universität wien

Dissertation / Doctoral Thesis

Titel der Dissertation / Title of the Doctoral Thesis

Algorithm Engineering for Cut Problems

verfasst von / submitted by

Alexander Noe, MSc BSc

angestrebter akademischer Grad / in partial fulfillment of the requirement for the degree of

Doktor der Technischen Wissenschaften (Dr. techn.)

Wien, 2021 / Vienna, 2021

Studienkennzahl lt. Studienblatt: / degree programme code as it appears on the student record sheet:	A 786 880
Dissertationsgebiet lt. Studienblatt: / field of study as it appears on the student record sheet:	Informatik
Betreuerin: / Supervisor:	Univ.-Prof. Dr. Monika Henzinger

Abstract

Graphs are a natural representation of data from various contexts, such as social connections, the web, road networks, and many more. In the last decades, many of these networks have become enormous, requiring efficient algorithms to cut networks into smaller, more readily comprehensible blocks. In this work, we aim to partition the vertices of a graph into multiple blocks while minimizing the number of edges that connect different blocks. There is a multitude of cut or partitioning problems that have been the focus of research for multiple decades. This work develops highly-efficient algorithms for the *(global) minimum cut problem*, the *balanced graph partitioning problem* and the *multiterminal cut problem*. All of these algorithms are efficient in practice and freely available for use[1]. In particular, we obtain the following results and algorithms:

- Fast heuristic and exact shared-memory parallel algorithms for the *(global) minimum cut problem*. We present efficient implementations of existing techniques and combine them with novel approaches to give algorithms that find a minimum cut in huge networks significantly faster than state-of-the-art algorithms. Our heuristic algorithm has a lower empirically observed error rate than existing inexact algorithms for the problem.
- The first engineered algorithm that finds *all* (global) minimum cuts and returns a compact *cactus graph* data structure which represents all of them in graphs with billions of edges in a few minutes. With a multitude of data reduction techniques, we improve the running time of state-of-the-art algorithms by up to multiple orders of magnitude. Based on the representation of all minimum cuts, we are able to find the most balanced minimum cut in time linear to the size of the cactus graph.
- A *fully-dynamic minimum cut* algorithm that efficiently maintains the minimum cut on a graph under edge insertions and deletions. While there is theoretical work, our algorithm is the first implementation of a fully-dynamic algorithm for the problem. Our algorithm uses the theoretical foundation and builds on it with efficient and finely-tuned implementations to give an algorithm that gives up to multiple orders of magnitude speedup to static recomputation.
- An integer linear programming (ILP) based meta-heuristic for the *balanced graph partitioning problem*. As ILPs do not scale to large

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

inputs, we define a much smaller model that allows us to use symmetry breaking and make the approach more scalable. This gives a powerful local search meta-heuristic that can improve given high-quality partitionings even further. We incorporate this meta-heuristic into an existing evolutionary algorithm to give an algorithm that computes state-of-the-art partitionings from scratch.

- A shared-memory parallel exact branch-and-reduce algorithm for the *multiterminal cut problem*. For this algorithm, we develop and engineer highly-efficient data reduction rules to transform a problem into a much smaller equivalent problem. Additionally we give an inexact algorithm that gives high-quality solutions for very hard problems in reasonable time.

Zusammenfassung

Graphen sind eine natürliche Representation von Daten aus zahlreichen Kontexten, zum Beispiel Verbindungen in sozialen Netzwerken, Web-Netzwerken, Straßennetzwerken und vielen weiteren. In den letzten Jahrzehnten sind viele dieser Netzwerke zu enormer Größe gewachsen, was effiziente Algorithmen zu ihrer Partitionierung in kleinere, eher begreifliche Teile erforderlich macht. In dieser Arbeit versuchen wir, die Knoten von Graphen in mehrere Blöcke zu partitionieren, so dass die Anzahl von Kanten, welche Blockgrenzen schneiden, minimiert wird. Es gibt eine Vielzahl von Schnitt- und Partitionierungsproblemen auf Graphen, welche Bereits seit Jahrzehnten erforscht werden. Diese Arbeit entwickelt hocheffiziente Algorithmen für das *(Global) Minimum Cut Problem*, das *Balanced Graph Partitioning Problem* und das *Multiterminal Cut Problem*. Alle hierbei entwickelten Algorithmen sind effizient in der Praxis und frei nutzbar[2]. Im Einzelnen haben wir die folgenden Ergebnisse erzielt und Algorithmen entwickelt:

- Schnelle *heuristische* und *exakte* shared-memory parallele Algorithmen für das *(Global) Minimum Cut Problem*. Wir präsentieren effiziente Implementierungen bestehender Methoden und kombinieren diese mit neuartigen Verfahren, um Algorithmen zu entwickeln, die einen minimalen Schnitt signifikant schneller finden können als der bisherige Stand der Forschung. Die heuristische Variante unseres Algorithmus hat hierbei auch eine deutlich niedrigere empirisch beobachtete Fehlerrate als bestehende inexakte Algorithmen für das Problem.
- Der erste praktisch effiziente Algorithmus, welcher *alle* global minimalen Schnitte eines Graphen findet und eine kompakte *Cactus Graph Datenstruktur* bildet, welche diese Schnitte repräsentiert. Unser Algorithmus findet alle minimalen Schnitte in Graphen mit bis zu mehreren Milliarden Kanten und mehreren Millionen minimalen Schnitten in wenigen Minuten. Mithilfe einer Vielzahl von Datenreduktionstechniken verbessern wir die Laufzeit von bestehenden Algorithmen um bis zu mehreren Größenordnungen. Ausgehend von der Cactus Graph Repräsentation sind wir auch in der Lage, den *Most Balanced Minimum Cut* in Laufzeit linear zur Größe des Kaktusgraphen zu finden.
- Ein *fully-dynamic Minimum Cut* Algorithmus, welcher effizient einen minimalen Schnitt eines Graphen unter Kanteneinfügungen und -löschungen aufrecht erhält. Während es bereits theoretische Forschung zu diesem Problem gibt, ist unser Algorithmus der

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

erste implementierte fully-dynamic Algorithmus für das Problem. Unsere Arbeit nutzt die bestehenden theoretischen Grundlagen und kombiniert sie mit effizienten und fein abgestimmten Implementierungen, um zu einem Algorithmus zu gelangen, welcher um bis zu mehrere Größenordnungen schneller ist als Neuberechnung mit statischen Algorithmen.

- Eine Metaheuristik auf Basis von ganzzahliger linearer Optimierung für das *Balanced Graph Partitioning Problem*. Da ganzzahlige lineare Programme nicht für große Eingaben skalieren, definieren wir ein deutlich kleineres Modell, auf welchem wir das Problem unter Zuhilfenahme von Symmetry Breaking skalierbar machen. Dies resultiert in einer mächtigen Metaheuristik zur lokalen Suche, welche existierende hochqualitative Partitionierungen noch weiter verbessern kann. Wir binden diese Metaheuristik in einen existierenden evolutionären Algorithmus ein und erhalten so einen Algorithmus, der Partitionierung hoher Qualität selbst erzeugen kann.
- Ein shared-memory paralleler exakter Algorithmus für das *Multi-terminal Cut Problem*. Für diesen branch-and-reduce Algorithmus entwickeln wir hocheffiziente Datenreduktionsregeln, um ein Problem in ein viel kleineres äquivalentes Problem umzuwandeln. Außerdem präsentieren wir einen inexakten Algorithmus, welcher hochqualitative Lösungen für extrem schwere Instanzen in annehmbarer Zeit liefert.

Acknowledgments

Thank you to everyone who made these last four years a very enjoyable time - I learned a lot and got to explore very interesting problems!

First and foremost, I would like to thank my advisors Monika Henzinger and Christian Schulz for their support and guidance with my research. You gave me many opportunities to learn new things and approach fascinating problems. Thank you for being so incredibly generous with your time and expertise!

Also, I would like to thank Darren Strash for great collaboration on the papers we wrote together! Thank you also to Ulrich Meyer and Ulrik Brandes who agreed to review this thesis. I could not have wished for a better thesis committee.

I am deeply thankful to Andrew Goldberg for providing me with the incredible opportunity to do an internship at Amazon. On this note, I am very thankful to Quico Spaen, Nhat Le, Larissa Petroianu, Mauricio Resende, Tim Jacobs, and many others for making this internship memorable and enjoyable. I learned so much from all of you!

Being a part of the TAA research group over the past few years has been a great experience! I want to thank Stefan Neumann, Gramoz Goranci, Bernhard Schuster, Alexander Svozil, Marcelo Fonseca Faraj, Wolfgang Ost, and Richard Paul for being great office mates! Thank you to Stefan, Alexander, and Gramoz for helping me find my way in the wonderful city of Vienna; and to Marcelo and Wolfgang for many fruitful discussions. I would also like to thank Sebastian Forster, Kathrin Hanauer, Rudolf Hürner, Sagar Kale, Shahbaz Khan, Ami Paz, Pan Peng, Xiaowei Wu, Vaidehi Srinivas, Ulrike Frolik-Steffan, Iris Gundacker, and Christina Licayan for being wonderful colleagues and making these years very enjoyable!

I am very grateful to my friends and family for their endless support during my years of university and graduate school. I thank my parents Birgitt and Wolfgang for always being there for me.

Finally, I would like to wholeheartedly thank my partner Anique-Marie Cabardos for her love and support. Thank you for keeping me happy and motivated and thank you for your valuable help with proofreading manuscripts and parts of this thesis!

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.

Partially supported by DFG grant SCHU 2567/1-2.

Moreover, we gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time on the GCS Supercomputer SuperMUC at Leibniz Supercomputing Centre (www.lrz.de).

We further thank the Vienna Scientific Cluster (VSC) for providing high performance computing resources.

Bibliographic Note

Several results in this thesis were already published in conference and journal papers and thus the chapters of this thesis are based on the following papers:

- **Chapter 3:** Monika Henzinger, Alexander Noe, Christian Schulz and Darren Strash. *"Practical Minimum Cut Algorithms"*. In: *ALENEX.*, 2018, pp. 48–61
  https://arxiv.org/abs/1708.06127
  Monika Henzinger, Alexander Noe, Christian Schulz and Darren Strash *"Practical Minimum Cut Algorithms"*. In: *ACM JEA.*, 2018, Vol. 23, Article 1.8 pp. 1-22
  https://doi.org/10.1145/3274662
- **Chapter 4:** Monika Henzinger, Alexander Noe and Christian Schulz. *"Shared-memory Exact Minimum Cuts"*. In: *IPDPS.*, 2019., pp. 13–22
  https://arxiv.org/abs/1808.05458
- **Chapter 5:** Monika Henzinger, Alexander Noe, Christian Schulz and Darren Strash. *"Finding All Global Minimum Cuts in Practice"*. In *ESA.*, 2020., Article 59, pp. 1–20
  https://arxiv.org/abs/2002.06948
- **Chapter 6:** Monika Henzinger, Alexander Noe and Christian Schulz. *"Practical Fully Dynamic Minimum Cut Algorithms"*. Manuscript., 2021.
  https://arxiv.org/abs/2101.05033
- **Chapter 7:** Alexandra Henzinger, Alexander Noe and Christian Schulz. *"ILP-based Local Search for Graph Partitioning"*. In: *SEA.*, 2018., Article 4, pp. 1–15
  https://arxiv.org/abs/1802.07144
  Alexandra Henzinger, Alexander Noe and Christian Schulz. *"ILP-based Local Search for Graph Partitioning"*. In: *ACM JEA.*, 2020., Vol. 25, Article 9, pp. 1–26
  https://doi.org/10.1145/3398634
- **Chapter 8:** Monika Henzinger, Alexander Noe and Christian Schulz. *"Shared-memory Branch-and-reduce for Multiterminal Cuts"*. In: *ALENEX.*, 2020., pp. 42–55
  https://arxiv.org/abs/1908.04141
  Monika Henzinger, Alexander Noe and Christian Schulz. *"Faster Parallel Multiterminal Cuts"*, Manuscript., 2020.
  https://arxiv.org/abs/2004.11666

Authors appear in alphabetical order in all listed publications.

Contents

CHAPTER 1

Introduction

1.1 Motivation

In the last few decades, world-spanning networks have created a plethora of structured and unstructured data. One very prominent example is the internet, which has seen the creation and growth of many networks, some of them to immense scale. This immense scale makes extracting information from the networks a hard task and necessitates the *partitioning* of networks into smaller, more readily comprehensible blocks. *Graphs* are a good abstraction to constitute such networks in a way that is understandable both for humans and machines. In a graph, we have a set of *vertices*, where each vertex represents an entity, such as a person, street address or a work package in a computer program. If two vertices are linked, such as friends in a social network or street addresses that are connected by a road, they are connected by an *edge*. This work focuses on *undirected graphs*, i.e. edges do not have a direction and a connection from A to B implies that B is also connected to A. In some graphs, vertices and edges have *weights*, for example if we have a graph that depicts a complex program where vertices are subprograms and connections represent communication, vertex weights indicate the computational complexity of a subprogram and edge weights indicate communication volume.

Graph algorithms aim to solve problems on such a graph. In this work, we look at various *cut problems* or *partitioning problems*, problems in which we want to partition the set of vertices into two or more subsets. Due to the large scale of global connections we want to be able to partition them into more manageable subgraphs. In all of the problems discussed in this dissertation, we aim to partition the set of vertices in such a way that the total weight of *cut edges*, i.e. edges that connect vertices in different blocks, or number of cut edges in graphs without edge weights, is minimized. We call the weight sum of cut edges the *cut size*. This allows the partitioning

of networks in such a way that communication over block boundaries in computing networks or separated relationships in social networks is as small as possible.

In this dissertation, we look at three important cut problems. In Part I we look at the *minimum cut problem* or *global minimum cut problem* where the aim is to find the smallest cut between two non-empty blocks of vertices without making any restrictions on the size of either block. In Part II, we look at the *balanced graph partitioning problem*. In this problem we aim to partition the vertex set into k blocks of roughly equal size so that the cut size is minimal. Part III deals with the *multiterminal cut problem*, where, given a set of k vertices called *terminals*, we want to find the smallest cut that pairwisely separates all terminals. The three parts of the dissertation are mostly independent; however, some techniques and ideas are shared between algorithms for different problems. We then give a brief re-introduction in the latter part and also cross-reference to the previous usage for further details.

We use the methodology and techniques of *algorithm engineering* [164] to give algorithms which give fast and strong solutions on a wide variety of different real-world instances but also stand on a sound theoretical base. In the methodology of algorithm engineering, algorithms are designed and analyzed using realistic machine models. In contrast to algorithm theory, these algorithms are then implemented and evaluated using experiments on data from real-world applications. Based on these experiments, we amend our design and repeat this inductive cycle until our algorithm is satisfactory. One important aspect is that the results of the implementation can be published as algorithm libraries so that other people can use them. As we develop algorithms for fundamental graph problems in this dissertation, we publish all of our algorithms under the permissive MIT license so that they can be used as building blocks for complex systems. The implementations in Parts I and III are available as the *VieCut* (Vienna Minimum Cuts) library [1], the implementations in Part II are integrated into the KaHIP [2] graph partitioning framework [161, 163]. For a detailed description of the methodology of algorithm engineering we refer the reader to [164].

For the minimum cut problem and the multiterminal cut problem, we develop and use a multitude of *local reduction rules* or *kernelization rules*. These reduction rules are related to the concept of fixed-parameter tractable (FPT) algorithms, where a hard problem can be solved efficiently as long as some problem parameter is not too large. FPT algorithms have long been a well-established field in algorithm theory, however only few of the techniques are implemented and tested on real datasets, and their practical potential is far from understood. More recently, the engineering aspect has gained some momentum. There are several experimental studies in this area that take up ideas from FPT or kernelization theory, e.g. for independent sets (or equivalently vertex cover) [27, 35, 44, 97, 98, 122], for cut tree construction[5], for treewidth computations [16, 117, 183], for the feedback vertex set problem [62, 114], for the dominating set problem [2], for the maximum cut

[1] https://github.com/VieCut/VieCut
[2] https://github.com/KaHIP/KaHIP

problem [57], for the cluster editing problem [23], and the matching problem [116]. In this dissertation, we make heavy use of data reduction techniques to improve the performance of algorithms for the minimum cut problem and the multiterminal cut problem. A recent survey on data reduction rules in practice is given in [1]. This survey covers data reduction for the global minimum cut problem and the multiterminal cut problem, as well as a multitude of other problems.

1.2 Main Contributions and Outline

This thesis consists of three individual parts, each addressing a fundamental cut problem. In this section, we give a brief overview where we briefly introduce the problems and then give the main contributions in this dissertation. In the introductory sections or chapters of each part we will give a more detailed outline.

1.2.1 Part I: Minimum Cut

In the first part of this dissertation we study the *(global) minimum cut problem*. This part is larger than the others, as we give inexact and exact shared-memory parallel algorithms for the problem, as well as an algorithm that finds all minimum cuts and an algorithm that maintains a minimum cut on a dynamically changing graph in which edges are inserted and deleted in arbitrary order. The minimum cut problem on a graph is to partition the vertices into two non-empty sets so that the sum of edge weights between the two sets is minimized. The minimum cut problem is one of the most fundamental graph problems and has seen a large amount of research. In Chapter 2, we give a brief overview over this research and introduce in a bit more detail algorithms that we use in the following chapters. We first give a practical shared-memory parallel heuristic algorithm in Chapter 3. This algorithm repeatedly reduces the input graph size with both heuristic and exact techniques by identifying and contracting edges that are likely or provably not part of a minimum cut. It is significantly faster than existing algorithms and has a lower empirically observed error rate than other inexact algorithms. Based on this inexact algorithm and practically efficient parallelization of an existing sequential algorithm, in Chapter 4, we then give a shared-memory parallel *exact* algorithm that provably finds a minimum cut for large graphs. Using 12 cores, this algorithm outperforms the state-of-the-art for exact minimum cut algorithms by a factor of up to 12.9 on some graphs.

In Chapter 5 we follow that up with an exact shared-memory parallel algorithm that finds *all* global minimum cuts in a graph and returns a compact *cactus graph* data structure that represents them all. This algorithm is able to solve instances with more than a billion edges and millions of minimum cuts in a few minutes on a single shared-memory parallel machine. We also give a new linear-time algorithm that, given a cactus graph data structure that represents all minimum cuts, gives the most balanced minimum cut.

Chapter 6 then details our algorithm that maintains a global minimum cut on a dynamically changing graph under edge insertions and deletions. As an edge insertion increases the value of some cuts but leaves most cuts untouched, it is useful to have a data structure with all minimum cuts, so that we only remove the minimum cuts whose value changed and retain all others without expensive recomputation. Our dynamic algorithm outperforms existing static algorithms by up to multiple orders of magnitude. While there have been various theoretical algorithms for finding all minimum cuts in a graph as well as for maintaining the minimum cut on a dynamically changing graph, to the best of our knowledge, our algorithms are the first publically available implementations for these problems.

1.2.2 Part II: Balanced Graph Partitioning

In the second part of this dissertation, we study the *balanced graph partitioning problem*. The balanced graph partitioning problem on an undirected graph with positive vertex and edge weights is to partition the vertex set into $k \geq 2$ blocks so that every block has roughly the same sum of contained node weights. More precisely, every block has a weight limit of $(1 + \epsilon)$ times the average block weight, i.e. the sum of all node weights in the graph divided by the number of blocks, for a given $\epsilon \geq 0$. In this dissertation, we present a novel meta-heuristic for the balanced graph partitioning problem. Our approach is based on integer linear programs that solve the partitioning problem to optimality. However, since those programs typically do not scale to large inputs, we adapt them to heuristically improve a given partition. We do so by defining a much smaller model that allows us to use symmetry breaking and other techniques that make the approach scalable. For example, in Walshaw's well-known benchmark tables [190], we are able to improve roughly half of all entries when the number of blocks is high. Additionally, we include our techniques in a memetic framework [163] and develop a crossover operation based on the proposed techniques. This extended evolutionary algorithm produces high-quality partitions from scratch. For half of the hard problems from Walshaw's graph partitioning benchmark, the result of our algorithm is at least as good as the previous best result. For 17%, the solution given is better than the previous best solution.

1.2.3 Part III: Multiterminal Cut Problem

In the third and final part of this dissertation we study the *multiterminal cut problem*. The multiterminal cut problem, given an undirected graph with positive edge weights and a set of k terminal vertices, is to partition the vertex set into k blocks so that each block contains exactly one terminal vertex. We present a fast shared-memory parallel exact algorithm for the multiterminal cut problem. In particular, we engineer existing as well as new efficient data reduction rules to transform the graph into a smaller equivalent instance. We use these reduction rules within a branch-and-reduce framework and combine this framework with an integer linear programming solver to give an algorithm that can solve a wide variety of large

instances. Additionally, we present an inexact heuristic algorithm that gives high-quality solutions for very hard instances in reasonable time. Among other techniques, we use local search to significantly improve a given solution to the problem. Our algorithms achieve improvements in running time of up to multiple orders of magnitude over the ILP formulation without data reductions.

Part I

The (Global) Minimum Cut Problem

CHAPTER 2

Minimum Cut

2.1 Introduction

Given an undirected graph with non-negative edge weights, the *minimum cut problem* is to partition the vertices into two sets so that the sum of edge weights between the two sets is minimized. An edge that crosses the partition boundary is called a *cut edge*. A cut that minimizes the weight sum of cut edges for all possible cuts is called the *minimum cut* or *global minimum cut* of the graph. In graphs where each edge has unit weight, a minimum cut is often also referred to as the *edge connectivity* of a graph [95, 143]. A variant of the minimum cut problem is the problem of finding *all global minimum cuts* in a graph.

The minimum cut problem has applications in many fields. In particular, for network reliability [106, 158], assuming equal failure chance on edges, the smallest edge cut in the network has the highest chance to disconnect the network; in VLSI design [120], a minimum cut can be used to minimize the number of connections between microprocessor blocks; and it is further used as a subproblem in the branch-and-cut algorithm for solving the Traveling Salesman Problem and other combinatorial problems [151]. Minimum cuts in similarity graphs can be used to find clusters [84, 194]. In community detection, the absence of a small cut inside a cluster can indicate a likely community in a social network [32]. In graph drawing [101], minimum cuts are used to separate the network. Finding all minimum cuts is an important subproblem for edge-connectivity augmentation algorithms [67, 148].

Part I of this dissertation is based on our papers on the global minimum cut problem. This chapter gives a brief overview of preliminaries and related work. In Section 2.2, we will introduce the notation and preliminaries used throughout this part of the dissertation. We give an overview of related work on the minimum cut problem and related problems in Section 2.3. We aim to give a general overview of algorithms and research and give some more detail about some of the algorithms and

techniques used in later chapters of this part. We then give a fast heuristic shared-memory parallel algorithm for the global minimum cut problem in Chapter 3 and based on this work, an exact shared-memory parallel algorithm in Chapter 4. In Chapter 5, we give an algorithm that finds all minimum cuts in a graph and gives their compact cactus graph representation. We use this cactus graph representation to maintain the global minimum cut in a dynamic graph, i.e. a graph in which edges are deleted and inserted over time. This dynamic algorithm is given in Chapter 6.

2.2 Preliminaries

Let $G = (V, E, c)$ be a weighted undirected simple graph with vertex set V, edge set $E \subset V \times V$ and non-negative edge weights $c : E \to \mathbb{N}$. We extend c to a set of edges $E' \subseteq E$ by summing the weights of the edges; that is, let $c(E') := \sum_{e=(u,v)\in E'} c(u, v)$ and let $c(u)$ denote the sum of weights of all edges incident to vertex 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 brevity, we simply call this the *degree* of the vertex. For a set of vertices $A \subseteq V$, we denote by $E[A] := \{(u, v) \in E \mid u \in A, v \in V\backslash A\}$; that is, the set of edges in E that start in A and end in its complement. A cut $(A, V\backslash A)$ is a partitioning of the vertex set V into two non-empty *partitions* A and $V\backslash A$, each being called a *side* of the cut. The *capacity* or *weight* of a cut $(A, V\backslash A)$ is $c(A) = \sum_{(u,v)\in E[A]} c(u, v)$. A *minimum cut* is a cut $(A, V\backslash A)$ that has smallest capacity $c(A)$ among all cuts in G. For two non-overlapping vertex sets $A \subset V$ and $B \subset V$, the capacity of the cut $c(A, B) = \sum_{(u,v)\in E, u\in A, v\in B} c(u, v)$ is the weight of all edges that connect vertices in A with vertices in B.

We use $\lambda(G)$ (or simply λ, when its meaning is clear) to denote the value of the minimum cut over all non-empty $A \subset V$. For two vertices s and t, we denote $\lambda(G, s, t)$ as the capacity of the smallest cut of G, where s and t are on different sides of the cut. $\lambda(G, s, t)$ is also known as the *minimum s-t-cut* of the graph. $\lambda(G, s, t)$ is also called the *connectivity* of vertices s and t. The connectivity $\lambda(G, e)$ of an edge $e = (s, t)$ is defined as $\lambda(G, s, t)$, the connectivity of its incident vertices. At any point in the execution of a minimum cut algorithm, $\widehat{\lambda}(G)$ (or simply $\widehat{\lambda}$) denotes the smallest upper bound of the minimum cut that the algorithm discovered up to that point. For a vertex $u \in V$, the size of the *trivial cut* $(\{u\}, V\backslash\{u\})$ is equal to the vertex degree of u. For most minimum cut algorithms, $\widehat{\lambda}(G)$ is initially set to the value of the minimum degree in G, as this is the weight of the trivial cut which separates the minimum degree vertex from the rest of the vertex set. When *clustering* a graph, we are looking for *blocks* of nodes V_1,...,V_k that partition V, that is, $V_1 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. The parameter k is usually not given in advance.

Many algorithms for the minimum cut problem use *graph contraction*. Given an edge $e = (u, v) \in E$, we define $G/(u, v)$ (or G/e) to be the graph after *contracting*

edge (u, v). In the contracted graph, we delete vertex v and all edges incident to this vertex. For each edge $(v, w) \in E$, we add an edge (u, w) with $c(u, w) = c(v, w)$ to G or, if the edge already exists, we give it the edge weight $c(u, w) + c(v, w)$. Given an edge $e \in (V \times V) \backslash E$, we define $G + e$ to be the graph after *inserting* edge e and given an edge $e \in E$ we define $G - e$ to be the graph after *deleting* edge e.

A graph with n vertices can have up to $\Omega(n^2)$ minimum cuts [104]. To see that this bound is tight, consider an unweighted cycle with n vertices. Each set of 2 edges in this cycle is a minimum cut of G. This yields a total of $\binom{n}{2}$ minimum cuts. However, all minimum cuts of an arbitrary graph G can be represented by a cactus graph C_G with up to $2n$ vertices and $\mathcal{O}(n)$ edges [146]. A cactus graph is a connected graph in which any two simple cycles have at most one vertex in common. In a cactus graph, each edge belongs to at most one simple cycle.

To represent all minimum cuts of a graph G in an edge-weighted cactus graph $C_G = (V(C_G), E(C_G))$, each vertex of C_G represents a possibly empty set of vertices of G and each vertex in G belongs to the set of one vertex in C_G. Let Π be a function that assigns to each vertex of C_G a set of vertices of G. Then every cut $(S, V(C_G) \backslash S)$ corresponds to a minimum cut $(A, V \backslash A)$ in G where $A = \cup_{x \in S} \Pi(x)$. In C_G, all edges that do not belong to a cycle have weight λ and all cycle edges have weight $\frac{\lambda}{2}$. A minimum cut in C_G consists of either one tree edge or two edges of the same cycle. We denote by n^* the number of vertices in C_G and m^* the number of edges in C_G. The weight $c(v)$ of a vertex $v \in C_G$ is equal to the number of vertices in G that are assigned to v.

2.3 Related Work

We now review algorithms for the global minimum cut and related problems. A closely related problem is the *minimum s-t-cut* problem, which asks for a minimum cut with nodes s and t in different partitions. Ford and Fulkerson [63] proved that minimum s-t-cut is equal to maximum s-t-flow. Gomory and Hu [78] observed that the (global) minimum cut can be computed with $n - 1$ minimum s-t-cut computations. For the following decades, this result by Gomory and Hu was used to find better algorithms for global minimum cut using improved maximum flow algorithms [105]. One of the fastest known maximum flow algorithms is the push-relabel algorithm [77] by Goldberg and Tarjan, which computes a maximum s-t-flow in $\mathcal{O}\left(mn \log \frac{n^2}{m}\right)$. Using their algorithm to find maximum s-t-flows, the algorithm of Gomory and Hu finds a global minimum cut in $\mathcal{O}\left(mn^2 \log \frac{n^2}{m}\right)$.

Hao and Orlin [81] adapt the push-relabel algorithm to pass information to future flow computations. When an iteration of the push-relabel algorithm is finished, they implicitly merge the source and sink vertices to form a new sink and find a new source vertex. Vertex heights are maintained over multiple iterations of push-relabel. With these techniques they achieve a total running time of

$\mathcal{O}\left(mn \log \frac{n^2}{m}\right)$ for a graph with n vertices and m edges, which is asymptotically equal to a single run of the push-relabel algorithm.

Padberg and Rinaldi [152] give a set of heuristics to find edges which can be contracted without affecting the minimum cut. Chekuri et al. [37] give an implementation of these heuristics that can be performed in time linear in the graph size. Using these heuristics it is possible to sparsify a graph while preserving at least one minimum cut in the graph. In Section 2.4.2 we outline their results, as our algorithms for the minimum cut problem make use of them.

Nagamochi et al. [143, 147] give a minimum cut algorithm which does not use any flow computations. Instead, their algorithm uses maximum spanning forests to find a non-empty set of contractible edges. This contraction algorithm is run until the graph is contracted into a single node. The algorithm has a running time of $\mathcal{O}(mn + n^2 \log n)$. As our exact algorithm is partially based on their contraction routine, we summarize their results in Section 2.4.1. Wagner and Stoer [178] give a simpler variant of the algorithm of Nagamochi, Ono and Ibaraki [147], which has the same asymptotic time complexity. The performance of this algorithm on real-world instances, however, is significantly worse than the performance of the algorithms of Nagamochi, Ono and Ibaraki or Hao and Orlin, as shown independently in experiments conducted by Jünger et al. [100] and Chekuri et al. [37]. In fact, both the algorithms of Hao and Orlin or Nagamochi, Ono and Ibaraki achieve close to linear running time on most benchmark instances [37, 100]. Based on the algorithm of Nagamochi, Ono and Ibaraki, Matula [136] gives a $(2+\varepsilon)$-approximation algorithm for the minimum cut problem. The algorithm contracts more edges than the algorithm of Nagamochi, Ono and Ibaraki to guarantee a linear time complexity while still guaranteeing a $(2 + \varepsilon)$-approximation factor.

