

Shared-Memory Branch-and-Reduce for Multiterminal Cuts*

Monika Henzinger[†] Alexander Noe[‡] Christian Schulz[§]

Abstract

We introduce the fastest known exact algorithm for the multiterminal cut problem with k terminals. In particular, we engineer existing as well as new data reduction rules. We use the rules within a branch-and-reduce framework and to boost the performance of an ILP formulation. Our algorithms achieve improvements in running time of up to *multiple orders of magnitudes* over the ILP formulation without data reductions, which has been the de facto standard used by practitioners. This allows us to solve instances to optimality that are significantly larger than was previously possible.

1 Introduction

We consider the multiterminal cut problem with k terminals. Its input is an undirected edge-weighted graph $G = (V, E, w)$ with edge weights $w : E \mapsto \mathbb{N}_{>0}$ and its goal is to divide its set of nodes into k blocks such that each block contains exactly one terminal and the weight sum of the edges running between the blocks is minimized. The problem has applications in a wide range of areas, for example in multiprocessor scheduling [38], clustering [36] and bioinformatics [24, 31, 41]. It is a fundamental combinatorial optimization problem which was first formulated by Dahlhaus et al. [12] and Cunningham [11]. It is NP-hard for $k \geq 3$ [12], even on planar graphs, and reduces to the minimum s - t -cut problem, which is in P, for $k = 2$. The minimum s - t -cut problem aims to find the minimum cut in which the vertices s and t are in different blocks. Most algorithms for the minimum multiterminal cut problem use minimum s - t -cuts as a subroutine. Dahlhaus et al. [12] give a $2(1 - 1/k)$ approximation algorithm with polynomial running time. Their approximation algorithm uses the notion of *isolating cuts*, i.e. the minimum cut separating a terminal from all other terminals. They prove that the union of the $k - 1$ smallest isolating cuts yields a valid multiterminal cut with the desired approximation ratio. The currently best known approximation algorithm by Buchbinder et al. [6] uses linear program relaxation to achieve an approximation ratio of 1.323.

While the multiterminal cut problem is NP-hard, it is *fixed-parameter tractable* (FPT), parameterized by the multiterminal cut weight $\mathcal{W}(G)$. A problem is fixed-parameter tractable if there is a parameter σ so that there is an algorithm with runtime $f(\sigma) \cdot n^{\mathcal{O}(1)}$. Marx [29] proves that the multiterminal cut problem is FPT and Chen et al. [9] give the first FPT algorithm with a running time of $4^{\mathcal{W}(G)} \cdot n^{\mathcal{O}(1)}$, later improved by Xiao [42] to $2^{\mathcal{W}(G)} \cdot n^{\mathcal{O}(1)}$ and by Cao et al. [7] to $1.84^{\mathcal{W}(G)} \cdot n^{\mathcal{O}(1)}$. However, to the best of our knowledge, there is no actual implementation for any of these algorithms.

*The research leading to these results has received funding from the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) /ERC grant agreement No. 340506

[†]University of Vienna, Faculty of Computer Science, Vienna, Austria

[‡]University of Vienna, Faculty of Computer Science, Vienna, Austria

[§]University of Vienna, Faculty of Computer Science, Vienna, Austria

The minimum s - t -cut problem and its equivalent counterpart, the maximum s - t -flow problem [15] were first formulated by Harris et al. [19]. Ford and Fulkerson [14] gave the first algorithm for the problem with a running time of $\mathcal{O}(mnW)$. One of the fastest known algorithms in practice is the push-relabel algorithm of Goldberg and Tarjan [18] with a running time of $\mathcal{O}(mn \log(n^2/m))$.

Problems related to the minimum multiterminal cut problem also appear in the data mining community, namely the very similar and heavily studied *seed expansion problem*, for which the aim is to find ground-truth clusters when given a small subset of the cluster vertices. In contrast to the minimum multiterminal cut problem, these clusters might overlap. There is a multitude of approaches adding and removing vertices greedily [2, 10, 27, 30]. PageRank [35] is reported to be well suited for the problem [25] and there are multiple approaches that aim to make PageRank perform even better [1, 4, 26]. Another approach is to use machine learning methods such as geometric [43] or relational [28] neighborhood classifiers.

Closely related to the problem is also the minimum cut problem. For this problem, the goal is to divide the set of nodes in an undirected edge-weighted graph into two blocks while minimizing the weight sum of the cut edges. Both Padberg et al. [34] and Nagamochi et al. [32, 33] give local conditions that are sufficient to contract edges such that the global minimum cut is maintained (and hence the problem size is reduced). An efficient implementation of those conditions is given by Henzinger et al. [23]. In this work, we adapt the conditions from their works that are applicable to the minimum multiterminal cut problem and use them to reduce the size of the problem.

Our paper has the following *main contributions*: We engineer existing as well as new data reduction rules for the minimum multiterminal cut problem with k terminals. These reductions are used within a branch-and-reduce framework as well as to boost the performance of an ILP formulation for the problem. Through extensive experiments we show that kernelization has a significant impact on both, the branch-and-reduce framework as well as the ILP formulation. Our experiments also show a clear trade-off: combining reduction rules with the ILP is very fast for problems which have a small kernel but a high cut value and the fixed-parameter tractable branch-and-reduce algorithm is highly efficient when the cut value is small. Overall, we obtain algorithms that are multiple orders of magnitude faster than the ILP formulation which is de facto standard to solve the problem to optimality.

2 Preliminaries

2.1 Basic Concepts

Let $G = (V, E, w)$ be a weighted undirected graph with vertex set V , edge set $E \subset V \times V$ and non-negative edge weights $w : E \rightarrow \mathbb{N}$. We extend w to a set of edges $E' \subseteq E$ by summing the weights of the edges; that is, $w(E') := \sum_{e=(u,v) \in E'} w(u,v)$. Let $n = |V|$ be the number of vertices and $m = |E|$ be the number of edges in G . The *neighborhood* $N(v)$ of a vertex v is the set of vertices adjacent to v . The *weighted degree* of a vertex is the sum of the weights of its incident edges. For a set of vertices $A \subseteq V$, we denote by $E[A] := \{(u,v) \in E \mid u \in A, v \in V \setminus A\}$; that is, the set of edges in E that start in A and end in its complement. A k -*cut*, or *multicut*, is a partitioning of V into k disjoint non-empty blocks, i.e. $V_1 \cup \dots \cup V_k = V$. The weight of a k -cut is defined as the weight sum of all edges crossing block boundaries, i.e. $w(E \cap \bigcup_{i < j} V_i \times V_j)$.

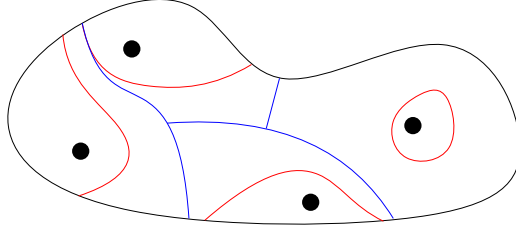


Figure 1: Graph with 4 terminals. Minimum s - T -cut for each terminal shown in red, \mathcal{C} in blue

2.2 Multiterminal Cuts

A *multiterminal cut* for k terminals $T = \{t_1, \dots, t_k\}$ is a multicut with $t_1 \in V_1, \dots, t_k \in V_k$. Thus, a multiterminal cut pairwise separates all terminals from each other. The edge set of the multiterminal cut with minimum weight of G is called $\mathcal{C}(G)$ and the associated optimal partitioning of vertices is denoted as $\mathcal{V} = \{\mathcal{V}_1, \dots, \mathcal{V}_k\}$. \mathcal{C} can be seen as the set of all edges that cross block boundaries in \mathcal{V} , i.e. $\mathcal{C}(G) = \bigcup \{e = (u, v) \mid \mathcal{V}_u \neq \mathcal{V}_v\}$. The weight of the minimum multiterminal cut is denoted as $\mathcal{W}(G) = w(\mathcal{C}(G))$. At any point in time, the best currently known upper bound for $\mathcal{W}(G)$ is denoted as $\widehat{\mathcal{W}}(G)$ and the best currently known multiterminal cut is denoted as $\widehat{\mathcal{C}}(G)$. If graph G is clear from the context, we omit it in the notation. There may be multiple minimum multiterminal cuts, however, we aim to find one multiterminal cut with minimum weight.

