \{1, ..., k\}$).

1: Initialize array π of length k with entries -1
2: **for** $v \in V_1 \cap V_2$ **do**
3: $\quad \pi[p''(v)] = p'(v)$
4: Fill entries of π that are still -1 with the numbers from $\{1, ..., k\}$ that are not yet in π
5: Compute the k-partition p of $V_1 \cup V_2$ with

$$p(v) = \begin{cases} p'(v) & \text{if } v \in V_1 \\ \pi[p''(v)] & \text{otherwise} \end{cases}$$

6: **return** p

p clearly satisfies $p_{|V_1} \cong p'$. As $p'_{|V_1\cap V_2} \cong p''_{|V_1\cap V_2}$ we have that the array π we compute stores a permutation such that $p'_{|V_1\cap V_2} = p''^{\pi}_{|V_1\cap V_2}$. We then have for all $v \in V_2$ that $p(v) = \pi[p''(v)]$, therefore $p_{|V_2} \cong p''$.

By realising partitions such that read and write access to $p(v)$ is possible in $\mathcal{O}(1)$ (for example as arrays), this algorithm runs in $\mathcal{O}(|V_1 \cup V_2|) = \mathcal{O}(n)$. ◀

B Data Reduction Rules

B.1 Proof of Theorem 11

Proof. In [3] this was proven using their framework based on graph separators. Our proof is very similar, but based on our proof framework based on graph addition.

We assume w.l.o.g. that $\{u, v\} \in E$ (and if not insert it with weight 0). Let $G[V'] = G' = (V', E', w')$ and $\Delta = w'(p'') - w'(p')$. We now set $\hat{w}(\{u, v\}) = w'(\{u, v\}) - \Delta$ and $\hat{w}(e) = w'(e) = w(e)$ for all other $e \in E'$. The graph $\hat{G} = (V', E', \hat{w})$ is G', except we subtracted Δ from the weight of $\{u, v\}$. Let G'' be the graph that remains after the rule has been applied. Note that $G = \hat{G} + G''$.

We begin by claiming that p' and p'' are both optimum solutions for $\hat{G}$. To show this, we first show that they are the optimum solution in which u, v have the same resp. different colours and then show that they have equal cut value.

As changing the weight of $\{u, v\}$ has no impact on the cut value of any solution that does not cut $\{u, v\}$, p' remains an optimum among these. For the solutions that cut $\{u, v\}$ all of their objective values change by $-\Delta$, so p'' remains an optimum among them. We then get

$$\hat{w}(p'') = w'(p'') - \Delta = w'(p'') - (w'(p'') - w'(p')) = w'(p')$$

Therefore, given an optimum solution p' for G', we either have $p'_{|\{u,v\}} \cong p_{|\{u,v\}}$ or $p''_{|\{u,v\}} \cong p_{|\{u,v\}}$. In either case we can combine an optimum solution for G'' with an optimum solution for G' using Lemma 9. By Theorem 8, the resulting partition p is optimum for G. ◀

B.2 Larger Vertex Separators

A natural question that arises is whether or not we can also use 3 vertex separators or even larger separators for a similar preprocessing strategy. Our clique based rules work on larger separators, but rely on a very special structure of the graphs on the other side that we can not hope for in general. In [3] the usage of 3 vertex cuts was employed for MaxCut. Unfortunately, this can not be generalised to higher values of k.

For MaxCut, given a 3 vertex separator u, v, w, the weights of $\{u, v\}, \{u, w\}$ and $\{v, w\}$ could be modified. Together with the offset, this gives 4 degrees of freedom to encode the differences in objective values for the 4 different optimum values for "u, v, w are in the same group", "u, v are in the same group, w is in the other group", "u, w are in the same group, v is in the other group" and "v, w are in the same group, u is in the other group". For Max k-Cut with $k \geqslant 3$ we would have to encode a fifth optimum value for the case "u, v, w are all in different groups", which makes using 3 vertex cuts in general not possible.

C Proofs and Further Notes for Structured Cut Sets

C.1 Proof of Theorem 14

Proof. We begin with the runtime, as it is the easiest. Computing G_{CR} takes $\mathcal{O}(|C| + k)$ time and its bipartite complement can be computed in time $\mathcal{O}(k^2)$. A maximum matching can be computed in $\mathcal{O}(k^{2.5})$ [19] and, if a matching is found, all further steps take time $\mathcal{O}(n + k)$, leading to a total runtime of $\mathcal{O}(n + |C| + k^{2.5})$.

With this we now move on to the proof of correctness. We will begin by showing that if we find a perfect matching, we find a k-partition p that cuts all $e \in C$ with $p_{|V_1} \cong p'$ and $p_{|V_2} \cong p''$. We then go on to show that we find a perfect matching if such a partition p exists.

If we find a perfect bipartite matching in line 3, then π defines a permutation on $\{1, ..., k\}$. By construction the returned partition p has $p_{|V_1} = p'$ and $p_{|V_2} = p''$, so we only need to show that p cuts every edge $e \in C$.

