Faster Parallel Multiterminal Cuts *

Monika Henzinger† Alexander Noe ‡ Christian Schulz §

Abstract

We give an improved branch-and-bound solver for the multiterminal cut problem, based on the recent work of Henzinger et al. [12]. We contribute new, highly effective data reduction rules to transform the graph into a smaller equivalent instance. In addition, we present a local search algorithm that can significantly improve a given solution to the multiterminal cut problem. Our exact algorithm is able to give exact solutions to more and harder problems compared to the state-of-the-art algorithm by Henzinger et al. [12]; and give better solutions for more than two third of the problems that are too large to be solved to optimality. Additionally, we give an inexact heuristic algorithm that computes high-quality solutions for very hard instances in reasonable time.

1 Introduction

The multiterminal cut problem is a fundamental combinatorial optimization problem which was first formulated by Dahlhaus et al. [8] and Cunningham [7]. Given an undirected edge-weighted graph $G = (V, E, w)$ with edge weights $w : E \mapsto \mathbb{N}_{>0}$ and a set T, $|T| = k$, of terminals, the *multiterminal cut* problem 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. There are many applications of the problem: for example multiprocessor scheduling [22], clustering [19] and bioinformatics [13, 16, 25].

The problem is known to be NP-hard for $k \geq 3$ [8]. For $k = 2$ the problem reduces to the well known minimum s-t-cut problem, which is in P. 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. [8] give a $2(1 - 1/k)$ approximation algorithm with polynomial running time based on the notion of *minimum isolating cuts*, i.e. the minimum cut separating a terminal from all other terminals. The currently best known approximation algorithm due to Buchbinder et al. [4] uses a linear program relaxation to achieve an approximation ratio of 1.323. Recently, Henzinger et al. [12] introduced a branch-and-reduce framework for the problem that is multiple orders of magnitudes faster than classic ILP formulations for the problem. This allows researchers to solve instances to optimality that are significantly larger than was previously possible and hence enables the use of multiterminal cut algorithms in practical applications.

Contribution. We give an improved solver for the multiterminal cut problem, based on the recent work of Henzinger et al. [12]. We contribute new, highly effective reductions to transform the graph into a smaller equivalent instance. In addition, we present a local search algorithm that can significantly improve a given solution to the multiterminal cut problem. Additionally, we combine the branch-and-reduce solver with an integer linear program solver to more efficiently solve subproblems emerging over the course of the algorithm. With our newly introduced reductions, the state-of-the-art algorithm by Henzinger et al. [12] is able to solve significantly harder instances to optimality and give better solutions to instances that are too large to solve to optimality. Additionally, we give an inexact algorithm that gives high-quality solutions to hard problems in reasonable time.

2 Preliminaries

2.1 Basic Concepts Let $G = (V, E, w)$ be a weighted undirected graph with vertex set V, edge set $E \subseteq V \times V$ and positive edge weights $w : E \to \mathbb{N}_{>0}$. We extend w to a set of edges $E' \subseteq E$ by summing the weights of the edges; that is, $w(E') := \sum_{e=(u,v)\in E'} w(u, v)$ and sets of nodes where $w(V_1, V_2)$ is the sum of edge weights connecting sets V_1 and V_2. Let $n = |V|$ be the number of vertices and $m = |E|$

*Research supported by the Austrian Science Fund (FWF) and netIDEE SCIENCE project P 33775-N
Partially supported by DFG grant SCHU 2567/1-2.
We further thank the Vienna Scientific Cluster (VSC) for providing high performance computing resources.

†University of Vienna, Faculty of Computer Science, Vienna, Austria, monika.henzinger@univie.ac.at
‡University of Vienna, Faculty of Computer Science, Vienna, Austria, alexander.noe@univie.ac.at
§Heidelberg University, Heidelberg, Germany, christian.schulz@informatik.uni-heidelberg.de

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. $V_1 \cup \cdots \cup V_k = V$. The weight of a k-cut is defined as the weight sum of all edges crossing block boundaries, i.e. $w(E \cap \bigcup_{i<j} V_i \times V_j)$.

2.2 Multiterminal Cuts A *multiterminal cut* for k terminals $T = \{t_1, ..., t_k\}$ is a multicut with $t_1 \in V_1, ..., t_k \in V_k$. Thus, a multiterminal cut pairwisely 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 $\mathcal{V} = \{\mathcal{V}_1, \ldots, \mathcal{V}_k\}$. For a vertex $v \in V$, $\mathcal{V}_v$ denotes the block affiliation of v in the optimal partitioning $\mathcal{V}$. $\mathcal{C}$ 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. Note that the optimal partitioning $\mathcal{V}$ and the corresponding cut $\mathcal{C}$ are not necessarily unique, our aim is to find *one* minimum multiterminal cut, even if multiple cuts of equal value exist.

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 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 $\lambda(G, s, T)$. This cut is also called the *minimum isolating cut* [8] for vertex s and vertex set T and the minimum isolating cut where the source side is the largest is called the *largest minimum isolating cut* for s and T.

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. An *articulation point* is a vertex whose removal disconnects the graph G into multiple disconnected components. For a given multiterminal cut S, the graph $G \backslash S$ splits G into k connected components, as defined by the cut edges in S, each containing exactly one terminal.

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 with respect to some parameters σ so that there is an algorithm with runtime $f(\sigma) \cdot n^{\mathcal{O}(1)}$ and f is a computable function. Marx [15] proved that the multiterminal cut problem is FPT and Chen et al. [6] gave the first FPT algorithm with a running time of $4^{\mathcal{W}(G)} \cdot n^{\mathcal{O}(1)}$, later improved by Xiao [26] to $2^{\mathcal{W}(G)} \cdot n^{\mathcal{O}(1)}$ and by Cao et al. [5] to $1.84^{\mathcal{W}(G)} \cdot n^{\mathcal{O}(1)}$.