Based on the observations that the contraction of an edge not in a minimum cut does not affect the value of said cut and that a minimum cut contains by definition only a small fraction of the edge set, Karger [103] gives a simple algorithm that contracts random edges until the graph has only two vertices left and then evaluates the cut value between them. They prove that by repeating this process $\mathcal{O}(n^2 \log n)$ times, the contraction algorithm finds a minimum cut with high probability. Thus, one can find a minimum cut in $\mathcal{O}(mn^2 \log n)$ in unweighted and $\mathcal{O}\left(mn^2 \log^3 n\right)$ in weighted graphs with high probability. Karger and Stein [105] show that minimum cut edges are contracted more often near the end of the contraction routine when the graph has only few vertices left. Their random contraction algorithm contracts a small set of edges, recurses twice and continues the contraction in both subproblems. Therefore the later stages are performed more often and the recursive contraction process only needs to be performed $\mathcal{O}\left(\log^2 n\right)$ times to find a minimum cut with high probability. This algorithm finds a minimum cut with high probability in $\mathcal{O}\left(n^2 \log^3 n\right)$ and was the first algorithm to break the $\tilde{\mathcal{O}}(mn)$ barrier. The $\tilde{\mathcal{O}}()$ notation ignores logarithmic factors. Gianinazzi et al. [75] give a parallel implementation of the algorithm of Karger and Stein. Other than that, there are no parallel

implementation of either algorithm known to us. More recently, the randomized contraction-based algorithm of Ghaffari et al. [74] solves the minimum cut problem on unweighted graphs in $\mathcal{O}(m \log n)$ or $\mathcal{O}\left(m + n \log^3 n\right)$.

Kawarabayashi and Thorup [111] give a deterministic near-linear time algorithm for the minimum cut problem on unweighted graphs, which runs in $\mathcal{O}\left(m \log^{12} n\right)$. Their algorithm works by growing contractible regions using a variant of PageRank [153]. It was improved by Henzinger et al. [95] to run in $\mathcal{O}\left(m \log^2 n \log\log^2 n\right)$ time, which is the currently fastest deterministic algorithm on unweighted graphs. Li and Panigrahi [128] give a deterministic algorithm that finds a global minimum cut on weighted graphs in $\mathcal{O}(m^{1+\epsilon})$ plus poly-logarithmic maximum flows for any constant $\epsilon > 0$.

Another approach to the global minimum cut problem is *tree packing*. Nash-Williams [150] proves that any graph with minimum cut λ contains a set of $\lambda/2$ edge-disjoint spanning trees. Such a tree packing can be found using Gabow's algorithm [66] in $\mathcal{O}(m\lambda \log n)$. Karger [106] introduces the concept of *k-respecting cuts*, where a cut k-respects a tree if it only cuts up to k tree edges. In his algorithm, Karger [106] finds a set of $\mathcal{O}(\log n)$ spanning trees so that the minimum cut 1- or 2-respects any of them with high probability. For each of the spanning trees, the algorithm computes the minimum cut that 1- or 2-respects it. This algorithm finds a minimum cut with high probability in $\mathcal{O}\left(m \log^3 n\right)$. Gawrychowski et al. [71] improve the running time of this algorithm to $\mathcal{O}\left(m \log^2 n\right)$, which is the currently fastest algorithm for the global minimum cut problem on weighted graphs. Bhardwaj et al. [19] give a simpler tree-packing-based algorithm with a running time of $\mathcal{O}\left(m \log^3 n\right)$ – matching the algorithm of Karger [106] – and implement a version with a running time of $\mathcal{O}\left(m \log^4 n\right)$. This implementation compares favorably against the algorithms of Karger and Stein [104] and Stoer and Wagner [178], however they do not compare their algorithm to algorithms that outperformed these by up to multiple orders of magnitudes in other experimental evaluations [37, 100], such as the algorithms of Nagamochi et al. [147] or the algorithm of Hao and Orlin [81]. Mukhopadhyay and Nanongkai [141] give an algorithm to find a minimum 2-respecting cut in $\mathcal{O}\left(m \log n + n \log^4 n\right)$. They also give a streaming variant of their algorithm that requires $\tilde{\mathcal{O}}(n)$ space and $\mathcal{O}(\log n)$ passes to compute the global minimum cut. Recently, Li [127] gave a deterministic algorithm using the techniques of Karger that finds a minimum cut in weighted graphs in $\mathcal{O}\left(m^{1+o(1)}\right)$.

Recently, Georgiadis et al. [73] carried out an experimental study of global minimum cut algorithms on directed graphs. Their experimental study shows that the directed version of Gabow's algorithm [66] performs well in practice; and for graphs with a low minimum cut value λ, local search based algorithms [36, 64] also perform well.

2.3.1 Finding *All* Global Minimum Cuts

Even though a graph can have up to $\binom{n}{2}$ minimum cuts [104], there is a compact representation of all minimum cuts of a graph called *cactus graph* with $\mathcal{O}(n)$ vertices and edges, as described earlier in Section 2.2. Karzanov and Timofeev [110] give the first polynomial time algorithm to construct the cactus representation for all minimum cuts. Picard and Queyranne [156] show that all minimum cuts separating two specified vertices can be found from a single maximum flow between them. Thus, similar to the classical algorithm of Gomory and Hu [78] for the minimum cut problem, we can find all minimum cuts in $n-1$ maximum flow computations. The algorithm of Karzanov and Timofeev [110] combines all those minimum cuts into a cactus graph representing all minimum cuts. Nagamochi and Kameda [144] give a representation of all minimum cuts separating two vertices s and t in a so-called (s,t)-cactus representation. Based on this (s,t)-cactus representation, Nagamochi et al. [146] give an algorithm that finds all minimum cuts and gives the minimum cut cactus in $\mathcal{O}(nm + n^2 \log n + n^* m \log n)$, where n^* is the number of vertices in the cactus. Fleischer [61] gives an algorithm based on the flow algorithm of Hao and Orlin that gives the cactus representation of all minimum cuts in a graph in the same asymptotic running time, $\mathcal{O}\left(mn \log \frac{n^2}{m}\right)$.

The aforementioned recursive contraction algorithm of Karger and Stein [105] above not only finds a single minimum cut, but is able to find all minimum cuts of a graph in $\mathcal{O}\left(n^2 \log^3 n\right)$ with high probability. Based on the algorithm of Karzanov and Timofeev [110] and its parallel variant given by Naor and Vazirani [149], they show how to give the cactus representation of the graph in the same asymptotic time. Likewise, the recent algorithm of Ghaffari et al. [74] finds all *non-trivial minimum cuts* (i.e. minimum cuts where each side contains at least two vertices) of a simple unweighted graph in $\mathcal{O}\left(m \log^2 n\right)$ time. Using the techniques of Karger and Stein, the algorithm can trivially give the cactus representation of all minimum cuts in $\mathcal{O}(n^2 \log n)$.

While there are implementations of the algorithm of Karger and Stein [37, 75] for the minimum cut problem, to the best of our knowledge there are no published implementations of either of the algorithms to find the cactus graph representing all minimum cuts.

A closely related problem is the cut tree problem (or Gomory-Hu tree problem), which aims to find a tree $T = (V, E_T, c_T)$, such that for each two vertices $u, v \in V$, the weight of the minimum u-v-cut is equal to the lightest edge weight on the unique path from u to v on T. This problem was first solved by Gomory and Hu [78] using $n-1$ minimum s-t-cuts and has been studied experimentally by Goldberg and Tsioutsiouliklis [76] and Akiba et al. [6], who solve the cut tree problem for graphs with millions of vertices and up to one billion edges in a few hours. Hartmann and Wagner [83] give a fully-dynamic algorithm to construct and maintain a cut tree under edge insertions, deletions, and weight changes.

2.3.2 Dynamic Minimum Cut

The field of dynamic graph algorithms [53] gives algorithms that maintain a solution to a graph problem on *dynamic graphs*, i.e. graphs that are undergoing updates such as the insertion or deletion of edges in the graph. A dynamic algorithm allows an efficient update of the solution instead of recomputing the solution from scratch. An algorithm performs an *update* when an edge is inserted or deleted and a *query* when we ask for a solution, e.g. the value of the minimum cut on the graph. A dynamic graph algorithm is called *incremental* if edges are only inserted and *decremental* if edges are only deleted. If edges are both inserted and deleted, we call the algorithm *fully dynamic*.

Henzinger [96] gives the first incremental minimum cut algorithm, which maintains the exact minimum cut with an amortized update time of $\mathcal{O}(\lambda \log n)$ per edge insertion and query time of $\mathcal{O}(1)$. The algorithm of Henzinger maintains the cactus graph of all minimum cuts and invalidates minimum cuts whose weight was increased due to an edge insertion. If there are not remaining minimum cuts, the algorithm recomputes all minimum cuts from scratch. Goranci et al. [79] manage to remove the dependence on λ from the update time and give an incremental algorithm with $\mathcal{O}\left(\log^3 n \log\log^2 n\right)$ amortized time per edge insertion and $\mathcal{O}(1)$ query time. They combine techniques of the incremental minimum cut algorithm of Henzinger with the quasi-linear static minimum cut algorithms of Kawarabayashi and Thorup [111] and Henzinger et al. [95].

For minimum cut values up to polylogarithmic size, Thorup [187] gives a fully dynamic algorithm with $\tilde{\mathcal{O}}(\sqrt{n})$ worst-case time per edge update. The algorithm of Thorup uses tree packing similar to the static algorithm of Karger [106]. Note that all of these algorithms are limited to unweighted graphs. For *planar* graphs with arbitrary edge weights, Łącki and Sankowski [121] give a fully-dynamic algorithm with $\mathcal{O}\left(n^{5/6} \log^{5/3} n\right)$ time per update and query. To the best of our knowledge, there exists no implementation of any of these algorithms.

2.4 Further Detail on Some Algorithms for the Minimum Cut Problem

2.4.1 Algorithm of Nagamochi, Ono and Ibaraki

We discuss the algorithm by Nagamochi, Ono and Ibaraki [143, 147] in greater detail since our work relies heavily on their results. The minimum cut algorithm of Nagamochi et al. works on graphs with positive integer weights. The intuition behind the algorithm is as follows: imagine you have an unweighted graph with minimum cut value exactly one. Then any spanning tree must contain at least one edge of each of the minimum cuts. Hence, after computing a spanning tree, every remaining edge can be contracted without losing the minimum cut. Nagamochi et al. extend this idea to the case where the graph can have edges with positive weight as well as the

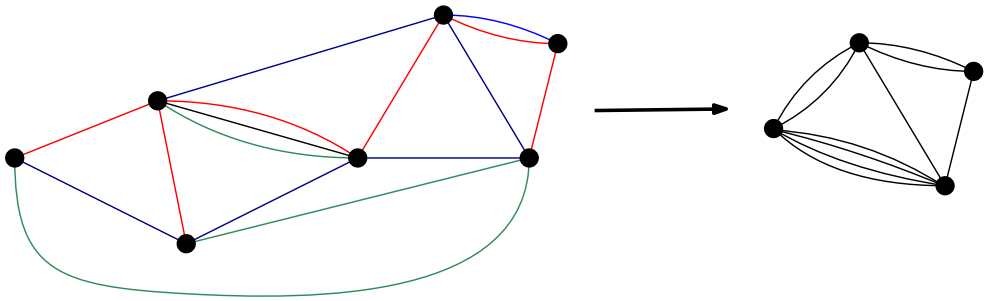


Figure 2.1: Left: Partition of edge-set into edge-disjoint spanning forests. Right: resulting graph after contracting black and green spanning forests.

case in which the minimum cut is bounded by $\hat{\lambda}$. The first observation is the following: assume that you already found a cut in the current graph of size $\hat{\lambda}$ and you want to find out whether there is a cut of size $< \hat{\lambda}$. Then the contraction process only needs to ensure that the contracted graph contains all cuts having a value *strictly* smaller than $\hat{\lambda}$. To do so, Nagamochi et al. build *edge-disjoint maximum spanning forests* and contract all edges that are not in one of the $\hat{\lambda} - 1$ first spanning forests, as those connect vertices that have connectivity of at least $\hat{\lambda}$. Note that the edge-disjoint maximum spanning forest *certifies* for any edge $e = (u, v)$ that is not in the forest that the minimum cut between u and v is at least $\hat{\lambda}$. Hence, the edge can be "safely" contracted. As weights are integral, this guarantees that the contracted graph still contains all cuts that are *strictly* smaller than $\hat{\lambda}$. Figure 2.1 shows a small graph where the edge set is partitioned into edge-disjoint maximum spanning forests. For this, an edge e of weight $c(e)$ is replaced with $c(e)$ unweighted edges. The first two spanning forests (red, blue) are trees, the subsequent ones (green, black) are not. As the minimum vertex degree is 3, the upper bound for the minimum cut $\hat{\lambda} = 3$. For each green and black edge $e = (u, v)$, we can find a path from u to v that only consists of red edges and one that only consists of blue edges. Thus, the connectivity $\lambda(u, v) \geq 3$ and no cut of value < 3 can separate u and v. Using this information, we can contract all green and black edges and repeat this process on the resulting graph until there are only two vertices left.

Since it would be inefficient to directly compute $\hat{\lambda} - 1$ edge disjoint maximum spanning trees and the running time would then depend on the value of the minimum cut λ, the authors give a modified algorithm CAPFOREST to be able to detect contractable edges faster. This is done by computing a lower bound for the connectivity of the endpoints of an edge which serves as a certificate for an edge to be contractable. If the lower bound for an edge e is $\geq \hat{\lambda}$, then e can be contracted, as no cut smaller than $\hat{\lambda}$ contains it. The minimum cut algorithm of Nagamochi et al. has a worst case running time of $\mathcal{O}(mn + n^2 \log n)$. In experimental evaluations [37, 94, 100], it is one of the fastest exact minimum cut algorithms, both on real-world and generated instances.

We now take a closer look at details of the algorithm. To find contractable edges, the algorithm uses a modified *breadth-first graph traversal* (BFS) algorithm CAPFOREST. The CAPFOREST algorithm starts at an arbitrary vertex. In each step, the algorithm visits (scans) the vertex v that is most strongly connected to the already visited vertices. For this purpose, a priority queue $\mathcal{Q}$ is used, in which the connectivity strength of each vertex $r : V \to \mathbb{N}$ to the already discovered vertices is used as a key. When scanning a vertex v, the value $r(w)$ is kept up to date for every unscanned neighbor w of v by setting i.e. $r(w) := r(w) + c(e)$. Moreover, for each edge $e = (v, w)$, the algorithm computes a lower bound $q(e)$ for the connectivity, i.e. the smallest cut $\lambda(G, v, w)$, which places v and w on different sides of the cut. To be precise, it is set to the connectivity strength of w to the already scanned vertices $q(e) := r(w)$. The vertices are scanned in an order such that the next scanned vertex is the unscanned vertex with the highest connection strength value r (the order used by the algorithm). Using this order, Nagamochi et al. [143, 147] show that $r(w)$ is a lower bound on $\lambda(G, v, w)$. The order in which the vertices are scanned is important for the correctness of the algorithm.

For an edge that has connectivity $\lambda(G, v, w) \geq \hat{\lambda}$, we know that there is no cut smaller than $\hat{\lambda}$ that places v and w in different partitions. If an edge e is not in a given cut $(A, V \backslash A)$, it can be contracted without affecting the cut. Thus, we can contract edges with connectivity of at least $\hat{\lambda}$ without losing any cuts smaller than $\hat{\lambda}$. As $q(e) \leq \lambda(G, u, v)$ (lower bound), all edges with $q(e) \geq \hat{\lambda}$ are contracted.

Afterwards, the algorithm continues on the contracted graph. A single iteration of the subroutine can be performed in $\mathcal{O}(m + n \log n)$ time. The authors show that in each BFS run, at least one edge of the graph can be contracted [143]. This yields a total running time of $O(mn + n^2 \log n)$. However, in practice the number of iterations is typically much less than $n - 1$, rather it is often proportional to $\log n$.

2.4.2 Exact Reductions by Padberg and Rinaldi

Padberg and Rinaldi [152] give conditions that allow for shrinking the size of a graph. They prove the following lemma which allows the contraction of an edge $e = (u, v)$.

Lemma 2.4.1. *[Padberg and Rinaldi [152], Corollary 2.2] Let $u \neq v \in V$. If there exists $Y \subseteq N(u) \cap N(v)$ so that*
(a) $c(u) \leq 2c(\{u\}, Y + \{v\} - T)$, or
(b) $c(v) \leq 2c(\{v\}, T + \{u\})$
holds for all $T \subseteq Y$, then either $c(u)$ or $c(v)$ is a minimum cut or there exists a minimum cut X,Y such that both $u \in X$ and $v \in X$.

If Lemma 2.4.1 holds for an edge $e = (u, v)$, it can be contracted since the trivial cuts $c(u)$ and $c(v)$ were already evaluated and an edge that is not part of a minimum cut can be contracted without affecting the value of said minimum cut. Unfortunately, checking Lemma 2.4.1 is NP-complete in general ([152], Remark 2.3) as the knapsack problem can be reduced to the problem. It is thus not feasible to

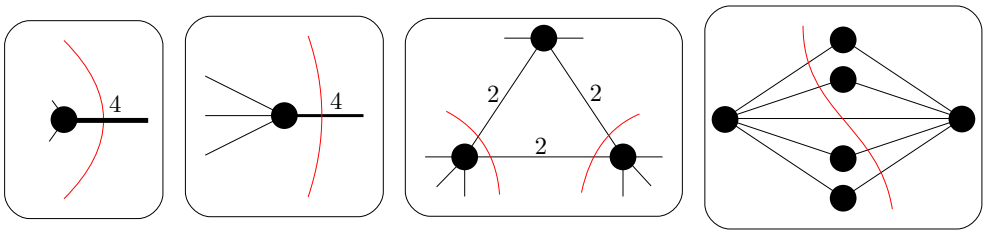


Figure 2.2: Reductions of Padberg and Rinaldi [152]

check Lemma 2.4.1 for every edge, especially not for edges whose incident vertices have a large shared neighborhood. In their work, Padberg and Rinaldi give a set of conditions that follow from Lemma 2.4.1 and can be checked faster. These conditions are given in Lemma 2.4.2 and Figure 2.2.

Lemma 2.4.2. *[Padberg and Rinaldi [152]] If two vertices $v, w \in V$ with an edge $(v, w) \in E$ satisfy at least one of the following four conditions and (v, w) is not the only edge adjacent to either v or w, then they can be contracted without increasing the value of the minimum cut:*

1. $c(v, w) \geq \hat{\lambda}$,
2. $c(v) \leq 2c(v, w)$ *or* $c(w) \leq 2c(v, w)$,
3. $\exists u \in V$ *such that* $c(v) \leq 2\{c(v, w) + c(v, u)\}$ *and* $c(w) \leq 2\{c(v, w) + c(w, u)\}$, *or*
4. $c(v, w) + \sum_{u \in V} \min\{c(v, u), c(w, u)\} \geq \hat{\lambda}$.

Condition 1 contracts every edge e whose weight is $\geq \hat{\lambda}$. By definition of a cut, we know that no cut that contains edge e can have weight $< \hat{\lambda}$. It is therefore safe to contract edge e without losing any cuts smaller than the smallest cut already found.

Condition 2 contracts an edge $e = (v, w)$, if its weight is at least half the degree of one of its incident vertices. In other words, e is at least as heavy as all other edges incident to one of v or w. Without loss of generality, let v be that vertex. For every cut that contains e, we can find another cut that replaces it with all other edges incident to v. As $c(v) \leq 2c(v, w)$, this cut is at most as heavy as the original cut. Edge e can therefore be contracted, as there is at least one minimum cut that does not contain it. Condition 3 is closely related but additionally uses information from the shared neighborhood of v and w. If there is a vertex u in the shared neighborhood of v and w (i.e. u, v, and w form a triangle), so that the two triangle edges incident to v and w respectively each have weight of at least half of their respective vertex degree, every cut that separates v and w can be replaced with one of smaller or equal weight that does not separate them. Note that the condition does not require

that u is in the shared neighborhood of v and w; however, if it is not, condition 2 already detects every contractible edge that condition 3 does.

Condition 4 uses the whole shared neighborhood of v and w. Each cut that separates vertices v and w has to contain (v, w) and for every shared neighbor $u \in N(v) \cap N(w)$, either edge (u, v) or (u, w). Thus, we can sum up over the lighter edge for each shared neighbor and find a lower bound for the connectivity $\lambda(u, v)$. If this bound is already $\geq \widehat{\lambda}$, (u, v) can be contracted.

In their experimental evaluation of various algorithms for the minimum cut problem, Chekuri et al. [37] use these reductions to improve the performance of the algorithms by contracting edges that fulfill either of the criteria in Lemma 2.4.2. Conditions 1 and 2 can be exhaustively checked in linear time.

In order to check conditions 3 and 4 exhaustively, potentially all triangles need to be checked. As an arbitrary graph can have up to $\Theta(m^{\frac{3}{2}})$ triangles [166], an exhaustive check introduces excessive running time penalties. Chekuri et al. [37] thus perform linear-time passes that check these conditions on a subset of vertex sets as follows. In the beginning of a pass, their algorithm marks each vertex as unscanned and then scans vertices in order. When scanning vertex v, their algorithm checks conditions 3 and 4 for each unscanned neighbor w of v. In this check of v and w, they test condition 3 for all vertices u in the common neighborhood $N(v) \cap N(w)$. When iterating over all vertices in the common neighborhood, they compute the sum in condition 4 by adding up the smaller of the two edge weights for each vertex in the common neighborhood. Afterwards they mark both v and w as scanned. This ensures a time complexity of $\mathcal{O}(n + m)$, as each edge is processed at most twice. However, not all possible edges (v, w) are tested to see whether the incident vertices v and w can be contracted.

2.5 Graph Instances

In our experiments, we use a wide variety of large static and dynamic graph instances. These are social graphs, web graphs, co-purchase matrices, cooperation networks and some generated instances. These large graphs from [15, 24, 25, 46, 159, 160] are detailed in Table 2.1. All instances are undirected. If the original graph is directed, we generate an undirected graph by removing edge directions and then removing duplicate edges. In our experiments, we use three families of graphs for different subproblems. In Table 2.1, we show the number of vertices n and edges m for each graph, the minimum cut λ and the minimum degree δ. Additionally, we also show the number of vertices in the cactus graph n^* for all minimum cuts. This number is an indication of how many minimum cuts exist. A value $n^* = 2$ indicates that there is a single minimum cut that separates two sides. A larger value indicates that there are multiple minimum cuts in the graph.

Graph family A consists of problems for finding *some* minimum cut. These graphs generally have multiple connected components and contain vertices with

Graph Family A					
Graph	n	m	λ	δ	n^*
com-orkut	$2.4M$	$112M$	14	16	2
	114 190	$18M$	89	95	2
	107 486	$17M$	76	98	2
	103 911	$17M$	70	100	2
eu-2005	605 264	$15M$	1	10	63
	271 497	$10M$	2	25	3
	58 829	$3.7M$	29	60	2
	5 289	464 821	19	100	2
gsh-2015-host	$25M$	$1.3B$	1	10	175
	$5.3M$	$944M$	1	50	32
	$2.6M$	$778M$	1	100	16
	98 275	$188M$	1	1 000	3
hollywood-2011	$1.3M$	$109M$	1	20	13
	576 111	$87M$	6	60	2
	328 631	$71M$	77	100	2
	138 536	$47M$	27	200	2
twitter-2010	$13M$	$958M$	1	25	2
	$10M$	$884M$	1	30	3
	$4.3M$	$672M$	3	50	3
	$3.5M$	$625M$	3	60	2
uk-2002	$9M$	$226M$	1	10	1 940
	$2.5M$	$115M$	1	30	347
	783 316	$51M$	1	50	138
	98 275	$11M$	1	100	20
uk-2007-05	$68M$	$3.1B$	1	10	3 202
	$16M$	$1.7B$	1	50	387
	$3.9M$	$862M$	1	100	134
	223 416	$183M$	1	1 000	2
Graph Family B					
amazon	64 813	153 973	1	1	10 068
auto	448 695	$3.31M$	4	4	43
	448 529	$3.31M$	5	5	102
	448 037	$3.31M$	6	6	557
	444 947	$3.29M$	7	7	1 128
	437 975	$3.24M$	8	8	2 792
	418 547	$3.10M$	9	9	5 814
caidaRouterLevel	190 914	607 610	1	1	49 940
cfd2	123 440	$1.48M$	7	7	15
citationCiteseer	268 495	$1.16M$	1	1	43 031
	223 587	$1.11M$	2	2	33 423
	162 464	862 237	3	3	23 373
	109 522	435 571	4	4	16 670
	73 595	225 089	5	5	11 878
	50 145	125 580	6	6	8 770
cnr-2000	325 557	$2.74M$	1	1	87 720
	192 573	$2.25M$	2	2	33 745
	130 710	$1.94M$	3	3	11 604
	110 109	$1.83M$	4	4	9 256
	94 664	$1.77M$	5	5	4 262
	87 113	$1.70M$	6	6	5 796
	78 142	$1.62M$	7	7	3 213
	73 070	$1.57M$	8	8	2 449
coAuthorsDBLP	299 067	977 676	1	1	45 242
cs4	22 499	43 858	2	2	2
delaunay_n17	131 072	393 176	3	3	1 484
fe_ocean	143 437	409 593	1	1	40
kron-logn16	55 319	$2.46M$	1	1	6 325
luxembourg	114 599	239 332	1	1	23 077
vibrobox	12 328	165 250	8	8	625
wikipedia	35 579	495 357	1	1	2 172

Graph Family B (continued)					
Graph	n	m	λ	δ	n^*
amazon-2008	735 323	$3.52M$	1	1	82 520
	649 187	$3.42M$	2	2	50 611
	551 882	$3.18M$	3	3	35 752
	373 622	$2.12M$	5	5	19 813
	145 625	582 314	10	10	64 657
coPapersCiteseer	434 102	$16.0M$	1	1	6 372
	424 213	$16.0M$	2	2	7 529
	409 647	$15.9M$	3	3	7 495
	379 723	$15.5M$	5	5	6 515
	310 496	$13.9M$	10	10	4 579
eu-2005	862 664	$16.1M$	1	1	52 232
	806 896	$16.1M$	2	2	42 151
	738 453	$15.7M$	3	3	21 265
	671 434	$13.9M$	5	5	18 722
	552 566	$11.0M$	10	10	23 798
hollywood-2009	$1.07M$	$56.3M$	1	1	11 923
	$1.06M$	$56.2M$	2	2	17 386
	$1.03M$	$55.9M$	3	3	21 890
	942 687	$49.2M$	5	5	22 199
	700 630	$16.8M$	10	10	19 265
in-2004	$1.35M$	$13.1M$	1	1	278 092
	909 203	$11.7M$	2	2	89 895
	720 446	$9.2M$	3	3	45 289
	564 109	$7.7M$	5	5	33 428
	289 715	$5.1M$	10	10	12 947
uk-2002	$18.4M$	$261.6M$	1	1	$2.5M$
	$15.4M$	$254.0M$	2	2	$1.4M$
	$13.1M$	$236.3M$	3	3	938 319
	$10.6M$	$207.6M$	5	5	431 140
	$7.6M$	$162.1M$	10	10	298 716
	657 247	$26.2M$	50	50	24 139
	124 816	$8.2M$	100	100	3 863

Graph Family C					
Dynamic Graph	n	Insertions	Deletions	Batches	λ
aves-weaver-social	445	1 423	0	23	0
ca-cit-HepPh	28 093	$4.60M$	0	2 337	0
ca-cit-HepTh	22 908	$2.67M$	0	219	0
comm-linux-kernel-r	63 399	$1.03M$	0	839 643	0
copresence-InVS13	987	394 247	0	20 129	0
copresence-InVS15	1 870	$1.28M$	0	21 536	0
copresence-LyonS	1 922	$6.59M$	0	3 124	0
copresence-SFHH	1 924	$1.42M$	0	3 149	0
copresence-Thiers	1 894	$18.6M$	0	8 938	0
digg-friends	279 630	$1.73M$	0	$1.64M$	0
edit-enwikibooks	134 942	$1.16M$	0	$1.13M$	0
fb-wosn-friends	63 731	$1.27M$	0	736 675	0
ia-contacts_dublin	10 972	415 912	0	76 944	0
ia-enron-email-all	87 273	$1.13M$	0	214 908	0
ia-facebook-wall	46 952	855 542	0	847 020	0
ia-online-ads-c	$15.3M$	133 904	0	56 565	0
ia-prosper-loans	89 269	$3.39M$	0	1 259	0
ia-stackexch-user	545 196	$1.30M$	0	1 154	1
ia-sx-askubuntu-a2q	515 273	257 305	0	257 096	0
ia-sx-mathoverflow	88 580	390 441	0	390 051	0
ia-sx-superuser	567 315	$1.11M$	0	$1.10M$	0
ia-workplace-cts	987	9 827	0	7 104	0
imdb	150 545	296 188	0	7 104	0
insecta-ant-colony1	113	111 578	0	41	4 285
insecta-ant-colony2	131	139 925	0	41	3 742
insecta-ant-colony3	160	241 280	0	41	1 539
insecta-ant-colony4	102	81 599	0	41	1 838
insecta-ant-colony5	152	194 317	0	41	6 671
insecta-ant-colony6	164	247 214	0	39	2 177
mammalia-voles-kcs	1 218	4 258	0	64	0
SFHH-conf-sensor	1 924	70 261	0	3 509	0
soc-epinions-trust	131 828	717 129	123 670	939	0
soc-flickr-growth	$2.30M$	$33.1M$	0	134	0
soc-wiki-elec	8 297	83 920	23 093	101 014	0
soc-youtube-growth	$3.22M$	$12.2M$	0	203	0
sx-stackoverflow	$2.58M$	392 515	0	384 680	0

Table 2.1: Statistics of the static and dynamic graphs used in experiments.

very low degree. To create instances with $\lambda > 0$, we use the largest connected component. As we want to find *some* minimum cut, instances in which the minimum cut is equal to the minimum degree are trivial to solve. Thus, we use a k-core decomposition [18, 171] to generate versions of the graphs with a minimum degree of k and use versions where $k > \lambda$, i.e. there exists at least one cut

strictly smaller than the minimum degree and the problem is therefore not trivial to solve. Generally these instances have very few minimum cuts, and in many cases, there is only a single minimum cut.