Let $\{v, w\} \in C$ be an edge with $v \in V_1$ and $w \in V_2$. From this it follows that the edge $\{(p'(v), L), (p''(w), R)\}$ is in G_{CR} and therefore not in its bipartite complement, so it can't be in the matching. Therefore we get $\pi(p''(w)) \neq p'(v)$, so $p(v) = p'(v) \neq \pi(p''(w)) = p(w)$, so e is cut by p.

Now we just have to show that we find a perfect matching if a partition p that cuts all $e \in C$ with $p_{|V_1} \cong p'$ and $p_{|V_2} \cong p''$ exists.

Suppose that such a k-partition p of V exists. W.l.o.g. $p_{|V_1} = p'$ and let π be the permutation such that $p_{|V_2} = p''^{\pi}$. For better readability let $\mathcal{L} = \{(i, L) \mid i \in \{1, ..., k\}\}$ and $\mathcal{R} = \{(i, R) \mid i \in \{1, ..., k\}\}$ be the different sides of the colour relation graph and $H = \overline{G_{CR}[\mathcal{L}, \mathcal{R}]}$ be the

bipartite complement of G_{CR}. We claim that $M = \{\{(i, L), (\pi^{-1}(i), R)\}\}$ is a perfect matching for H. Suppose that there was an edge $\{(i, L), (\pi^{-1}(i), R)\} \in M$ that is not in H. Then there must be an edge $e = \{v, w\} \in C$ with $v \in V_1, w \in V_2$ such that $p'(v) = i, p''(w) = \pi^{-1}(i)$. But then $p(v) = i = \pi(\pi^{-1}(i)) = \pi(p''(i)) = p(w)$, so p does not cut e, a contradiction. Therefore M must be a perfect matching.

With this we have shown that ReconstructStructured Cut Sets finds a k-partition p of V that cuts all $e \in C$ with $p_{|V_1} \cong p'$ and $p_{|V_2} \cong p''$ if one exists in time $\mathcal{O}(n + |C| + k^{2.5})$. and returns an error otherwise. ◀

C.2 Proof of Theorem 18

Proof. Let w.l.o.g. $|\partial_G(V_1)| \geqslant k$ and let $v_1, ..., v_k \in \partial_G(V_1)$ be distinct vertices. We now define $p'(v_i) = i$ and assign all other vertices in V_1 to some arbitrary colour. We define p'' with $p''(v) = 1$ for all $v \in V_2$.

Suppose now that there was a k-partition p of V such that p cuts all edges in C, $p_{|V_1} \cong p_1$ and $p_{|V_2} \cong p_2$. Then w.l.o.g. we assume that $p_{|V_1} = p_1$. Let $j = p(w)$ for all $w \in V_2$. Then there exists an edge $\{v_j, w\}$ for some $w \in V_2$ (as otherwise $v_j \notin \partial_G(V_1)$), so $p(v_j) = j = p(w)$. Therefore p does not cut $\{v_j, w\}$, a contradiction. ◀

C.3 Proof of Theorem 16

Proof. Let p', p'' be optimum solutions for $G_1 = (V_1, E_1, w_1), G_2 = (V_2, E_2, w_2)$ respectively.

If $G[C]$ has at most k vertices, then we can permute p'' such that no two vertices $v \in \partial_G(V_1), w \in \partial_G(V_2)$ have the same colour and therefore all edges in C are cut.

For the case that $G[C]$ has at most $k-1$ edges we show that algorithm 3 always computes a perfect matching, despite only guaranteeing a $\frac{1}{2}$ approximation on general bipartite graphs.

Algorithm 3 ByDegreeAscending($G = (L \sqcup R, E)$).

```
1: Set M = ∅
2: Set G' = G
3: for v_i ∈ L by degree ascending do
4:     Choose any edge e = {v_i, w} in G'
5:     Set M = M ∪ {e}
6:     Delete v_i and w from G
7: return M
```

▶ **Lemma 19.** *Given a bipartite graph $G = (L \sqcup R, E), |L| = |R| = k$ with $|E| \geqslant k^2 - k + 1$ the ByDegereeAscending algorithm 3 computes a perfect matching.*

Proof. To show that the algorithm produces a perfect matching it suffices to show that we can always find an edge e in line 4. Suppose that in iteration i vertex v_i has no more incident edges. As every iteration before has reduced the degree of v_i by at most 1 we can conclude that $d_G(v) \leqslant i - 1$. As every previous iteration selected a different vertex than v_i we can conclude that there at least i vertices in L with degree $\leqslant i - 1$. Therefore we have that $|E| \leqslant k^2 - i \cdot (k - i + 1)$ and therefore

$$i \cdot (k - i + 1) \leqslant k - 1 \Leftrightarrow (k - i) \cdot (i - 1) + 1 \leqslant 0$$

However, $k \geqslant i$ and $i \geqslant 1$, therefore $(k - i) \cdot (i - 1) + 1 \geqslant 1$, a contradiction. Therefore we always find an edge in line 4 and compute a perfect matching. ◀

Algorithm 3 is very similar to the GreedyMin algorithm [35], which selects a vertex of minimum degree in the remaining graph uniformly at random, matches it to a random neighbour and then removes both vertices. We show that GreedyMin also finds an optimum solution under the described circumstances in the full version of this paper.