2.3 `VieCut-MTC` We present an improved solver for the multiterminal cut problem. Our work is based on a recent result by Henzinger et al. [12], in the following named `VieCut-MTC`. In this section we give a short summary of their results, for further details we refer the reader to their original work [12]. The `VieCut-MTC` multiterminal cut solver is a shared-memory parallel solver for the multiterminal cut problem. `VieCut-MTC` is a branch-and-reduce algorithm that performs a set of local contraction routines to transform the graph G into an instance of smaller size H, where the minimum multiterminal cut $\mathcal{W}(G) = \mathcal{W}(H)$, i.e. the minimum multiterminal cut $\mathcal{C}(G)$ can still be found on the smaller instance H. For this purpose, they use the following lemmas:

LEMMA 2.1. *[5][12] If an edge $e = (u,v) \in G$ is guaranteed not to be in at least one multiterminal cut $\mathcal{C}(G)$ (i.e. $\mathcal{V}_u = \mathcal{V}_v$), we can contract e and $\mathcal{W}(G/e) = \mathcal{W}(G)$.*

LEMMA 2.2. *[5][12] 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 $\mathcal{V}_u \neq \mathcal{V}_v$, we can delete e from G and $\mathcal{C}(G-e)$ is still a valid minimum multiterminal cut.*

Lemma 2.1 allows the contraction of edges that are guaranteed not to be in at least on multiterminal cut and Lemma 2.2 allows the deletion of edges that are guaranteed to be in a multiterminal cut. An example for such an edge is an edge that connects two terminal vertices.

Largest Minimum Isolating Cut Dahlhaus et al. [8] show that there exists a minimum multiterminal cut $\mathcal{C}(G)$ for a graph G such

that for every terminal $t \in T$ all vertices on the source side of the largest minimum isolating cut are in block t. Thus, according to Lemma 2.1, the source sides can be contracted into their respective terminals. The cut value of this problem is equal to the sum of all isolating cuts minus the heaviest, as any set of $t-1$ isolating cuts pairwisely separates all terminals form each other. A lower bound for the optimal solution is the sum of all isolating cuts divided by two [8, 12].

Reductions A variety of reductions in the work of Henzinger et al. [12] use Lemma 2.1 to contract edges and effectively reduce the size of the input graph. For every *low degree vertex* v with $|N(v)| \leq 2$, one can contract the heaviest edge incident to v as there is at least one multiterminal cut that does not contain it. Every *heavy edge* $e = (v, u)$ with $w(e) \cdot 2 \geq w(E[v])$ can also be contracted. This condition can be relaxed to *heavy triangles*, where an edge $e = (v_1, v_2)$ that is part of a triangle (v_1, v_2, v_3) can be contracted if $w(e) + w(v_1, v_3) \cdot 2 \geq w(E[v_1])$ and $w(e) + w(v_2, v_3) \cdot 2 \geq w(E[v_2])$. A more global reduction uses the CAPFOREST algorithm of Nagamochi et al. [17, 18] to find a connectivity lower bound of every edge in G in almost linear time. If an edge $e = (u, v)$ has *high connectivity*, i.e. there is no small cut that separates u and v, and no multiterminal cut that separates its incident vertices can be better than $\widehat{\mathcal{W}}(G)$, the edge can be contracted according to Lemma 2.1. For full descriptions and proofs of these reductions we refer the reader to Section 4 of [12].

Branching When it is not possible to find any more edges to contract or delete, `VieCut-MTC` selects an edge e incident to a terminal and creates two subproblems: G/e represents the problem in which e is not part of the multiterminal cut $\mathcal{C}(G)$ and $G - e$ represents the problem in which it is. Both subproblems are added to a shared-memory parallel problem queue $\mathcal{Q}$ and solved independently from each other.

3 Improved Reductions and Branching

We now introduce a set of new reductions to further decrease the problem size. Additionally, we give an alternative branching rule that allows for faster branching.

3.1 New Reductions

`VieCut-MTC` contracts edges incident to low degree vertices, edges with high weight and edges whose incident vertices have a high connectivity. Additionally, `VieCut-MTC` contracts the largest minimum isolating cut for each terminal to the remainder of the terminal set. We now introduce additional reductions that are able to further shrink the graph and thus speed up the algorithm.

3.1.1 Articulation Points

Let $\phi \in V$ be an articulation point in G whose removal disconnects the graph into multiple connected components. For any of these components that does not contain any terminals, we show that all vertices in the component can be contracted into ϕ.

LEMMA 3.1. *For an articulation point ϕ whose removal disconnects the graph G into multiple connected components $(G_1, \ldots, G_p)$ and a component G_i with $i \in \{1, \ldots, p\}$ that does not contain any terminals, no edge in G_i or connecting G_i with ϕ can be part of $\mathcal{C}(G)$.*

Proof. Let e be an edge that connects two vertices in $\{V_i \cup \phi\}$. Assume $e \in \mathcal{C}(G)$, i.e. e is part of the minimum multiterminal cut of G. This means that vertices in $\{V_i \cup \phi\}$ are not all in the same block. By changing the block affiliation of all vertices in $\{V_i \cup \phi\}$ to $\mathcal{V}_\phi$ we can remove all edges connecting vertices in $\{V_i \cup \phi\}$ from the multiterminal cut, thus decrease the weight of the multiterminal cut by at least $w(e)$. As ϕ is an articulation point, G_i is only connected to the rest of G through ϕ and thus no new edges are introduced to the multiterminal cut. This is a contradiction to the minimality of $\mathcal{C}(G)$, thus no edge e that connects two vertices in $\{V_i \cup \phi\}$ is in the minimum multiterminal cut $\mathcal{C}(G)$. □

Using Lemmas 2.1 and 3.1 we can contract all components that contain no terminals into the articulation point ϕ. All articulation points of a graph can be found in linear time using an algorithm by Tarjan and Vishkin [23] based on depth-first search. The algorithm performs a depth-first search and checks in the backtracking step whether for a vertex v there exists an alternative path from the parent of v to every of descendant of v. If there is no alternative path, v is an articulation point in G.

3.1.2 Equal Neighborhoods

In many cases, the resulting graph of the reductions contains groups of vertices that are connected to the same neighbors. If the neighborhood and respective edge weights of two vertices are equal, we can use Lemmas 2.1 and 3.2 to contract them into a single vertex.