In this paper we use *minimum s-T-cuts*. For a vertex s (*source*) and a non-empty vertex set T (*sinks*), the minimum s-T-cut is the smallest cut in which s is one side of the cut and all vertices in T (except for s , if $s \in T$) are on the other side. This is a generalization of minimum s-t-cuts that allows multiple vertices in t and can be easily replaced by a minimum s-t-cut by connecting every vertex in T with a new super-sink by infinite-capacity edges. We denote the capacity of a minimum-s-T-cut, i.e. the sum of weights in the smallest cut separating s from T , by $\lambda(G, s, T)$.

The example in Figure 1 shows a graph with 4 terminals. The minimum s-T-cut for each terminal with T being the set of all terminals is shown in red and the minimum multiterminal cut is shown in blue. We can see that any $k - 1$ minimum s-T-cuts (in red) separate all terminals and are thus a valid multiterminal cut. In our algorithm we use *graph contraction* and *edge deletions*. Given an edge $e = (u, v) \in E$, we define G/e to be the graph after *contracting* e . In the contracted graph, we delete vertex v and all incident edges. For each edge $(v, x) \in E$, we add an edge (u, x) with $w(u, x) = w(v, x)$ to G or, if the edge already exists, we give it the edge weight $w(u, x) + w(v, x)$. For the *edge deletion* of an edge e , we define $G - e$ as the graph G in which e has been removed. Other vertices and edges remain the same.

For a given multiterminal cut S , the graph $G \setminus S$ splits G into k blocks as defined by the cut edges in S , each containing exactly one terminal. Let the residual $R(t_i)$ be the connected component of $G \setminus S$ containing t_i and $\delta(t_i) = |E(R(t_i), V \setminus R(t_i))|$ be the edges in S incident to t_i .

3 Branch and Reduce for Multiterminal Cut

In this section we give an overview of our approach to find the optimal multiterminal cut in large graphs. Our algorithm combines kernelization techniques with an engineered bounded search.

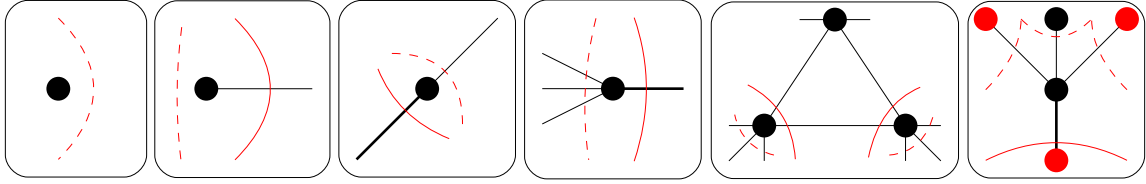


Figure 2: Reductions. Solid line cannot be minimal as dashed line has smaller weight: (1) `IsolatedVertex`, (2) `DegreeOne`, (3) `DegreeTwo`, (4) `HeavyEdge`, (5) `HeavyTriangle` and (6) `SemiEnclosedVertex`

We begin by finding all connected components of G . We can then look at all connected components independently from each other, as there is a trivial cut of weight 0 between different connected components. If a connected component contains only one terminal t , it can be separated from all other terminals by using the whole connected component as the block \mathcal{V}_t belonging to terminal t . Due to it being not connected to any other terminals, the cut value is 0. If a connected component contains no terminals, the result \mathcal{W} is identical no matter which block \mathcal{V} the connected component belongs to. For a connected component C with two terminals s and t , we can run a minimum s-t-cut algorithm on C to find the minimum cut. The optimal blocks \mathcal{V}_s and \mathcal{V}_t then consist of the two sides of the s-t-cut. On a connected component with more than two terminals, the problem is NP-hard [12]. We run our branch and reduce algorithm on this component. As those runs are completely independent, we only look at one connected component in the following and disregard the rest of the graph for now.

For a graph G , we can find an upper bound $\hat{\mathcal{W}}$ which is equal to the sum of minimum s-T-cut weights minus the heaviest of them. $\hat{\mathcal{W}}(G) = \sum_{s \in T} \lambda(G, s, T \setminus \{s\}) - \arg \max_{s \in T} \lambda(G, s, T \setminus \{s\})$. However, this is not necessarily the minimum multiterminal cut.

We can also give a lower bound for the minimum multiterminal cut: as $\lambda(G, s, T \setminus \{s\})$ is by definition minimal, \mathcal{C} has at least as many edges incident to terminal s as $\lambda(G, s, T \setminus \{s\})$. As this is true for every terminal (and every edge is only incident to two vertices), $\mathcal{C}(G) \cdot 2 \geq \sum_{s \in T} \lambda(G, s, T \setminus \{s\})$, so that $\mathcal{C}(G) \geq \sum_{s \in T} \lambda(G, s, T \setminus \{s\}) / 2$.

In our algorithm, we keep a queue \mathcal{Q} of problems. A problem in \mathcal{Q} consists of a graph $G_{\mathcal{Q}}$, a set of terminals, the upper and lower bound for $\mathcal{W}(G_{\mathcal{Q}})$ and the weight sum of all deleted edges in $G_{\mathcal{Q}}$. When our algorithm is initialized, \mathcal{Q} is initialized with a single problem, whose graph is G and whose set of terminals is T . The problem has 0 deleted edges and its lower and upper bound for $\mathcal{W}(G)$ can be set as previously described. As the problem is currently the only one, the global upper bound $\hat{\mathcal{W}}(G)$ is equal to the upper bound of G . Over the course of the algorithm, we repeatedly take a problem from \mathcal{Q} and check whether we can reduce the graph size using our kernelization techniques outlined in Section 4. When possible, we perform the kernelization and push the kernelized problem to \mathcal{Q} . Otherwise, we branch on an edge e adjacent to one of the terminals.

The kernelization techniques detailed in Section 4 reduce the size of the graph by finding edges that are (1) either guaranteed to be in a minimum multiterminal cut or (2) guaranteed not to be part of at least one minimum multiterminal cut. As we only want to find a single multiterminal cut with minimum sum of edge weights, we can delete edges in (1) and contract edges in (2).

In Section 5 we detail the branching procedure which is used if these reduction techniques are unable to find any further reduction possibilities. For any edge e , either it is in the multiterminal cut or it is not. We create two subproblems for G : G/e and $G-e$. We aim to find the minimum mul-

titerminal cut on either. Further details on the branching and edge selection are given in Section 5. We compute upper and lower bounds for each of the problems and follow the branches whose lower bounds are lower than \mathcal{W} , the best cut weight previously found. In Section 8.3 we discuss queue implementation and whether using a priority queue to first process 'promising' problems is useful in practice. We employ shared-memory parallelism by having multiple threads pull problems from \mathcal{Q} .

4 Kernelization

We now show how to reduce the size of our graph to make the problem more manageable. This is achieved by contracting edges that are guaranteed not to be in the minimum multiterminal cut and deleting edges that are guaranteed to be in it. Before we detail the kernelization rules we show that edges not in \mathcal{C} can be safely contracted and edges in \mathcal{C} can be safely deleted if we store the weight sum of all deleted edges so far. The kernelization rules given in the following and outlined in Figure 2 are used to identify such edges.

Lemma 1 [7] *If an edge $e = (u, v) \in G$ is guaranteed not to be in at least one multiterminal cut $\mathcal{C}(G)$ (i.e. $P_u = P_v$), we can contract e and $\mathcal{W}(G/e) = \mathcal{W}(G)$.*

Proof 1 *As $e \notin \mathcal{C}(G)$, $\mathcal{C}(G/e)$ is equal to $\mathcal{C}(G)$ and thus still has weight equal to $w(\mathcal{C}(G)) = \mathcal{W}(G)$. As an edge contraction only removes cuts and does not create any new cuts, an edge contraction can not lower the weight of the minimum multiterminal cut, i.e. $\mathcal{W}(G/e) \geq \mathcal{W}(G)$. As $\mathcal{C}(G/e)$ has weight $\mathcal{W}(G)$, it is a multiterminal cut in G/e with weight equal to $\mathcal{W}(G)$. Thus it is definitely a minimum multiterminal cut with weight $\mathcal{W}(G)$.*

Lemma 1 allows us to reduce the graph size by contracting an edge if we can prove that both incident vertices are in the same partition in \mathcal{V} . The lemma can be generalized trivially to contract a connected vertex set by applying the lemma to each edge connecting two vertices of the set.