The k-core of a graph $G = (V, E)$ is the largest subgraph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$, which fulfills the condition that every vertex in G' has a degree of at least k. We perform our experiments on the largest connected component of G'. For every real-world graph we use, we compute a set of 4 different k-cores, in which the minimum cut is not equal to the minimum degree.

We generate a diverse set of graphs with different sizes. For the large graphs `gsh-2015-host` and `uk-2007-05`, we use cores with k in 10, 50, 100, and 1000. In the smaller graphs we use cores with k in 10, 30, 50, and 100. `twitter-2010` and `com-orkut` had only a few cores for which the minimum cut is not equal to the minimum degree. Therefore we used those cores. As `hollywood-2011` is very dense, we used $k = 20, 60, 100, 200$.

Graph family B consists of problems for finding *all* minimum cuts. Thus, the problem does not become trivial when the minimum cut is equal to the minimum degree. We therefore do not compute k-cores of the graphs and instead run the algorithms on the largest connected component of the source graph. However, as most large real-world networks have cuts of size 1, finding all minimum cuts becomes essentially the same as finding all bridges, which can be solved in linear time using depth-first search [184]. Usually there is one huge block that is connected by minimum cuts to a set of small and medium size blocks. Thus, we use our minimum cut algorithms to generate a more balanced set of instances. We find all minimum cuts and contract each edge that does not connect two vertices of the largest block. Thus, the remaining graph only contains the huge block and is guaranteed to have a minimum cut value $> \lambda$. We use this method to generate multiple graphs with different minimum cuts for each instance. These graphs usually have $\lambda = \delta$, i.e. the value of the minimum cut is equal to the minimum degree, and have a large set of minimum cuts. Thus, finding some minimum cut on these graphs is very easy, but finding all of them is a significantly harder problem.

Graph family C consists of a set of 36 dynamic graphs from Network Repository [159, 160]. These graphs consist of a sequence of edge insertions and deletions. While edges are inserted and deleted, all vertices are static and remain in the graph for the whole time. Each edge update has an associated timestamp and a set of updates with the same timestamp is called a *batch*. Most of the graphs in this dataset have multiple connected components, i.e. their minimum cut λ is 0.

CHAPTER 3

VieCut: Shared-memory Parallel Heuristic Minimum Cut

In this chapter, we give a *practical* shared-memory parallel algorithm for the minimum cut problem. Our algorithm is heuristic (i.e., there are no guarantees on solution quality), randomized, and has a running time of $\mathcal{O}(n + m)$ when run sequentially. The algorithm works in a multilevel fashion: we repeatedly reduce the input graph size with both heuristic and exact techniques, and then solve the smaller remaining problem with exact methods. Our heuristic technique identifies edges that are unlikely to be in a minimum cut using the label propagation technique introduced by Raghavan et al. [157] and contracts them in bulk. We further combine this technique with exact reduction routines by Padberg and Rinaldi [152], as discussed in Section 2.4.2. We perform extensive experiments comparing our algorithm with other heuristic algorithms as well as exact algorithms on real-world and generated instances, which include graphs of up to 70 million vertices and 5 billion edges. Results indicate that our algorithm finds optimal cuts for almost all instances and also that the empirically observed error rate is lower than for competing approximation algorithms (i.e., that come with guarantees on the solution quality). At the same time, even when run sequentially, our algorithm is significantly faster (up to a factor of 4.85) than other state-of-the-art algorithms. To further speedup computations, we also give a version of our algorithm that performs random edge contractions as preprocessing. This version achieves a lower running time and has better parallel scalability at the expense of a higher error rate.

The content of this chapter is based on [93] and [94].

3.1 VieCut: A Parallel Heuristic Minimum-Cut Algorithm

In this section we introduce our new approach to the minimum cut problem. Our algorithm is based on edge contractions: we find densely connected vertices in the graph and contract those into single vertices. Due to the way contractions are defined, we ensure that a minimum cut of the contracted graph corresponds to a minimum cut of the input graph. Once the graph is contracted, we apply exact reductions. These two contraction steps are repeated until the graph has a constant number of vertices. We apply an exact minimum cut algorithm to find the optimal cut in the contracted graph.

Throughout our algorithm we maintain a variable $\hat{\lambda}$, which denotes the current lowest upper bound for the minimum cut. In the beginning, $\hat{\lambda}$ equals the minimum node degree of G. After every contraction, if the minimum node degree in the contracted graph is smaller than $\hat{\lambda}$, we set $\hat{\lambda}$ to the minimum node degree of the contracted graph. As we only perform contractions and therefore do not introduce any new cuts we can guarantee that our algorithm will never output a value that is lower than the minimum cut.

This chapter is organized as follows. In Section 3.2, we give a general overview of our algorithm `VieCut` for the global minimum cut problem and discuss in detail the parts that form the algorithm. Additionally we give insight into parallelization and implementation details. In Section 3.3, we discuss a variant which combines `VieCut` with random edge contraction to achieve an even lower running time at the expensive of a higher error rate. We then show experiments and results in Section 3.4 before we conclude in Section 3.5.

3.2 Fast Minimum Cuts

The algorithm of Karger and Stein [105] spends a large amount of time computing graph contractions recursively. One idea to speed up their algorithm therefore is to increase the number of contracted edges per level. However, this strategy is undesirable: it increases the error both in theory and in practice, as their algorithm selects edges for contraction at random. We solve this problem by introducing an aggressive coarsening strategy that contracts a large number of edges that are unlikely to be in a minimum cut.

We first give a high level overview before diving into the details of the algorithm. Our algorithm starts by using the label propagation algorithm [157] to cluster the vertices into densely connected clusters. We then use a correcting algorithm to find misplaced vertices that should form a singleton cluster. Finally, we contract the graph and apply the exact reductions of Padberg and Rinaldi [152], as discussed in Section 2.4.2. We repeat these contraction steps until the graph has at most a constant number n_0 of vertices. When the contraction step is finished we apply the

algorithm of Nagamochi, Ono and Ibaraki [147], as discussed in Section 2.4.1, to find the minimum cut of the contracted graph. Finally, we transfer the resulting cut into a cut in the original graph. Overview pseudocode can be found in Algorithm 1.

Algorithm 1 `VieCut`

Input: $G = (V, E, c : V \to \mathbb{N}_{>0}), n_0$: bound for exact algorithm,

1: $\mathcal{G} \leftarrow G$
2: **while** $|V_{\mathcal{G}}| > n_0$ **do** ▷ compute inexact kernel
3: $\mathcal{C} \leftarrow$ computeClustering($\mathcal{G}$) ▷ label propagation clustering
4: $\mathcal{C}' \leftarrow$ fixMisplacedVertices($\mathcal{G}, \mathcal{C}$)
5: $G_{\mathcal{C}'} \leftarrow$ contractClustering($\mathcal{G}, \mathcal{C}'$)
6: $\mathcal{E} \leftarrow$ findContractableEdges($G_{\mathcal{C}'}$) ▷ further apply exact reductions
7: $\mathcal{G} \leftarrow$ contractEdges($G_{\mathcal{C}'}, \mathcal{E}$)
8: **end while**
9: $(A, B) \leftarrow$ NagamochiOnoIbaraki($\mathcal{G}$) ▷ solve minimum cut problem on final kernel
10: $(A', B') \leftarrow$ solutionTransfer(A, B) ▷ transfer solution to input network
11: **return** (A', B')

The *label propagation algorithm* (LPA) was proposed by Raghavan et al. [157] for graph clustering. It is a fast algorithm that locally minimizes the number of edges cut. We outline the algorithm briefly. Initially, each node is in its own cluster/block, i.e. the initial block ID of a node is set to its node ID. The algorithm then works in rounds. In each round, the nodes of the graph are traversed in a random order. When a node v is visited, it is *moved* to the block that has the strongest connection to v, i.e. it is moved to the cluster C that maximizes $c(\{(v, u) \mid u \in N(v) \cap C\})$. Ties are broken uniformly at random. The block IDs of round i are used as initial block IDs of round $i + 1$.

In the original formulation [157], the process is repeated until the process converges and no vertices change their labels in a round. Kothapalli et al. [118] show that label propagation finds all clusters in few iterations with high probability, when the graph has a distinct cluster structure. Hence, we perform at most ℓ iterations of the algorithm, where ℓ is a tuning parameter. One LPA round can be implemented to run in $\mathcal{O}(n + m)$ time. As we only perform ℓ iterations, the algorithm runs in $\mathcal{O}(n + m)$ time as long as ℓ is constant. In this formulation the algorithm has no bound on the number of clusters. However, we can modify the first iteration of the algorithm, so that a vertex i is not allowed to change its label when another vertex already moved to block i. In a connected graph this guarantees that each cluster has at least two vertices and the contracted graph has at most $\frac{|V|}{2}$ vertices. The only exceptions are connected components consisting of only a single vertex (*isolated* vertices with a degree of 0) which can not be contracted by the label propagation algorithm. However, when such a vertex is detected, our minimum cut algorithm

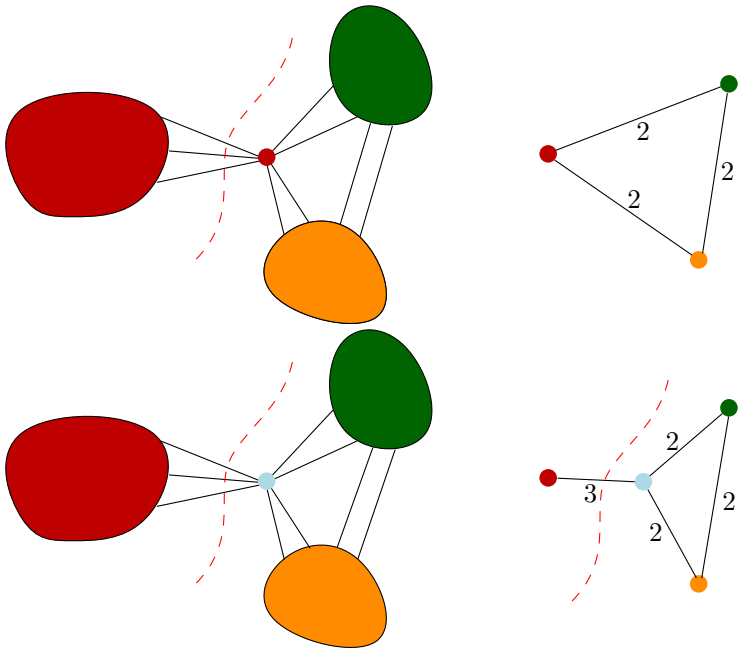


Figure 3.1: A case in which label propagation misplaces vertices. Top: label propagation assigns the centered vertex correctly to the left (red) cluster. However, this results in a situation in which the contracted graph no longer contains the minimum cut. Setting the centered vertex to be a singleton fixes this problem.

terminates immediately and returns a cut of size 0. In practice we do not use the modification, as label propagation usually returns far fewer than $\frac{|V|}{2}$ clusters.

Once we have computed the clustering with label propagation, we search for single misplaced vertices using a *correcting algorithm*. A misplaced vertex is a vertex, whose removal from its cluster improves the minimum weighted degree of the contracted graph. Figure 3.1 gives an example in which the clustering misplaces a vertex. To find misplaced vertices, we sweep over all vertices and check for each vertex whether it is misplaced. We only perform this correcting algorithm on small clusters, which have a size of up to $\log_2(n)$ vertices, as it is likely that large clusters would have more than a single node misplaced at a time. In general, one can enhance this algorithm by starting at any node whose removal would lower the cluster degree and greedily adding neighbors whose removal further lowers the remaining cluster degree. However, even when performing this greedy search on all clusters, this did not yield further improvement over the single vertex version on small clusters. This correcting step never makes a solution worse and on several instances it improved the value of the final result.

After we computed the final clustering, we contract it to obtain a coarser graph. Contracting the clustering works as follows: each block of the clustering is contracted into a single node. There is an edge between two nodes u and v in the

contracted graph if the two corresponding blocks in the clustering are adjacent to each other in G, i.e. block u and block v are connected by at least one edge. The weight of an edge (A, B) is set to the sum of the weight of edges that run between block A and block B of the clustering. Our contractions ensure that a minimum cut of the coarse graph corresponds to a cut of the finer graph with the same value, but not vice versa: we can not guarantee that a minimum cut of the contracted graph is equal to a minimum cut of the original graph. It is possible that a single cluster contains nodes from both sides of the cut. In this case, contracting the cluster eliminates this minimum cut. If all minimum cuts are eliminated, $\lambda(G_{\mathcal{C}}) > \lambda(G)$. Thus our newly introduced reduction for the minimum cut problem is *inexact*. However, the following lemma holds:

Lemma 3.2.1. *If there exist a minimum cut of G such that each cluster of the clustering $\mathcal{C}$ is completely contained in one side of the minimum cut of G and $|V_{\mathcal{C}}| > 1$, then $\lambda(G) = \lambda(G_{\mathcal{C}})$.*

Proof. As node contraction removes cuts but does not add any new cuts, $\lambda(G_{\mathcal{C}}) \geq \lambda(G)$ for each contraction with $|V_{\mathcal{C}}| > 1$. For an edge e in G, which is not part of some minimum cut of G, $\lambda(G) = \lambda(G/e)$ [105]. Contraction of a cluster C in G can also be represented as the contraction of all edges in any spanning tree of C. If the cluster C is on one side of the minimum cut, none of the spanning edges are part of the minimum cut. Thus we can contract each of the edges without affecting the minimum cut of G. We can perform this contraction process on each of the clusters and $\lambda(G_{\mathcal{C}}) = \lambda(G)$. □

Exact Reductions by Padberg and Rinaldi

We use the Padberg-Rinaldi reductions to further shrink the size of the graph. These are exact reductions, which do not modify the size of the minimum cut. Our algorithm contracts all edges which are marked by the Padberg-Rinaldi heuristics. In our experiments, we also tried to run the exact reductions first and cluster contraction last. However, this resulted in a slower algorithm since not many exact reductions could be applied on the initial unweighted network. These reductions are described in Section 2.4.2. Conditions 1 and 2 contract individual heavy edges and conditions 3 and 4 use the shared neighborhood of the incident vertices to certify whether an edge can be contracted.

We iterate over all edges of G and check conditions 1 and 2. Whenever we encounter an edge (u, v) that satisfies either condition 1 or 2 we mark it as contractible. After finishing the pass, we build the contracted graph. More precisely, we perform contraction in linear time by deleting all unmarked edges, contracting connected components and then re-adding the deleted edges as defined in the contraction process. In practice, we achieve better performance using a union-find data structure [70], which results in a running time of $\mathcal{O}(n\alpha(n) + m)$

It is not possible to perform an exhaustive check for conditions 3 and 4 in all triangles in an arbitrary graph G in linear time, as the graph might have as many as $\Theta(m^{\frac{3}{2}})$ triangles [166]. We therefore perform linear time passes similar to the implementation of Chekuri et al. [37], as discussed in Section 2.4.2.

Final Step: Exact Minimum Cut Algorithm.

To find the minimum cut of the final problem kernel, we use the minimum cut algorithm of Nagamochi, Ono and Ibaraki, as discussed in Section 2.4.1.

Lemma 3.2.2. *The algorithm* `VieCut` *has a running time complexity of* $\mathcal{O}(n + m)$.

Proof. One round of all reduction and contraction steps (Algorithm 1, lines 2-8) can be performed in $\mathcal{O}(n + m)$. The label propagation step contracts the graph by at least a factor of 2, which yields geometrically shrinking graph size and thus a total running time of $\mathcal{O}(n + m)$. We break this loop when the contracted graph has less than some constant n_0 number of vertices. The exact minimum cut of this graph with constant size can therefore be found in constant time. The solution transfer can be performed in linear time by performing the coarsening in reverse and pushing the two cut sides from each graph to the next finer graph.

If the graph is not connected, throughout the algorithm one of the contracted graphs can contain isolated vertices, which our algorithm does not contract. However, when we discover an isolated vertex, there exists a cut of size 0 that separates the connected components. As no cut can be smaller than 0, this cut is minimum and our algorithm terminates and reports it. □

3.2.1 Parallelization

We describe how to parallelize `VieCut`. We parallelize each part of the algorithm, except the final invocation of the algorithm of Nagamochi, Ono, and Ibaraki.

Parallel Label Propagation

To perform the label update for vertex v, we only need to consider vertices in the neighborhood $N(v)$. Therefore the label propagation algorithm can be implemented in parallel on shared-memory machines [175] using the `parallel for` directive from the OpenMP [42] API. We store the cluster affiliation for all vertices in an array of size n, where position i denotes the cluster affiliation of vertex i. We explicitly do not perform label updates in a `critical` section, as each vertex is only traversed once and the race conditions are not critical but instead introduce another source of randomness.

Parallel Correcting Step

As the clusters are independent of each other for this correcting step, we parallelize it on a cluster level, that is, a cluster is checked by a single thread but each thread can check a different cluster without the need for locks or mutexes.

Parallel Graph Contraction

After label propagation has partitioned the graph into c clusters, we build the cluster graph. As the time to build this contracted graph is not negligible, we parallelize graph contraction as well. One of the p threads performs the memory allocations to store the contracted graph, while the other $p - 1$ threads prepare the data for this contracted graph. When $c^2 > n$, we parallelize the graph on a cluster level. To build the contracted vertex for cluster C, we iterate over all outgoing edges $e = (u, v)$ for all vertices $u \in C$. If $v \in C$ then e is an intra-cluster edge and we discard the edge, otherwise we add $c(u, v)$ to the edge weight between C and the cluster of vertex v. When $c^2 < n$, we achieve lower running time and better scaling when using a shared-memory parallel hash table [4, 134]. We generate the contracted graph $G_C = (V_C, E_C)$, in which each block is represented by a single vertex - first we assign each block a vertex ID in the contracted graph in $[0, |V_C|)$. For each edge $e = (u, v)$, we compute a hash of the block IDs of u and v to uniquely identify the edge in E_C. We use this identifier to compute the weights of all edges between blocks. Every thread iterates over a distinct block of edges and we use the parallel hash table to sum up the edge weights between vertices in the contracted graph. If the contracted graph contains two extremely heavy vertices, i.e. two vertices that each encompass at least 20% of the vertices of the original graph, we noticed slowdown due to the many accesses to the same hash table entry. We therefore compute the edge weight between those two blocks separately on each processor and sum up these local values at the end.

Parallel Padberg-Rinaldi Reductions

In parallel, we run the Padberg-Rinaldi reductions on the contracted graph. As these criteria are local and independent, they can be parallelized trivially. We use a parallel wait-free union-find data structure [12] to avoid locking. Reductions 2 and 3 use the weighted vertex degree which changes when edges are contracted. Updating the vertex degrees before performing the actual bulk contraction would entail additional locks. These reductions are therefore only performed on edges where both incident vertices where not yet affected by a contraction. We use a compare-and-swap mechanism to make sure this holds in parallel.

3.2.2 Further Implementation Details

The label propagation algorithm by Raghavan et al. [157] traverses the graph vertices in random order. Other implementations of the algorithm [175] omit this ex-

plicit randomization and rely on implicit randomization through parallelism, as the vertex processing order in parallel label propagation is non-deterministic. Our implementation to find the new label of a vertex v in uses an array, in which we sum up the weights for all clusters in the neighborhood $N(v)$. Therefore randomizing the vertex traversal order would destroy any graph locality, leading to many random reads in the large array, which is very cache inefficient. Thus we trade off randomness and graph locality by randomly shuffling small blocks of vertex ids but traversing each of these shuffled blocks successively.

Using a time-forward processing technique [198] the label propagation as well as the contraction algorithm can be implemented in external memory [3] using Sort($|E|$) I/Os overall. Hence, if we only use the label propagation contraction technique in external memory and use the whole algorithm as soon as the graph fits into internal memory, we directly obtain an external memory algorithm for the minimum cut problem. We do not further investigate this variant of the algorithm as our focus is on fast internal memory algorithms for the problem.

3.3 Random Edge Contraction

We now propose an additional variant of our algorithm, which aims to achieve a lower running time at the expense of a higher error rate. Similar to the algorithm of Karger and Stein [105], we shrink the graph by contracting random edges and then perform the `VieCut` algorithm on the contracted graph.

In contrast to Karger and Stein's original algorithm [105], our implementation of random contraction does not perform the edge contractions independently. Instead, we use a wait-free parallel union-find data structure [12] to mark contracted blocks and perform contractions in bulk, as discussed in the previous section.

In detail, the process works as follows: we draw a random integer $i \in [0, \dots, m)$ and use the union-find data structure to check whether the vertices u and v incident to edge i are in the same block. If they are not, we unite the blocks containing u and v and decrement the number of blocks. We repeat this process until the number of blocks is smaller than the number of vertices multiplied by a given contraction factor $\alpha \in (0, 1)$. We then perform all contractions in a single operation, similar to the contraction in Section 3.2. This implementation of the random contraction algorithm for the minimum cut problem was first employed by Chekuri et al. [37].

For edge-weighted graphs we draw each edge with probability proportional to its weight. To do this efficiently, we build the prefix sum of all edge weights. This prefix sum p_e of an edge e is defined as the weight of all previous edges as given by the edge order in the graph data structure, more formally defined in Equation 3.1.

$$p_e = \sum_{i=0}^{e-1} c(i) \tag{3.1}$$

In weighted graphs we can then draw edges from the range $i \in [0, \dots, \sum_{e \in E} c(e))$. If $p_j < i \leq p_{j+1}$, we contract edge j similar to the unweighted case. This can be implemented in $\mathcal{O}(\log n)$ time using binary search on the array of the prefix sums.

We also tested other techniques to achieve a speedup by contracting the graph: in expectation, random edge sampling approximately preserves the minimum cut with high probability [99]. However, in order to actually achieve a speedup, we need very low sampling rates and the approximation factor deteriorates both in theory and practice. Removing high-degree vertices and their incident edges often disconnects the graph. Greedily re-adding the removed vertices to the partition with stronger connection does not result in cuts with low weight. Hence, we omit further investigation of those techniques here.

3.4 Experiments

In this section we compare our algorithm `VieCut` with existing algorithms for the minimum cut problem on real-world and synthetic graphs. We compare the sequential variant of our algorithm to efficient implementations of existing algorithms and show how our algorithm scales on a shared-memory machine.

3.4.1 Experimental Setup and Methodology

We implemented the algorithms using C++-17. Our experiments are conducted on two machines: Machine A, which is used for nearly all experiments, has two Intel Xeon E5-2643 v4 with 3.4GHz with 6 CPU cores each and 1.5 TB RAM in total. On this machine we compiled our code using g++-7.1.0 with full optimization (`-O3`). Machine B contains 4 Intel Xeon E7-8677 v3 with 2.5GHz with 16 cores each. It has 1TB of RAM in total. This machine is used for the parallel experiments in Section 3.4.6 with up to 128 threads. On this machine, we compiled all code with g++-6.3.0 with full optimization (`-O3`). In general, we perform five repetitions per instance and report the average running time as well as the cut size.

3.4.2 Algorithms

We compare our algorithm with our implementations of the algorithm of Nagamochi, Ono and Ibaraki (`NOI`) [147] and the $(2 + \varepsilon)$-approximation algorithm of Matula (`Matula`) [136]. In addition, we compare against the preflow-based algorithm of Hao and Orlin (`HO`) [81] by using the implementation of Chekuri et al. [37]. We also performed experiments with Chekuri et al.'s implementations of `NOI`, but our implementation is generally faster. For `HO`, Chekuri et al. give variants with and without Padberg-Rinaldi tests and with an excess detection heuristic [37], which contracts nodes with large preflow excess. We use three variants of the algorithm of Hao and Orlin in our experiments: `HO_A` uses Padberg-Rinaldi tests, `HO_B` uses excess detection and `HO_C` uses both. We also use their implementation of the algorithm of Karger and Stein [37, 105, 177] (`KS`) without Padberg-Rinaldi tests.

The variant of Karger-Stein with Padberg-Rinaldi tests decomposed most graphs in preprocessing with repeated Padberg-Rinaldi tests. It therefore performed very similar to HO_A and HO_C and was omitted. We only perform a single iteration of the Karger-Stein algorithm, as this is already slower than all other algorithms. Note that performing more iterations yields a smaller error probability, but also makes the algorithm even slower. The implementation crashes on very large instances due to overflows in the graph data structure used for edge contractions. We do not include the algorithm by Stoer and Wagner [178], as it is far slower than NOI and HO in the experiments of Chekuri et al. [37] and Jünger et al. [100] and was also slower in preliminary experiments we conducted. We also do not include the near-linear algorithm of Henzinger et al. [95], as the other algorithms are quasi linear in most instances examined and the algorithm of Henzinger et al. has large constant factors in the running time. We performed, however, preliminary experiments with the core of the algorithm, which indicate that the algorithm is slower in practice. We also performed preliminary experiments with an ILP formulation using Gurobi 8.0.0. On an RHG graph with $n = 2^{15}$ and an average density of 2^5 that was solved exactly in 0.04 seconds using HO_A, the ILP was solved in 3 500 seconds. We therefore did not further investigate using ILP formulations to solve the minimum cut problem. Finally, we note that the MPI-parallel implementation of KS by Gianinazzi et al. [75] finds the minimum cut of RMAT graphs with n =16 000 and an average degree of 4 000 in 5 seconds using 1 536 cores [75]. This is significantly slower than our `VieCut` algorithm, which finds the minimum cut on a similar-sized RMAT graph [113] in 0.2 seconds using just 24 threads. Given this stark difference in running time, we exclude their algorithm from our experiments.

3.4.3 Instances

We perform experiments on clustered Erdős-Rényi graphs that are generated using the generator from Chekuri et al. [37], which are commonly used in the literature [37, 100, 147, 152]. We also perform experiments on random hyperbolic graphs [119, 131] and on large undirected real-world graphs taken from the 10th DIMACS Implementation Challenge [15] and from the Laboratory for Web Algorithmics [24, 25]. As these graphs contain vertices with low degree (and therefore trivial cuts), we use the k-core decomposition [18], which gives the largest subgraph, in which each vertex has a degree of at least k, to generate input graphs. We use the largest connected components of these core graphs to generate graphs in which the minimum cut is not trivial. For every real-world graph, we use k-cores for four different values of k. In Section 2.5 we show the instances in further detail and in Table 2.1 (Graph Family A) we give sizes and cut values for each instance used.

The graphs used in our experiments have up to 70 million vertices (`uk-2007-05`, $k = 10$) and up to 5 billion edges (Clustered Erdős-Rényi, $n = 100$K, $d = 100\%$). To the best of our knowledge, these graphs are the largest instances reported in literature to be used for experiments on global minimum cuts.

	VCut1	VCut2	VCut3	VCut5	VCut10	VCut25
# of non optimal cuts	29	14	15	19	19	18
average dist. to opt.	16.2%	2.44%	2.46%	3.80%	3.37%	3.14%

Table 3.1: Error rate for configurations of VieCut in RHG graphs (out of 300 instances). The number in configuration name indicates the number of iterations in the label propagation step.

Clustered Erdős-Rényi Graphs

Many prior experimental studies of minimum cut algorithms used a family of clustered Erdős-Rényi graphs with $m = O(n^2)$ [37, 100, 147, 152]. This family of graphs is specified by the following parameters: number of vertices $n = |V|$, d the graph density as a percentage where $m = |E| = \frac{n \cdot (n-1)}{2} \cdot \frac{d}{100}$ and the number of clusters k. For each edge (u, v), the integral edge weight $c(u, v)$ is generated independently and uniformly in the interval $[1, 100]$. When the vertices u and v are in the same cluster, the edge weight is multiplied by n, resulting in edge weights in the interval $[n, 100n]$. Therefore the minimum cut can be found between two clusters with high probability. We performed three experiments on this family of graphs. In each of these experiments we varied one of the graph parameters and fixed the other two parameters. These experiments are similar to older experiments [37, 100, 147, 152] but scaled to larger graphs to account for improvements in machine hardware. We use the generator *noigen* of Andrew Goldberg [177] to generate the clustered Erdős-Rényi graphs for these experiments. This generator was also used in the study conducted by Chekuri et al. [37]. As our code uses the METIS [109] graph format, we use a script to translate the graph format. All experiments exclude I/O times.

Random Hyperbolic Graphs (RHG) [119]

Random hyperbolic graphs replicate many features of real-world networks [34]: the degree distribution follows a power law, they often exhibit a community structure and have a small diameter. In denser hyperbolic graphs, the minimum cut is often equal to the minimum degree, which results in a trivial minimum cut. In order to prevent trivial minimum cuts, we use a power law exponent of 5. We use the generator of von Looz et al. [131], which is a part of NetworKit [176], to generate unweighted random hyperbolic graphs with 2^{20} to 2^{25} vertices and an average vertex degree of 2^5 to 2^8. These graphs generally have very few small cuts and in most instances there is only one unique minimum cut. Removal of the minimum cut partitions the set of nodes into two sets of similar size.

3.4.4 Configuring the Algorithm

We performed experiments to tune the number of label propagation iterations and to find an appropriate amount of randomness for our algorithm. We con-

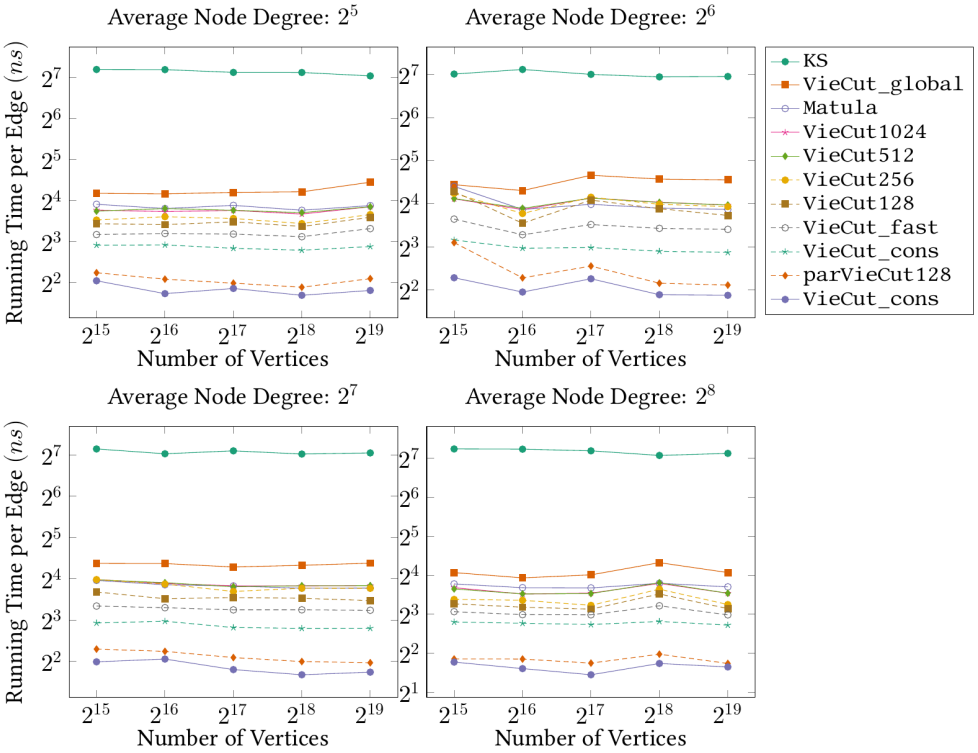


Figure 3.2: Total running time in nanoseconds per edge in small RHG graphs for different configurations of `VieCut`

ducted these experiments with different configurations on generated hyperbolic graphs (see Section 3.4.3) with 2^{15} to 2^{19} vertices with an average degree of 2^5 to 2^8 and compared error rate and running time. The instances used here are different to the ones used in later sections.