LEMMA 3.2. *For two non-terminal vertices v_1 and v_2 with $\{N(v_1) \backslash v_2\} = \{N(v_2) \backslash v_1\}$ where for all $v \in \{N(v_1) \backslash v_2\}$, $w(v_1, v) = w(v_2, v)$, there is at least one minimum multiterminal cut where $\mathcal{V}_{v_1} = \mathcal{V}_{v_2}$.*

Proof. Let C be a partitioning of the vertices in G with $C(v_1) \neq C(v_2)$, let ζ be the corresponding cut, where $e = (u, v) \in \zeta$, if $C(u) \neq C(v)$ and let $cw(v)$ be the

total weight of edges in ζ incident to a vertex $v \in V$. W.l.o.g. let v_2 be the vertex with $cw(v_2) \geq cw(v_1)$. We analyze this in two steps: We assume that when moving v_2 to $C(v_1)$ that all edges incident to v_2 in its old location are removed from ζ, which drops the weight of ζ by $cw(v_2)$ and then all edges incident to v_2 in its new location are added to ζ, which is exactly $cw(v_1)$ by the conditions of the lemma. Thus the weight of ζ changes by $cw(v_1) - cw(v_2) \leq 0$. If the edge $e_{12} = (v_1, v_2)$ exists, both $cw(v_1)$ and $cw(v_2)$ are furthermore decreased by $w(e_{12})$, as the edge connecting them is not a cut edge anymore. As we only moved the block affiliation of v_2, the only edges newly introduced to ζ are edges incident to v_2. Thus, the total weight of the multiterminal cut was not increased by moving v_1 and v_2 into the same block and we showed that for each cut ζ, in which $C(v_1) \neq C(v_2)$ there exists a cut of equal or better value in which v_1 and v_2 are in the same block. Thus, there exists at least one multiterminal cut where $\mathcal{V}_{v_1} = \mathcal{V}_{v_2}$. ∎

We detect equal neighborhoods for all vertices with neighborhood size smaller or equal to a constant c_N using two linear time routines. To detect neighboring vertices v_1 and v_2 with equal neighborhood, we sort the neighborhood vertex IDs including edge weights by vertex IDs (excluding the respective other vertex) for both v_1 and v_2 and check for equality. To detect non-neighboring vertices v_1 and v_2 with equal neighborhood, we create a hash of the neighborhood sorted by vertex ID for each vertex with neighborhood size smaller or equal to c_N. If hashes are equal, we check whether the condition for contraction is actually fulfilled. As the neighborhoods to sort only have constant size, they can be sorted in constant time and thus the procedures can be performed in linear time. We perform both tests, as the neighborhoods of neighboring vertices contain each other and therefore do not result in the same hash value; and non-neighboring vertices are not in each others neighborhood and therefore finding them requires checking the neighborhood of every neighbor, which results in a large search space. We set $c_N = 5$, as in most cases where we encountered equal neighborhoods they are in vertices with neighborhood size ≤ 5.

3.1.3 Maximum Flow from Non-terminal Vertices Let v be an arbitrary vertex in $V \backslash T$, i.e. a non-terminal vertex of G. Let $(V_v, V \backslash V_v)$ be the largest minimum isolating cut of v and the set of terminal vertices T. Lemma 3.3 shows that there is at least one minimum multiterminal cut $\mathcal{C}(G)$ so that $\forall x \in V_v : \mathcal{V}_x = \mathcal{V}_v$ and thus V_v can be contracted into a single vertex.

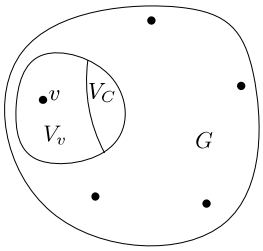


Figure 1: Illustration of vertex sets in Lemma 3.3

LEMMA 3.3. *Let v be a vertex in $V \backslash T$. Let $(V_v, V \backslash V_v)$ be the largest minimum isolating cut of v and the set of terminal vertices T and let $\lambda(G, v, T)$ be the weight of the minimum isolating cut $(V_v, V \backslash V_v)$. There exists at least one minimum multiterminal cut $\mathcal{C}(G)$ in which $\forall x \in V_v : \mathcal{V}_x = \mathcal{V}_v$.*

Proof. As $(V_v, V \backslash V_v)$ is a minimum isolating cut with the terminal set as sinks, we know that no terminal vertex is in V_v. Assume that $\mathcal{C}(G)$ cuts V_v, i.e. there is a non empty vertex set $V_C \in V_v$ so that $\forall x \in V_C : \mathcal{V}_x \notin \mathcal{V}_v$. We will show that the existence of such a vertex set contradicts the minimality of $\mathcal{C}(G)$. Figure 1 gives an illustration of the vertex sets defined here.

Due to the minimality of the minimum isolating cut, we know that $w(V_C, V_v \backslash V_C) \geq w(V_C, V \backslash V_v)$ (i.e. the connection of V_C to the rest of V_v is at least as strong as the connection of V_C to $(V \backslash V_v)$), as otherwise we could remove V_C from V_v and find an isolating cut of smaller size.

We now show that by changing the block affiliation of all vertices in V_C to $\mathcal{V}_v$, i.e. removing all vertices from the set V_C, we can construct a multiterminal cut of equal or better cut value. By changing the block affiliation of all vertices in V_C to $\mathcal{V}_v$, we remove all edges connecting V_C to $(V_v \backslash V_C)$ from $\mathcal{C}(G)$ and potentially more, if there were edges in $\mathcal{C}(G)$ that connect two vertices both in V_C. At most, the edges connecting V_C and $(V \backslash V_v)$ are newly added to $\mathcal{C}(G)$. As $w(V_C, V_v \backslash V_C) \geq w(V_C, V \backslash V_v)$, the cut value of $\mathcal{C}(G)$ will be equal or better than previously. Thus, there is at least one multiterminal cut in which V_C is empty and therefore $\forall x \in V_v : \mathcal{V}_x = \mathcal{V}_v$. ∎

We can therefore run a maximum s-T-flow from a non-terminal vertex to the set of all terminals T and contract the source side of the largest minimum isolating cut into a single vertex. These flow problems can be solved embarrassingly parallel, in which every processor solves an independent maximum s-T-flow problem for a different non-terminal vertex v.

While it is possible to run a flow problem from every vertex in V, this is obviously not feasible as it would entail excessive running time overheads. Promising vertices to use for maximum flow computations are

either high degree vertices or vertices with a high distance from every terminal. High degree vertices are promising, as due to their high degree it is more likely that we can find a minimum isolating cut of size less than their degree. Vertices that have a high distance to all terminals are on 'the edge of the graph', potentially in a subgraph only weakly connected to the rest of the graph. Running a maximum flow then allows us to contract this subgraph. In every iteration, we run 5 flow problems starting from high-distance vertices and 5 flow problems starting from high-degree vertices.

3.2 Vertex Branching When the `VieCut-MTC`-algorithm is initialized, it only has a single problem containing the whole graph G. While independent minimum isolating cuts are computed in parallel, most of the shared-memory parallelism in `VieCut-MTC` comes from the embarrassingly parallel solving of different problems on separate threads. When branching, `VieCut-MTC` selects the highest degree vertex that is adjacent to a terminal and branches on the heaviest edge connecting it to one of the terminals. The algorithm thus creates only up to two subproblems and is still not able to use the whole machine.