Lemma 2 [7] *If an edge $e = (u, v) \in E$ is guaranteed to be in a minimum multiterminal cut, i.e. there is a minimum multiterminal cut $\mathcal{C}(G)$ in which $P_u \neq P_v$, we can delete e from G and $\mathcal{C}(G - e)$ is still a valid minimum multiterminal cut.*

Proof 2 *Let $\mathcal{W}(G)$ be the weight of the minimum multiterminal cut $\mathcal{C}(G)$. We show that for an edge $e \in \mathcal{C}(G)$, $\mathcal{W}(G - e) = \mathcal{W}(G) - w(e)$. Thus, we can delete e (and thus replace G with $G - e$) and store the weight of the deleted edge. Obviously, $\mathcal{C}(G - e)$ has weight equal to $\mathcal{W}(G) - w(e)$, as we just deleted e and all other edges in $\mathcal{C}(G)$ are still in G . By deleting e , the weight of any multiterminal cut can be decreased by at most $w(e)$ (as a multiterminal cut is a set of edges and e can at most be once in that set). As $\mathcal{W}(G)$ is minimal by definition and no cut weight can be decreased by more than $w(e)$, $G - e$ cannot have a minimum multiterminal cut with weight $< \mathcal{W}(G) - w(e)$. Thus, $\mathcal{C}(G - e)$ is a minimum multiterminal cut of $G - e$ with weight $\mathcal{W}(G - e)$.*

Minimum Isolating Cuts When we look at a problem, we first solve the minimum s-T-cut problem for each terminal $s \in T$. This results in one or multiple minimum cuts that separate s from all other terminals. We call the side of the cut containing s the *isolating cut* of s . Dahlhaus et al. [12] prove that there is a minimum multiterminal cut \mathcal{C} in which the complete isolating cut is in \mathcal{V}_s . Thus, according to Lemma 1 we can contract all vertices of the largest isolating cut into a single vertex. In Figure 1 this would result in contracting the red areas into their respective terminals.

This contraction might result in edges connecting terminals. Such an edge $e = (u, v)$, where both u and v are terminal vertices is guaranteed to be a part of $\mathcal{C}(G)$. This comes from the fact that we know $\mathcal{V}_u \neq \mathcal{V}_v$, i.e. u and v are not in the same block in the minimum multiterminal cut, as both u and v are terminals. According to Lemma 2 they can therefore be deleted.

4.1 Local Contraction

We aim to find edges that cannot be part of the minimum multiterminal cut. If we find an edge that can be contracted, we mark it in a union find data structure [17]. This union-find structure is initialized with each vertex as its own block, an edge contraction then merges the two blocks of incident vertices. After all kernelization criteria are tested, we contract all edges that are marked as contractible. As a contraction might open up new contractions in its neighborhood, we run the contraction routines until they do not find any more contractible edges. To ensure low overhead, we run only the first iteration completely and subsequently check only the neighborhoods of vertices that were changed in the previous iteration.

Low-Degree Vertices [7] Figures 2.(1), 2.(2) and 2.(3) show examples of why non-terminal vertices with degree ≤ 2 can be contracted. A non-terminal vertex with no neighbors (**IsolatedVertex**) can be deleted as there is no incident edge that could affect a cut. For a non-terminal vertex v with only one adjacent edge $e = (v, x)$ (**DegreeOne**), e can not be part of the minimum multiterminal cut $\mathcal{C}(G)$. Any multiterminal cut that contains e can be improved by removing e and moving v to the block of its neighbour x . Thus, we can contract e . On a non-terminal vertex with two adjacent edges e_1 and e_2 (**DegreeTwo**), the heavier edge e_1 can not be part of \mathcal{C} , as replacing it with e_2 improves the cut value. If e_1 and e_2 have equal weight, we can contract either (but not both!). These reductions are performed in a single run, which we denote as **Low**.

Heavy Edges We now look to contract heavy edges. **HeavyEdge** (2.(4)) and **HeavyTriangle** (2.(5)) are reductions that were originally used for the minimum cut problem [8, 22, 34]. We adapt them and transfer them to the minimum multiterminal cut problem.

HeavyEdge says that an edge $e = (u, v)$ which has a weight of at least half of the total edge degree of a non-terminal vertex u can be contracted, as any cut containing e can instead also contain all other edges incident to u . If e has at least $\frac{deg(u)}{2}$, all other incident edges together are not heavier.

For a **HeavyTriangle** with vertices v_1, v_2 and v_3 , we can relax the condition. If for two of the vertices the incident triangle edges together are at least as heavy as all other incident edges, we can contract those, as shown in Figure 2.(5). Each of the continuous lines between v_1 and v_2 can be replaced with the dashed line without increasing the value of the cut. Thus, in every case (v_3 can be on either side of the cut), there is an optimal solution in which v_1 and v_2 are in the same block. Thus, we can contract the edge according to Lemma 1.

The condition **SemiEnclosed**, shown in Figure 2.(6), considers a vertex v which is mostly incident to terminal vertices. Let t_1 be the terminal that is most strongly connected to v and t_2 the terminal with second highest connection strength. Now say that v is contracted into any terminal vertex. All edges connecting v with other terminals are then edges connecting terminals and are guaranteed to be in \mathcal{C} . If $w(v, t_1) > w(v, t_2) + \sum_{u \in V \setminus T} w(v, u)$, i.e. (v, t_1) is heavier than the sum of (v, t_2) and all edges connecting v with non-terminals, we can contract v into t_1 . This follows from the fact that the weight of cut edges incident to v is at most $deg(v) - w(v, t_1)$ if v is in the same block as t_1 . If we instead add v to the block of t_2 (or any other block), at most

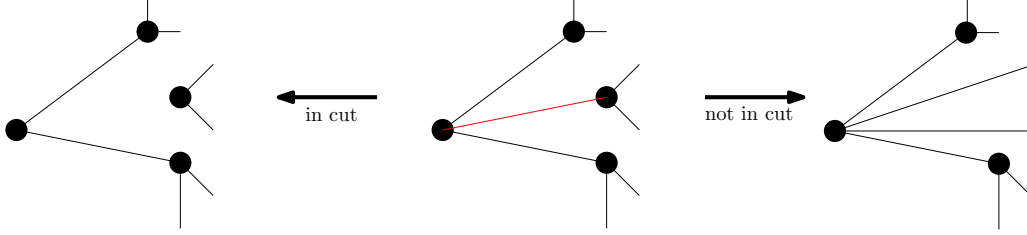


Figure 3: Branch on marked edge e in G , adjacent to a terminal - create two subproblems, (1) G/e and (2) $G - e$.

$w(v, t_2) + \sum_{u \in V \setminus T} w(v, u)$ of the edges incident to v would not be part of the cut. Thus, the locally best choice is contracting v into t_1 . As this does not affect any other graph areas, this choice is guaranteed to be optimal. We check both `HeavyEdge` and `SemiEnclosed` in a single run labelled `High`. `HeavyTriangle` is checked in a run named `Triangle`.

High-connectivity edges The *connectivity* of an edge $e = (u, v)$ is the value of the minimum cut separating u and v . If an edge has connectivity $\geq \widehat{W}(G)$, it is guaranteed that u and v are in the same block in \mathcal{V} , as there can not be a multiterminal cut that separates them and has value $< \widehat{W}(G)$. We can therefore contract u and v . We now show how to improve the bound.

Lemma 3 *If for a graph G with best known multiterminal cut $\widehat{C}(G)$, vertices u and v belong to different connected components of the minimum multiterminal cut $G \setminus \mathcal{C}$, then $\lambda(u, v) + \frac{\sum_{i \in \{1, \dots, t\} \setminus \max_2} \lambda(G, t_i, T \setminus \{t_i\})}{4} \leq |\mathcal{W}(G)|$, where \max_2 is the set of the indices of the largest 2 values $\lambda(G, t_i, T \setminus \{t_i\})$ in the sum.*

Proof 3 *In Appendix A*

We can use Lemma 3 to contract high-connectivity edges. This condition is denoted as `HighConnectivity`. For any edge $e = (u, v)$, if $\lambda(u, v) + \frac{\sum_{i \in \{1, \dots, k\} \setminus \max_2} \lambda(G, t_i, T \setminus \{t_i\})}{4} > |\mathcal{W}| \geq |\widehat{W}|$, u and v are guaranteed to be in the same block in \mathcal{V} . Thus, we can contract them into a single vertex according to Lemma 1.