Table 3.1 shows the number of non-optimal cuts returned by `VieCut` with different numbers of label propagation iterations indicated by the integer in the name. Each implementation traverses the graph in blocks of 256 randomly shuffled elements as described in Section 3.2.2. The variant `VieCut25` performs up to 25 iterations or until the label propagation converges so that only up to $\frac{1}{10000}$ of all nodes change their cluster. On average the variant performed 20.4 iterations. The results for all variants with 2 to 25 iterations are very similar with 14 to 19 non-optimal results and 2.44% and 3.80% average distance to the optimum. As the largest part of the total running time is in the label propagation step, running the algorithm with a lower amount of iterations is obviously faster. Therefore we use 2 iterations of label propagation in all following experiments.

To compare the effect of graph traversal strategies, we compared different configurations of our algorithm. `VieCut_cons` does not randomize the traversal order, i.e. it traverses vertices consecutively by ID, `VieCut_global` performs global

shuffling, VieCut_fast swaps each vertex with a random vertex with a index distance up to 20. The configurations VieCut128, VieCut256, VieCut512, VieCut1024 randomly shuffle blocks of 128, 256, 512, or 1024 vertices and introduce randomness without losing too much data locality. We also include the configurations parVieCut_cons and parVieCut128, which are shared-memory parallel implementation with 12 threads. As a comparison, we also include the approximation algorithm of Matula and a single run of the randomized algorithm of Karger and Stein.

Figure 3.2 shows the total running time for different configurations of VieCut. From the sequential algorithms, VieCut_cons has the lowest running time for all algorithms. The algorithm, however, returns non-optimal cuts in more than $\frac{1}{3}$ of all instances, with an average distance to the minimum cut of 44% over all graphs. The best results were obtained by VieCut128, which has an average distance of 0.83% and only 10 non-optimal results out of 300 instances. The results are very good compared to Matula, which has 57 non-optimal results in these 300 instances and an average distance of 5.57%. VieCut128 is 20% faster on most graphs than Matula, regardless of graph size or density. In the following we use the configuration VieCut128 with 2 iterations, there named VieCut. On these small graphs, the parallel versions have a speedup factor of 2 to 3.5 compared to their sequential version. parVieCut128 has 17 non-optimal results and an average distance of 4.91% while parVieCut_cons has 29 non-optimal results and 20% average distance to the minimum cut. Therefore we use parVieCut128 for all parallel experiments (named parVieCut). We set the bound n_0 to 10 000 and did not encounter a single instance with more than a single bulk contraction step.

3.4.5 Experimental Results

Clustered Erdős-Rényi Graphs

Clustered Erdős-Rényi graphs have distinct small cuts between the clusters and do not have any other small cuts. We perform three experiments varying one parameter of the graph class and use default parameters for the other two parameters. Our default parameters are $n = 100\,000$, $d = 10\%$ and $k = 2$. The code of Chekuri et al. [37] uses 32 bit integers to store vertices and edges. We could therefore not perform the experiments with $m \geq 2^{31}$ with HO. Figure 3.3 shows the results for these experiments. First of all, on 20% of the instances KS returns non-optimal results. No other algorithm returned any non-optimal minimum cuts on any graph of this dataset. Moreover, seqVieCut is the fastest algorithm on all of these instances, followed by Matula, which is 40% to 100% slower on these instances.

Our algorithm seqVieCut is faster on graphs with a lower number of vertices, as the array containing cluster affiliations – which has one entry per vertex and is accessed for each edge – fits into cache. In graphs with $k = 2, 4, 8$, the final number of clusters in the label propagation algorithm is equal to k, as label propagation correctly identifies the clusters. In the graph contraction step, we iterate over all edges and check whether the incident vertices are in different clusters. For this

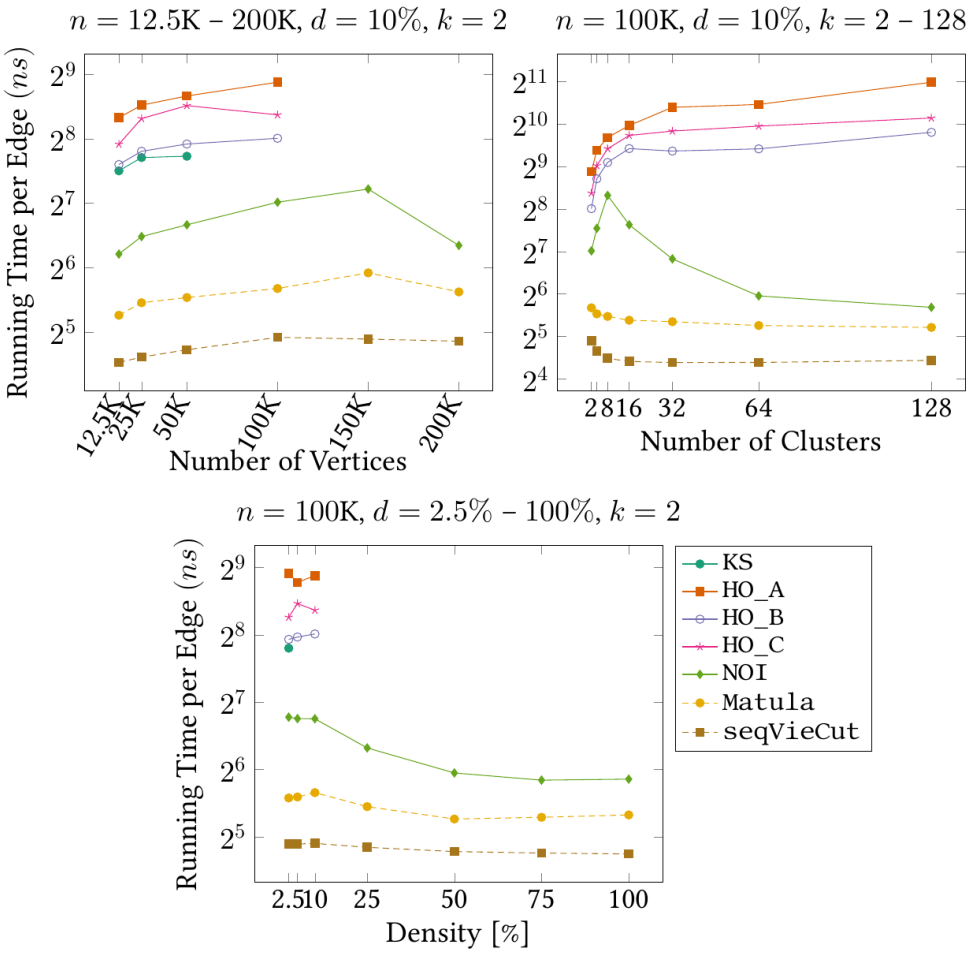


Figure 3.3: Total running time in nanoseconds per edge in clustered Erdős-Rényi graphs

branch, the compiler assumes that they are indeed in different cluster. However, in these graphs, the chance for any two adjacent nodes being in the same cluster is $\frac{1}{k}$, which is far from zero. This results in a large amount of branch misses (for $n = 100\,000$, $d = 10\%$, $k = 2$: average 14% branch misses, in total 1.5 billion missed branches). Thus the performance is better with higher values of k. The fastest exact algorithm is NOI. This matches the experimental results obtained by Chekuri et al. [37] on graphs generated with the same instance generator.

Random Hyperbolic Graphs

We also performed experiments on random hyperbolic graphs with $n = 2^{20} - 2^{25}$ and an average degree of $2^5 - 2^8$. We generated 3 graphs for each of the 24 possible combinations of n and average degree yielding a total of 72 RHG graphs. Note that these graphs are hard instances for the inexact algorithms, as they contain few –

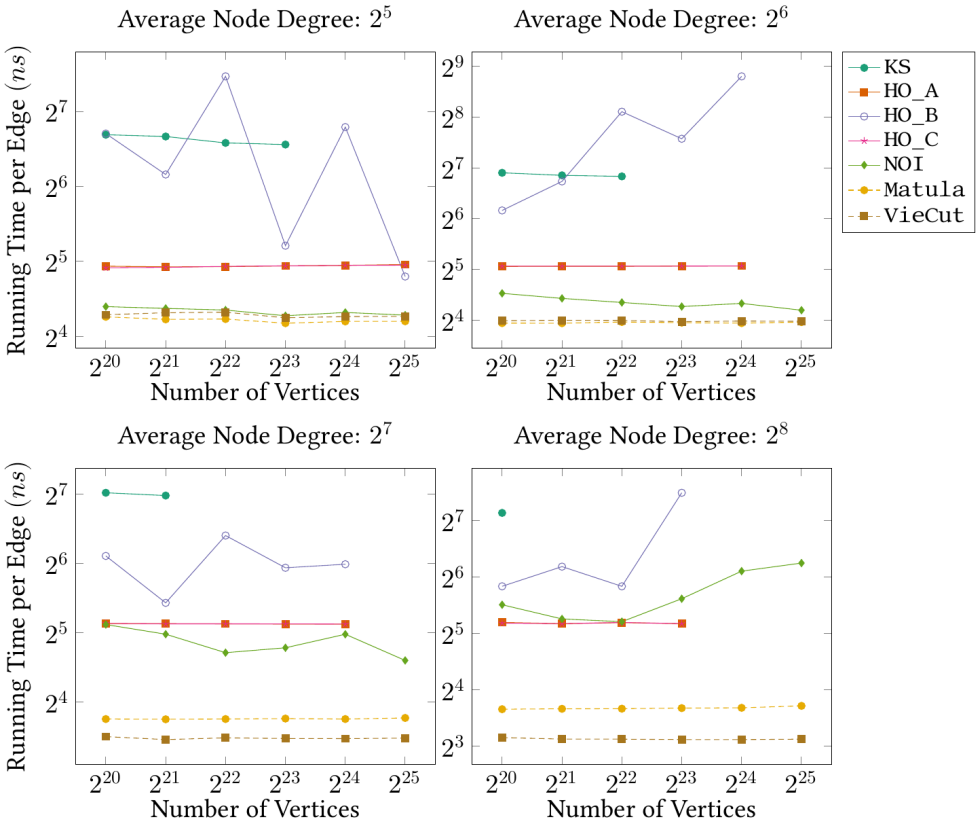


Figure 3.4: Total running time in nanoseconds per edge in RHG graphs

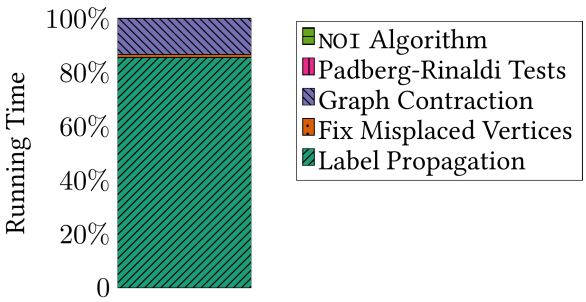


Figure 3.5: Running Time Breakdown for RHG Graphs with $n = 2^{25}$ and $m = 2^{32}$

usually only one – small cuts and both sides of the cut are large. From a total of 360 runs, `seqVieCut` does not return the correct minimum cut in 1% of runs and `Matula` does not return the correct minimum cut in 31% of runs. KS, which crashes on large instances, returns non-optimal cuts in 52% of the runs where it ran to completion.

Figure 3.4 shows the results for these experiments. On nearly all of these graphs, NOI is faster than HO. On sparse graphs with an average degree of 2^5, `seqVieCut`,

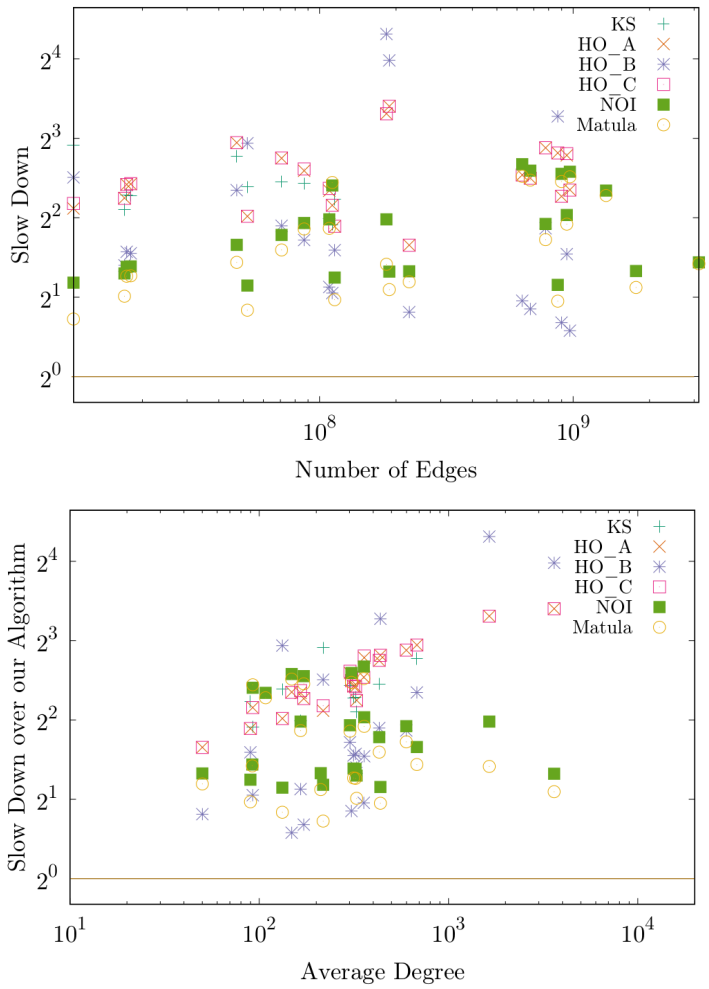


Figure 3.6: Slowdowns of competitors to `VieCut` in large real-world graphs. We display slowdowns based on the absolute number of edges (top), and by the average vertex degree (bottom) in the graph

`Matula` and `NOI` nearly have equal running time. On denser graphs with an average degree of 2^8, `seqVieCut` is 40% faster than `Matula` and 4 to 10 times faster than `NOI`. `HO_A` and `HO_C` use preprocessing with the Padberg-Rinaldi heuristics. Multiple iterations of this preprocessing contract the RHG graph into two nodes. The running time of those algorithms is 50% higher on sparse graphs and 4 times higher on dense graphs compared to `seqVieCut`. Figure 3.5 shows a time breakdown for `seqVieCut` on large RHG graphs with $n = 2^{25}$. Around 85% of the running time is in the label propagation step and the rest is mostly spent in graph contraction. The correcting step has low running time on most graphs, as it is not performed on large clusters.

Real-World Graphs

The third set of graphs we use in our experiments are k-cores of large real-world social and web graphs. On these graphs, no non-optimal minimum cuts were returned by any algorithm except for KS, which gave 36% non-optimal results. However, as most of these graph instances have multiple minimum cuts, even exact algorithms usually output different cuts on multiple runs. Figure 3.6 gives slowdown plots to the fastest algorithm (seqVieCut in each case) for the real-world graphs. On these graphs, seqVieCut is the fastest algorithm, far faster than the other algorithms. Matula is not much faster than NOI, as most of the running time is in the first iteration of their CAPFOREST algorithm, which is similar for both algorithms. On the largest real-world graphs, seqVieCut is approximately 3 times faster than the next fastest algorithm Matula. We also see that seqVieCut, Matula and NOI all perform better on denser graphs. For Matula and NOI, this can most likely be explained by the smaller vertex priority queue. For seqVieCut, this is mainly due to better cache locality. As HO does not benefit from denser graphs, it has high slow down on dense graphs.

The highest speedup in our experiments is in the 10-core of gsh-2015-host, where seqVieCut is faster than the next fastest algorithm (Matula) by a factor of 4.85. The lowest speedup is in the 25-core of twitter-2010, where seqVieCut is 50% faster than the next fastest algorithm (HO_B). The average speedup factor of seqVieCut to the next fastest algorithm is 2.37. NOI and Matula perform badly on the cores of the graph twitter-2010. This graph has a very low diameter (average distance on the original graph is 4.46), and as a consequence the priority queue used in these algorithms is filled far quicker than in graphs with higher diameter. Therefore the priority queue operations become slow and the total running time is very high.

To summarize, both in generated and real-world graphs, even in sequential runs seqVieCut is up to a factor of 6 faster than the state of the art, while achieving a high solution quality even for hard instances such as the hyperbolic graphs. The performance of seqVieCut is especially good on the real-world graphs, presumably as these graphs have high locality.

Shared-Memory Parallelism

Figure 3.7 shows the speedup of parVieCut compared to the sequential variant and to the next fastest algorithm, which is Matula in all of the large graph examined. We examine the largest graphs from each of the three graph classes and perform parallel runs using $1, 2, 3, \ldots, 12$ threads. We also perform experiments with 24 threads, as the machine has 12 cores and supports multi-threading. The harmonic mean of the speedup of parVieCut on large graphs with 12 threads is 5.01. (24 threads: 5.5) and all runs computed the exact minimum cut. Compared to the next fastest sequential algorithm Matula, this is an average harmonic speedup factor of 9.5 (24 threads: 11.1). parVieCut scales especially well on the clustered Erdős-

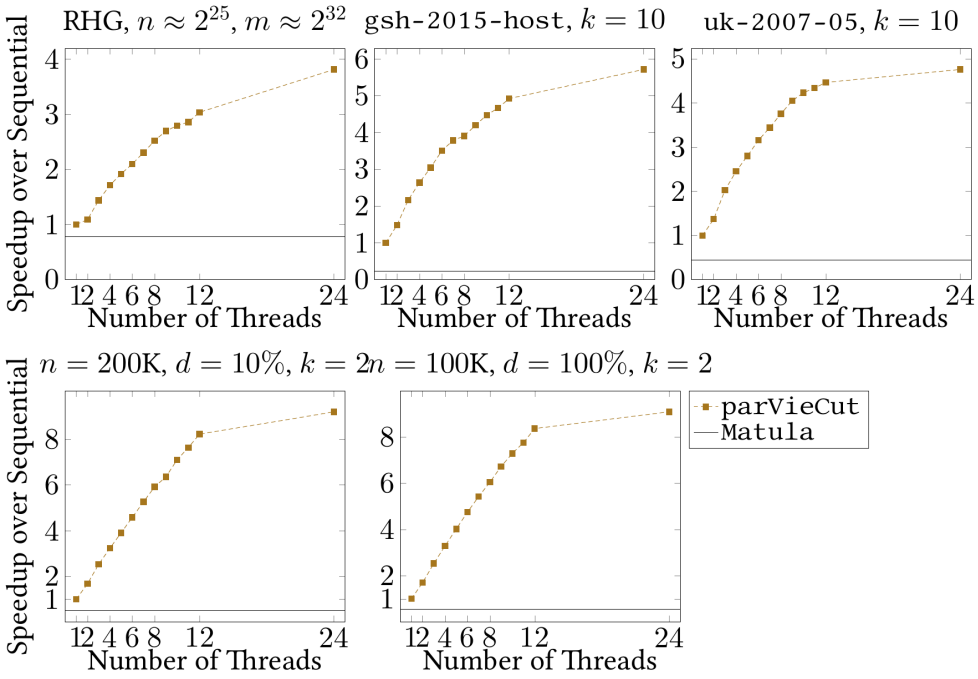


Figure 3.7: Speedup on large graphs over VieCut using 1 thread.

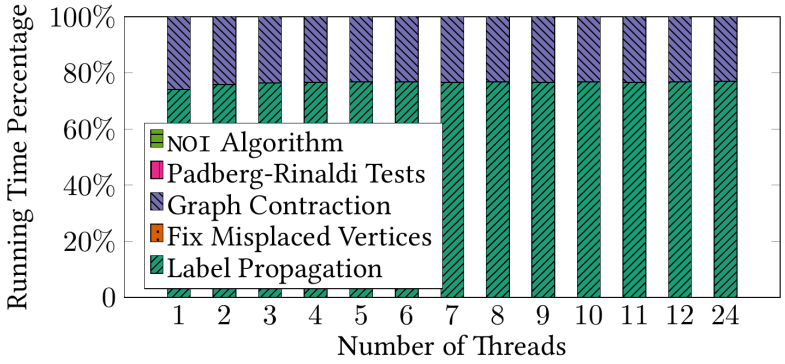


Figure 3.8: Parallel Running Time Breakdown

Rényi graphs, presumably as these dense graphs contain many high-degree vertices and have a rather low number of vertices. Figure 3.8 shows average running time breakdowns averaged over all graphs. For this figure, the correcting algorithm is turned off for the two Erdős-Rényi graphs. With one thread, label propagation uses 74% of the total running time and with 24 threads, 77% of the total running time. Thus the different parts of the algorithm parallelize equally well.

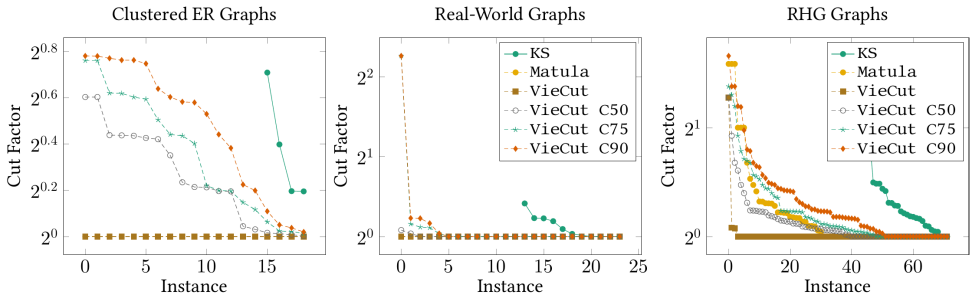


Figure 3.9: Average factor of result to minimum cut for inexact algorithms

3.4.6 Random Edge Contraction

We now evaluate the variant of our algorithm that uses random edge contractions similar to the algorithm of Karger and Stein before running VieCut on the contracted graph. The edge contractions promise faster results but increase the error rate of the algorithm. This section shows experiments which detail error rate, error severity and running times for the heuristic and approximation algorithms. We repeat the experiments of the previous section, but now with only inexact algorithms. In addition to (the one iteration-only version of) KS, Matula and VieCut, we add VieCut C50, VieCut C75, VieCut C90, which contract 50, 75 and 90% of all vertices before running VieCut.

Clustered Erdős-Rényi Graphs

Figure 3.10 shows the running time for the random edge contraction algorithm variants compared to VieCut on dense clustered Erdős-Rényi graphs. We can see that VieCut C75 and VieCut C90 are always faster than VieCut, with VieCut C90 being faster by a factor of 2.3 to 2.7 than VieCut and VieCut C75 being faster by a factor of 1.25 to 1.8.

Figure 3.9 (left) shows the average distance to the optimal cut for all algorithms which do not guarantee optimality, both as the difference and the factor of the returned cut to the optimal. In these highly regular graphs, we find the optimal cut if none of the low-weight edges between the clusters is contracted. Otherwise we find a cut where one side is only a single vertex. On average, this cut is around twice the value of the minimum cut. The algorithms all have an average cut factor of up to 1.7 on all graphs, depending on how many vertices we contract. KS has a similar error rate on the graphs where it finishes. Both VieCut and Matula have no errors on these graphs.

Real-World Graphs

Figure 3.11 shows the average running time for the random edge contraction variants of VieCut on real-world graphs. VieCut C90 is faster than VieCut by a factor

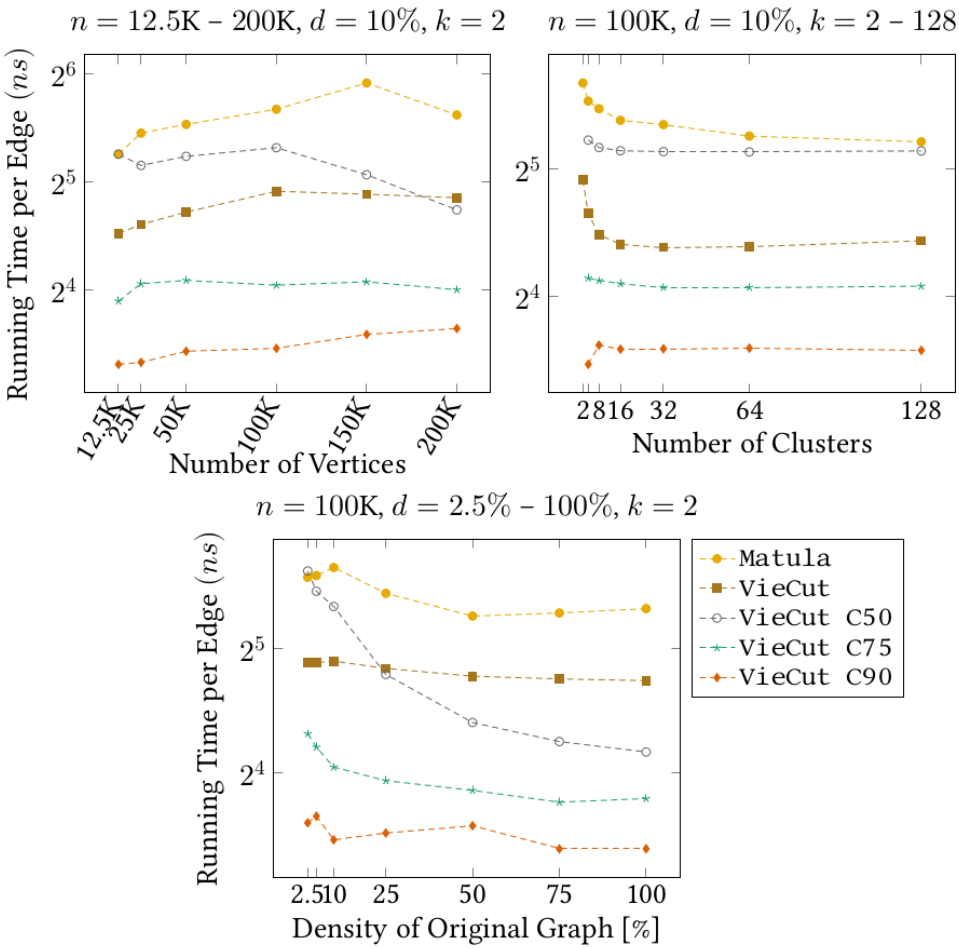


Figure 3.10: Total running time in nanoseconds per edge in clustered Erdős-Rényi graphs

of up to 3.40. The lowest speedup factor is 0.80 (a slowdown of 25%). `VieCut C75` has speedup factors between 1.12 to 2.35 compared to `VieCut`.

Figure 3.9 (middle) shows the average error rates. On these graphs, the average ratio of cut size to the optimal cut size is very low for the random contraction algorithms. The outlier for `VieCut C75` and `VieCut C90` is a single run of the graph `twitter-2010` with $k = 60$, where one of the edges in the unique small cut (of value 3) is contracted. The next smallest cut in the graph is a trivial cut with a cut value of 60, which is found. The algorithm finds optimal results in the other four iterations of this graph. On all other graphs, the cut factor is below 1.18. KS has an average cut rate of up to 1.33. `VieCut` and `Matula` have no errors on these graphs.

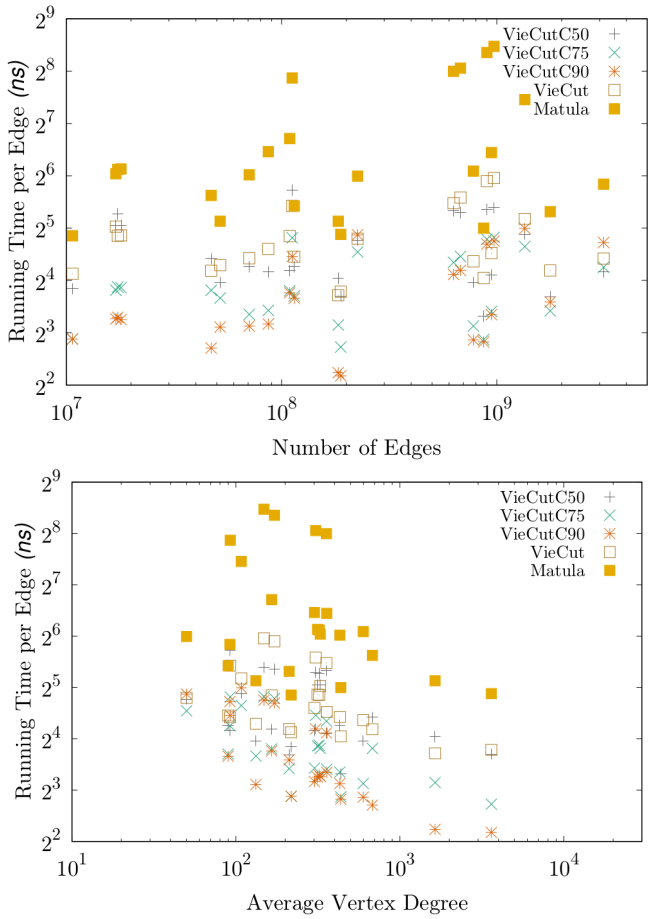


Figure 3.11: Running time per edge in nanoseconds on real-world graphs. We display running times based on the absolute number of edges (top), and by the average vertex degree (bottom) in the graph

Random Hyperbolic Graphs

Figure 3.12 shows the running time of the algorithms on random hyperbolic graphs. `VieCut C75` has a speedup of 1.05 to 1.67 compared to `VieCut` and `VieCut C90` has a speedup of 1.15 to 2.54.