We propose a new branching rule that overcomes these limitations by selecting the highest degree vertex incident to at least one terminal and use it to create multiple subproblems to allow for faster startup. Let x be the vertex used for branching, $\{t_1, \dots, t_i\}$ for some $i \geq 1$ be the adjacent terminals of x and w_M be the weight of the heaviest edge connecting x to a terminal. We now create up to $i+1$ subproblems as follows:

For each terminal t_j with $j \in \{1, \dots, i\}$ with $w(x, t_j) + w(x, V \backslash T) > w_M$ create a new problem P_j where edge (x, t_j) is contracted and all other edges connecting x to terminals are deleted. Thus in problem P_j, vertex x belongs to block $\mathcal{V}_{t_j}$. If $w(x, t_j) + w(x, V \backslash T) \leq w_M$, i.e. the weight sum of the edges connecting x with t_j and all non-terminal vertices is not heavier than w_M, the assignment to block $\mathcal{V}_{t_j}$ cannot be optimal and thus we do not need to create the problem P_j, also called *pruning* of the problem. The following Lemma 3.4 proves the correctness of this pruning step.

LEMMA 3.4. *Let $G = (V, E)$ be a graph, $T \subseteq V$ be the set of terminal vertices in G, and $x \in V$ be a vertex that is adjacent to at least one terminal and for an $i \in \{1, \dots, |T|\}$ be the index of the terminal for which $e_i = (x, t_i)$ is the heaviest edge connecting x with any terminal. Let w_M be the weight of e_i. If there exists a terminal t_j adjacent to x with $j \in \{1, \dots, |T|\}$ with $w(x, t_j) + w(x, V \backslash T) \geq w_M$, there is at least one minimum multiterminal cut $\mathcal{C}(G)$ so that $\mathcal{V}_x \neq j$, i.e. x is not in block j.*

Proof. If $\mathcal{V}_x = i$, i.e. x is in the block of the terminal it has the heaviest edge to, the sum of cut edge weights incident to x is $\leq E(x) - w_M$, as edge e_i of weight w_M is not a cut edge in that case. If $\mathcal{V}_x = j$, i.e. x is in the block of terminal j, the sum of cut edge weights incident to x is $\geq E(x) - (w(x, V \backslash T) + w(x, t_j))$, as all edges connecting x with other terminals than t_j are guaranteed to be cut edges. As $w(x, t_j) + w(x, V \backslash T) \geq w_M$, even if all non-terminal neighbors of x are in block j, the weight sum of incident cut edges is not lower than when x is placed in block i. As the block affiliation of x can only affect its incident edges, the cut value of every solution that sets $\mathcal{V}_x = j$ would be improved or remain the same by setting $\mathcal{V}_x = i$. ∎

If $w(x, V \backslash T) > w_M$ and $i < |T|$, we also create problem P_{i+1}, in which all edges connecting x to a terminal are deleted. This problem represents the assignment of x to a terminal that is not adjacent to it. We add each subproblem whose lower bound is lower than the currently best found solution $\widehat{\mathcal{W}}$ to the problem queue $\mathcal{Q}$. As we create up to $|T|$ subproblems, this allows for significantly faster startup of the algorithm and allows us to use the whole parallel machine after less time than before.

3.3 Integer Linear Programming Integer Linear Programming can be used as an alternative to branch-and-reduce [12] and for some problems this is faster than branch-and-reduce. We integrate the ILP formulation from the work of [12] and include it directly into `VieCut-MTC` as an alternative to branching. We give the ILP solver a time limit and if it is unable to find an optimal solution within the time limit, we instead perform a branch operation. In Section 5.2 we study which subproblems to solve with an ILP first.

3.4 Improving Bounds with Greedy Optimization The `VieCut-MTC` algorithm prunes problems which cannot result in a solution which is better than the best solution found so far. Therefore, even though it is a deterministic algorithm that will output the optimal result when it terminates, performing greedy optimization on intermediate solutions allows for more aggressive pruning of problems that cannot be optimal. Additionally, `VieCut-MTC` has reductions that depend on the value of $\widehat{\mathcal{W}}(G)$ and can thus contract more vertices if the cut value $\widehat{\mathcal{W}}(G)$ is lower.

For a subproblem $H = (V_H, E_H)$ with solution ρ, the original graph $G = (V_G, E_G)$ and a mapping $\pi : V_G \to V_H$ that maps each vertex in V_G to the vertex in V_H that encompasses it, we can transfer the solution ρ to a solution γ of G by setting the block affiliation of

every vertex $v \in V_G$ to $\gamma(v) := \pi(\rho(v))$. The cut value of the solution $w(\gamma)$ is defined as the sum of weights of the edges crossing block boundaries, i.e. the sum of edge weights where the incident vertices are in different blocks. Let $\xi_i(V_G)$ be the set of all vertices $v \in V_G$ where $\gamma(v) = i$.

We introduce the following greedy optimization operators that can transform γ into a better multiterminal cut solution γ_{IMP} with $w(\gamma_{\text{IMP}}) < w(\gamma)$.

3.4.1 Kernighan-Lin Local Search Kernighan and Lin [14] give a heuristic for the traveling-salesman problem that has been adapted to many hard optimization problems [20, 24, 27, 11], where each vertex $v \in V_G$ is assigned a gain $g(v) = \max_{i \in \{i,\dots,|T|\}, i \neq \gamma(v)} \sum w(v, \xi_i(V_G)) - w(v, \xi_{\gamma(v)}(V_G))$, i.e. the improvement in cut value to be gained by moving v to another block, the best connected other block. We perform runs where we compute the gain of every vertex that has at least another neighbor in a different block and move all vertices with non-negative gain. Additionally, if a vertex v has a negative gain, we store its gain and associated best connected other block. For any neighbor u of v that also has the same best connected other block, we check whether $g(w)+g(v)+2 \cdot w(v,u) > 0$, i.e. moving both u and v at the same time is a positive gain move. If it is, we perform the move.

3.4.2 Pairwise Maximum Flow For any pair of blocks $1 \leq i < j \leq |T|$ where $w(\xi_i(V_G), \xi_j(V_G)) > 0$, i.e. there is at least one edge from block i to block j, we can create a maximum s-t flow problem between them: we create a graph F_{ij} that contains all vertices in $\xi_i(V_G)$ and $\xi_j(V_G)$ and all edges that connect these vertices.