As it is very expensive to compute the connectivity for every edge, we use the CAPFOREST algorithm of Nagamochi et al. [21, 32, 33] to compute a connectivity lower bound $\gamma(u, v)$ for each edge $e = (u, v)$ in G in near-linear time. If the lower bound $\gamma(u, v)$ fulfills Equation 1, we can use Lemma 3 to contract u and v .

$$\gamma(u, v) > |\widehat{W}| - \frac{\sum_{i \in \{1, \dots, k\} \setminus \max_2} \lambda(G, t_i, T \setminus \{t_i\})}{4} \quad (1)$$

5 Branching Tree Search

If our reductions detailed in Section 4 are unable to contract any edges in G , we branch on an edge adjacent to a terminal. Figure 3 shows an example in which we chose an edge to branch on. For each edge, there are two options: either the edge is part of the minimum multiterminal cut $\mathcal{C}(G)$ or it is

not. Lemmas 1 and 2 show that we can delete an edge that is in $\mathcal{C}(G)$ and contract an edge that is not. Therefore we can build two subproblems, G/e and $G-e$ and add them to the problem queue \mathcal{Q} .

Both of the subproblems will have a higher lower bound and thus, the algorithm will definitely terminate. For $G-e$, we know that e is adjacent to a terminal s but not an edge connecting two terminals (otherwise it would have been deleted). Thus, it is in exactly one minimum s-T-cut $\lambda(G, s, T \setminus \{s\})$. For the lower bound, we half the value of all minimum s-T-cuts. Deleting the edge indicates that it is definitely part of the multiterminal cut. Thus, we increased the lower bound by $w(e) - \frac{w(e)}{2} = \frac{w(e)}{2}$.

For G/e we know that $e = (s, v)$ is part of the largest isolating cut of s (as we contract the largest isolating cut). In G/e terminal s is guaranteed to have a larger minimum s-T-cut, as otherwise there would be an isolating cut of equal value containing v , which contradicts the maximality of the contracted isolating cut. Thus $\lambda(G/e, s, T \setminus \{s\}) > \lambda(G, s, T \setminus \{s\})$ and no other minimum s-T-cut can be decreased by an edge contraction. Thus, the lower bound of $\mathcal{W}(G/e)$ and $\mathcal{W}(G-e)$ are both guaranteed to be higher than the lower bound of $\mathcal{W}(G)$.

Edge Selection In Section 8.2 we evaluate the following edge selection strategies: **HeavyEdge** branches on the heaviest edge incident to a terminal; **HeavyVertex** branches on the edge between the heaviest vertex that is in the neighborhood of a terminal to that terminal; **Connection** searches the vertex that is most strongly connected to the set of terminals and branches on the heaviest edge connecting it to a terminal; **NonTerminalWeight** branches on the edge between the vertex that has the highest weight sum to non-terminal vertices and the terminal it is most strongly connected with; and **HeavyGlobal** branches on the heaviest edge in the graph.

Sub-problem order In Section 8.3 we evaluate the following comparators for the priority queue \mathcal{Q} , i.e. the order in which we look at the problems. A straightforward indicator on whether a problem can lead to a low cut is the current lower and upper bound for the best solution. If a problem has a good lower bound, it has a large potential for improvement and if it has a good upper bound there is already a good solution, potentially close to an even better solution in the neighborhood. Thus, **LowerBound** orders the problems by their lower bound and solves the ones with a better lower bound first while **UpperBound** first examines problems with a lower bound. In either comparator, the respective other bound acts as a tie breaker. **BoundSum** orders problems by the sum of their upper and lower bound.

BiggerDistance first examines problems in which the distance between lower and upper bound is very large. The conceptual idea is that those problems still have many unknowns and thus could be interesting to examine. In contrast to that, **LowerDistance** first examines problems with a lower distance of upper and lower bound, as those branches will likely have fewer sub-branches. Following the same idea, **MostDeleted** first explores the problem that has the highest deleted weight. **SmallerGraph** orders the graphs by the number of vertices and first examines the smallest graph. As over the course of the algorithm a terminal might become isolated (as all incident edges were deleted), not all problems have the same amount of terminals. The isolated terminals are inactive and thus do not need any more flow computations. **FewTerminals** first examines problems with a lower number of active terminals. As there are many solutions with the same amount of terminals, ties are broken using **LowerBound**.

6 Parallel Branch and Reduce

Our algorithm is shared-memory parallel. As we maintain a queue of problems which are independent from each other, we can run our algorithm embarrassingly parallel. The shared-memory priority queue of problems is implemented as a separate queue for each thread to pull from. When a thread adds a problem to the priority queue, it is added to a random queue with minimum queue size. In order to exploit data and cache locality, we add problems to the queue of the local thread if it is one of the queues with minimum size. Additionally, we fix each thread to a single CPU thread in order to actually use those locality benefits. In the beginning of the algorithm, there is only a single problem, which would leave all except for one processors idle, potentially for a long time, as we have to solve k flow problems on the whole (potentially very large) graph. Thus, if there are idle processors, we distribute the flow problems over different threads.

7 Combining Kernelization with ILP

Multiterminal cut problems are generally solved in practice using integer linear programs [31]. The following ILP formulation is adapted from [20] and implemented using Gurobi 8.1.1. It is functionally equal to [31].

$$\begin{aligned} \min \quad & \sum_{\{u,v\} \in E} e_{uv} \cdot w(\{u,v\}) \\ & \forall \{u,v\} \in E, \forall k : e_{uv} \geq x_{u,k} - x_{v,k} \\ & \forall \{u,v\} \in E, \forall k : e_{uv} \geq x_{v,k} - x_{u,k} \\ & \forall v \in V : \sum_k x_{v,k} = 1 \\ & \forall i, j : x_{t_i,j} = [i = j] \end{aligned}$$

Here, $x_{u,k}$ is 1 iff vertex u is in V_k and 0 otherwise and e_{uv} is 1 iff (u,v) is a cut edge. We use this ILP formulation as a baseline of comparison. Additionally, we also create a new algorithm that combines the kernelization of our algorithm with integer linear programming. Using flow computations and kernelization routines, we are able to significantly reduce the size of most graphs while still preserving the minimum multiterminal cut. As the complexity of the ILP depends on the size of the graph and the complexity of the branch-and-reduce algorithm also depends on the value of the cut, this is fast on graphs with a high cut value in which the kernelization routines can reduce the graph to a very small size but with a large cut value. In the following, our algorithm `Kernel+ILP` first runs kernelization until no further reduction is possible and then solves the problem using the above integer linear programming formulation.

8 Experiments and Results

We now perform an experimental evaluation of the proposed algorithms. This is done in the following order: first analyze the impact of algorithmic components on our branch-and-reduce algorithm in a non-parallel setting, i.e. we compare different variants for branching edge selection,

priority queue comparator and the effects of the kernelization operators. We then report the speedup over ILP formulation and as well as parallel speedup on a variety of graphs. Lastly, we perform experiments on large real-world networks of various sources and protein-protein interaction graphs comparing `Kernel+ILP` and the branch-and-reduce algorithm.

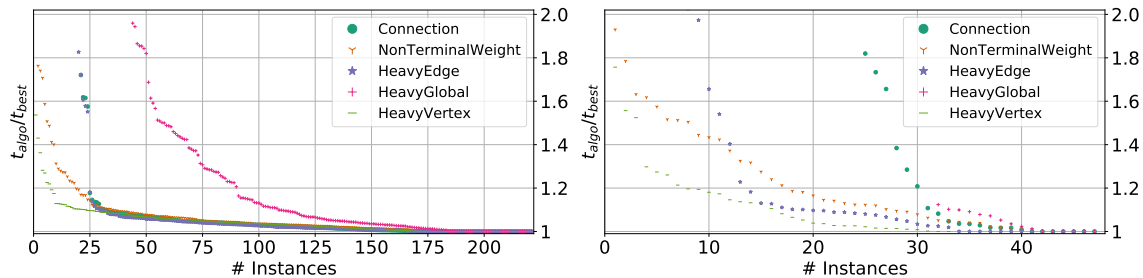
8.1 Experimental Setup and Methodology

We implemented the algorithms using C++-17 and compiled all codes using g++-7.4.0 with full optimization (`-O3`). Our experiments are conducted on a machine with two Intel Xeon Gold 6130 with 2.1GHz with 16 CPU cores each and 256 GB RAM in total. We perform five repetitions per instance and report average running time. In this section we first describe experimental methodology. Afterwards, we evaluate different algorithmic choices in our algorithm and then we compare our algorithm to the state of the art. When we report a mean result we give the geometric mean as problems differ strongly in result and time.