Figure 3.9 (right) shows the error rate for RHG graphs. The error rate of `VieCut` is far lower than all other non-exact algorithms. `Matula` has a non-optimality factor that is between `VieCut C50` and `VieCut C75` but a smaller distance to the optimal solution than both of them. However, `VieCut C75` is much faster than `Matula`, which is consistently slower than `VieCut`. Note that the 3 graphs in which `Matula` has a cut 3 times as large as optimal are graphs with a minimum cut of 1, where `Matula` consistently returns cuts of value 3. Our implementation of `Matula` contracts all edges in the spanning forest with index $\lfloor \frac{\hat{\lambda}}{2} \rfloor$.

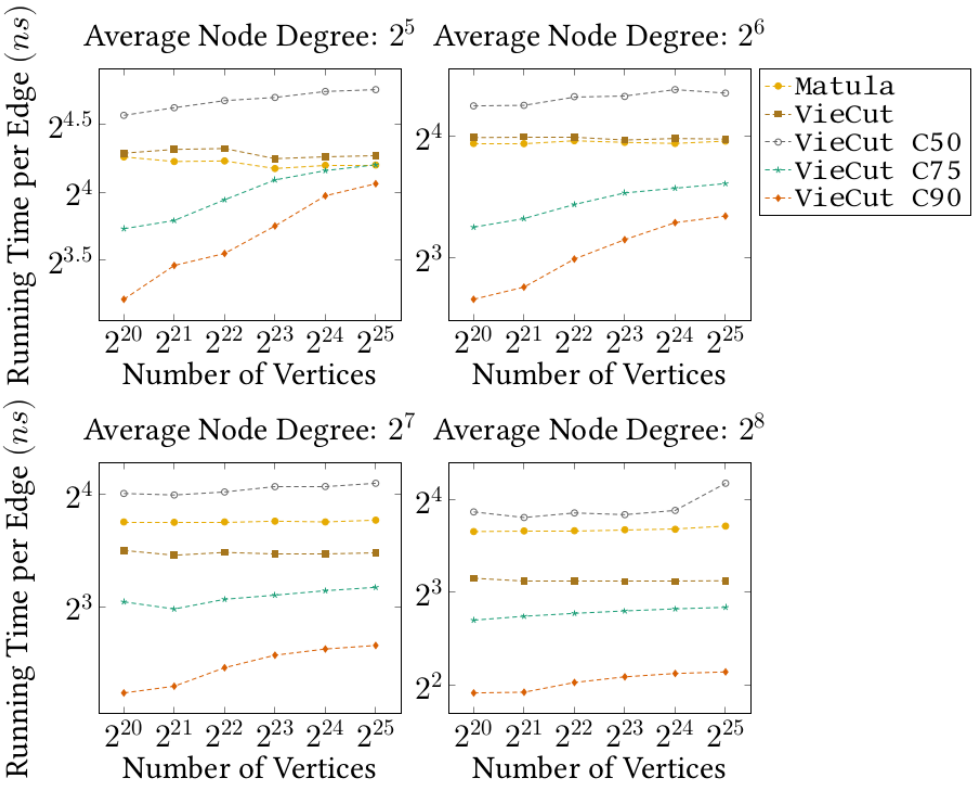


Figure 3.12: Total running time in nanoseconds per edge in RHG graphs

Shared-Memory Parallelism

Figure 3.13 shows the speedup of the random contraction variants in comparison to VieCut. The RHG graph has a single smallest cut with value 73, followed by trivial cuts with a degree of 139 each. VieCut and Matula return the correct minimum cut each run, VieCut C50 34 out of 40 times, VieCut C75 28 times, VieCut C90 24 times and 139 otherwise. Only VieCut C90 scales better than VieCut and has a speedup of up to 13.4 compared to sequential VieCut. Due to the large number of clusters, where each cluster has many incident edges, the contraction step in VieCut C75 and VieCut C50 takes a long time.

The random contraction variants scale better in the large real-world graphs, where all algorithms return the minimum cut in all runs. On graph gsh-2015-host, VieCut C90 has a speedup of over 21, while VieCut C75 has a speedup of 16 and VieCut has a speedup of 9.5 with 32 threads. On uk-2007-05, VieCut C90 has a speedup of over 12, VieCut C75 of over 8.5 and VieCut a speedup of up to 5.7.

On the clustered Erdős-Rényi graphs, the random edge contraction creates one or few very large blocks of vertices. Again, VieCut and Matula always return the correct minimum cut. Out of 80 runs on clustered Erdős-Rényi graphs, VieCut C50

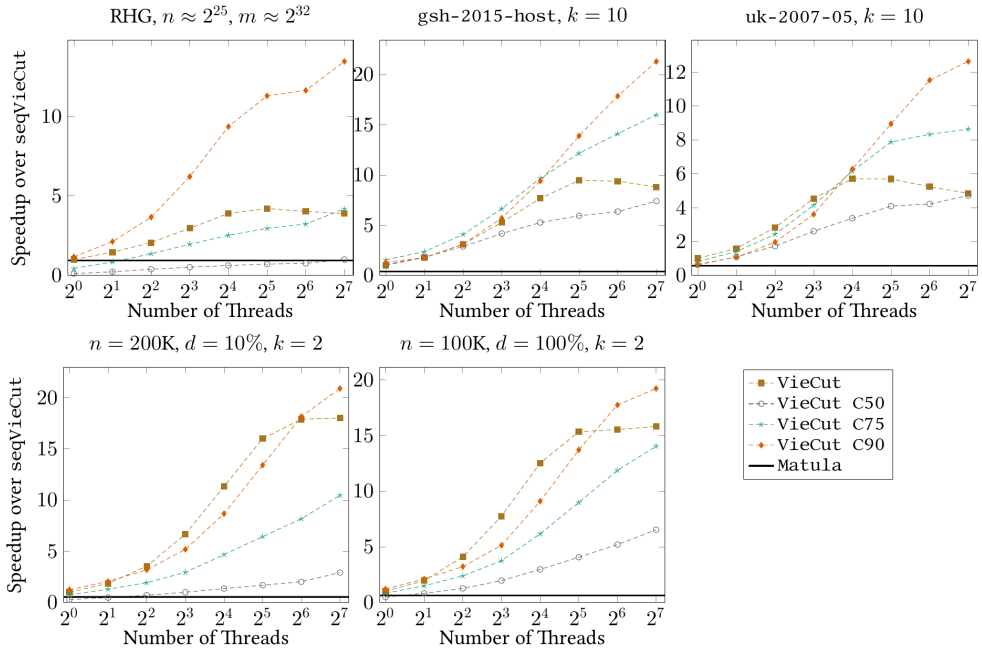


Figure 3.13: Speedup on large graphs over VieCut using 1 thread.

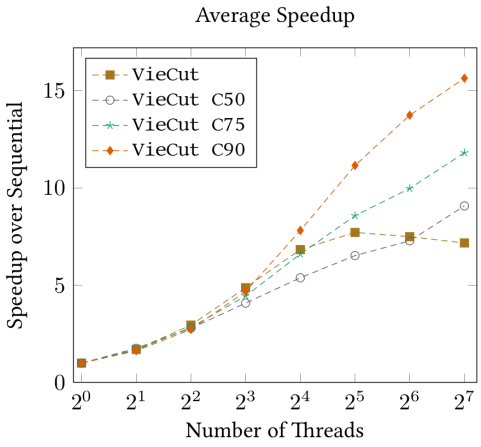


Figure 3.14: Average (harmonic) speedup relative to algorithm performance using 1 thread.

returns the minimum cut in 46 cases, VieCut C75 in 8 cases and VieCut C90 in 12 cases. On these very dense and unstructured graphs, random contraction has a high error rate and does not significantly speed up VieCut. VieCut however has a speedup of up to 18 on these graphs. Figure 3.14 shows the average speedup of the algorithms compared to their performance with one thread on machine B.

In summary, the random contraction variants can improve the running time of VieCut even further, especially when we have many threads. However, the random contraction increases the error rate of the algorithm by a large margin and should therefore only be used if running time is more important than absolute solution quality.

3.5 Conclusion

We presented the linear-time heuristic algorithm VieCut for the minimum cut problem. VieCut is based on the label propagation algorithm [157] and the Padberg-Rinaldi heuristics [152]. Both for real-world graphs and a varied family of generated graphs, VieCut is significantly faster than the state of the art. The algorithm has far higher solution quality than other heuristic algorithms while also being faster. Additionally, we propose a variant of our algorithm to further speed up computations at the expense of higher error rates. Important future work includes checking whether using different clustering techniques affect the observed error probability. However, these clustering algorithms generally have higher running time.

CHAPTER 4

Exact Global Minimum Cut

In the previous chapter, we introduced a heuristic shared-memory parallel algorithm for the global minimum cut problem called `VieCut`. In this chapter, we combine techniques from that algorithm, the algorithm of Padberg and Rinaldi [152] and the algorithm of Nagamochi, Ono and Ibaraki [143, 147], as introduced in Section 2.4, to engineer an exact shared-memory parallel algorithm for the minimum cut problem. Our algorithm achieves improvements in running time over existing exact algorithms by a multitude of techniques. First, we use our fast and parallel *inexact* minimum cut algorithm `VieCut` to obtain a better bound for the problem. Afterwards, we use reductions that depend on this bound to reduce the size of the graph much faster than previously possible. We use improved data structures to further lower the running time of our algorithm. Additionally, we parallelize the contraction routines of Nagamochi et al. [143, 147]. Overall, we arrive at a system that outperforms the state-of-the-art by a factor of up to 2.5 sequentially, and when run in shared-memory parallel, by a factor of up to 12.9 using 12 cores.

The content of this chapter is based on [90].

In the following sections we detail our *exact shared-memory parallel algorithm* for the minimum cut problem that is based on the algorithms of Nagamochi et al., as described in Section 2.4.1 and the `VieCut` algorithm described in Chapter 3 of this thesis. We aim to modify the algorithm of Nagamochi et al. in order to find exact minimum cuts faster and in parallel.

We start this chapter with optimizations to the sequential algorithm of Nagamochi et al. First we show how to save work by first performing the inexact `VieCut` algorithm to lower the minimum cut upper bound $\hat{\lambda}$. As shown in Chapter 3, `VieCut` often already finds a cut of value λ. We then give different implementations of the priority queue $\mathcal{Q}$ and detail the effects of the choice of queue on the algorithm. We show that the algorithm remains correct, even if we limit the priorities in the queue to $\hat{\lambda}$, meaning that elements in the queue having a key larger than that will not be

updated. This significantly lowers the number of priority queue operations necessary. Then we adapt the algorithm so that we are able to detect contractible edges in parallel efficiently. In Section 4.3, we put everything together and present a full system description. We then give experimental setup and results of our work in Section 4.4 before we briefly conclude this chapter in Section 4.5.

4.1 Sequential Optimizations

4.1.1 Lowering the Upper Bound $\hat{\lambda}$

The upper bound $\hat{\lambda}$ for the minimum cut is an important parameter for contraction based minimum cut algorithms. For example, the algorithm of Nagamochi et al. [147] computes a lower bound for the connectivity of the two incident vertices of each edge and contracts all edges whose incident vertices have a connectivity of at least $\hat{\lambda}$. Thus, it is possible to contract more edges if we manage to lower $\hat{\lambda}$ beforehand.

A trivial upper bound $\hat{\lambda}$ for the minimum cut is the minimum vertex degree, as it represents the trivial cut which separates the minimum degree vertex from all other vertices. We run `VieCut` to lower $\hat{\lambda}$ in order to allow us to find more edges to contract. Although `VieCut` is an *inexact algorithm*, in most cases it already finds the minimum cut [94] of the graph. As there are by definition no cuts smaller than the minimum cut, the result of `VieCut` is guaranteed to be at least as large as the minimum cut λ. We set $\hat{\lambda}$ to the result of `VieCut` when running the CAPFOREST routine and can therefore guarantee a correct result.

A similar idea is employed by the linear time $(2+\epsilon)$-approximation algorithm of Matula [136], which initializes the algorithm of Nagamochi et al. [147] with $\hat{\lambda} = (\frac{1}{2} - \epsilon)\cdot$min degree. The algorithm of Matula does not guarantee optimality, as this value can be smaller than the minimum cut.

4.1.2 Bounded Priority Queues

Whenever we visit a vertex in the CAPFOREST algorithm, we update the priority of all of its neighbors in $\mathcal{Q}$ by adding the respective edge weight. Thus we perform a total of $|E|$ priority queue increase-weight operations in one call of the CAPFOREST algorithm. In practice, many vertices reach priority values much higher than $\hat{\lambda}$ and perform many priority increases until they reach their final value. We limit the values in the priority queue by $\hat{\lambda}$, i.e. we do not update priorities that are already $\hat{\lambda}$. Lemma 4.1.1 shows that this does not affect correctness of the algorithm.

Let $\tilde{q}_G(e)$ be the value $q(e)$ assigned to e in the modified algorithm on graph G and let $\tilde{r}_G(x)$ be the r-value of a node x in the modified algorithm on G.

Lemma 4.1.1. *Limiting the values in the priority queue $\mathcal{Q}$ used in the CAPFOREST routine to a maximum of $\hat{\lambda}$ does not interfere with the correctness of the algorithm. For*

every edge $e = (v, w)$ with $\tilde{q}_G(e) \geq \hat{\lambda}$, it holds that $\lambda(G, e) \geq \hat{\lambda}$. Therefore the edge can be contracted.

Proof. As we limit the priority queue $\mathcal{Q}$ to a maximum value of $\hat{\lambda}$, we cannot guarantee that we always pop the element with highest value $r(v)$ if there are multiple elements that have values $r(v) \geq \hat{\lambda}$ in $\mathcal{Q}$. However, we know that the vertex x that is popped from $\mathcal{Q}$ is either maximal or has $r(x) \geq \hat{\lambda}$.

We prove Lemma 4.1.1 by creating a graph $G' = (V, E, c')$ by lowering edge weights (possibly to 0, effectively removing the edge) while running the algorithm, so that CAPFOREST on G' visits vertices in the same order (assuming equal tie breaking) and assigns the same q values as the modified algorithm on G.

We first describe the construction of G'. We initialize the weight of all edges in graph G' with the weight of the respective edge in G and run CAPFOREST on G'. Whenever we check an edge $e = (x, y)$ and update a value $r_{G'}(y)$, we check whether we would set $r_{G'}(y) > \hat{\lambda}$. If this is the case, i.e. when $r_{G'}(y)+c(e) > \hat{\lambda}$, we set $c'(e)$ in G' to $c(e) - (r_{G'}(y) - \hat{\lambda})$, which is lower by exactly the value by which $r_G(y)$ is larger than $\hat{\lambda}$, and non-negative. Thus, $r_{G'}(y) = \hat{\lambda}$. As we scan every edge exactly once in a run of CAPFOREST, the weights of edges already scanned remain constant afterwards. This completes the construction of G'

Note that during the construction of G' edge weights were only decreased and never increased. Thus it holds that $\lambda(G', x, y) \leq \lambda(G, x, y)$ for any pair of nodes (x, y). If we ran the unmodified CAPFOREST algorithm on G' each edge would be assigned a value $q_{G'}(e)$ with $q_{G'}(e) \leq \lambda(G', e)$. Thus for every edge e it holds that $q_{G'}(e) \leq \lambda(G', e) \leq \lambda(G, e)$.

Below we will show that $\tilde{q}_G(e) = q_{G'}(e)$ for all edges e. It then follows that for all edges e it holds that $\tilde{q}_G(e) \leq \lambda(G, e)$. This implies that if $\tilde{q}_G(e) \geq \hat{\lambda}$ then $\lambda(G, e) \geq \hat{\lambda}$, which is what we needed to show.

It remains to show that for all edges e $\tilde{q}_G(e) = q_{G'}(e)$. To show this claim we will show the following stronger claim. For any i with $1 \leq i \leq m$ after the $(i-1)$th and before the ith scan of an edge the modified algorithm on g and the original algorithm on G' with the same tie breaking have visited all nodes and scanned all edges up to now in the same order and for all edges e it holds that $\tilde{q}_G(e) = q_{G'}(e)$ (we assume that before scanning an edge e, $q(e) = 0$) and for all nodes x it holds that $\tilde{r}_G(x) = r_{G'}(x)$. We show this claim by the induction on i.

For $i = 1$ observe that before the first edge scan $\tilde{q}_G(e) = q_{G'}(e) = 0$ for all edges e and the same node is picked as first node due to identical tie breaking and the fact that $G = G'$ at that point. Now for $i > 1$ assume that the claim holds for $i - 1$ and consider the scan of the $(i - 1)^{\text{th}}$ edge. If for the $(i - 1)^{\text{th}}$ edge scan a new node needs to be chosen from the priority queue by one of the algorithms then note that both algorithms will have to choose a node and they pick the same node y as $\tilde{r}_G(x) = r_{G'}(x)$ for all nodes x. Then both algorithms scan the same incident edge of y as in both algorithms the set of unscanned neighbors of y is identical. If

neither algorithm has to pick a new node then both have scanned the same edges of the same current node y and due to identical tie breaking will pick the same next edge to scan. Let this edge be (y, w). By induction $\tilde{r}_G(w) = r_{G'}(w)$ at this time. As (y, w) is unscanned $c'(y, w) = c(y, w)$ which implies that $\tilde{r}_G(w) + c(y, w) = r_{G'}(w) + c'(y, w)$. If $\tilde{r}_G(w) + c(y, w) \leq \hat{\lambda}$ then the modified algorithm on G and the original algorithm on G' will set the r value of w to the same value, namely $\tilde{r}_G(w)+c(y, w)$. If $\tilde{r}_G(w)+c(y, w) > \hat{\lambda}$, then $\tilde{r}_G(w)$ is set to $\hat{\lambda}$ and $c'(y, w)$ is set to $c(y, w)-(r_{G'}(w)-\hat{\lambda})$, which leads to $r_{G'}(w)$ being set to $\hat{\lambda}$. Thus $\tilde{r}_G(w) = r_{G'}(w)$ and by induction $\tilde{r}_G(x) = r_{G'}(x)$ for all x. Additionally the modified algorithm on G sets $\tilde{q}_G(y, w) = \tilde{r}_G(w)$ and the original algorithm on G' sets $q_{G'}(y, w) = r_{G'}(w)$. It follows that $\tilde{q}_G(y, w) = q_{G'}(y, w)$ and, thus, by induction $\tilde{q}_G(e) = q_{G'}(e)$ for all e. This completes the proof of the claim. □

Lemma 4.1.1 allows us to considerably lower the number of priority queue operations, as we do not need to update priorities that are bigger than $\hat{\lambda}$. This optimization has even more benefit in combination with running `VieCut` to lower the upper bound $\hat{\lambda}$, as we further lower the number of priority queue operations.

4.1.3 Priority Queue Implementations

Nagamochi et al. [147] use an addressable priority queue $\mathcal{Q}$ in their algorithm to find contractible edges. In this section we now address variants for the implementation of the priority queue. As the algorithm often has many elements with maximum priority in practice, the implementation of this priority queue can have major impact on the order of vertex visits and thus also on the edges that will be marked contractible.

Bucket Priority Queue

As our algorithm limits the values in the priority queue to a maximum of $\hat{\lambda}$, we observe integer priorities in the range of $[0, \hat{\lambda}]$. Hence, we can use a bucket queue that is implemented as an array with $\hat{\lambda}$ buckets. In addition, the data structure keeps the id of the highest non-empty bucket, also known as the *top bucket*, and stores the position of each vertex in the priority queue. Priority updates can be implemented by deleting an element from its bucket and pushing it to the bucket with the updated priority. This allows constant time access for all operations except for deletions of the maximum priority element, which have to check all buckets between the prior top bucket and the new top bucket, possibly up to $\hat{\lambda}$ checks. We give two possible implementations to implement the buckets so that they can store all elements with a given priority.

The first implementation, `BStack` uses a dynamic array (`std::vector`) as the container for all elements in a bucket. When we add a new element to the array, we push it to the back of the array. $\mathcal{Q}$`.pop_max()` returns the last element of the

top bucket. Thus, our algorithm will always visit the element next whose priority was just increased. It thus does not fully explore all vertices in a region and instead behaves more similar to a depth-first search.

The other implementation, `BQueue` uses a double ended queue (`std::deque`) as the container instead. A new element is pushed to the back of the queue and $\mathcal{Q}$`.pop_max()` returns the *first* element of the top bucket. This results in a variant of our algorithm, which behaves more similar to a breadth-first search in that it first explores the vertices that have been discovered earlier, i.e. are closer to the source vertex in the graph.

Bottom-Up Binary Heap

A binary heap [193] is a binary tree (implemented as an array, where element i has its children in index $2i$ and $2i+1$) which fulfills the heap property, i.e. each element has priority that is not lower than either of its children. Thus the element with highest priority is the root of the tree. The tree can be made addressable by using an array of indices, in which we save the position of each vertex. We use a binary heap using the bottom-up heuristics [192], in which we sift down holes that were created by the deletion of the top priority vertex. Priority changes are implemented by sifting the addressed element up or down in the tree. Operations have a running time of up to $\mathcal{O}(\log n)$ to sift an element up or down to fix the heap property.

In $\mathcal{Q}$`.pop_max()`, the `Heap` priority queue does not favor either old or new elements in the priority queue and therefore this implementation can be seen as a middle ground between the two bucket priority queues.

4.2 Parallel CAPFOREST

We modify the algorithm in order to quickly find contractible edges using shared-memory parallelism. The pseudocode can be found in Algorithm 2. The proofs in this section show that the modifications do not violate the correctness of the algorithm. Detailed proofs for the original CAPFOREST algorithm and the modifications of Nagamochi et al. for weighted graphs can be found in [147].

The idea of the our algorithm is as follows: We aim to find contractible edges using shared-memory parallelism. Every processor selects a random vertex and runs Algorithm 2, which is a modified version of CAPFOREST [143, 147] where the priority values are limited to $\widehat{\lambda}$, the current upper bound of the size of the minimum cut. We want to find contractible edges without requiring that every process looks at the whole graph. To achieve this, every vertex will only be visited by one process. Compared to limiting the number of vertices each process visits this has the advantage that we also scan the vertices in sparse regions of the graph which might otherwise not be scanned by any process.

Algorithm 2 Parallel CAPFOREST

Input: $G = (V, E, c)$ ← undirected graph $\hat{\lambda}$ ← upper bound for minimum cut, $\mathcal{T}$ ← shared array of vertex visits
Output: $\mathcal{U}$ ← union-find data structure to mark contractible edges

1: Label all vertices $v \in V$ "unvisited", blacklist $\mathcal{B}$ empty
2: $\forall v \in V : r(v) \leftarrow 0$
3: $\forall e \in E : q(e) \leftarrow 0$
4: $\mathcal{Q} \leftarrow$ empty priority queue
5: Insert random vertex into $\mathcal{Q}$
6: **while** $\mathcal{Q}$ not empty **do**
7: $x \leftarrow \mathcal{Q}$.pop_max() ▷ Choose unvisited vertex with highest priority
8: Mark x "visited"
9: **if** $\mathcal{T}(x) =$ True **then** ▷ Every vertex is visited only once
10: $\mathcal{B}(x) \leftarrow$ True
11: **else**
12: $\mathcal{T}(x) \leftarrow$ True
13: $\alpha \leftarrow \alpha + c(x) - 2r(x)$
14: $\hat{\lambda} \leftarrow min(\hat{\lambda}, \alpha)$
15: **for** $e = (x, y) \leftarrow$ unscanned edge, where $y \notin \mathcal{B}$ **do**
16: **if** $r(y) < \hat{\lambda} \leq r(y) + c(e)$ **then**
17: $\mathcal{U}$.union(x,y) ▷ Mark edge e to contract
18: **end if**
19: $r(y) \leftarrow r(y) + c(e)$
20: $q(e) \leftarrow r(y)$
21: $\mathcal{Q}(y) \leftarrow min(r(y), \hat{\lambda})$
22: **end for**
23: **end if**
24: **end while**

Figure 4.1 shows an example run of Algorithm 2 with $p = 5$. Every process randomly chooses a start vertex and performs Algorithm 2 on it to "grow a region" of scanned vertices. As we want to employ shared-memory parallelism to speed up the algorithm, we share an array $\mathcal{T}$ between all processes to denote whether a vertex has already been visited. If a vertex v has already been visited by a process, it will not be visited by any other processes. Additionally, every process keeps a local blacklist $\mathcal{B}$ for vertices that the process attempted to visit but that were already visited by another process before and were thus ignored by this process. Note that $\mathcal{B}$ is not shared between processes. We need this blacklist to ensure correctness, as a process may only contract edges that are not adjacent to a vertex previously blacklisted by that process (proof in Lemma 4.2.2). For every vertex v we keep a value $r(v)$, which denotes the total weight of edges connecting v to already scanned vertices. Over the course of a run of the algorithm, every edge $e = (v, w)$ is given a value $q(e)$ (equal to $r(w)$ right after scanning e) which is a lower bound for the smallest cut $\lambda(G, v, w)$.

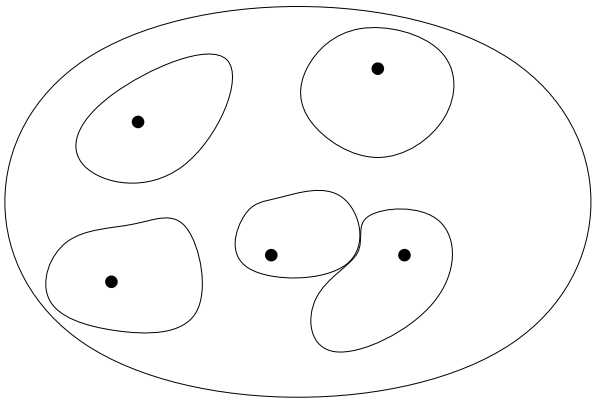


Figure 4.1: Example run of Algorithm 2. Every process starts at a random vertex and scans the region around the start vertex. These regions do not overlap.

We mark an edge e as contractible (more accurately, we union the incident vertices in the shared concurrent union-find data structure [12]), if $q(e) \geq \hat{\lambda}$. Note that this does not modify the graph, it just remembers which nodes to collapse. The actual node collapsing happens in a postprocessing step. Nagamochi and Ibaraki showed [147] that contracting only the edges that fulfill the condition in line 16 is equivalent.

As the set of disconnected edges is different depending on the start vertex, we looked into visiting every vertex by a number of processes up to a given parameter to find more contractible edges. However, this did generally result in higher total running times and thus we only visit every vertex once.

After all processes are finished, every vertex was visited exactly once (or possibly zero times, if the graph is disconnected). On average, every process has visited roughly $\frac{n}{p}$ vertices and all processes finish at the same time. We do not perform any form of locking of the elements of $\mathcal{T}$, as this would come with a running time penalty for every write and the only possible race condition with concurrent writes is that a vertex is visited more often, which does not affect correctness of the algorithm.

However, as we terminate early and no process visits every vertex, we cannot guarantee that the algorithm actually finds a contractible edge. However, in practice, this only happens if the graph is already very small (< 50 vertices in all of our experiments). We can then run the (sequential) CAPFOREST routine to find at least one edge which can be contracted. In line 13 and 14 of Algorithm 2 we compute the value of the cut between the scanned and unscanned vertices and update $\hat{\lambda}$ if this cut is smaller than it. This optimization to the CAPFOREST algorithm was first given by Nagamochi et al. [147].

In practice, many vertices reach values of $r(y)$ that are much higher than $\hat{\lambda}$ and therefore need to update their priority in $\mathcal{Q}$ often. As previously detailed, we limit the values in the priority queue by $\hat{\lambda}$ and do not update priorities that are

already greater or equal to $\hat{\lambda}$. This allows us to considerably lower the number of priority queue operations per vertex.

Theorem 4.2.1. *Algorithm 2 is correct.*

As Algorithm 2 is a modified variant of CAPFOREST [143, 147], we use the correctness of their algorithm and show that our modifications cannot result in incorrect results. In order to show this we need the following lemmas:

Lemma 4.2.2. *The following modifications to the CAPFOREST algorithms do not result in incorrect results.*

1. *Multiple instances of Algorithm 2 can be run in parallel with all instances sharing a parallel union-find data structure.*
2. *Early termination does not affect correctness*
3. *For every edge $e = (v, w)$, where neither v nor w are blacklisted, $q(e)$ is a lower bound for the connectivity $\lambda(G, v, w)$, even if the set of blacklisted vertices $\mathcal{B}$ is not empty.*
4. *When limiting the priority of a vertex in Q to $\hat{\lambda}$, it still holds that the vertices incident to an edge $e = (x, y)$ with $q(e) \geq \hat{\lambda}$ have connectivity $\lambda(G, x, y) \geq \hat{\lambda}$.*

Proof. A run of the CAPFOREST algorithm finds a non-empty set of edges that can be contracted without contracting a cut with value less than $\hat{\lambda}$ [143]. We show that none of our modifications can result in incorrect results:

1. The CAPFOREST routine can be started from an arbitrary vertex and finds a set of edges that can be contracted without affecting the minimum cut λ. This is true for any vertex $v \in V$. As we do not change the underlying graph but just mark contractible edges, the correctness is obviously upheld when running the algorithm multiple times starting at different vertices. This is also true when running the different iterations in parallel, as long as the underlying graph is not changed.

   Marking the edge $e = (u, v)$ as contractible is equivalent to performing a `Union` of vertices u and v. The `Union` operation in a union-find data structure is commutative and therefore the order of unions is irrelevant for the final result. Thus performing the iterations successively has the same result as performing them in parallel.

2. Over the course of the algorithm we set a value $q(e)$ for each edge e and we maintain a value $\hat{\lambda}$ that never increases. We contract edges that have value $q(e) \geq \hat{\lambda}$ at the time when $q(e)$ is set. For every edge, this value is set exactly once. If we terminate the algorithm prior to setting $q(e)$ for all edges,

the set of contracted edges is a subset of the set of edges that would be contracted in a full run and all contracted edges e fulfill $q(e) \geq \hat{\lambda}$ at termination. Thus, no edge contraction contracts a cut that is smaller than $\hat{\lambda}$.