Let H be the current problem graph created by performing reductions and branching on the original graph G. All vertices that are encompassed in the same vertex in problem graph H as the terminals i and j are hereby contracted into the corresponding terminal vertex. We perform a maximum s-t-flow between the two terminal vertices and re-assign vertex assignments in γ according to the minimum s-t-cut between them. As we only model blocks $\xi_i(V_G)$ and $\xi_j(V_G)$, this does not affect other blocks in γ. In the first run we perform a pairwise maximum flow between every pair of blocks i and j where $w(\xi_i(V_G), \xi_j(V_G)) > 0$ in random order. We continue on all pairs of blocks where $w(\xi_i(V_G), \xi_j(V_G))$ was changed since the end of the previous maximum flow iteration between them.

We first perform Kernigham-Lin local search until there is no more improvement, then pairwise maximum flow until there is no more improvement, followed by another run of Kernigham-Lin local search. As pairwise maximum flow has significantly higher running time, we spawn a new thread to perform the optimization if there is a CPU core that is not currently utilized.

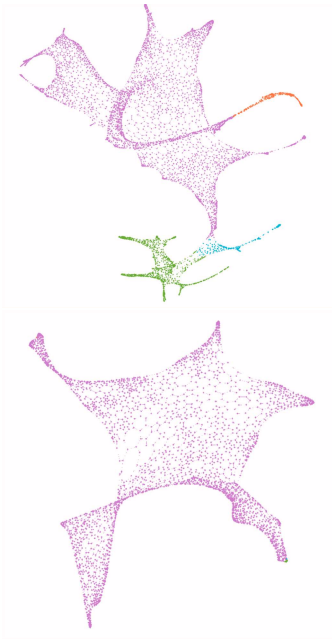


Figure 2: Minimum multiterminal cut for graph `uk` [21] and four terminals - on original graph (top) and remaining graph at time of first branch operation (bottom), visualized using Gephi-0.9.2 [2]

4 Fast Inexact Solving

`VieCut-MTC` in an exact algorithm, i.e. when it terminates the output is guaranteed to be optimal. As the multiterminal cut problem is NP-complete [8], it is not feasible to expect termination in difficult instances of the problem. Henzinger et al. [12] report that their algorithm often does not terminate with an optimal result but runs out of time or memory and returns the best result found up to that point. Thus, it makes sense to relax the optimality constraint and aim to find a high-quality (but not guaranteed to be optimal) solution faster.

A key observation herefor is that in many problems, most, if not all vertices that are not already contracted into a terminal at the time of the first branch, will be assigned to a few terminals whose weighted degree at that point is highest. See Figure 2 for an example with 4 terminals (selected with high distance to each other) on graph `uk` from the Walshaw Graph Partitioning Archive [21]. As we can see, at the time of the first branch (right figure), most vertices that are not assigned to the pink terminal in the optimal solution are already contracted into their respective terminals.

The remainder is mostly assigned to a single terminal. As we can observe similar behavior in many problems, we propose the following heuristic speedup operations:

Let $\delta \in (0, 1)$ be a contraction factor and T_H be the set of all terminals that are not yet isolated in graph H. In each branching operation on an intermediate graph H, we delete all edges around the $\lceil \delta \cdot |T_H| \rceil$ terminals with lowest degree. Additionally, we contract all vertices adjacent to the highest degree terminal that are not adjacent to any other terminal into the highest degree terminal. This still allows us to find all solutions in which no more vertices were added to the lowest degree terminals and the adjacent vertices are in the same block as the highest degree terminals.

Additionally, in a branch operation on vertex v, we set a maximum branching factor β and only create problems where v is contracted into the β adjacent terminals it has the heaviest edges to and one problem in which it is not contracted into either adjacent terminal. This is based on the fact that all other edges connecting v to other terminals will be part of the multiterminal cut and the greedy assumption that it is likely that the optimal solution does not contain at least one of these heavy edges. By default, we set $\delta = 0.1$ and $\beta = 5$.

5 Experiments and Results

We now perform an experimental evaluation of the proposed work. This is done in the following order: first we analyze the impact of different reductions introduced in the work of Henzinger et al. [12] and in this work. We then analyze which subproblems to solve using integer linear programming and then compare the results of `VieCut-MTC` with our exact and inexact algorithms on a variety of graphs from different sources. Here, `VieCut-MTC` denotes the algorithm of Henzinger et al. [12], `Exact-MTC` denotes the exact version of our algorithm and `Inexact-MTC` denotes the heuristic algorithm proposed in Section 4

We implemented the algorithms using C++-17 and compiled all code using g++ version 7.3.0 with full optimization (`-O3`). Our experiments are conducted on two machine types. Machine A is a machine with two Intel Xeon E5-2643v4 with 3.4 GHz with 6 CPU cores each and 1.5 TB RAM in total. Machine B is a machine in the Vienna Scientific Cluster with two Intel Xeon E5-2650v2 with 2.6GHz with 8 CPU cores each and 64 GB RAM in total. We limit the maximum amount of memory used for each problem to 32 GB. ILP problems are solved using Gurobi 8.1.0. When we report a mean result we give the geometric mean as problems differ significantly in cut size and time. Our code is freely available under the permissive MIT license[1].

To evaluate the performance of different multiterminal cut algorithms, we use a wide variety of graphs from different sources. We re-use a large subset of the map, social and web graphs graphs used by Henzinger et al. [12]. Additionally, we add numerical graphs from the Walshaw Graph Partitioning Benchmark [21] and a set of graphs from the 10^{th} DIMACS implementation challenge [1] and the SuiteSparse Matrix Collection (formerly UFSparse Matrix Collection) [9]. Table 1 gives an overview over the graphs used in this work.