Performance plots relate the fastest running time to the running time of each other algorithm on a per-instance basis. For each algorithm, these ratios are sorted in increasing order. The plots show the ratio $t_{\text{algorithm}}/t_{\text{best}}$ on the y-axis. A point significantly above one indicates that the running time of the algorithm was considerably worse than the fastest algorithm on the same instance. A value of one therefore indicates that the corresponding algorithm was one of the fastest algorithms

Table 1: Large Real-world Benchmark Instances

Graph	n	m
Section 8.7: Social, Web and Map Graphs		
bcstk30 [37]	28 924	1.01M
ca-2010 [3]	710K	1.74M
ca-CondMat [13]	23 133	93 439
cit-HepPh [13]	34 546	422K
eu-2005 [5]	862K	16.1M
higgs-twitter [13]	457K	14.9M
in-2004 [5]	1.38M	13.6M
ny-2010 [3]	350K	855K
uk-2002 [5]	18.5M	261M
vibrobox [37]	12 328	165K
Section 8.8: Protein-protein Interaction		
Acidithiobacillus ferrivorans	3 093	5 394
Agaricus bisporus	11 271	14 636
Candida maltosa	5 948	19 462
Escherichia coli	4 127	13 488
Erinaceus europaeus	19 578	68 066
Homo sapiens	19 566	324K
Mesoplasma florum	683	2 365
Saccharomyces cerevisiae	6 691	69 809
Toxoplasma gondii	7 988	11 779
Vitis vinifera	29 697	70 206



(a) RHG graphs with partition centers as terminals (b) Map graphs with partition centers as terminals

Figure 4: Performance plots for branching edge selection variants

to compute the solution. Thus an algorithm is considered to outperform another algorithm if its corresponding values are below those of the other algorithm.

8.1.1 Instances

We use multiple set a instances to avoid overtuning the branch-and-reduce algorithm. To analyze the impact of algorithmic components in Sections 8.2 to 8.4, we generate random hyperbolic graphs using the KaGen graph generator [16]. These graphs have $n = 2^{14} - 2^{18}$ and an average degree of 8, 16 and 32. For each graph size, we use three generated graphs and compute the multiterminal cut, each with $k \in \{3, 4, 5, 6, 7\}$. We use random hyperbolic graphs as they have power-law degree distribution and resemble a wide variety of real-world networks. Additionally, we also use a family of weighted graphs from the 10th DIMACS implementation challenge [3]. These graphs depict US states, where a vertex depicts a census block and a weighted edge denotes the length of the border between two blocks. We use the 10 states with fewest census blocks (AK, CT, DE, HI, ME, NH, NV, RI, SD, VT). For each state, we set the number of terminals $k \in \{3, 4, 5, 6, 7\}$. A multiterminal cut on these graphs depicts the shortest border that respects census blocks and separates a set of pre-defined blocks (or groups of blocks). Here, we use one processor and set a timeout of 3 minutes and a memory limit of 20GiB. On these instances we also run the ILP and compare the results in Section 8.5 – note that running the ILP itself without the kernelization rules on the large instances below is not feasible. We then look at the impact of parallelization for our algorithms in Section 8.6.

When comparing `Kernel+ILP` with our branch and reduce framework in Sections 8.7 and 8.8 on large instances, we use all 32 cores of the machine (for the ILP as well as the branch and reduce framework). Here, we set a time limit of 1 hour and a memory limit of 250GiB and use the following graphs In Section 8.7 we perform experiments on 10 large real-world networks of various sources. Table 1 shows the sources and properties of the graphs. For each graph, we solve the minimum multiterminal cut problem for $k \in \{3, 4, 5, 8\}$ terminals and $p \in \{10\%, 15\%, 20\%, 25\%\}$ vertices in the terminal. In Section 8.8 we perform experiments on protein-protein interaction networks generated from the STRING protein interaction database [39, 40] by using all edges they predict with a high certainty. We use the protein description to assign functions (block terminal affiliations) to proteins (vertices). We use the first occurrence of a set of pre-defined function classes. For each graph, we examine problems with the 4, 5, 6, 7, 8 most often occurring functions and with all (up to 15, if all occurring in an organism) classes.

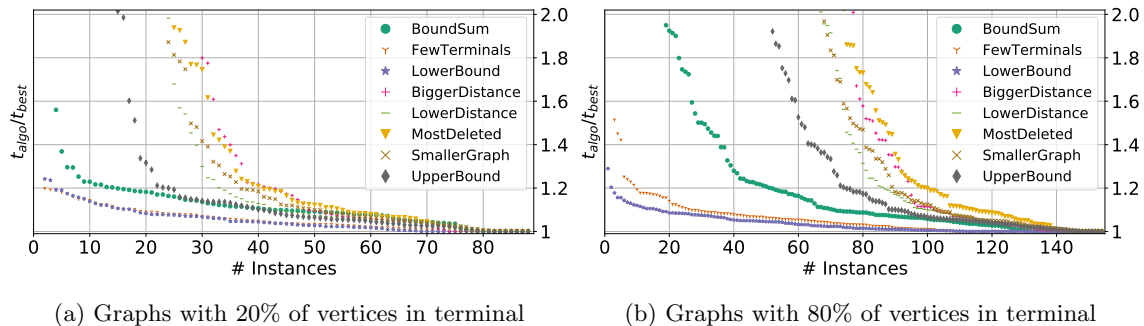


Figure 5: Performance plots for priority queue comparator variants

8.2 Branching Edge Selection

Figure 4 shows the results for the branching edge selection rules. In Subfigure 4a, we show performance plots for RHG graphs and in Subfigure 4b we show performance plots for map graphs. To find terminals, we partition the RHG graphs into k parts and perform a breadth-first search starting in the block boundary. We define the vertex encountered last as the block center and use it as a terminal. In this experiment we use the `BoundSum` comparator and enable all kernelization rules.

As the minimum multiterminal cut of those problems usually turns out to be the trivial multiterminal cut of $k - 1$ blocks of size 1 and one block that comprises of the rest of the graph, we instead pick the last 10 vertices encountered by the breadth-first search per block and contract them into a terminal. The minimum multiterminal cut of the resulting graph is usually not equal to the trivial multiterminal cut.

In general, we aim to increase the lower bound by a large margin to reduce the number of subproblems that need to be checked. When we branch on a heavy edge, this increases the lower bound for $G - e$ by a large amount. For G/e , the lower bound is increased by half the amount of flow that is now added to the network. For a vertex that has a large number of edges to non-terminal vertices, contracting it into a terminal is expected to increase the flow by a large margin. The variant `HeavyVertex` chooses the edge e , for which the sum of edge weight and outgoing weights are maximized. It thus outperforms all other variants in both experiments. The only variant that is not guaranteed to be fixed-parameter tractable is `HeavyGlobal`, as this variant can also contract edges that are not incident to a terminal (and thus do not necessarily increase the lower bound). However, most edge contractions happen near terminals, so most heavy edges occur near terminals and thus `HeavyGlobal` often performs similar to `HeavyEdge`.

In all following experiments we use `HeavyVertex`, as it outperforms all other variants consistently.

8.3 Priority Queue Comparator

We now explore the effect of the comparator used in the priority queue \mathcal{Q} . The choice of comparator decides which problems are highest priority and will be explored first. We want to first explore the problems and branches which will result in an improved solution, as this allows us to prune more branches. However, it is not obvious which criterion correctly identifies problems that

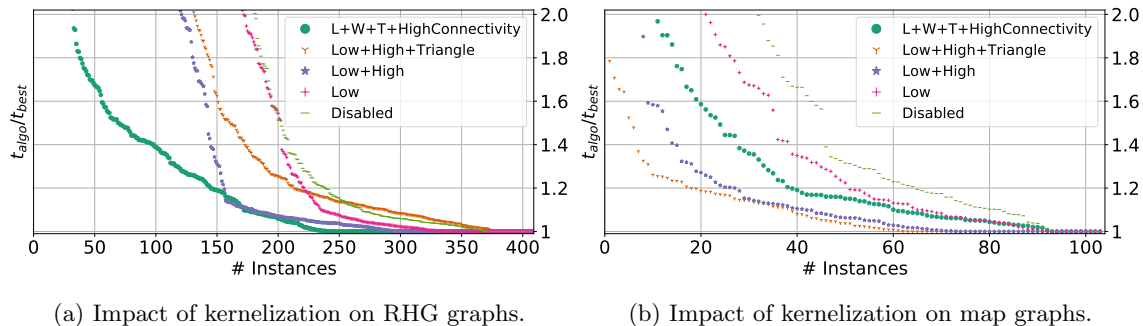


Figure 6: Performance plots for kernelization variants

might yield improved solutions, either directly or indirectly. Thus, we perform experiments on the same set of random hyperbolic and map graphs.