3. Let $e = (v, w)$ be an edge and let $\mathcal{B}_e$ be the set of nodes blacklisted at the time when e is scanned. We show that for an edge $e = (v, w)$, $q(e) \leq \lambda(\bar{G}, v, w)$, where $\bar{G} = (\bar{V}, \bar{E})$ with vertices $\bar{V} = V \backslash \mathcal{B}_e$ and edges $\bar{E} = \{e = (u, v) \in E : u \notin \mathcal{B}_e \text{ and } v \notin \mathcal{B}_e\}$ is the graph G with all blacklisted vertices and their incident edges removed. As the removal of vertices and edges can not increase edge connectivities $q_{\bar{G}}(e) \leq \lambda(\bar{G}, v, w) \leq \lambda(G, v, w)$ and e is contractible.

   Whenever we visit a vertex b, we decide whether we blacklist the vertex. If we blacklist the vertex b, we immediately leave the vertex and do not change any values $r(v)$ or $q(e)$ for any other vertex or edge. As vertex b is marked as blacklisted, we will not visit the vertex again and the edges incident to b only affect $r(b)$.

   As edges incident to any of the vertices in $\mathcal{B}_e$ do not affect $q(e)$, the value of $q(e)$ in the algorithm with the blacklisted in G is equal to the value of $q(e)$ in $\bar{G}$, which does not contain the blacklisted vertices in $\mathcal{B}_e$ and their incident edges. On $\bar{G}$ this is equivalent to a run of CAPFOREST without blacklisted vertices and due to the correctness of CAPFOREST [147] we know that for every edge $e \in \bar{E} : q_{\bar{G}}(e) \leq \lambda(\bar{G}, v, w) \leq \lambda(G, v, w)$.

   Note that in $\bar{G}$ we only exclude the vertices that are in $\mathcal{B}_e$. It is possible that a node y that was unvisited when e was scanned might get blacklisted later, however, this does not affect the value of $q(e)$ as the value $q(e)$ is set when an edge is scanned and never modified afterwards.

4. Proof in Lemma 4.1.1.

We can combine the sub-proofs (3) and (4) by creating the graph $\bar{G}'$, in which we remove all edges incident to blacklisted vertices and decrease edge weights to make sure no $q(e)$ is strictly larger than $\hat{\lambda}$. As we only lowered edge weights and removed edges, for every edge between two not blacklisted vertices $e = (u, v)$, $q_G(e) \leq \lambda(\bar{G}', x, y) \leq \lambda(G, x, y)$ or $q_G(e) > \hat{\lambda}$ and thus we only contract contractible edges. As none of our modifications can result in the contraction of edges that should not be contracted, Algorithm 2 is correct. □

4.3 Putting Things Together

Algorithm 3 shows the overall structure of the algorithm. We first run `VieCut` to find a good upper bound $\hat{\lambda}$ for the minimum cut. Afterwards, we run Algorithm 2 to find contractible edges. In the unlikely case that none were found, we run CAPFOREST [147] sequentially to find at least one contractible edge. We create a new contracted graph using parallel graph contraction with the hash-based shared-memory

Algorithm 3 Parallel Minimum Cut

Input: $G = (V, E, c)$

```
1: λ̂ ← VieCut(G)
2: G_C ← G
3: while G_C has more than 2 vertices do
4:     λ̂ ← Parallel CAPFOREST(G_C, λ̂)
5:     if no edges marked contractible then
6:         λ̂ ← CAPFOREST(G_C, λ̂)
7:     end if
8:     G_C, λ̂ ← Parallel Graph Contract(G_C)
9: end while
10: return λ̂
```

parallel contraction technique outlined in the previous chapter. This process is repeated until the graph has only two vertices left. Whenever we encounter a collapsed vertex with a degree of lower than $\hat{\lambda}$, we update the upper bound. We return the smallest cut we encounter in this process.

If we also want to output the minimum cut, for each collapsed vertex v_C in G_C we store which vertices of G are included in v_C. When we update $\hat{\lambda}$, we store which vertices are contained in the minimum cut. This allows us to see which vertices are on one side of the cut.

4.4 Experiments and Results

4.4.1 Experimental Setup and Methodology

We implemented the algorithms using C++-17 and compiled all codes using g++-7.1.0 with full optimization (-O3). Our experiments are conducted on a machine with two Intel Xeon E5-2643 v4 with 3.4GHz with 6 CPU cores each and hyperthreading enabled, and 1.5 TB RAM in total. We perform five repetitions per instance and report average running 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 $t_{\text{best}}/t_{\text{algorithm}}$ on the y-axis. A point close to zero 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 to compute the solution. Thus an algorithm is considered to outperform another algorithm if its corresponding ratio values are above those of the other algorithm. In order to include instances that were too big for an algorithm, i.e. some implementations are limited to 32bit integers, we set the corresponding ratio below *zero*.

4.4.2 Algorithms

There have been multiple experimental studies that compare exact algorithms for the minimum cut problem [37, 94, 100]. All of these studies report that the algorithm of Nagamochi et al. and the algorithm of Hao and Orlin outperform other algorithms, such as the algorithms of Karger and Stein [105] or the algorithm of Stoer and Wagner [178], often by multiple orders of magnitude. Among others, we compare ourselfs against two available implementations of the sequential algorithm of Nagamochi et al. [143, 147]. We use our own implementation of the algorithm of Nagamochi et al. [143, 147], written in C++ (NOI-HNSS) which was implemented as part of VieCut (Chapter 3) and uses a binary heap. We use this algorithm with small optimizations in the priority queue as a base of our implementation. Chekuri et al. [37] give an implementation of the flow-based algorithm of Hao and Orlin using all optimizations given in the paper (variant ho in [37]), implemented in C, in our experiments denoted as HO-CGKLS. They also give an implementation of the algorithm of Nagamochi et al. [143, 147], denoted as NOI-CGKLS, which uses a heap as its priority queue data structure (variant ni-nopr in [37]). As their implementations use signed integers as edge ids, we include their algorithms only for graphs that have fewer than 2^{31} edges. Most of our discussions focus on comparisons to the NOI-HNSS implementation as this outperforms the implementations by Chekuri et al. .

Gianinazzi et al. [75] give a MPI implementation of the algorithm of Karger and Stein [105]. We performed preliminary experiments on small graphs which can be solved by NOI-HNSS, NOI-CGKLS and HO-CGKLS in less than 3 seconds. On these graphs, their implementation using 24 processes took more than 5 minutes, which matches other studies [37, 94, 100] that report bad real-world performance of (other implementations of) the algorithm of Karger and Stein. Gianinazzi et al. report a running time of 5 seconds for RMAT graphs with $n = 16000$ and an average degree of 4000, using 1536 *cores*. As NOI-HNSS can find the minimum cut on RMAT graphs [113] of equal size in less than 2 seconds using *a single core*, we do not include the implementation in [75] in our experiments.

As our algorithm solves the minimum cut problem exactly, we do not include the $(2 + \epsilon)$-approximation algorithm of Matula [136] and our inexact algorithm VieCut in the experiments.

4.4.3 Instances

We use a set of graph instances that was also used for experiments in Chapter 3. The set of instances contains k-cores [18] of large undirected real-world graphs taken from the 10th DIMACS Implementation Challenge [15] as well as the Laboratory for Web Algorithmics [24, 25]. Additionally it contains large random hyperbolic graphs [119, 131] with $n = 2^{20} - 2^{25}$ and $m = 2^{24} - 2^{32}$. A detailed description of the graph instances is given in Section 2.5 (Graph family A). These graphs are unweighted, however contracted graphs that are created in the course of the algorithm have edge weights.

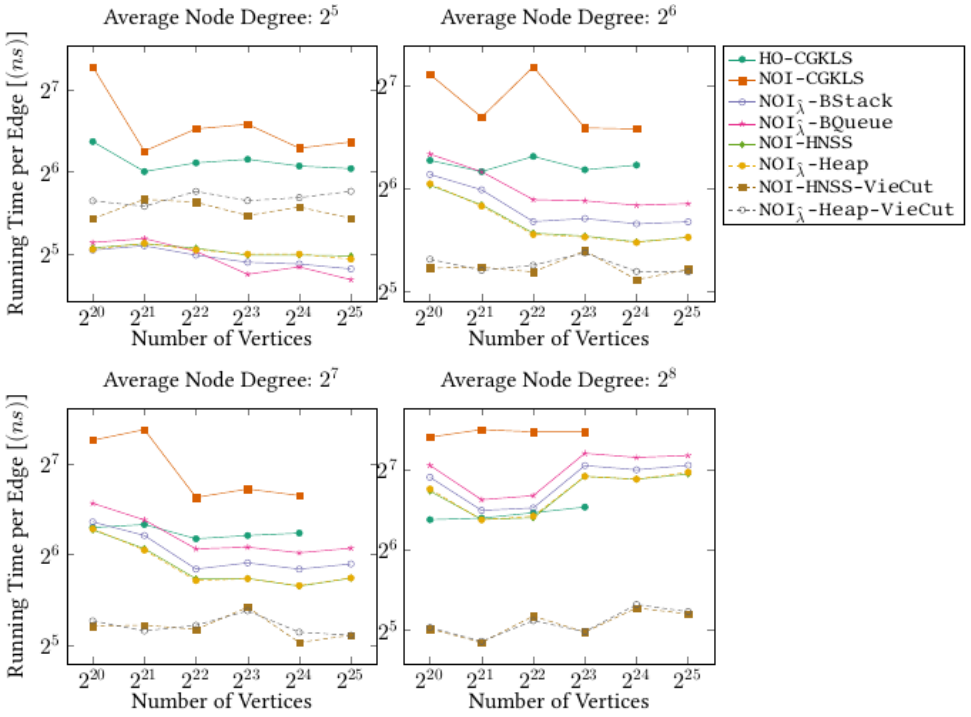


Figure 4.2: Total running time in nanoseconds per edge in random hyperbolic graphs.

4.4.4 Sequential Experiments

We limit the values in the priority queue $\mathcal{Q}$ to $\hat{\lambda}$, in order to significantly lower the number of priority queue operations needed to run the contraction routine. In this experiment, we want to examine the effects of different priority queue implementations and limiting priority queue values have on sequential minimum cut computations. We also include variants which run VieCut first to lower $\hat{\lambda}$.

We start with sequential experiments using the implementation of NOI-HNSS. We use two variants: NOI$_{\hat{\lambda}}$ limits values in the priority queue to $\hat{\lambda}$ while NOI-HNSS allows arbitrarily large values in $\mathcal{Q}$. For NOI$_{\hat{\lambda}}$, we test the three priority queue implementations, BQueue, Heap and BStack. As the priority queue for NOI-HNSS has priorities of up to the maximum degree of the graph and the contracted graphs can have very large degrees, the bucket priority queues are not suitable for NOI-HNSS. Therefore we only use the implementation of NOI-HNSS [94].

The variants NOI-HNSS-VieCut and NOI$_{\hat{\lambda}}$-Heap-VieCut first run the shared-memory parallel algorithm VieCut using all 24 threads to lower $\hat{\lambda}$ before running the respective sequential algorithm. We report the total running time, e.g. the sum of VieCut and NOI.

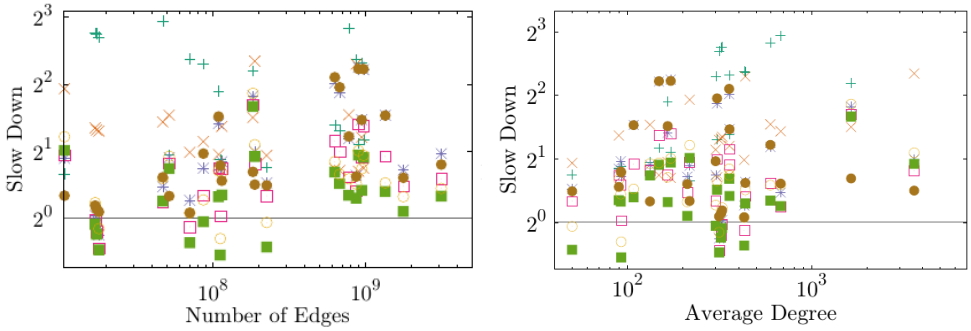


Figure 4.3: Total running time in real-world graphs, normalized by the running time of NOI$_{\hat{\lambda}}$-Heap-VieCut. (Legend shared with Figure 4.4 below)

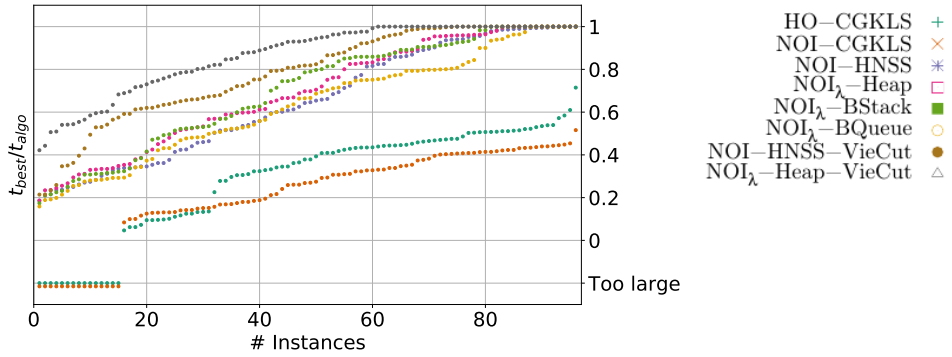


Figure 4.4: Performance plot for all graphs.

Priority Queue Implementations

Figure 4.2 shows the results for hyperbolic graphs and Figure 4.3 shows the results for real-world graphs, normalized by the running time of NOI$_{\hat{\lambda}}$-Heap-VieCut. Figure 4.4 gives performance plots for all graphs from both graph families. We can see that in nearly all sequential runs, NOI$_{\hat{\lambda}}$-BStack is $5 - 10\%$ faster than NOI$_{\hat{\lambda}}$-BQueue. This can be explained as this priority queue uses `std::vector` instead of `std::deque` as its underlying data structure and thus has lower access times to add and remove elements. As all vertices are visited by the only thread, the scan order does not greatly influence how many edges are contracted.

In the random hyperbolic graphs, nearly no vertices in NOI-HNSS reach priorities in $\mathcal{Q}$ that are much larger than $\hat{\lambda}$. Usually, fewer than 5% of edges do not incur an update in $\mathcal{Q}$. Thus, NOI-HNSS and NOI$_{\hat{\lambda}}$-Heap have practically the same running time. NOI$_{\hat{\lambda}}$-BStack is usually 5% slower.

As the real-world graphs are social network and web graphs, they contain vertices with very high degrees. In these vertices, NOI-HNSS often reaches priority values of much higher than $\hat{\lambda}$ and NOI$_{\hat{\lambda}}$ can actually save priority queue operations.

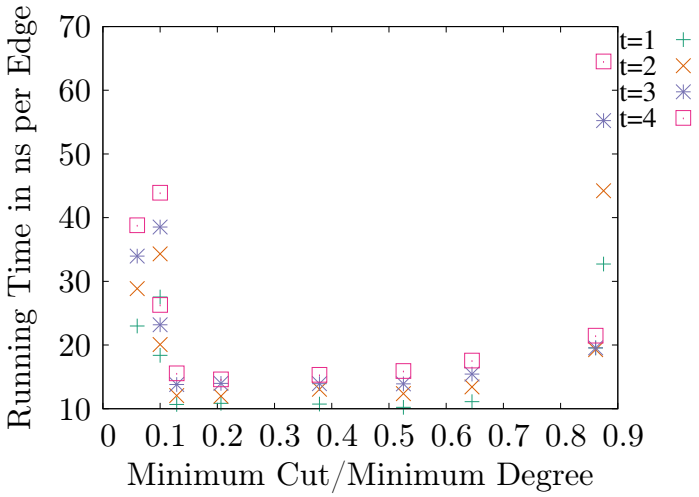


Figure 4.5: Results for different values for t

Thus, NOI$_{\hat{\lambda}}$-Heap is up to 1.83 times faster than NOI-HNSS with an average (geometric) speedup factor of 1.35. Also, in contrast to the random hyperbolic graphs, NOI$_{\hat{\lambda}}$-BStack is faster than NOI-HNSS on real-world graphs. Due to the low diameter of web and social graphs, the number of vertices in $\mathcal{Q}$ is very large. This favors the BStack priority queue, as it has constant access times. The average geometric speedup of NOI$_{\hat{\lambda}}$-BStack compared to NOI$_{\hat{\lambda}}$-Heap is 1.22.

Reduction of $\hat{\lambda}$ by VieCut

In this experiment we aim to reduce $\hat{\lambda}$ by running VieCut before NOI. While the other algorithms are slower for denser random hyperbolic graphs, both algorithms NOI-HNSS-VieCut and NOI$_{\hat{\lambda}}$-Heap-VieCut are faster in these graphs with higher density. This happens as the variants without VieCut find fewer contractible edges and therefore need more rounds of CAPFOREST. The highest speedup compared to NOI$_{\hat{\lambda}}$-Heap is reached in random hyperbolic graphs with $n = 2^{23}$ and an average density of 2^8, where NOI$_{\hat{\lambda}}$-Heap-VieCut has a speedup of factor 4.

NOI$_{\hat{\lambda}}$-Heap-VieCut is fastest on most real-world graphs, however when the minimum degree is very close to the minimum cut λ, running VieCut can not significantly lower $\hat{\lambda}$. Thus, the extra work to run VieCut takes longer than the time saved by lowering the upper bound $\hat{\lambda}$. The average geometric speedup factor of NOI$_{\hat{\lambda}}$-Heap-VieCut on all graphs compared to the variant without VieCut is 1.34.

In the performance plots in Figure 4.4 we can see that NOI$_{\hat{\lambda}}$-Heap-VieCut is fastest or close to the fastest algorithm in all but the very sparse graphs, in which the algorithm of Nagamochi et al. [147] is already very fast [94] and therefore us-

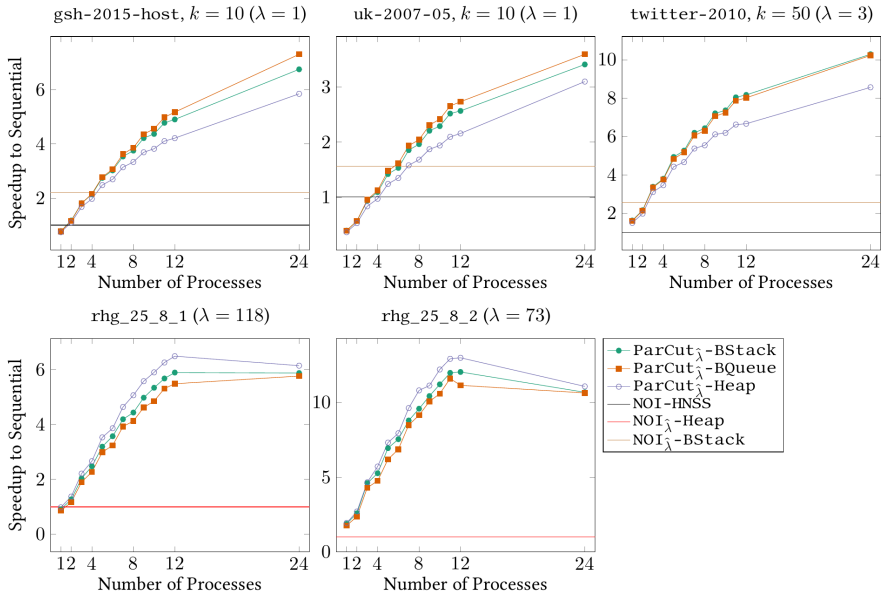


Figure 4.6: Scaling plots for large graphs - Scalability

ing VieCut cannot sufficiently lower $\hat{\lambda}$ and thus the running time of the algorithm. NOI-CGKLS and HO-CGKLS are outperformed on all graphs.

4.4.5 Shared-memory parallelism

Overlapping Scan Regions

We examine whether it is useful to overlap the regions scanned by each thread. For this purpose we introduce a parameter t which indicates how many threads can scan each processor. A value of $t = 1$ executes algorithm 2, any larger value replaces $\mathcal{T}(x)$ with a counter indicating how many threads already scanned vertex x. If $\mathcal{T}(x) \geq t$, no further threads may scan it.

Figure 4.5 shows results for ParCut$_{\hat{\lambda}}$-BQueue with 24 processes for values of t from 1 to 4 on a set of 10 large graphs (4 real-world graphs, 6 RHG graphs). In general, lower values of t have lower running times for Algorithm 2, however the amounts of contracted edges can be lower, especially when many vertices have degree not too much higher than $\hat{\lambda}$, as those can only be contracted depending on the order of vertex scans. On 9 out of the 10 graphs, $t = 1$ has the best performance, just in the graph rhg_25_8_1 with minimum degree 137 and $\lambda = 118$, $t = 2$ and $t = 3$ are slightly faster. Thus we set parameter t to 1 and do not use overlapping scan regions.

We run experiments on 5 of the largest graphs in the data sets using up to 24 threads on 12 cores. First, we compare the performance of Algorithm 3 using different priority queues: ParCut$_{\hat{\lambda}}$-Heap, ParCut$_{\hat{\lambda}}$-BStack and ParCut$_{\hat{\lambda}}$-BQueue

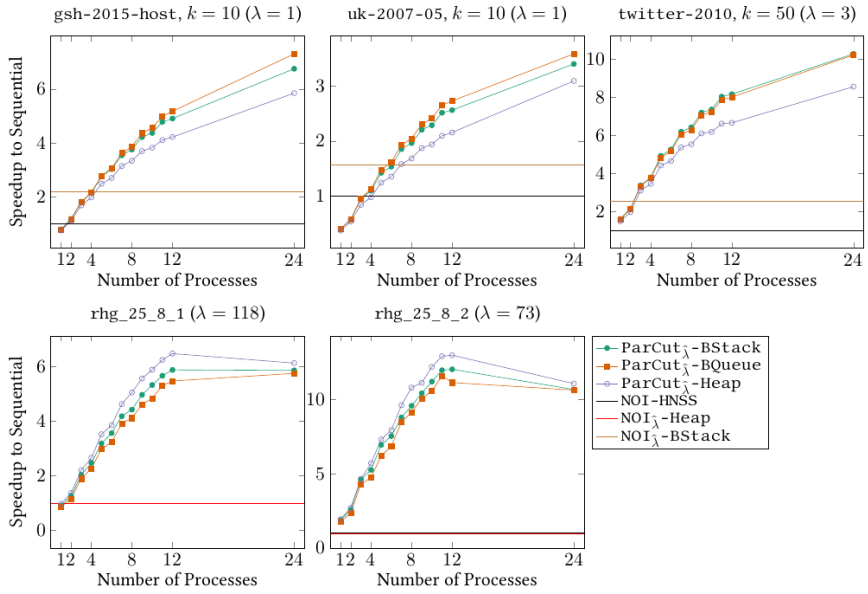


Figure 4.7: Scaling plots for large graphs - Speedup compared to NOI-HNSS and fastest sequential algorithm (first 3 graphs: NOI$_{\hat{\lambda}}$-BStack, last 2 graphs: NOI$_{\hat{\lambda}}$-Heap).

all limit the priorities to $\hat{\lambda}$, the result of VieCut. In these experiments, VieCut takes up between $19 - 83\%$ of the total running time with an average of 51%. Figure 4.6 shows how well the algorithms scale with increased number of processors. Figure 4.7 shows the speedup compared to the fastest sequential algorithm of Section 4.4.4. On all graphs, ParCut$_{\hat{\lambda}}$-BQueue has the highest speedup when using 24 threads. On real-world graphs, ParCut$_{\hat{\lambda}}$-BQueue also has the lowest total running time. In the large random hyperbolic graphs, in which the priority queue is usually only filled with up to 1000 elements, the worse constants of the double-ended queue cause the variant to be slightly slower than ParCut$_{\hat{\lambda}}$-Heap also even when running with 24 threads. In the two large real-world graphs that have a minimum degree of 10, the sequential algorithm NOI$_{\hat{\lambda}}$-BStack contracts most edges in a single run of CAPFOREST - due to the low minimum degree, the priority queue operations per vertex are also very low. Thus, ParCut$_{\hat{\lambda}}$ using only a single thread has a significantly higher running time, as it runs VieCut first and performs graph contraction using a concurrent hash table, as described in Section 3.2.1, which is slower than sequential graph contraction when using just one thread. In graphs with higher minimum degree, NOI needs to perform multiple runs of CAPFOREST. By lowering $\hat{\lambda}$ using VieCut we can contract significantly more edges and achieve a speedup factor of up to 12.9 compared to the fastest sequential algorithm NOI$_{\hat{\lambda}}$-Heap. On twitter-2010, $k = 50$, ParCut$_{\hat{\lambda}}$-BQueue has a speedup of 10.3 to NOI-HNSS, 16.8 to NOI-CGKLS and a speedup of 25.5 to HO-CGKLS. The other graphs have more than 2^{31} edges and are thus too large for NOI-CGKLS and HO-CGKLS.

4.5 Conclusion

We presented a shared-memory parallel exact algorithm for the minimum cut problem. Our algorithm is based on the algorithms of Nagamochi et al. [143, 147] and our work described in Chapter 3. We use different data structures and optimizations to decrease the running time of the algorithm of Nagamochi et al. by a factor of up to 2.5. Using additional shared-memory parallelism we further increase the speedup factor to up to 12.9. Future work includes checking whether our sequential optimizations and parallel implementation can be applied to the $(2 + \epsilon)$-approximation algorithm of Matula [136].

CHAPTER 5

Finding All Minimum Cuts

We present a practically efficient algorithm that finds all global minimum cuts in huge undirected graphs. Our algorithm uses a multitude of kernelization rules to reduce the graph to a small equivalent instance and then finds all minimum cuts using an optimized version of the algorithm of Nagamochi, Nakao and Ibaraki [146]. Some of these techniques are adapted from techniques for the global minimum cut problem [147, 152] which we discussed in the previous chapters of this dissertation. Using these and newly developed reductions we are able to decrease the running time by up to multiple orders of magnitude compared to the algorithm of Nagamochi et al. [146] and are thus able to find all minimum cuts on graphs with up to billions of edges in a few minutes. Based on the cactus representation of all minimum cuts, we are able to find the most balanced minimum cut in time linear to the size of the cactus. As our techniques are able to find the most balanced minimum cut of graphs with billions of edges in minutes, this allows the use of minimum cuts as a subroutine in sophisticated data mining and graph analysis.

The content of this chapter is based on [92].

5.1 Algorithm Description

Our algorithm combines a variety of techniques and algorithms in order to find all minimum cuts in a graph. The algorithm is based on the contractions of edges which cannot be part of *any* minimum cut. Thus, we first show that an edge e that is not part of any minimum cut in graph G can be contracted. In contrast to the previous chapters, we now aim to maintain *all* minimum cuts.

Lemma 5.1.1. *[105] If an edge $e = (u, v)$ is not part of any minimum cut in graph G, all minimum cuts of G remain in the resulting graph G/e.*

Proof. Let (A, B) be an arbitrary minimum cut of G. For an edge $e = (u, v)$, which is not part of any minimum cut, we know that $e \notin E[A]$, so either u and v are both in vertex set A or both in vertex set B. This is still the case in G/e. Thus, the edge e can be contracted even if we aim to find every minimum cut of G. □

Lemma 5.1.1 is very useful to reduce the size of the graph with the usage of techniques to identify such edges. We now give a short overview of our algorithm and then explain the techniques in more detail. First, we use our shared-memory parallel heuristic minimum cut algorithm `VieCut` (as described in Chapter 3) in order to find an upper bound $\hat{\lambda}$ for the minimum cut which is likely to be the correct value. Having a tight bound for the minimum cut allows the contraction of many edges, as multiple reduction techniques depend on the value of the minimum cut. We adapt contraction techniques originally developed by Nagamochi et al. [143, 147] and Padberg et al. [151] (see Section 2.4) to the problem of *finding all minimum cuts*. Section 5.1.1 details these contraction routines. On the resulting graph we find all minimum cuts using an optimized variant of the algorithm of Nagamochi, Nakao and Ibaraki [146] and return the cactus graph which represents them all. A short description of the algorithm and an explanation of our engineering effort are given in Section 5.1.2. Afterwards, in Section 5.1.3 we show how we combine the parts into a fast algorithm to find all minimum cuts of large networks.

5.1.1 Edge Contraction

As shown in Lemma 5.1.1, edges that are not part of any minimum cut can be safely contracted. We build a set of techniques that aim to find contractible edges and run these in alternating order until neither of them finds any more contractible edges. We now give a short introduction to these.

For efficiency, we perform contractions in bulk. If our algorithm finds an edge that can be contracted, we merge the incident vertices in a thread-safe union-find data structure [12]. After each run of a contraction technique that finds contractible edges, we create the contracted graph using a shared-memory parallel hash table [134]. In this contracted graph, each set of vertices of the original graph is merged into a single node. The contraction of this vertex set is equivalent to contracting a spanning tree of the set. After contraction we check whether a vertex in the contracted graph has degree $< \hat{\lambda}$. If it does, we found a cut of smaller value and update $\hat{\lambda}$ to this value.

Connectivity-based Contraction

The connectivity of an edge $e = (s, t)$ is the weight of the minimum cut that separates s and t, i.e. the *minimum s-t-cut*. For an edge that has connectivity $> \hat{\lambda}$, we thus know that there is no cut separating s and t (i.e. no cut that contains e) that has value $\leq \hat{\lambda}$. Thus, we know that there cannot be a minimum cut that contains e, as $\hat{\lambda}$ is by definition at least as large as λ. However, solving the minimum s-t-cut

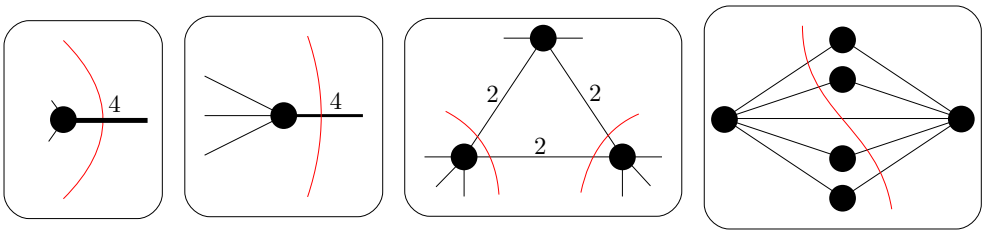


Figure 5.1: Local reduction rules: (1) `HeavyEdge`, (2) `ImbalancedVertex`, (3) `ImbalancedTriangle`, (4) `HeavyNeighborhood`.

problem takes significant time, so computing the connectivity of each edge does not scale to large networks. Hence, as part of their algorithm for the global minimum cut problem, Nagamochi et al. [143, 147] give a subroutine that computes a lower bound $q(e)$ for the connectivity of every edge e of G in a total running time of $\mathcal{O}(m + n \log n)$. Both the algorithm of Nagamochi et al. and the CAPFOREST subroutine that computes the connectivity lower bounds $q(e)$ are outlined in Section 2.4.1. Each of the edges whose connectivity lower bound is already larger than $\widehat{\lambda}$ can be contracted as it cannot be part of any minimum cut.