Table 1: Large Real-world Benchmark Instances

Graph	Source	n	m
Map Graphs			
ak2010	[1]	45 292	$109K$
ca2010	[1]	$710K$	$1.74M$
ct2010	[1]	67 578	$168K$
de2010	[1]	24 115	58 028
hi2010	[1]	25 016	62 063
luxembourg.osm	[9]	$115K$	$120K$
me2010	[1]	69 518	$168K$
netherlands.osm	[9]	$2.22M$	$2.44M$
nh2010	[1]	48 837	$117K$
nv2010	[1]	84 538	$208K$
ny2010	[1]	$350K$	$855K$
ri2010	[1]	25 181	62 875
sd2010	[1]	88 360	$205K$
vt2010	[1]	32 580	77 799
Social, Web and Numerical Graphs			
598a	[21]	$111K$	$742K$
astro-ph	[9]	16 706	$121K$
bcsstk30	[21]	28 924	$1.01M$
ca-CondMat	[9]	23 133	93 439
caidaRouterLevel	[9]	$192K$	$609K$
citationCiteseer	[9]	$268K$	$1.16K$
cit-HepPh	[9]	34 546	$422K$
cnr-2000	[9]	$326K$	$2.74M$
coAuthorsCiteseer	[9]	$227K$	$814K$
cond-mat-2005	[9]	40 421	$176K$
coPapersCiteseer	[9]	$434K$	$16.0M$
cs4	[21]	22 499	43 858
eu-2005	[3]	$862K$	$16.1M$
fe_body	[21]	45 087	$164K$
higgs-twitter	[9]	$457K$	$14.9M$
in-2004	[3]	$1.38M$	$13.6M$
NACA0015	[9]	$1.04M$	$3.11M$
uk-2002	[3]	$18.5M$	$261M$
venturiLevel3	[9]	$4.03M$	$8.05M$
vibrobox	[21]	12 328	$165K$

[1] `https://github.com/VieCut/VieCut`

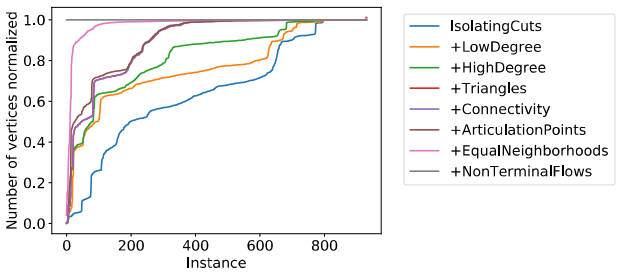


Figure 3: Number of vertices in graph after reductions are finished, normalization by (# vtx with all reductions / # vtx in variant), sorted by normalized value.

As the instances generally do not have any terminals, we find random vertices that have a high distance from each other in the following way: we start with a random vertex r, run a breadth-first search starting at r and select the vertex v encountered last as first terminal. While the number of terminals is smaller than desired, we add another terminal by running a breadth-first search from all current terminals and adding the vertex encountered last to the list of terminals. We then run a bounded-size breadth-first search around each terminal to create instances where the minimum multiterminal cut does not have $k-1$ blocks consisting of just a single vertex each. This results in problems in which well separated clusters of vertices are partitioned and the task consists of finding a partitioning of the remaining vertices in the boundary regions between already partitioned blocks. This relates to clustering tasks, in which well separated clusters are labelled and the task consists of labelling the remaining vertices in-between. Additionally, we use a subset of the generated instances of Henzinger et al. [12] to compare our work to `VieCut-MTC`. These graphs have unit-weight edges, however contracted subproblems often have weighted edges.

5.1 Reductions We first analyze the impact of the different reductions on the size of the graph at the time of first branch. For this, we run experiments on all graphs in Table 1 with $k = \{4, 8, 10\}$ terminals and 10% of all vertices added to the terminals on machine B. On these instances, we run subsets of all contractions exhaustively and check which factor of vertices remain in the graph. A value of 1 thus indicates that the reductions were unable to find any edges to contract, a value close to 0 shows that almost no vertices remain and the resulting problem is far smaller than the original problem. Figure 3 gives the result with 8 different variants, starting with a version that only runs isolating cuts and adding one reduction family per version. For this, we sorted the reductions by their impact on the total running time. In Figure 3, we can see that using all reductions allows us to reduce the number of vertices by more than a half in about half of all instances and can find a sizable number of reductions on almost any instance.

We can see that running the local reductions in `VieCut-MTC` are very effective on almost all instances. In average, `IsolatingCuts` reduce the number of vertices by 33%, `LowDegree` reduces the number of vertices in the remaining graph by 17%, `HighDegree` by 7% and `Triangles` by 8%. In contrast, `Connectivity` only has a negligible effect, as it contracts edges whose connectivity is larger than a value related to the difference of upper bound to total weight of deleted edges. As there are almost no deleted edges in the beginning, this value is very high and almost no edge has high enough connectivity.

In average, `ArticulationPoints` reduces the number of vertices on the already contracted graphs by 1.9%, `EqualNeighborhoods` reduces the number of vertices by 7.8% - mostly in sparse regions of the graph, and `NonTerminalFlows` reduces the number of vertices by 2.0%. However, there are some instances in which the newly introduced reductions reduce the number of vertices remaining by more than 99%.

5.2 Integer Linear Programming In order to get a wide variety of ILP problems, we run the `Inexact-MTC` algorithm on all instances in Table 1 with $k = 10$ terminals and 10% of vertices added to the terminals on machine B. As `Inexact-MTC` removes low-degree terminals and contracts edges, we have subproblems with very different sizes and numbers of terminals. In this experiment, whenever `Inexact-MTC` chooses between branching and ILP on graph G, we select a random integer $r \in (1{,}200\,000)$. We use a random integer as we want to have problems of varying sizes. We select 200000 as the maximum, as we did not encounter any larger instances that were solved in the allotted time. If $|E| < r$, the problem is solved with ILP, otherwise the algorithm branches on a vertex incident to a terminal. The timeout is set to 60 seconds.

Figure 4 shows the time needed to solve the ILP problems in relation to the number of edges in the graph. We can see that there is a strong correlation between problem size and total running time, but there are still a large number of outliers that cannot be solved in the allotted time even though the instances are rather small. In the following, we set the limit to 50 000 edges and solve all instances with fewer than 50 000 edges with an integer linear program. If the instance has at least 50 000 edges, we branch on a vertex incident to a terminal and create more subproblems.

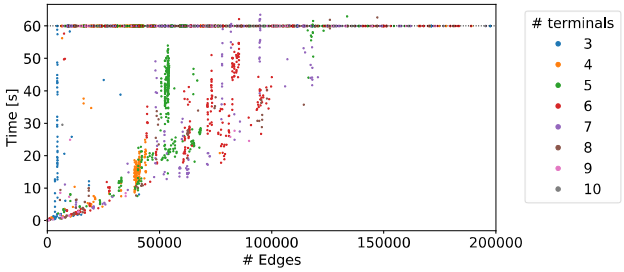


Figure 4: Running time of ILP subproblems in relation to $|E|$.