On the random hyperbolic graphs examined in the previous experiment, the minimum multi-terminal cut is often equal to the sum of all minimum-s-T-cuts excluding the heaviest. This is the cut that is found in the first iteration. If this is also the optimal cut, we definitely have to check all subproblems whose lower bound is lower than this cut. As the priority queue comparator only changes the order in which we examine those problems, the experimental results using the same problems as the previous section turned out very inconclusive. However, if we contract a sizable fraction of each block into its terminal, the minimum multiterminal cut is usually not equal to the union of s-T-cuts. Figure 5a shows results for 20% of vertices in the terminal on RHG graphs and Figure 5b shows results for 80% of vertices in the terminal.

LowerBound and **FewTerminals** are very competitive on most graphs. This indicates that problems with a low lower bound are very likely to yield improved results. The next fastest variant is **BoundSum**, which is almost competitive with 20% of vertices in the terminal but significantly slower with 80% of vertices in the terminal. However, **BoundSum** uses far less memory, as the lower bound of the newly created problems depends on the lower bound of the current problem. **BoundSum** examines many problems for which the lower bound is close to the currently best known solution. Thus, many newly created subproblems are immediately discarded when their lower bound is not lower than the currently best known solution. None of the other variants have noteworthy performance.

8.4 Kernelization

We now study the effects of the kernelization operations performed in this work. For this purpose, we compare our algorithm without any kernelization to variants that enable different subsets of the kernelization operators detailed in Section 4. Figure 6a shows the results on the RHG graphs and Figure 6b shows the results on the map graphs. We use **BoundSum** as the priority queue comparator. In both cases, we combine results of 10 vertices, 20% and 80% of vertices near block center in the terminal. The requirements in **Low** and **High** can be checked quickly whereas checking **Triangle** and **HighConnectivity** requires significant time. Thus, running **Low+High** is always useful, no matter how many edges can actually be contracted. On the RHG graphs in Figure 6a, **Triangle** does not find a lot of contractible edges that weren't already found by the previous kernelization operators. The high-degree vertices in the center of the hyperbolic plane have very high connectivity

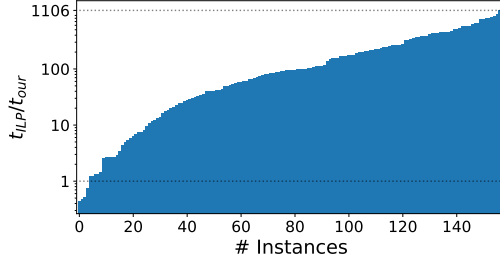


Figure 7: Speedup of optimized branch-and-reduce to ILP

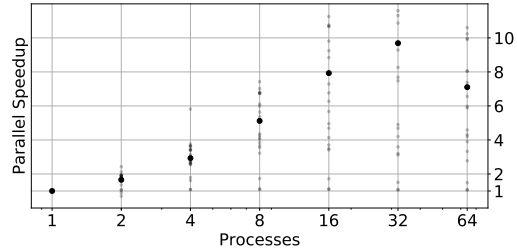


Figure 8: Parallel speedup on a variety of graphs. (low-alpha dot: one graph, solid dot: average speedup)

and thus, `HighConnectivity` is able to significantly reduce graph sizes and significantly improve running times compared to all other variants. In contrast, on the map graphs in Figure 6b, the connectivity of an edge can only be as high as the border length of the smaller vertex. Thus, we do not find many contractible edges. `Triangle`, however, is able to find many edges to contract, as vertices usually have few high degree neighbours. We can see that the utility of the kernelization operators depends heavily on the structure of the graph.

8.5 Comparison to ILP

Figure 7 shows the speedup of the engineered branch-and-reduce algorithm, using `HeavyVertex` edge selection, `LowerBound` priority queue comparator and all kernelization rules enabled, to the ILP on all graphs from the previous subsections in which the ILP managed to find the minimum multiterminal cut within 3 minutes. The branch-and-reduce algorithm outperforms the ILP on almost all graphs, often by multiple orders of magnitude. The ILP only solves 24% of all problems, our algorithm solves 61%; on the problems solved by both, our optimized algorithm has a mean speedup factor of 67, a median speedup factor of 95 and a maximum speedup factor of 1106. The mean speedup factor of the average of our algorithms compared to ILP is 43 with a median speedup factor of 71. Compared to the original ILP, `Kernel+ILP` is faster on all instances, has a mean speedup factor of 44 and a median speedup factor of 49. For figures, see Appendix B.

This allows us to solve instances with more than a million vertices, while the ILP was unable to solve any instance with more than 100 000 vertices. As the basic ILP is unable to solve any large instances, we do not use it in the following experiments on large graphs.

8.6 Parallel Branch and Reduce

The previous experiments were all performed sequentially on a single thread. To see parallel speedup, we chose the 14 RHG and 9 map graphs that required the longest running time out of all terminated instances and solve them with varying amounts of processes. Figure 8 shows the speedup for all graphs. The machine has 2 processors with 16 cores each and hyperthreading enabled. We can see that the average speedup factor is 7.9x with 16 threads, i.e. one thread for each core of a single CPU. The speedup only slightly increases when using both CPUs and 32 cores, to a factor of 9.7x. This is caused by the large amount of data which needs to be

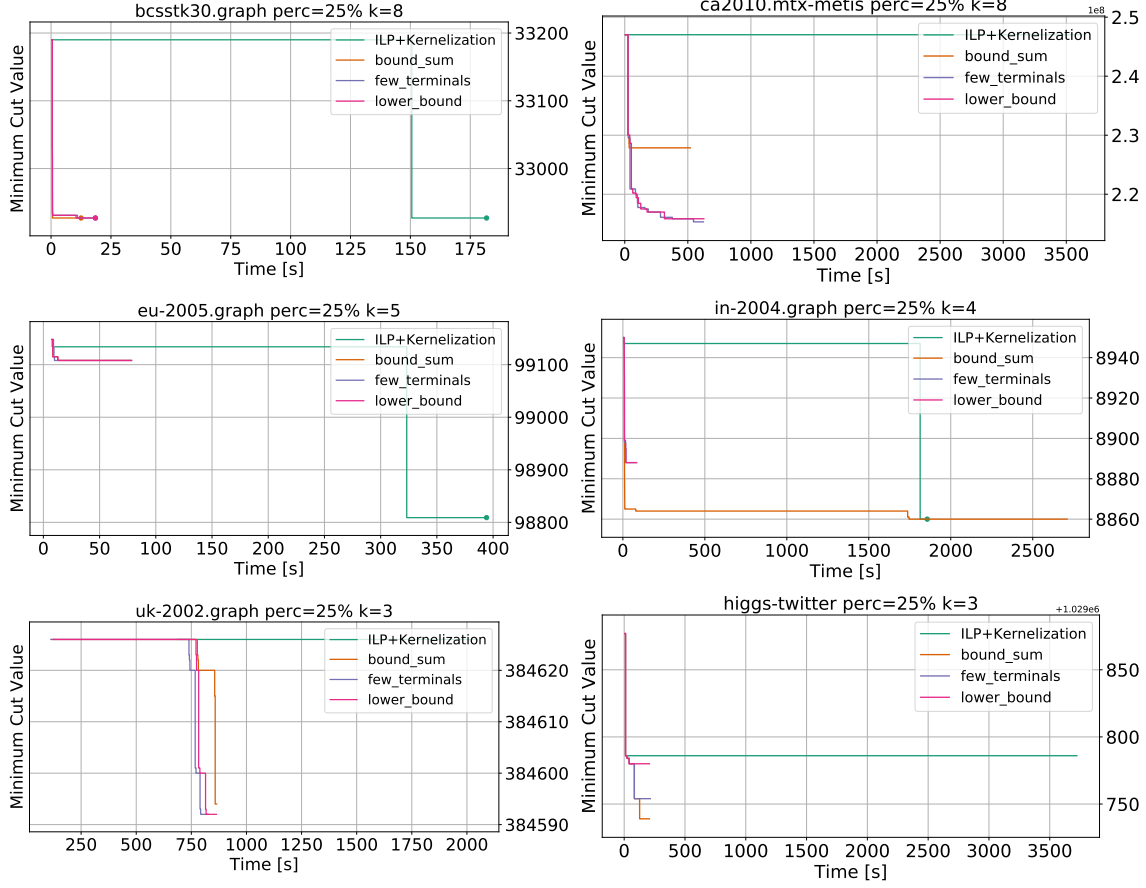


Figure 9: Progression of best result over time. Dot at end symbolizes that algorithm certifies optimality.

transferred on the comparatively slower connection between the processors. When using hyper-threading, the average speedup even slows down to a factor of 7.1x.