In Chapter 4 we give a fast shared-memory parallel variant of their algorithm. As that algorithm aims to find a single minimum cut, it also contracts edges that have connectivity equal to $\widehat{\lambda}$, as the only relevant cuts are ones better than the best cut known previously. As we want to find all minimum cuts, we can only contract edges whose connectivity is strictly larger than $\widehat{\lambda}$. Nagamochi et al. could prove that at least one edge has value $\widehat{\lambda}$ in their routine and can thus be contracted. We do not have such a guarantee when trying to find edges that have connectivity $> \widehat{\lambda}$. Consider for example an unweighted tree, whose minimum cut has a value of 1 and each edge has connectivity 1 as well.

Local Contraction Criteria

Padberg and Rinaldi [151] give a set of *local reduction* routines which determine whether an edge can be contracted without affecting the minimum cut. We describe these reductions in Section 2.4.2. Their reduction routines were shown to be very useful in order to find a minimum cut fast in practice [37, 94, 100] and are also used in our `VieCut` algorithm in Chapter 3. We adapt the routines originally developed for the minimum cut problem so that they hold for the problem of for finding all minimum cuts. Thus, we have to make sure that we do not contract cuts of value $\widehat{\lambda}$, as they might be minimal and additionally make sure that we do not contract edges incident to vertices that could have a *trivial minimum cut*, i.e. a minimum cut, where one side contains only a single vertex. Figure 5.1 shows examples and Lemma 5.1.2 gives a formal definition of these reduction rules.

Lemma 5.1.2. *For an edge $e = (u, v) \in E$, e is not part of any minimum cut, if e fulfills at least one of the following criteria. Thus, all minimum cuts of G are still present in G/e and e can be contracted.*

1. `HeavyEdge`: $c(e) > \hat{\lambda}$
2. `ImbalancedVertex`:
   - $c(v) < 2c(e)$ *and* $c(v) > \hat{\lambda}$, *or*
   - $c(u) < 2c(e)$ *and* $c(u) > \hat{\lambda}$
3. `ImbalancedTriangle`:
   $\exists w \in V$ *with*
   - $c(v) < 2\{c(v, w) + c(e)\}$ *and* $c(v) > \hat{\lambda}$, *and*
   - $c(u) < 2\{c(u, w) + c(e)\}$ *and* $c(u) > \hat{\lambda}$
4. `HeavyNeighborhood`:
   $c(e) + \sum_{w \in V} min\{c(v, w), c(u, w)\} > \hat{\lambda}$

Proof.

1. If $c(e) > \hat{\lambda}$, every cut that contains e has capacity $> \hat{\lambda}$. Thus it can not be a minimal cut.
2. Without loss of generality let v be the vertex in question. The condition $c(v) < 2c(e)$ means that e is heavier than all other edges incident to v combined. Thus, for any non-trivial cut that contains e, we can find a lighter cut by replacing e with all other incident edges to v, i.e. moving v to the other side of the cut. As this is not true for the trivial minimum cut $(v, V \backslash v)$, we cannot contract an edge incident to a vertex that has weight $\leq \hat{\lambda}$.
3. This condition is similar to (2). Let there be a triangle u, v, w in the graph in which it holds for both u and v that the two incident triangle edges are heavier than the sum of all other incident edges. Then, every cut that separates u and v can be improved by moving u and v into the same side. As the cut could have vertex w on either side, both vertices need to fulfill this condition. To make sure that we do not contract any trivial minimum cut, we check that both v and u have weighted vertex degree $> \hat{\lambda}$ and thus can not represent a trivial minimum cut.
4. In this condition we check the whole shared neighborhood of vertices u and v. Every cut that separates u and v must contain e and for each shared neighbor w at least one of the edges connecting them to w. Thus, we sum over the lighter edge connecting them to the shared neighbors and have a lower bound of the minimum cut that separates u and v. If this is heavier than $\hat{\lambda}$, we know that no minimum cut separates u and v.

□

The conditions `HeavyEdge` and `ImbalancedVertex` can both be checked for the whole graph in a single run in linear time. While we can check condition `ImbalancedTriangle` when summing up the lighter incident edges for condition `HeavyNeighborhood`, exhaustively checking all triangles incurs a strictly worse than linear runtime, as a graph can have up to $\Theta(m^{3/2})$ triangles [166]. Thus, we only perform linear-time runs as developed by Chekuri et al. [37] by marking the neighborhood of u and v while we check the conditions and do not perform the test on marked vertices.

Vertices with One Neighbor

Over the run of the algorithm, we occasionally encounter vertices that have only a single neighbor. Let v be this vertex with one neighbor and $e = (v, w)$ be the only incident edge. As we update $\widehat{\lambda}$ to the minimum degree whenever we perform a bulk edge contraction, $c(e) \geq \widehat{\lambda}$: for an edge whose weight is $> \widehat{\lambda}$, condition `HeavyEdge` will contract it. For an edge whose weight is $\widehat{\lambda}$, the edge represents a trivial minimum cut iff $\widehat{\lambda} = \lambda$. This is the only minimum cut that contains e, as every non-trivial cut containing e has higher weight. Thus, we can contract e for now and remember that it was contracted. If $\widehat{\lambda}$ is decreased, we can forget about these vertices as the cuts are not minimal. When we are finished, we can re-insert all contracted vertices that have a trivial minimum cut. We perform this reinsertion in a bottom-up fashion (i.e. in reverse order to how they were contracted), as the neighbor w could be contracted in a later contraction.

5.1.2 Finding All Minimum Cuts

We apply the reductions in the previous section exhaustively until they are not able to find a significant number of edges to contract. On the remaining graph we aim to find the cactus representation of all minimum cuts. Our algorithm for this purpose is based on the algorithm of Nagamochi, Nakao and Ibaraki [146]. While there is a multitude of algorithms for the problem of finding all minimum cuts, to the best of our knowledge there are no implementations accessible to the public and there is no practical experimentation on finding all minimum cuts. We base our algorithm on the algorithm of Nagamochi, Nakao and Ibaraki [146], as their algorithm allows us to run the reduction routines previously detailed in between recursion steps.

We give a quick sketch of their algorithm, for further details we refer the reader to [146]. To find all minimum cuts in graph G, the algorithm chooses an edge $e = (s, t)$ in G and uses a maximum flow f to find the minimum s-t-cut $\lambda(s, t)$. If $\lambda(s, t) > \lambda$ there is no minimum cut that separates s and t and thus e can be contracted. If $\lambda(s, t) = \lambda$, the edge is part of at least one minimum cut. They show that the strongly connected components $(V_1, \ldots, V_k)$ of the residual graph G_f represent all minimum cuts that contain e (and potentially some more). For each connected component V_i, they build a graph C_i, in which all other connected components are contracted into a single vertex. We recurse on these component

subgraphs and afterwards combine the minimum cut cactus graphs of the recursive calls to a cactus representation for G. The combination of the cactus graphs begins by building a cactus graph C representing the set of strongly connected components, in which each V_i is represented by a single vertex v_i. Each cactus C_i is then merged with C by replacing v_i with C_i. Inside this algorithm we re-run the contraction routines of Section 5.1.1. As they incur some computational cost and the graph does not change too much over different recursion steps, we only run the contraction routines every 10 recursion levels.

As the contraction routines in Section 5.1.1 usually mark a large amount of edges that can be contracted in bulk, we represent the graph in the compressed sparse row format [186]. This allows for fast and memory-efficient accesses to vertices and edges, however, we need to completely rebuild the graph in each bulk contraction and also keep vertex information about the whole graph hierarchy to be able to see which vertices in the original graph are encompassed in a vertex in a coarser vertex and to be able to re-introduce the cactus edges that were removed. While this is efficient for the bulk contractions performed in the previous section, in this section we often perform single-edge contractions or contract a small block of vertices. For fast running times these operations should not incur a complete rebuild of the graph data structure. We therefore use a mutable adjacency list data structure where each vertex is represented by a dynamic array of edges to neighboring vertices. Each edge stores its weight, target and the ID of its reverse edge (as we look at undirected graphs). This allows us to contract edges and small blocks in time corresponding to the sum of vertex degrees. For each vertex in the original graph, we store information which vertex currently encompasses it and every vertex keeps a list of currently encompassed vertices of the original graph. All vertex and edge information is updated during each edge contraction. The same graph data structure is also used for the *multiterminal cut problem* in Part III of this work.

Edge Selection

The recursive algorithm of Nagamochi, Nakao and Ibaraki [146] selects an arbitrary edge for the maximum flow problem in each recursion step. If this edge has connectivity equal to the minimum cut, we create a recursive subproblem for each connected component of the residual graph. In order to reduce the graph size - and thus the amount of work necessary - quickly, we aim to select edges in which the largest connected component of the residual graph is as small as possible. The edge selection strategy `Heavy` searches for the highest degree vertex v and chooses the edge from v to its highest degree neighbor. The strategy `WeightedHeavy` does the same, but uses the vertices whose weighted degree is highest. The idea is that an edge between high-degree vertices is most likely 'central' to the graph and thus manages to separate sizable chunks from the graph. The edge selection strategy `Central` aims to find a central edge more directly: we aim to find two vertices u and v with a high distance and take the central edge in their shortest paths. We find those vertices by performing a breadth-first search from a random vertex w, after-

wards performing a breadth-first search from the vertex encountered last. We then take the central edge in the shortest path (as defined from the second breadth-first search) from the two vertices encountered last in the two breadth-first searches. The edge selection strategy `Random` picks a random edge.

Degree-two Reductions

Over the course of this recursive contraction-based algorithm, we often encounter vertices with just two neighbors. Let v be the vertex in question, which is connected to u_0 by edge e_0 and to u_1 by edge e_1. We look at four cases, each looking at whether the weight of e_0 being equal to the weight of e_1 and $c(v)$ being equal to λ, both conditions that can be checked in constant time. In three out of four cases, we are able to contract an incident edge.

$c(e_0) \neq c(e_1)$ and $c(v) > \lambda$: Without loss of generality let e_0 be the heavier edge. As $c(v) > \lambda$, the trivial cut $(\{v\}, V\backslash\{v\})$ is not a minimum cut. As by definition no cut in G is smaller than λ, $\lambda(u_0, u_1) \geq \lambda$. Thus, excluding the path through v, they have a connectivity of $\geq \lambda - c(e_1)$ and any cut containing e_0 has weight $\geq \lambda - c(e_1) + c(e_0) > \lambda$ and can thus not be minimal. We therefore know that e_0 is not part of any minimum cuts and can be contracted according to Lemma 5.1.1.

$c(e_0) \neq c(e_1)$ and $c(v) = \lambda$: Without loss of generality let e_0 be the heavier edge. Analogously to the previous case we can show that no nontrivial cut contains e_0. In this case, where $c(v) = \lambda$, the trivial cut $(\{v\}, V\backslash\{v\})$ is minimal and therefore should be represented in the cactus graph. For all other minimum cuts that contain e_1, we know that v and u_0 will be in the same block (as $c(e_0) > c(e_1)$). Thus, v will be represented in the cactus as a leaf incident to u_0. We contract e_0 calling the resulting vertex u^* and store which vertices of the original graph are represented by v. Then we recurse. On return from the recursion we check which cactus vertex now encompasses u^* and add an edge from this vertex to a newly added vertex representing all vertices encompassed by v.

$c(e_0) = c(e_1)$ and $c(v) > \lambda$: in this case we are not able to contract any edges without further connectivity information.

$c(e_0) = c(e_1)$ and $c(v) = \lambda$: as $c(v) = \lambda$, the trivial cut $(\{v\}, V\backslash\{v\})$ is minimal. If there are other minimum cuts that contain either e_0 or e_1 (e.g. that separate u_0 and u_1), we know that by replacing e_0 with e_1 (or vice-versa) the cut remains minimal. Such a minimum cut exists iff $\lambda(u_0, u_1) = \lambda$. We contract e_0 and remember this decision. As e_1 is still in the graph (merged with (u_0, u_1)), we are able to find each cut that separates u_0 and u_1. If none exists, $\lambda(u_0, u_1) > \lambda$ and u_0 and u_1 will be contracted later in the algorithm. When leaving the recursion, we can thus re-introduce vertex v as a leaf connected to the vertex encompassing u_0 and u_1. If u_0 and u_1 are in different vertices after leaving the recursion, there is at least one nontrivial cut that contains e_1. We thus re-introduce v as a cycle vertex connected to u_0 and u_1, each with weight $\frac{\lambda}{2}$, and subtract $\frac{\lambda}{2}$ from $c(u_0, u_1)$.

Algorithm 4 Algorithm to find all minimum cuts

```
1: procedure FindAllMincuts(G = (V, E))
2:     λ̂ ← VieCut(G) [94]
3:     while not converged do
4:         (G, D_1, λ̂) ← contract degree-one vertices(G, λ̂)
5:         (G, λ̂) ← connectivity-based contraction(G, λ̂)
6:         (G, λ̂) ← local contraction(G, λ̂)
7:     end while
8:     λ ← FindMinimumCutValue(G)
9:     C ← RecursiveAllMincuts(G, λ) ([146])
10:    C ← reinsert vertices(C, D_1)
11:    return (C, λ)
12: end procedure
```

In three out of the four cases presented here, we are able to contract an edge incident to a degree-two vertex. We can check these conditions in total time $\mathcal{O}(n)$ for the whole graph. Over the course of the algorithm, we perform edge contractions and thus routinely encounter vertices whose neighborhood has been contracted and thus have a degree of two. Thus, these reductions are able to reduce the size of the graph significantly even if the initial graph is rather dense and does not have a lot of low degree vertices.

5.1.3 Putting it All Together

Algorithm 4 gives an overview over our algorithm to find all minimum cuts. Over the course of the algorithm we keep an upper bound $\widehat{\lambda}$ for the minimum cut, initially set to the result of the inexact variant of the `VieCut` minimum cut algorithm [94] (Chapter 3). While the `VieCut` algorithm also offers an exact version [90] (Chapter 4), we use the inexact version, as it is considerably faster and gives a low upper bound for the minimum cut, usually equal to the minimum cut. As described in Section 5.1.1, we use this bound to contract degree-one vertices, high-connectivity edges and edges whose local neighborhood guarantees that they are not part of any minimum cut. We repeat this process until it is converged, as an edge contraction can cause other edges in the neighborhood to also become safely contractible. As this process often incurs a long tail of single edge contractions, we stop if the number of vertices was decreased by less than 1% over a run of all contraction routines.

We then use the minimum cut algorithm of Nagamochi, Ono and Ibaraki [143, 147] on the remaining graph, as the following steps need the correct minimum cut. To find all minimum cuts in the contracted graph, we call our optimized version of the algorithm of Nagamochi et al. [146], as sketched in Section 5.1.2, and afterwards re-insert all minimum cut edges that were previously deleted. Before each recursive call of the algorithm of Nagamochi et al. [146], we contract edges inci-

dent to degree-one and eligible degree-two vertices. Every 10 recursion levels we additionally check for connectivity-based edge contractions and local contractions.

5.1.4 Shared-Memory Parallelism

Algorithm 4 employs shared-memory parallelism in every step. When we run the algorithm in parallel, we use the parallel variant of `VieCut` [94]. Local contraction and marking of degree one vertices are parallelized using OpenMP [42]. For the first round of connectivity-based contraction, we use the parallel connectivity certificate used in the shared-memory parallel minimum cut algorithm detailed in Chapter 4 [90]. This connectivity certificate is essentially a parallel version of the connectivity certificate of Nagamochi et al. [143, 147], in which the processors divide the work of computing the connectivity bounds for all edges of the graph. In subsequent iterations every processor runs an independent run of the connectivity certificate of Nagamochi et al. on the whole graph starting from different random vertices in the graph. As the connectivity bounds given by the algorithm heavily depend on the starting vertex, this allows us to find significantly more contractible edges per round than running the connectivity certificate only once.

We use our exact shared-memory parallel minimum cut algorithm to find the exact minimum cut of the graph. The algorithm of Nagamochi et al. [146] is not shared-memory parallel, however we usually manage to contract the graph to a size proportional to the minimum cut cactus before calling them. Unfortunately it is not beneficial to perform the recursive calls embarrassingly parallel, as in almost all cases one of the connected components of the residual graph contains the vast majority of vertices and thus also has the overwhelming majority of work.

5.2 Applications

We can use the minimum cut cactus C_G to find a minimum cut fulfilling certain balance criteria, such as a most balanced minimum cut, e.g. a minimum cut $(A, V\backslash A)$ that maximizes $\min(|A|, |V\backslash A|)$. Note that this is not equal to the most balanced s-t-cut problem, which is NP hard [26]. Following that we show how to modify the algorithm to find the optimal minimum cut for other optimization functions.

One can find a most balanced minimum cut trivially in time $\mathcal{O}((n^*)^3)$, as one can enumerate all $\mathcal{O}((n^*)^2)$ minimum cuts [104] and add up the number of vertices of the original graph G on either side. We now show how to find a most balanced minimum cut of a graph G in $\mathcal{O}(n^* + m^*)$ time, given the minimum cut cactus graph C_G.

For every cut $(A, V\backslash A)$, we define the balance $b(A)$ (or $b(V\backslash A)$) of the cut as the number of vertices of the original graph encompassed in the lighter side of the cut. Recall that for any node $v \in V_G$, $c(v)$ is the number of vertices of G represented by v. For a leaf $v \in V_G$, we set its weight $w(v) = c(v)$ and set the balance $b(v)$ to be the minimum of $w(v)$ and $n - w(v)$. We root C_G in an arbitrary vertex and depending on that root define $w(v)$ as the sum of vertex weights in the subcactus rooted in

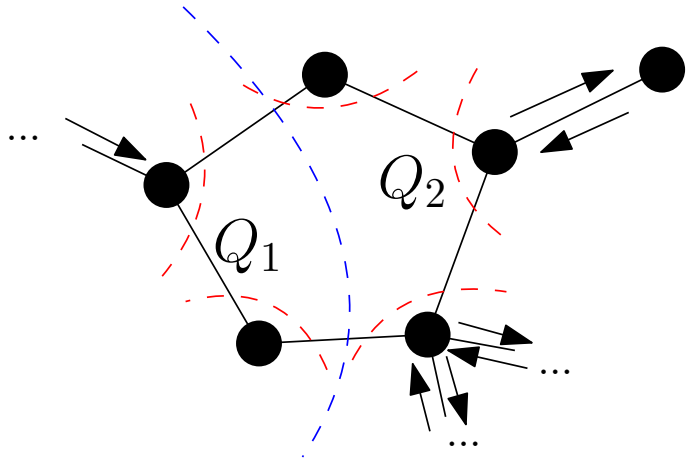


Figure 5.2: Cycle check in balanced cut algorithm

v; and $b(v)$ accordingly. For a cycle $C = \{c_1, \dots, c_i\}$, we define $b(c_j, \dots, c_{k \mod i})$ with $0 \geq j \geq k$ analogously as the balance of the minimum cut splitting the cycle so that the sub-cacti rooted in $c_j, \dots, c_{k \mod i}$ are on one side of the cut and the rest are on the other side (see blue line in Figure 5.2 for an example).

Let T_G be the tree representation of C_G where each cycle in C_G is contracted into a single vertex. We perform a depth-first search on T_G rooted on an arbitrary vertex and check the balance of every cut in T_G when backtracking.

As C_G is not necessarily a tree, we might encounter cycles and we explain next how to extend the depth first search to handle such cycles. Let $\mathcal{C} = \{c_0, \dots, c_{i-1}\}$ be a cycle and c_0 be the vertex encountered first by the DFS. Due to the cactus graph structure of C_G, the depth-first search backtracks from a vertex v_{cy} in T_G that represents $\mathcal{C}$ only after all subtrees rooted in $\mathcal{C}$ are explored. Thus, we know the weight of all subtrees rooted in vertices $c_1, \dots, c_{i-1}$ when backtracking. The weight of c_0 is equal to n minus the sum of these sub-cactus weights.

Examining all cuts in the cycle would take i^3 time, but as we only want to find the most balanced cut, we can check only a subset of them, as shown in Algorithm 5. Q_1 and Q_2 are *queues*, thus elements are ordered and the following operations are supported: *queue* adds an element to the back of the queue, called the *tail* of the queue, *dequeue* removes the element at the front of the queue, called the *head* of the queue. We implicitly use the fact that queues can only be appended to, thus an element q was added to the queue after all elements that are closer to the head of the queue and before all elements that are closer to its tail.

The weight of a queue $w(Q)$ is denoted as the weight of its contents. For queue $Q = \{c_{j \mod i}, \dots, c_{k \mod i}\}$ with $0 \leq j \leq k$, we use the notation $w_{j \mod i, k \mod i}$ to denote the weight of Q and $\overline{w_{j \mod i, k \mod i}}$ as the weight of the queue that contains all cycle vertices not in Q.

Algorithm 5 Algorithm to find most balanced cut in cycle $\{c_0, \dots, c_{i-1}\}$

```
1:  procedure BalanceInCycle(G = (V, E), C = {c_1, ..., c_i})
2:      b_OPT ← 0
3:      Q_1 = Queue({})
4:      Q_2 = Queue({c_0, c_1, ..., c_{i-1}})
5:      while c_0 not Q_1.head() for second time do
6:          b_OPT ← checkBalance(Q_1, Q_2)
7:          if w(Q_1) > w(Q_2) then
8:              Q_2.queue(Q_1.dequeue())
9:          else
10:             Q_1.queue(Q_2.dequeue())
11:         end if
12:     end whilereturn b_OPT
13: end procedure
```

In every step of the algorithm, the cut represented by the current state of the queues consists of the two edges connecting the queue heads to the tails of the respective other queue. Initially Q_1 is empty and Q_2 contains all elements, in order from c_0 to c_{i-1}. In every step of the algorithm, we dequeue one element and queue it in the other queue. Thus, at every step each cycle vertex is in exactly one queue. When we check the balance of a cut, we compute the weight of each queue at the current point in time; and update b_{OPT}, the best balance found so far, if (Q_1, Q_2) is more balanced. As we only move one cycle vertex in each step, we can check the balance of an adjacent cut in constant time by adding and subtracting the weight of the moved vertex to the weights of each set.

Lemma 5.2.1. *Algorithm 5 terminates after $O(i)$ steps.*

Proof. In each step of Algorithm 5, one queue head is moved to the other queue. The algorithm terminates when c_0 is the head of Q_1 for the second time. In the first step, c_0 is moved to Q_1, as the empty queue Q_1 is the lighter one. The algorithm terminates after c_0 then performs a full round through both queues and is the head of Q_1 again. At termination, c_0 was thus moved a total of three times, twice from Q_2 to Q_1 and once the other way. As no element can 'overtake' c_0 in the queues, every vertex will be moved at most three times. Thus, we enter the loop at most $3i$ times, each time only using a constant amount of time. □

In Algorithm 5, we only check the balance of a subset of cuts represented by edges in the cycle C. Lemma 5.2.3 shows that none of the disregarded cuts can have balance better than b_{OPT} and we thus find the most balanced minimum cut. We call a cut disregarded if its balance was never checked (Line 6), and considered otherwise. In order to prove correctness of Algorithm 5, we first show the following Lemma:

Lemma 5.2.2. *Each vertex in the cycle is dequeued from Q_1 at least once in the algorithm.*

Proof. The algorithm terminates when c_0 is the head of Q_1 for the second time. For this, it needs to be moved from Q_2 to Q_1 twice. As we queue elements to the back of a queue, all vertices are dequeued from Q_2 before c_0 is dequeued from it for the second time. In order for c_0 to become the head of Q_1 again, all elements that were added beforehand need to be dequeued from Q_1. □

Lemma 5.2.3. *Algorithm 5 finds the most balanced minimum cut represented by cycle C.*

Proof. We now prove for each $c_l \in \mathcal{C}$ that all disregarded cuts containing the cycle edge separating c_l from $c_{(l-1) \bmod i}$ are not more balanced than the most balanced cut found so far. As no disregarded cut can be more balanced than the most balanced cut considered in the algorithm, the output of the algorithm is the most balanced minimum cut; or one of them if multiple cuts of equal balance exist.

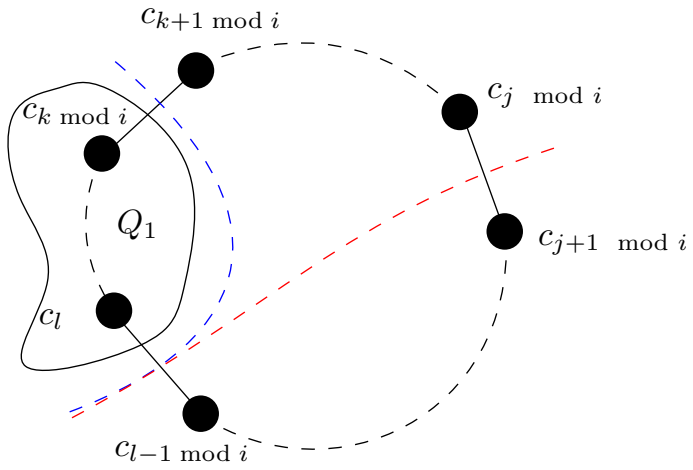


Figure 5.3: State of Q_1 at time t_l (cut in blue). Cut in red denotes cut considered at time t^*

.

Let t_l be the time that c_l becomes the head of Q_1 for the first time. Figure 5.3 shows the state of Q_1 at that point in time. Let $c_{k \bmod i}$ be the tail of Q_1 at time t_l for some integer k. Right before t_l, $c_{l-1 \bmod i}$ was head of the heavier queue Q_1 and thus dequeued, i.e. $Q_1 = \{c_{l-1 \bmod i}, \dots, c_{k \bmod i}\}$ has weight $w_{l-1 \bmod i, k \bmod i} \geq \overline{w_{l-1 \bmod i, k \bmod i}}$ and c_l is now head of Q_1.

From this point t_l the algorithm considers cuts that separate c_l from $c_{l-1 \bmod i}$. While Q_1 is not heavier than Q_2, we add more elements to the tail of Q_1 (and check the respective cuts) until Q_1 is the heavier queue. Let t^* be the time when this happens and $c_{j \bmod i}$ with $j \geq k$ be the tail of Q_1 at this point. Note that at time t^*,

c_l is about to be dequeued from Q_1. The red cut in Figure 5.3 shows the cut at time t^*, where $w_{c_l,c_{j \bmod i}} > \overline{w_{c_l,c_{j \bmod i}}}$.

We now prove that all cuts in which c_l is the head of Q_1 and its tail is not between $c_{k \bmod i}$ and $c_{j \bmod i}$ cannot be more balanced than the most balanced cut considered so far.

For all cuts where c_l is head of Q_1 and Q_1 also contains $c_{j+1 \bmod i}$, Q_1 is heavier than $w_{l,j \bmod i}$, as it contains all elements in $c_l, \dots, c_{j \bmod i}$ plus at least one more. As $w_{l,j \bmod i} > \overline{w_{l,j \bmod i}}$, i.e. Q_1 is already heavier when $c_{j \bmod i}$ is its tail, all of these cuts are less balanced than $(\{c_l, \dots, c_{j \bmod i}\}, \mathcal{C} \backslash \{c_l, \dots, c_{j \bmod i}\})$.

For the cuts in which $c_{k \bmod i}$ is in Q_2, i.e. Q_1 is lighter than at time t_l, we need to distinguish two cases, depending on whether $w_{l,k \bmod i}$ is larger than $\overline{w_{l,k \bmod i}}$ or not.

If $w_{l,k \bmod i} \leq \overline{w_{l,k \bmod i}}$, all cuts in which c_l is the head of Q_1 and $c_{k \bmod i}$ is in Q_2 are less balanced than $(\{c_l, \dots, c_{k \bmod i}\}, \mathcal{C} \backslash \{c_l, \dots, c_{k \bmod i}\})$, as Q_1 is lighter than it is at t_l, where it was already not the heavier queue.

If $w_{l,k \bmod i} > \overline{w_{l,k \bmod i}}$, there might be cuts in which c_l is the head of Q_1 that are more balanced than $(\{c_l, \dots, c_{k \bmod i}\}, \mathcal{C} \backslash \{c_l, \dots, c_{k \bmod i}\})$ in which Q_1 is lighter than at time t_l. Thus, consider time t' when $c_{k \bmod i}$ was added to Q_1. Such a time must exist, since Q_1 is initially empty. As $c_{k \bmod i}$ is already the tail of Q_1 at time t_l, $t' < t_l$. At that time Q_1 contained $c_{l-1 \bmod i}, \dots, c_{k-1 \bmod i}$ and potentially more vertices.

Still, $w_{l-1 \bmod i,k-1 \bmod i} \leq \overline{w_{l-1 \bmod i,k-1 \bmod i}}$, as otherwise $c_{k \bmod i}$ would not have been added to Q_1. Obviously $w_{l-1 \bmod i,k-1 \bmod i} > w_{l,k-1 \bmod i}$, as Q_1 is even lighter when $c_{l-1 \bmod i}$ is dequeued. As $w_{l-1 \bmod i,k-1 \bmod i}$ is already not heavier than its complement, $(\{c_l, \dots, c_{k-1 \bmod i}\}, \mathcal{C} \backslash \{c_l, \dots, c_{k-1 \bmod i}\})$ is more imbalanced than the cut examined just before time t'. Thus, all cuts where c_l is the head of Q_1 and $c_{k-1 \bmod i}$ is in Q_2 are even more imbalanced, as Q_1 is even lighter.

Coming back to the outline shown in Figure 5.3, we showed that for all cuts in which c_l is head of Q_1 and Q_1 is lighter than at time t_l (left of blue cut) and all cuts where Q_1 is heavier than at time t^* (below red cut) can be safely disregarded, as a more balanced cut than any of them was considered at some point between t' and t^*. The algorithm considers next all cuts with c_l as head of Q_1 and the tail of Q_1 between $c_{k \bmod i}$ and $c_{j \bmod i}$. Thus, the algorithm will return a cut that is at least as balanced as the most balanced cut that separates c_l and $c_{l-1 \bmod i}$. This is true for every cycle vertex $v_l \in \mathcal{C}$, which concludes the proof. □

This allows us to perform the depth-first search and find the most balanced minimum cut in C_G in time $\mathcal{O}(n^* + m^*)$. This algorithm can be adapted to find the minimum cut of any other optimization function of a cut that only depends on the (weight of the) edges on the cut and the (weight of the) vertices on either side of the cut. In order to retain the linear running time of the algorithm, the function

needs to be evaluable in constant time on a neighboring cut. For example, we can find the minimum cut of lowest conductance. The conductance of a cut $(S, V \backslash S)$ is defined as $\frac{\lambda(S,(V \backslash S))}{min(a(S),a(V \backslash S))}$, where $a(S)$ is the sum of degrees for all vertices in set S. Note that this is not the minimum conductance cut problem, which is NP-hard [10], as we only look at the minimum cuts. To find the minimum cut of lowest conductance, we set the weight of a vertex $v_{C_G} \in C_G$ to the sum of vertex degrees encompassed in v_{C_G}. Otherwise the algorithm remains the same.