5.3 Comparison to VieCut-MTC We use the experiment of Section 8.7 in the work of Henzinger et al. [12] to compare Exact-MTC to VieCut-MTC on the instances used in their work. The experiment uses a set of large social and web graphs with pre-defined clusters and $k = \{3, 4, 5, 8\}$ terminals, where $10 - 25\%$ of vertices are marked as terminal vertices initially, a total of 160 instances. We run the experiments on machine A using all 12 cores and set the time limit to 600 seconds.

Out of 160 instances, VieCut-MTC terminates with an optimal result in 32 instances, while Exact-MTC terminates with an optimal result in 46 instances. Out of the 115 instances that were not solved to optimality by both algorithms, Exact-MTC gives a better result on 75 instances and the same result on the other 38 instances. The geometric mean values of results given by Exact-MTC and Inexact-MTC are both about 1.5% lower than VieCut-MTC. Note that in the experiments performed in [12], which uses a larger machine (32 cores) and has a timeout of 3600 seconds, VieCut-MTC has a geometric mean of about 0.1% better than VieCut-MTC in this work. The largest part of the improvement of Exact-MTC and Inexact-MTC over VieCut-MTC is gained by the greedy optimization detailed in Section 3.4.

Figure 5a shows the performance profile [10] of this experiment. We can see that both Exact-MTC and Inexact-MTC are almost always optimal or very close to it. In contrast, VieCut-MTC gives noticeably worse results on about 20% of instances and more than 5% worse results on 10% of instances.

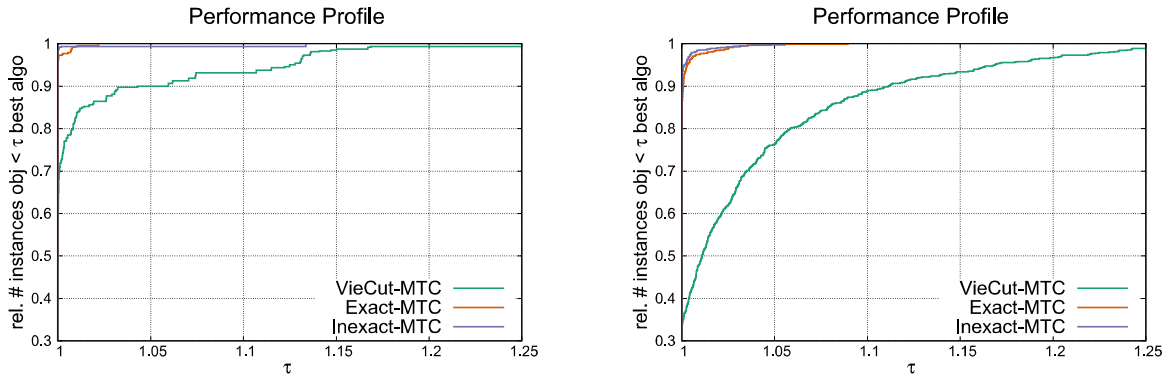
5.4 Large Multiterminal Cut Problems We compare VieCut-MTC, Exact-MTC and Inexact-MTC on all graphs with $k = \{4, 5, 8, 10\}$ terminals and 10% and 20% of vertices added to the terminal. For each combination of graph, number of terminals and factor of vertices in terminal, we create three problems with random seeds $s = \{0, 1, 2\}$. Thus, we have a total of 816 problems. We set the time limit per algorithm and problem to 600 seconds. We run the experiment on machine A using all 12 CPU cores. If the algorithm does not terminate in the allotted time or memory limit, we report the best intermediate result. Note that is a soft limit, in which the algorithm finishes the current operation and exits afterwards if the time or memory limit is reached. As many of these are very large instances, most instances in this section are not solved to optimality.

Table 2: Result overview for large multiterminal cut problems on graphs from Table 1.

# Terminals		VieCut-MTC	Exact-MTC	Inexact-MTC
4	Best Solution	109	**183**	175
	Mean Solution	161 799	**159 402**	159 499
	Better Exact	6	**94**	—
5	Best Solution	81	**173**	158
	Mean Solution	216 191	**210 928**	211 090
	Better Exact	6	**121**	—
8	Best Solution	42	139	**175**
	Mean Solution	346 509	331 112	**330 856**
	Better Exact	2	**162**	—
10	Best Solution	37	129	**173**
	Mean Solution	412 138	392 561	**391 822**
	Better Exact	1	**165**	—

Table 2 gives an overview of the results. For each algorithm, we give the number of times, where it gives the best (or shared best) solution over all algorithms; the geometric mean of the cut value; and for VieCut-MTC and Exact-MTC the number of instances in which they have a better result than the respective other. In all instances, in which VieCut-MTC and Exact-MTC terminate with the optimal result, Inexact-MTC also gives the optimal result. We can see that in the problems with 4 and 5 terminals, Exact-MTC slightly outperforms Inexact-MTC both in number of best results and mean solution value. In the problems with 8 and 10 terminals, Inexact-MTC has slightly better results in average. Thus, disregarding the optimality constraint can allow the algorithm to give better solutions faster especially in hard problems with a large amount of terminals.

However, both new algorithms outperform VieCut-MTC on almost all instances where not all algorithms give the same result. Here, Exact-MTC gives a better result than VieCut-MTC in 66% of all instances, while VieCut-MTC gives the better result in only 2% of all instances. As most problems do not terminate with an optimal result, we are unable to say how far the solutions are from the globally optimal solution. Note that Inexact-MTC gives an optimal result in all instances in which all algorithms terminate. Figure 6 shows the progress of the best solution for the algorithms in a set of problems. For both Exact-MTC and Inexact-MTC we can see large improvements to the cut value when the local search algorithm is finished on the first subproblem. In