On a few problems there is almost no parallel speedup. These are the problems in which there is a large amount of work in a single graph and the flow problems are of starkly different difficulties. Thus, we would need to solve the flow problems in parallel to achieve a speedup in these graphs. As we only observed this behaviour in a few cases in the smaller RHG graphs and never in any large real-world networks, we refrained from including that additional complexity in our algorithm. If we exclude these 3 problems from the problem set, we have an average speedup factor of 8.9x with 16 threads and of 10.9x with 32 threads. `Kernel+ILP` achieved almost no parallel speedup on these problems. This is the case as a large part of the solving time is spent in the sequential root relaxation.

Algorithm	K+ILP	BSum	FTerm	LBound
best result	118	136	126	125
terminated	46	35	33	33
mean result	146 570	145 961	146 052	146 025
mean time	18,69s	6,71s	6,97s	6,78s

Table 2: Overview of large real-world networks.

Algorithm	K+ILP	BSum	FTerm	LBound
best result	57	34	26	23
terminated	57	25	23	21
mean result	4 183	4 210	4 218	4 222
mean time	0,21s	0,33s	0,36s	0,40s

Table 3: Overview of protein-protein-networks.

8.7 Large Real-World Networks

We aim to solve multiterminal cut problems on the large real-world networks in Table 1 from a wide variety of graph and problem classes. Figure 9 shows the progression of the best result over time for a set of interesting problems. Table 2 gives an overview over the results. For each variant we show how often it produced the best result over all variants and how often it terminated with the optimal result. It also gives the mean result and time for all problems which were solved to optimality by all variants. Both in Figure 9 and Table 2 we can see that the branch and reduce variants find good solutions faster than `Kernel+ILP`. However, the variants often run out of memory in some of the largest instances. Especially in cases where the best multiterminal cut was already found (but not confirmed to be optimal) by the kernelization, `Kernel+ILP` managed to certify optimality more often than the branching variants. Thus it has the highest amount of terminated results, but reports significantly worse results in average. `Kernel+ILP` has about half as much improvements as the best variant `BoundSum`. In addition to giving the best results, variant `BoundSum` also has the lowest mean time in problems which were solved by all variants, however the improvement over the other branch-and-reduce variants is miniscule. The correlation between running time and number of vertices in the kernel graph is much stronger in `Kernel+ILP` compared to the branching variants.

8.8 Protein-Protein Interaction Networks

Multiterminal cuts can be used for protein function prediction by creating a terminal for each possible protein function and adding all proteins which have this function to this terminal [24, 31, 41]. Table 3 shows the results for these graphs. We can see that `Kernel+ILP` outperforms branch-and-reduce by a large margin on most graphs. This is the case because the kernelization is able to reduce the size of the graphs severely. These small problems with high cut values are better suited for `Kernel+ILP` than the branch-and-bound variants whose running time is more correlated with the value of the minimum multiterminal cut. The mean times are very low as some problems can be solved very quickly and thus drag the mean of all algorithms down.

9 Conclusion

In this paper, we engineered data reduction rules for the minimum multiterminal cut problem with k terminals. These reductions are used within a branch-and-reduce framework as well as to boost the performance of an ILP formulation for the problem. Our experiments a) show that kernelization – especially our newly introduced kernelization operators – has a significant impact on both, the branch-and-reduce framework as well as the ILP formulation and b) show a clear trade-off: combining reduction rules with the ILP is very fast for problems which have a small kernel but a high cut value and the fixed-parameter tractable branch-and-reduce algorithm is highly efficient when the cut value is small. Overall, we obtain algorithms that are multiple orders of magnitude faster than the ILP formulation which is de facto standard to solve the problem to optimality. Future work includes combining the branching algorithm with integer linear programming so that all occurring subproblems can be solved using the algorithm best suited for their problem properties. We also aim to introduce scalable distributed parallelism.

References

- [1] R. Andersen, F. Chung, and K. Lang. Local graph partitioning using pagerank vectors. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 475–486. IEEE, 2006.
- [2] R. Andersen and K. J. Lang. Communities from seed sets. In *Proceedings of the 15th international conference on World Wide Web*, pages 223–232. ACM, 2006.
- [3] D. A. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner. Benchmarking for graph clustering and partitioning. *Encyclopedia of Social Network Analysis and Mining*, pages 73–82, 2014.
- [4] Y. Bian, J. Ni, W. Cheng, and X. Zhang. Many heads are better than one: Local community detection by the multi-walker chain. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 21–30. IEEE, 2017.
- [5] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proceedings of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.
- [6] N. Buchbinder, J. S. Naor, and R. Schwartz. Simplex partitioning via exponential clocks and the multiway cut problem. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 535–544. ACM, 2013.
- [7] Y. Cao, J. Chen, and J.-H. Fan. An $o(1.84 k)$ parameterized algorithm for the multiterminal cut problem. *Information Processing Letters*, 114(4):167–173, 2014.
- [8] C. S. Chekuri, A. V. Goldberg, D. R. Karger, M. S. Levine, and C. Stein. Experimental study of minimum cut algorithms. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '97)*, pages 324–333. SIAM, 1997.
- [9] J. Chen, Y. Liu, and S. Lu. An improved parameterized algorithm for the minimum node multiway cut problem. *Algorithmica*, 55(1):1–13, 2009.

- [10] A. Clauset. Finding local community structure in networks. *Physical review E*, 72(2):026132, 2005.
- [11] W. H. Cunningham. The optimal multiterminal cut problem. In *Reliability of computer and communication networks*, pages 105–120, 1989.
- [12] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The complexity of multiterminal cuts. *SIAM Journal on Computing*, 23(4):864–894, 1994.
- [13] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.
- [14] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.
- [15] L. R. Ford Jr and D. R. Fulkerson. *Flows in networks*. Princeton university press, 2015.
- [16] D. Funke, S. Lamm, P. Sanders, C. Schulz, D. Strash, and M. von Looz. Communication-free massively distributed graph generation. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 336–347. IEEE, 2018.
- [17] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of computer and system sciences*, 30(2):209–221, 1985.
- [18] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988.
- [19] T. Harris and F. Ross. Fundamentals of a method for evaluating rail net capacities. Technical report, RAND CORP SANTA MONICA CA, 1955.
- [20] A. Henzinger, A. Noe, and C. Schulz. ILP-based Local Search for Graph Partitioning. *Proceedings of the 17th International Symposium on Experimental Algorithms (SEA 2018)*, 2018.
- [21] M. Henzinger, A. Noe, and C. Schulz. Shared-memory Exact Minimum Cuts. *Proceedings of the 33rd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019.
- [22] M. Henzinger, A. Noe, C. Schulz, and D. Strash. Practical minimum cut algorithms. In *2018 Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 48–61. SIAM, 2018.
- [23] M. Henzinger, S. Rao, and D. Wang. Local flow partitioning for faster edge connectivity. In *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1919–1938. SIAM, 2017.
- [24] U. Karaoz, T. Murali, S. Letovsky, Y. Zheng, C. Ding, C. R. Cantor, and S. Kasif. Whole-genome annotation by using evidence integration in functional-linkage networks. *Proceedings of the National Academy of Sciences*, 101(9):2888–2893, 2004.
- [25] I. M. Kloumann and J. M. Kleinberg. Community membership identification from small seed sets. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1366–1375. ACM, 2014.