As we are guaranteed to find a perfect matching, reconstruction is possible. ◀

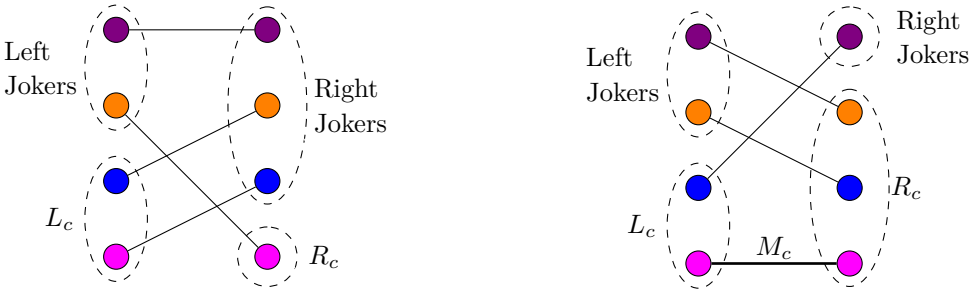
C.4 Proof of Theorem 17

Proof. We begin by noting that this criterion can only hold if $|L|, |R| \leqslant k - 1$.

The key idea of our proof is that we can construct G_{CR} from $G[C]$ by repeatedly contracting two vertices that are on the same side and are assigned to the same group by p' or p'', then adding vertices to both sides of the bipartition until each side has k vertices. We call these newly added vertices jokers, as in the bipartite complement they can be matched to any vertex on the other side. Let L_c, R_c be L and R after these contractions have taken place and let i be the number of contractions.

When contracting two vertices v, v' it may happen that one or both of them are endpoints of edges in M. In this case we remove the edges with one endpoint v or v' from M. We note that in each contraction we can loose at most 2 edges in M. Let M_c denote the matching after the contractions.

We now distinguish two cases based on the number of contractions necessary to obtain G_{CR} from $G[C]$:



(a) Matching construction in the case 1

(b) Matching construction in case 2

Figure 9 The choice of perfect matching on the bipartite complement of G_{CR} for $k = 4$.

Case 1: $i \geqslant |V_C| - k$. In this case M_c may be empty. However, we have $|L_c| + |R_c| \leqslant k$ and therefore the right side of G_{CR} contains at least $|L_c|$ jokers and the left side contains at least $|R_c|$ jokers. We can then find a perfect matching by matching all vertices in L_c with jokers on the right side, matching all vertices in R_c with jokers on the left side and matching any remaining vertices. A visualization can be seen in Figure 9a.

Case 2: $i < |V_C| - k$. In this case M_c still contains $\geqslant 2 \cdot (|V_H| - k) - 1 - 2 \cdot i = 2 \cdot (|L_c| + |R_c| - k) - 1$ edges. We construct our perfect matching by taking all edges in M_c, matching all unmatched vertices in L_c to jokers on the right side and vice versa and then match any remaining jokers. A visualization can be seen in Figure 9b.
For this to work we must show that there are enough jokers on the right side to match to the remaining vertices in L_c (the proof for R_c is symmetric). For this we show that L_c contains fewer vertices than the matching has edges + the number of jokers on the right.

$$
\begin{aligned}
& 0 \leqslant |L_c| + |R_c| - k - 1 \\
\Leftrightarrow \quad & |L_c| \leqslant 2 \cdot (|L_c| + |R_c| - k) - 1 + k - |R_c| \\
\Rightarrow \quad & |L_c| \leqslant |M_c| + k - |R_c|
\end{aligned}
$$

As $i < |V_C| - k$ we must have $|L_c| + |R_c| \geqslant k + 1$, so the first inequality holds, implying the last. ◀

C.5 Further Notes on Structured Cut Sets

A natural question is if we can also split structured cut setsthat contain negative weight edges. Unfortunately that is not the case. Given a cut set $C \subseteq E$ and $\{u, v\} = e \in C$ with $w(e) < 0$, we distinguish two cases:

Case 1: $|C| = 1$. In this case either the connected component containing e contains only two vertices, or at least one endpoint of e is a separating vertex and we can simply apply the rule for splitting biconnected components. Therefore, there is no need to consider the algorithms we developed around structured cut sets.

Case 2: $|C| \geqslant 2$. In this case let $\{u', v'\} = e' \in C, e \neq e'$ and we assume that w.l.o.g. $u \neq u'$, u, u' are on one side of the cut set and v, v' are on the other. If $w(e') < 0$, then if our solutions for G_1, G_2 require that $p(u) \neq p(u')$ and $p(v) = p(v')$, we must cut one of e and e'. If instead the solutions require $p(u) = p(u')$ and $p(v) = p(v')$ it is possible to cut neither.

As such we would somehow have to encode the trade off between cutting neither of e, e' and performing better on G_1 and G_2, which makes the techniques we developed unusable. If $w(e') > 0$ a similar argument can be made.