(a) Instances of Section 5.3

(b) Instances of Section 5.4

Figure 5: Performance profiles for multiterminal cut algorithms

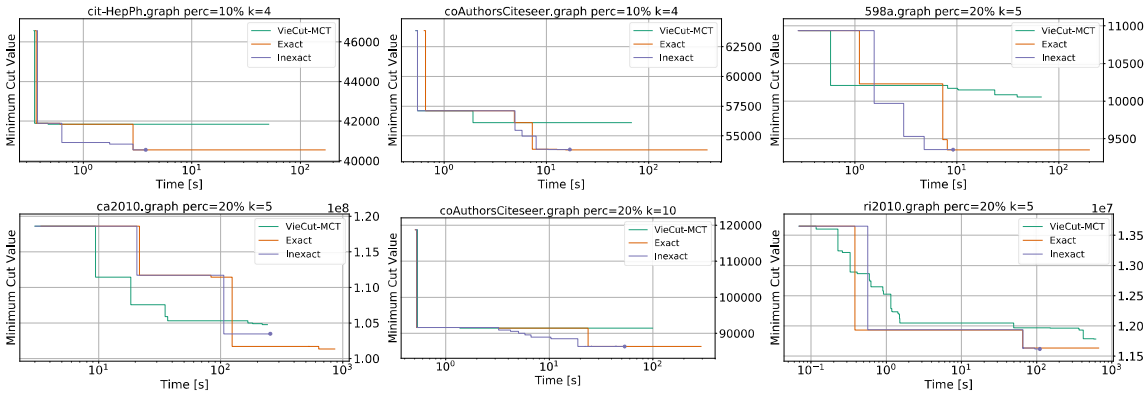


Figure 6: Progression of best result over time. Dot at end marks termination of algorithm.

contrast, `VieCut-MTC` has more small step-by-step improvements and generally gives worse results.

Figure 5b shows the performance profile for the instances in this section. Here we can see that `VieCut-MTC` has significantly worse results on a large subset of the instances, with more than 10% of instances where the result is worse by more than 10%. Also, on a few instances, the results given by `Exact-MTC` and `Inexact-MTC` differ significantly. In general, both of them outperform `VieCut-MTC` on most instances that are not solved to optimality by every algorithm.

6 Conclusion

In this paper, we give a fast parallel solver that gives high-quality solutions for large multiterminal cut problems. We give a set of highly-effective reduction rules that transform an instance into a smaller equivalent one. Additionally, we directly integrate an ILP solver into the algorithm to solve subproblems well suited to be solved using an ILP; and develop a flow-based local search algorithm to improve a given optimal solution. These optimizations significantly increase the number of instances that can be solved to optimality and improve the cut value of multiterminal cuts in instances that can not be solved to optimality. Our algorithm gives better solutions in more than two thirds of these instances, often improving the result by more than 5% on hard instances. Additionally, we give an inexact algorithm for the multiterminal cut problem that aggressively shrinks the graph instances and is able to even outperform the exact algorithm on many of the hardest instances that are too large to be solved to optimality while still giving the exact solution for most easier instances. Important future work consists of improving the scalability of the algorithm by giving a distributed memory version.

References

[1] David A Bader, Henning Meyerhenke, Peter Sanders, Christian Schulz, Andrea Kappes, and Dorothea Wagner. Benchmarking for graph clustering and partitioning. *Encyclopedia of Social Network Analysis and Mining*, pages 73–82, 2014.

[2] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: an open source software for exploring and manipulating networks. In *Third international AAAI conference on weblogs and social media*, 2009.

[3] Paolo Boldi and Sebastiano 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.

[4] Niv Buchbinder, Joseph Seffi Naor, and Roy 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.

[5] Yixin Cao, Jianer 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.

[6] Jianer Chen, Yang Liu, and Songjian Lu. An improved parameterized algorithm for the minimum node multiway cut problem. *Algorithmica*, 55(1):1–13, 2009.

[7] William H Cunningham. The optimal multiterminal cut problem. In *Reliability of computer and communication networks*, pages 105–120, 1989.

[8] Elias Dahlhaus, David S. Johnson, Christos H. Papadimitriou, Paul D. Seymour, and Mihalis Yannakakis. The complexity of multiterminal cuts. *SIAM Journal on Computing*, 23(4):864–894, 1994.

[9] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[10] Elizabeth D Dolan and Jorge J Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213, 2002.

[11] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *IEEE computational intelligence magazine*, 1(4):28–39, 2006.

[12] Monika Henzinger, Alexander Noe, and Christian Schulz. Shared-memory branch-and-reduce for multiterminal cuts. In *2020 Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 42–55. SIAM, 2020.

[13] Ulas Karaoz, TM Murali, Stan Letovsky, Yu Zheng, Chunming Ding, Charles R Cantor, and Simon Kasif. Whole-genome annotation by using evidence integration in functional-linkage networks. *Proceedings of the National Academy of Sciences*, 101(9):2888–2893, 2004.

[14] Shen Lin and Brian W Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498–516, 1973.

[15] Dániel Marx. Parameterized graph separation problems. *Theoretical Computer Science*, 351(3):394–406, 2006.

[16] Elena Nabieva, Kam Jim, Amit Agarwal, Bernard Chazelle, and Mona Singh. Whole-proteome prediction of protein function via graph-theoretic analysis of interaction maps. *Bioinformatics*, 21(suppl_1):i302–i310, 2005.

[17] Hiroshi Nagamochi and Toshihide Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM Journal on Discrete Mathematics*, 5(1):54–66, 1992.

[18] Hiroshi Nagamochi, Tadashi Ono, and Toshihide Ibaraki. Implementing an efficient minimum capacity cut algorithm. *Mathematical Programming*, 67(1):325–341, 1994.

[19] Ulrich Pferschy, Rüdiger Rudolf, and Gerhard J. Woeginger. Some geometric clustering problems. *Nord. J. Comput.*, 1(2):246–263, 1994.

[20] Peter Sanders and Christian Schulz. Think locally, act globally: Highly balanced graph partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA 2013)*, volume 7933 of *LNCS*, pages 164–175. Springer, 2013.

[21] Alan J Soper, Chris Walshaw, and Mark Cross. A combined evolutionary search and multilevel optimisation approach to graph-partitioning. *Journal of Global Optimization*, 29(2):225–241, 2004.

[22] Harold S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Trans. Software Eng.*, 3(1):85–93, 1977.

[23] Robert E Tarjan and Uzi Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14(4):862–874, 1985.

[24] Jesper Larsson Träff. Direct graph k-partitioning with a kernighan–lin like heuristic. *Operations Research Letters*, 34(6):621–629, 2006.

[25] Alexei Vazquez, Alessandro Flammini, Amos Maritan, and Alessandro Vespignani. Global protein function prediction from protein-protein interaction networks. *Nature biotechnology*, 21(6):697, 2003.

[26] Mingyu Xiao. Simple and improved parameterized algorithms for multiterminal cuts. *Theory of Computing Systems*, 46(4):723–736, 2010.

[27] Rui Xu and Donald Wunsch. Survey of clustering algorithms. *IEEE Transactions on neural networks*, 16(3):645–678, 2005.