- [26] J. Leskovec, K. J. Lang, and M. Mahoney. Empirical comparison of algorithms for network community detection. In *Proceedings of the 19th international conference on World wide web*, pages 631–640. ACM, 2010.
- [27] F. Luo, J. Z. Wang, and E. Promislow. Exploring local community structures in large networks. *Web Intelligence and Agent Systems: An International Journal*, 6(4):387–400, 2008.
- [28] S. A. Macskassy and F. Provost. A simple relational classifier. Technical report, NEW YORK UNIV NY STERN SCHOOL OF BUSINESS, 2003.
- [29] D. Marx. Parameterized graph separation problems. *Theoretical Computer Science*, 351(3):394–406, 2006.
- [30] A. Mislove, B. Viswanath, K. P. Gummadi, and P. Druschel. You are who you know: inferring user profiles in online social networks. In *Proceedings of the third ACM international conference on Web search and data mining*, pages 251–260. ACM, 2010.
- [31] E. Nabieva, K. Jim, A. Agarwal, B. Chazelle, and M. Singh. Whole-proteome prediction of protein function via graph-theoretic analysis of interaction maps. *Bioinformatics*, 21(suppl_1):i302–i310, 2005.
- [32] H. Nagamochi and T. Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM Journal on Discrete Mathematics*, 5(1):54–66, 1992.
- [33] H. Nagamochi, T. Ono, and T. Ibaraki. Implementing an efficient minimum capacity cut algorithm. *Mathematical Programming*, 67(1):325–341, 1994.
- [34] M. Padberg and G. Rinaldi. An efficient algorithm for the minimum capacity cut problem. *Mathematical Programming*, 47(1):19–36, 1990.
- [35] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.
- [36] U. Pferschy, R. Rudolf, and G. J. Woeginger. Some geometric clustering problems. *Nord. J. Comput.*, 1(2):246–263, 1994.
- [37] A. J. Soper, C. Walshaw, and M. Cross. A combined evolutionary search and multilevel optimisation approach to graph-partitioning. *Journal of Global Optimization*, 29(2):225–241, 2004.
- [38] H. S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Trans. Software Eng.*, 3(1):85–93, 1977.
- [39] D. Szklarczyk, A. Franceschini, M. Kuhn, M. Simonovic, A. Roth, P. Minguéz, T. Doerks, M. Stark, J. Muller, P. Bork, et al. The string database in 2011: functional interaction networks of proteins, globally integrated and scored. *Nucleic acids research*, 39(suppl_1):D561–D568, 2010.
- [40] D. Szklarczyk, A. L. Gable, D. Lyon, A. Junge, S. Wyder, J. Huerta-Cepas, M. Simonovic, N. T. Doncheva, J. H. Morris, P. Bork, et al. String v11: protein–protein association networks with increased coverage, supporting functional discovery in genome-wide experimental datasets. *Nucleic acids research*, 47(D1):D607–D613, 2018.

- [41] A. Vazquez, A. Flammini, A. Maritan, and A. Vespignani. Global protein function prediction from protein-protein interaction networks. *Nature biotechnology*, 21(6):697, 2003.
- [42] M. Xiao. Simple and improved parameterized algorithms for multiterminal cuts. *Theory of Computing Systems*, 46(4):723–736, 2010.
- [43] W. Ye, L. Zhou, D. Mautz, C. Plant, and C. Böhm. Learning from labeled and unlabeled vertices in networks. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1265–1274. ACM, 2017.

A Proofs

In order to prove Lemma 3 we first prove the following useful claim:

Claim 1 *For any two nodes u and v , if u and v belong to different connected components of $G \setminus \mathcal{C}(G)$, then $\lambda(u, v) \leq \frac{\sum_{i \in \{1, \dots, k\}} \delta(R(t_i))}{4} + \frac{\delta(R(u)) + \delta(R(v))}{4}$, where δ are the weighted node degrees in the quotient graph corresponding to $\mathcal{C}(G)$ and $R(x)$ is the block of a vertex x as defined by the cut $\mathcal{C}(G)$.*

Proof 4 *Let G_R be the contracted graph where every block $R(t_i)$ in G is contracted into a single vertex and let $|S(u, v)|$ be a minimum u - v -cut in G_R . By definition of the minimum cut $\lambda(u, v)$, $\lambda(u, v) \leq |S(u, v)|$. Slightly abusing the notation we denote by $R(t_i)$ the vertex of G_R that results from contracting the block $R(t_i)$.*

For every vertex $w \in G_R$ that does not represent a block that contains either u or v , at most $\frac{\deg(w)}{2}$ edges are in $|S(u, v)|$. This follows directly from the assumption that $|S(u, v)|$ is minimal. If more than $\frac{\deg(w)}{2}$ edges incident to w are in $|S(u, v)|$, moving w to the other side of the cut would give a better cut. Thus, at most half of the edges incident to w are in $|S(u, v)|$.

We can not make this argument for the blocks containing u and v , as potentially all edges incident to their blocks could be in the minimum multiterminal cut. Thus, $2 \cdot |S(u, v)| \leq \frac{\sum_{i \in \{1, \dots, k\}} \delta(R(t_i))}{2} + \frac{\delta(R(u))}{2} + \frac{\delta(R(v))}{2}$. The factor 2 on the left side is caused by the fact that every edge is incident to two blocks. As we do not know the multiterminal cut S , we need to assume that they could be the blocks with the largest cuts $\delta(R(t_i))$. Dividing each side by 2 finishes the proof.

Claim 2 *For any two nodes u and v , if u and v belong to different connected components of $G \setminus \mathcal{C}(G)$, then $\lambda(u, v) + \frac{\sum_{i \in \{1, \dots, k\}} \delta(R(t_i))}{4} \leq \mathcal{W}$.*

Proof 5 *Using Claim 1 we know that $\lambda(u, v) + \frac{\sum_{i \in \{1, \dots, k\}} \delta(R(t_i))}{4} \leq \frac{\sum_{i \in \{1, \dots, k\}} \delta(R(t_i))}{2}$. By definition of δ , $\frac{\sum_{i \in \{1, \dots, k\}} \delta(R(t_i))}{2} = \mathcal{W}(G)$.*

We now use Claims 1 and 2 to prove Lemma 3.

Proof 6 *Let vertices u and v be in different blocks. Then $\lambda(u, v) + \frac{\sum_{i \in \{1, \dots, t\} \setminus \max_2} \lambda(G, t_i, T \setminus \{t_i\})}{4} \leq \lambda(u, v) + \frac{\sum_{i \in \{1, \dots, t\} \setminus \max_2} \delta(R(t_i))}{4} \leq \frac{\sum_{i \in \{1, \dots, t\} \setminus \max_2} \delta(R(t_i))}{2} = \mathcal{W}(G)$.*

The first inequality follows from the fact that λ is per definition the minimal cut separating t from $T \setminus \{t_i\}$ and thus $\lambda(G, t_i, T \setminus \{t_i\}) \leq \delta(R(t_i))$.

Thus, we know that if $\lambda(u, v) + \frac{\sum_{i \in \{1, \dots, t\} \setminus \max_2} \lambda(G, t_i, T \setminus \{t_i\})}{4} > \mathcal{W}(G)$, u and v are in the same block and the edge connecting them can be safely contracted.

B Additional Figures

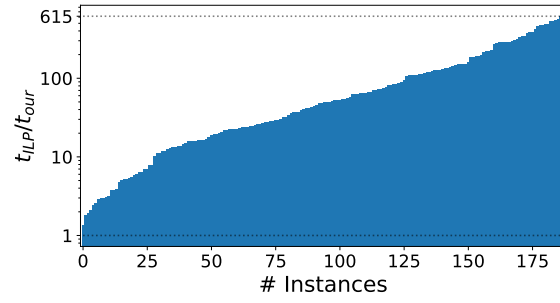


Figure 10: Speedup of Kernel+ILP to ILP

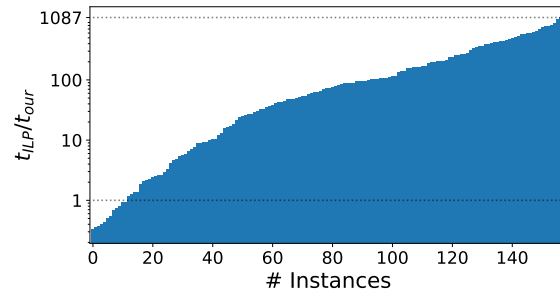


Figure 11: Speedup of avg. branch-and-reduce to ILP