5.3 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 minimum cut algorithm in a non-parallel setting, i.e. we compare different variants for edge selection and see the impact of the various optimizations detailed in this work. Afterwards, we report parallel speedup on a variety of large graphs.

Experimental Setup and Methodology

We implemented the algorithms using C++-17 and compiled all code using g++ version 8.3.0 with full optimization (`-O3`). Our experiments are conducted on a machine with two Intel Xeon Gold 6130 processors 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 our 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 significantly in cut size and time.

Instances

We use a variety of graphs from the 10th DIMACS Implementation challenge [15] and the SuiteSparse Matrix Collection [46]. These are social graphs, web graphs, co-purchase matrices, cooperation networks and some generated instances. If a network has multiple connected components, we run on the largest. The list of graphs can be found in Section 2.5, where graph family (2A) shows a set of smaller instances and graph family (2B) shows a set of larger and harder to solve instances.

5.3.1 Edge Selection

Figure 5.4 shows the results for graph family (2A). We compute the cactus graph representing all minimum cuts using the edge selection variants `Random`, `Central`, `Heavy` and `HeavyWeighted`, as detailed in Section 5.1.2. As we want a majority of the running time in the algorithm of Nagamochi et al. [146], where we actually select edges, we run a variant of our algorithm

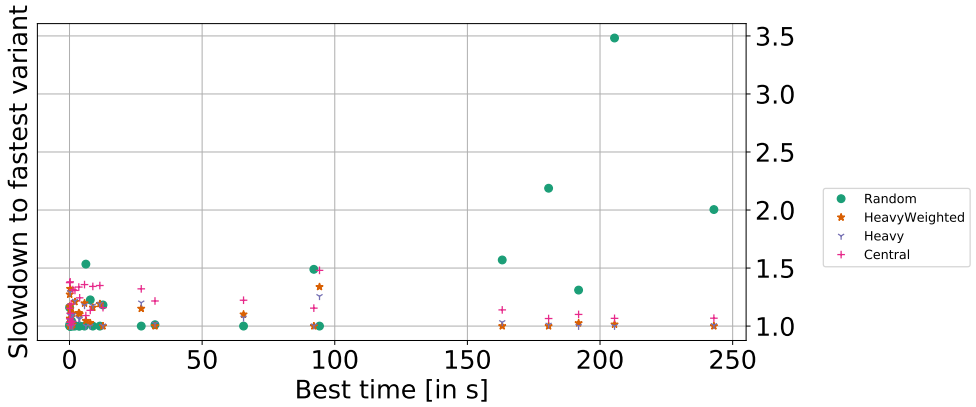


Figure 5.4: Effect of edge selection strategies.

that only contracts edges using connectivity-based contraction and then runs the algorithm of Nagamochi et al. [146].

We can see that in the graphs which cannot be contracted quickly, `Random` is significantly slower than all other variants. On `cnr-2000`, `Random` takes over 700 seconds in average, whereas all other variants finish in approximately 200 seconds. This happens independently of the random seed used, there is no large deviation in the running time on any of the graphs. On almost all graphs, the variants `Heavy` and `HeavyWeighted` are within 3% of each other, which is not surprising, as the variants are almost identical. While it optimizes for 'edge centrality' very directly, `Central` has two iterations of breadth-first search in each edge selection and thus a sizable overhead. For this reason it is usually $5 - 15\%$ slower than `Heavy` and is not the fastest algorithm on any graph. On graphs with large n^*, all three variants manage to shrink the graph significantly faster than `Random`.

On graphs with a low value of n^*, we can see that `Random` is slightly faster than the other variants. There is no significant difference in the shrinking of the graph, as almost all selected edges have connectivity larger than λ and thus only trigger a single edge contraction anyway. Thus, not spending the extra work of finding a 'good' edge results in a slightly lower running time. In the following we will use variant `Heavy`, which is the only variant that is never more than 30% slower than the fastest variant on any graph.

5.3.2 Optimization

We now examine the effect of the different optimizations. For this purpose, we benchmarks different variants on a variety of graphs. We hereby compare the following variants that build on one another: as a baseline, `BasicCactus` runs the algorithm of Nagamochi, Nakao and Ibaraki [146] using `Heavy` edge selection on the input graph. `+Connectivity` additionally runs `VieCut` [94] to find an upper bound

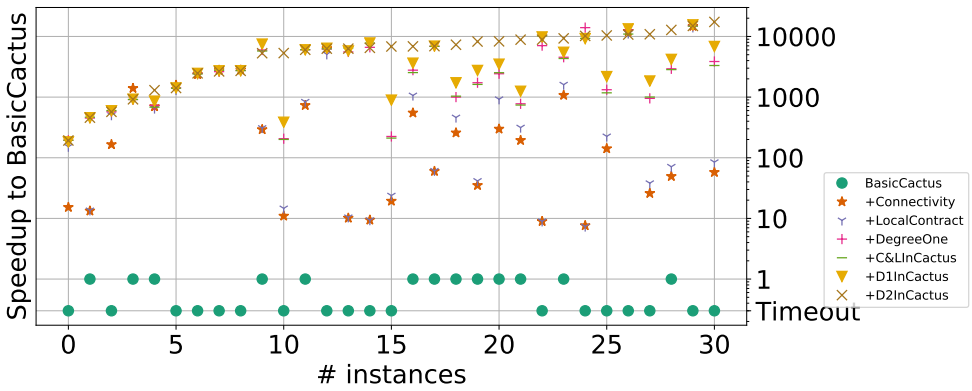


Figure 5.5: Speedup to `BasicC` on small graphs (Table 2.5, graph family 2A)

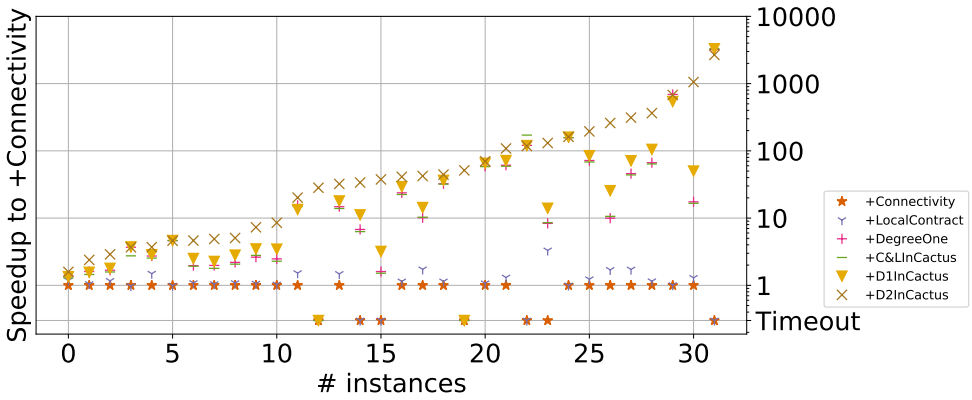


Figure 5.6: Speedup to `+Conn` on large graphs (Table 2.5, graph family 2B)

for the minimum cut and uses this to contract high-connectivity edges as described in Section 5.1.1. In addition to this, `+LocalContract` also contracts edges whose neighborhood guarantees that they are not part of any minimum cut, as described in Section 5.1.1 and Lemma 5.1.2. `+DegreeOne` runs also the last remaining contraction routine from Algorithm 4, contraction and re-insertion of degree-one vertices as described in Section 5.1.1. `+C&LInCactus` additionally runs high-connectivity and local contraction in every tenth recursion step. `+D1InCactus` additionally contracts and re-inserts degree-one vertices in every recursion step. `FullAlgorithm` also runs the degree-two contraction as described in Section 5.1.2. We compare these variants on the graph families (2A) and (2B) of Table 2.5. We use a timeout of 30 minutes for each problem. If the baseline algorithm does not finish in the allotted time, we report speedup to the timeout, so a lower bound for the actual speedup.

Figure 5.5 shows the speedup of all variants to the baseline `BasicCactus` on all small graphs. We can see that already just adding `+Connectivity` gives a speedup of more than an order of magnitude for each of the graphs in the dataset. Most of the

Table 5.1: Huge social and web graphs. n^* denotes number of vertices in cactus graph, max n and max m denote size of smaller block in most balanced cut

Name	n	m	n^*	λ	max. n	max. m	seq. t	par. t
friendster	65.6M	1.81B	13.99M	1	897	1 793	1266.35s	138.34s
twitter7	41.7M	1.20B	1.93M	1	47	1 893	524.86s	72.51s
uk-2007-05	104.3M	3.29B	9.66M	1	49 984	13.8M	229.18s	40.16s

other optimizations manage to improve the running time of at least some instances by a large margin. Especially +DegreeOne, which is the first contraction for edges that are in a minimum cut, has speedups of multiple orders of magnitude in some instances. This is the case as minimum cut edges that are incident to a degree-one vertex previously incur a flow problem on the whole graph each. However, it is very easy to see that the edge will be part of exactly one minimum cut, thus we can contract and re-insert it in constant time. Especially in graphs whose minimum cut is 1, all edges can be quickly contracted, as they will either be incident to a degree-one vertex or be quickly certified to have a connectivity value of > 1.

While rerunning `Connectivity` and `LocalContract` inside of the recursive algorithm of Nagamochi et al. [146] does usually not yield a large speedup, many graphs develop degree-one vertices by having their whole neighborhood contracted. Thus, +D1InCactus has a significant speedup for most graphs in which n^* is sufficiently large. `FullAlgorithm` has an even larger speedup on these graphs, even when the minimum cut is significantly higher than 2, as there are often cascading effects where the contraction of an edge incident to a degree-two vertex often lowers the degree of neighboring vertices to two.

Figure 5.6 shows the speedup of all variants on large graphs. As `BasicCactus` is not able to solve any of these instances in 30 minutes, we use +Connectivity as a baseline. The results are similar to Figure 5.5, but we can see even clearer how useful the contraction of degree-two vertices is in finding all minimum cuts: `FullAlgorithm` often has a speedup of more than an order of magnitude to all other variants and is the only variant that never times out.

5.3.3 Shared-memory Parallelism

Table 5.1 shows the average running times of our algorithm both sequential and with 16 threads on huge social and web graphs. Each of these graphs has more than a billion of edges and more than a million vertices in the cactus graph depicting all minimum cuts. On these graphs we have a parallel speedup factor of 5.7x to 9.1x using 16 threads. On all of these graphs, a large part of the running time is spent in the first iteration of the kernelization routines, which already manages to contract most dense blocks in the graph. Thus, all subsequent operations can be performed on significantly smaller problems and are therefore much faster.

5.4 Conclusion

We engineered an algorithm to find all minimum cuts in large undirected graphs. Our algorithm combines multiple kernelization routines with an engineered version of the algorithm of Nagamochi, Nakao and Ibaraki [146] to find all minimum cuts of the reduced graph. Our experiments show that our algorithm can find all minimum cuts of huge social networks with up to billions of edges and millions of minimum cuts in a few minutes on shared memory. We found that especially the contraction of high-connectivity edges and efficient handling of low-degree vertices can give huge speedups. Additionally we give a linear time algorithm to find the most balanced minimum cut given the cactus graph representation of all minimum cuts. Future work includes finding all near-minimum cuts.

CHAPTER 6

Dynamic Minimum Cut

In this chapter, we give the first implementation of a *fully-dynamic algorithm* for the *minimum cut problem* in a weighted graph. Our algorithm maintains an exact global minimum cut under edge insertions and deletions. For edge insertions, we use the approach of Henzinger [96] and Goranci et al. [79], who maintain a compact data structure of all minimum cuts in a graph and invalidate only the minimum cuts that are affected by an edge insertion. We use the algorithm presented in Chapter 5 to compute all minimum cuts in a graph. For edge deletions, we use the push-relabel algorithm of Goldberg and Tarjan [77] to certify whether the previous minimum cut is still a minimum cut. As we only need to certify whether an edge deletion changes the value of the minimum cut, we can perform optimizations that significantly improve the speed of the push-relabel algorithm for our application. In particular, we develop a fast initial labeling scheme and terminate early when the connecitivity value is certified.

An important observation for dynamic minimum cut algorithms is that graphs often have a large set of global minimum cuts. We can see this in the experimental section of Chapter 5, where we aim to find all minimum cuts in huge graphs. Thus, dynamic minimum cut algorithms can avoid costly recomputation by storing a compact data structure representing all minimum cuts [79, 96] and only invalidate changed cuts in edge insertion. The data structure we use is a *cactus graph*, i.e. a graph in which every vertex is part of at most one cycle. A minimum cut in the cactus graph is represented by either a tree edge or two edges of the same cycle. For a graph with multiple connected components, i.e. a graph whose minimum cut value $\lambda = 0$, the cactus graph $\mathcal{C}$ has an empty edge set and one vertex corresponding to each connected component.

The content of this chapter is based on [89].

The rest of this chapter is organized as follows. We start by explaining the incremental minimum cut algorithm in Section 6.1, followed by a description of the

decremental minimum cut algorithm in Section 6.2. In Section 6.3, we show how to combine the routines into a fully dynamic minimum cut algorithm. In Section 6.4, we perform an experimental evaluation of the algorithms detailed in this chapter.

6.1 Incremental Minimum Cut

For incremental minimum cuts, our algorithm is closely related to the exact incremental dynamic algorithms of Henzinger [96] and Goranci et al. [79]. Upon initialization of the algorithm with graph G, we run the algorithm detailed in Chapter 5 on G to find the weight of the minimum cut λ and the cactus graph $\mathcal{C}$ representing all minimum cuts in G. Each minimum cut in $\mathcal{C}$ corresponds to a minimum cut in G and each minimum cut in G corresponds to one or more minimum cuts in $\mathcal{C}$ [96].

The insertion of an edge $e = (u, v)$ with positive weight $c(e) > 0$ increases the weight of all cuts in which u and v are in different partitions, i.e. in different vertices of the cactus graph $\mathcal{C}$. The weight of cuts in which u and v are in the same partition remains unchanged. As edge weights are non-negative, no cut weight can be decreased by inserting additional edges.

If $\Pi(u) = \Pi(v)$, i.e. both vertices are mapped to the same vertex in $\mathcal{C}$, there is no minimum cut that separates u and v and all minimum cuts remain intact. If $\Pi(u) \neq \Pi(v)$, i.e. the vertices are mapped to different vertices in $\mathcal{C}$, we need to invalidate the affected minimum cuts by contracting the corresponding edges in $\mathcal{C}$.

6.1.1 Path Contraction

Dinitz [51] shows that for a connected graph with $\lambda > 0$ the minimum cuts that are affected by the insertion of (u, v) correspond to the minimum cuts on the path between $\Pi(u)$ and $\Pi(v)$. We find the path using alternating breadth-first searches from $\Pi(u)$ and $\Pi(v)$. For this path-finding algorithm, imagine the cactus graph $\mathcal{C}$ as a tree graph in which each cycle is contracted into a single vertex. On this tree, there is a unique path from $\Pi(u)$ to $\Pi(v)$.

For every cycle in $\mathcal{C}$ that contains at least two vertices of the path between $\Pi(u)$ and $\Pi(v)$, the cycle is "squeezed" by contracting the first and last path vertex in the cycle, thus creating up to two new cycles. Figure 6.1 shows an example in which a cycle is squeezed. In Figure 6.1, the cycle is squeezed by contracting the bottom left and top right vertices. This creates a new cycle of size 3 and a "cycle" of size 2, which is simply a new tree edge in the cactus graph $\mathcal{C}$. For details and correctness proofs we refer the reader to the work of Dinitz [51]. The intuition is that due to the insertion of the new edge, all cactus vertices in the path from $\Pi(u)$ and $\Pi(v)$ are now connected with a value $> \lambda$, as their previous connection was λ and the newly introduced edge increased it. For any cycle in the path, this also includes the first and last cycle vertices x and y in the path, as these two vertices now have a higher connectivity $\lambda(x, y)$. The minimum cuts that are represented by edges in this cycle that have x and y on the same side are unaffected, as all vertices in the path from

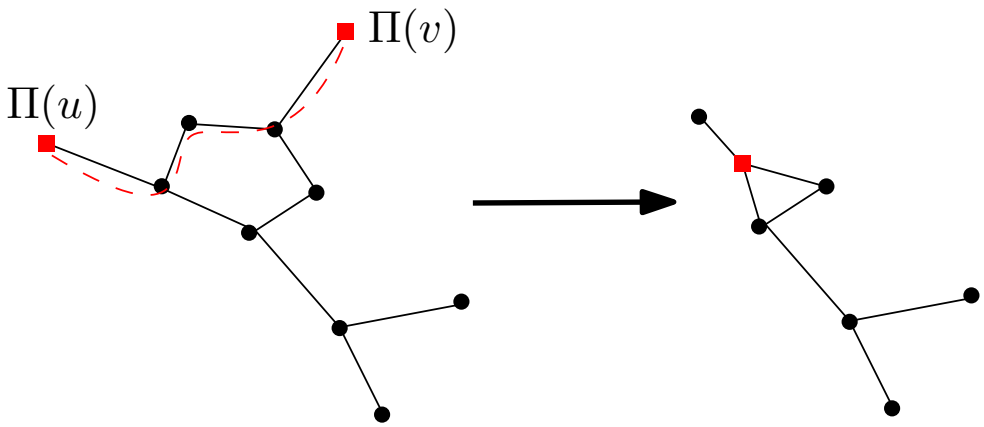


Figure 6.1: Insertion of edge $e = (u, v)$ - contraction of path in $\mathcal{C}$, squeezing of cycle

$\Pi(u)$ and $\Pi(v)$ are on the same side of this cut. As this is not true for cuts that separate x and y, we merge x and y (as well as the rest of the path from $\Pi(u)$ to $\Pi(v)$), which "squeezes" the cycle and creates up to two new cycles.

If the graph has multiple connected components, i.e. the graph has a minimum cut value $\lambda = 0$, $\mathcal{C}$ is a graph with no edges where each connected component is mapped to a vertex. The insertion of an edge between different connected components $\Pi(u)$ and $\Pi(v)$ merges the two vertices representing the connected components, as they are now connected.

If $\mathcal{C}$ has at least two non-empty vertices after the edge insertion, there is at least one minimum cut of value λ remaining in the graph, as all minimum cuts that were affected by the insertion of edge e were just removed from the cactus graph $\mathcal{C}$. As an edge insertion cannot decrease any connectivities, λ remains the value of the minimum cut. If $\mathcal{C}$ only has a single non-empty vertex, we need to recompute the cactus graph $\mathcal{C}$ using the algorithm detailed in Chapter 5.

Checking the set affiliation Π of u and v can be done in constant time. If $\Pi(u) = \Pi(v)$ and the cactus graph does not need to be updated, no additional work needs to be done. If $\Pi(u) \neq \Pi(v)$, we perform breadth-first search on $\mathcal{C}$ with $n^* := |V(\mathcal{C})|$ and $m^* := |E(\mathcal{C})|$ which has a asymptotic running time of $\mathcal{O}(n^* + m^*) = \mathcal{O}(n^*)$, contract the path from $\Pi(u)$ to $\Pi(v)$ in $\mathcal{O}(n^*)$ and then update the set affiliation of all contracted vertices. This update has a worst-case running time of $\mathcal{O}(n)$, however, contracting all vertices of the path from $\Pi(u)$ to $\Pi(v)$ into the cactus graph vertex that already corresponds to the most vertices of G, we often only need to update the affiliation of a few vertices. Both the initial computation and a full recomputation of the minimum cut cactus have a worst-case running time of $\mathcal{O}(nm + n^2 \log n + n^* m \log n)$.

6.2 Decremental Minimum Cut

The deletion of an edge $e = (u, v)$ with positive weight $c(e) > 0$ decreases the weight of all cuts in which u and v are in different partitions. This might lead to a decrease of the minimum cut value λ and thus the invalidation of the minimum cuts in the existing minimum cut cactus $\mathcal{C}$. The value of the minimum cut $\lambda(G, u, v)$ that separates vertices u and v is equal to the maximum flow between them and can be found by a variety of algorithms [50, 63, 77]. In order to check whether λ is decreased by this edge deletion, we need to check whether $\lambda(G-e, u, v) < \lambda(G)$. For this purpose, we use the push-relabel algorithm of Goldberg and Tarjan [77] which aims to push flow from u to v until there is no possible path remaining. We first give a short description of the push-relabel algorithm and then show the adaptions we performed to improve its performance in our application.

6.2.1 Push-relabel algorithm

In this work we use and adapt the push-relabel algorithm of Goldberg and Tarjan [77] for the minimum s-t-cut problem. The algorithm aims to push as much flow as possible from the *source* vertex s to the *sink* vertex t and returns the value of the maximum flow between s and t, which is equal to the value of the minimum cut separating them [45]. We now give a brief description of the algorithm, for more details we refer the reader to the original work [77].

Let $G = (V, E, c)$ be a directed edge-weighted graph. An undirected edge $e = (u, v)$ is hereby interpreted as two symmetric directed edges (u, v) and (v, u) with $c(e) = c(u, v) = c(v, u)$. In the push-relabel algorithm, each vertex $v \in V$ has a *distance* or *height label* $d(x)$, initially $d(x) = 0$ for every vertex except $d(s) = n$. The algorithm handles a *preflow*, a function f so that for each edge e, $0 \geq f(e) \geq c(e)$ and for each $v \in V \backslash s$, $\sum_{(v,x)\in E} f((v, x)) \leq \sum_{(y,v)\in E} f((y, v))$ there is at least as much ingoing as outgoing flow. The difference in ingoing and outgoing flow in a vertex is called the *excess flow* of this vertex.

First, the algorithm pushes flow from s to all neighboring vertices, afterwards vertices push their excess flow to neighbors with a lower distance d. If a vertex v has positive excess but no neighbors with a lower distance, the *relabel* function increases the distance of v until at least one outgoing preflow f can be increased. At termination, the push-relabel algorithm reaches a *flow*, where each edge e has $0 \leq f(e) \leq c(e)$ units of flow and the excess of each vertex except s and t is 0. The value of the minimum cut $\lambda(s, t)$ separating s and t is equal to the excess flow on t. Inherent to the push-relabel algorithm is the *residual graph* $G_f = (V, E_f)$ for a given preflow f, where E_f contains all edges $e = (u, v) \in E$ with $f(e) < c(e)$, i.e. edges that have capacity to handle additional flow, and a reverse-edge for every edge where $0 < f(e)$.

6.2.2 Early Termination

We terminate the algorithm as soon as $\lambda(G)$ units of flow reached v. If $\lambda(G)$ units of flow from u reached v, we know that $\lambda(G-e,u,v) \geq \lambda(G)$, i.e. the connectivity of u and v on $G-e$ is at least as large as the minimum cut on G, the minimum cut value λ remains unchanged. Note that iff $\lambda(G-e,u,v) = \lambda(G)$, the deletion of e introduces one or more new minimum cuts. We do not introduce these new cuts to $\mathcal{C}$. The trade-off hereby is that we are able to terminate the push-relabel algorithm earlier and do not need to perform potentially expensive operations to update the cactus, but do not necessarily keep all cuts and have to recompute the cactus earlier. As most real-world graphs have a large number of minimum cuts, there are far more edge deletions than recomputations of $\mathcal{C}$.

Each edge deletion calls the push-relabel algorithm using the lowest-label selection rule with a worst-case running time of $\mathcal{O}(n^2m)$ [77]. The lowest-label selection rule picks the active vertices whose distance label is lowest, i.e. a vertex that is close to the sink v. Using highest-level selection would improve the worst-case running time to $\mathcal{O}(n^2\sqrt{m})$, but we aim to push as much flow as possible to the sink early to be able to terminate the algorithm early as soon as λ units of flow reach the sink. Using lowest-level selection prioritizes the vertices close to the sink and thus increases the amount of flow which reaches the sink at a given point in time. Preliminary experiments show faster running times using the lowest-level selection rule.

6.2.3 Decremental Rebuild of Cactus Graph

If the push-relabel algorithm finishes with a value of $< \lambda(G)$, we update the minimum cut value $\lambda(G-e)$ to $\lambda(G-e,u,v)$. As the minimum cut value changed by the deletion of e and this deletion only affects cuts which contain e, we know that all minimum cuts of the updated graph $G-e$ separate u and v. We use this information to significantly speed up the cactus construction. Instead of running the full algorithm from Chapter 5, we run only the subroutine which is used to compute the (u,v)-cactus, i.e. the cactus graph which contains all cuts that separate u and v, as we know that all minimum cuts of $G-e$ separate u and v. This routine, developed by Nagamochi and Kameda [145], finds a u-v-cactus a running time of $\mathcal{O}(n+m)$.

Note that the routine of Nagamochi and Kameda [145] only guarantees to find all minimum u-v-cuts if an edge $e=(u,v)$ with $c(e)>0$ exists ([145, Lemma 3.4]). As this edge was just deleted in $G-e$ and therefore does not exist, it is possible that *crossing* u-v-cuts $(X,\overline{X})$ and $(Y,\overline{Y})$ with $u \in X$ and $u \in Y$ exist. Two cuts are *crossing*, if both $(\overline{X} \cap Y)$ and $(Y \cap \overline{X})$ are not empty. As we only find one cut in a pair of crossing cuts, the u-v-cactus is not necessarily maximal. However, the operation is significantly faster than recomputing the complete minimum cut cactus in which almost all edges are not part of any minimum cut. While it is not guaranteed that the decremental rebuild algorithm finds all minimum cuts in $G-e$, every cut of size $\lambda(G-e,u,v)$ that is found is a minimum cut. As we build the

minimum cut cactus out of minimum cuts, it is a valid (but potentially incomplete) minimum cut cactus and the algorithm is correct.

6.2.4 Local Relabeling

Many efficient implementations of the push-relabel algorithm use the global relabeling heuristic [39] in order to direct flow towards the sink more efficiently. The push-relabel algorithm maintains a distance label d for each vertex to indicate the distance from that vertex to the sink using only edges that can receive additional flow. The global relabeling heuristic hereby periodically performs backward breadth-first search to compute distance labels on all vertices.

This heuristic can also be used to set the initial distance labels in the flow network for a flow problem with source u and sink v. This has a running time of $\mathcal{O}(n + m)$ but helps lead the flow towards the sink. As our algorithm terminates the push-relabel algorithm early, we try to avoid the $\mathcal{O}(m)$ running time while still giving the flow some guidance. Thus, we perform *local relabeling* with a *relabeling depth* of γ for $\gamma \in [0, n)$, where we set $d(v) = 0$, $d(u) = n$ and then perform a backward breadth-first search around the sink v, in which we set $d(x)$ to the length of the shortest path between x and v (at this point, there is no flow in the network, so every edge in G is admissible). Instead of setting the distance of every vertex, we only explore the neighborhoods of vertices x with $d(x) < \gamma$, thus we only set the distance-to-sink for vertices with $d(x) \leq \gamma$. For every vertex y with a higher distance, we set $d(y) = (\gamma + 1)$. This results in a running time for setting the distance labels of $\mathcal{O}(n)$ plus the time needed to perform the bounded-depth breadth-first search.

This process creates a "funnel" around the sink to lead flow towards it, without incurring a running time overhead of $\Theta(m)$ (if γ is set sufficiently low). Note that this is useful because the push-relabel algorithm is terminated early in many cases and thus initializing the distance labels faster can give a large speedup. We give experimental results for different relabeling depths γ for local relabeling in our application in Section 6.4.1.

Correctness

Goldberg and Tarjan show that each push and relabel operation in the push-relabel algorithm preserve a *valid labeling* [77]. A valid labeling is a labeling d, where in a given preflow f and corresponding residual graph G_f, for each edge $e = (u, v) \in E_f$, $d(u) \leq d(v) + 1$. We therefore need to show that the labeling d that is given by the initial local relabeling is a valid labeling.

Lemma 6.2.1. *Let $G = (V, E, c)$ be a flow-graph with source s and sink t and let d be the vertex labeling given by the local relabeling algorithm. The vertex labeling d is a valid labeling.*

Proof. The vertex labeling d is generated using breadth-first search. Thus, for every edge $e = (u, v)$ where $u \neq s$ and $v \neq s$, $|d(u) - d(v)| \leq 1$. We prove this by contradiction. W.l.o.g. assume that $d(u) - d(v) > 1$. As $u \neq s$ and s is the only vertex with $d(s) > \gamma$, $d(u) \leq \gamma + 1$ and $d(v) < \gamma$. Thus, at some point of the breadth-first search, we set the distance labels of all neighbors of v that do not yet have a distance label to $d(v) + 1$. As edge $e = (u, v)$ exists, u and v are neighbors and the labeling sets $d(u) = d(v) + 1$. This contradicts $d(u) - d(v) > 1$.

This shows that the labeling is valid for every edge not incident to the source s, as distance labels of incident non-source vertices differ by at most 1. The only edges we need to check are edges incident to s. In the initialization of the push-relabel algorithm, all outgoing edges of the source s are fully saturated with flow and are thus no outgoing edge of s is in E_f. For ingoing edges $e = (v, s)$, we know that $0 \leq d(v) \leq \gamma + 1 = n$ and thus know that $d(v) \leq d(s)$. Thus e respects the validity of labeling d. □

Lemma 6.2.1 shows that local relabeling gives a valid labeling; which is upheld by the operations in the push-relabel algorithm [77]. Thus, correctness of the modified algorithm follows from the correctness proof of Goldberg and Tarjan.

Resetting the vertex data structures can be performed in $\mathcal{O}(n)$, however there are m edges whose current flow needs to be reset to 0. Using early termination we hope to solve some problems very fast in practice, as we can sometimes terminate early without exploring large parts of the graph. Thus, resetting of the edge flows in $\mathcal{O}(m)$ is a significant problem and is avoided using implicit resetting as described in the following paragraph.

Each flow problem that is solved over the course of the dynamic minimum cut algorithm is given a unique ID, starting at an arbitrary integer and incrementing from there. In addition to the current flow on an edge, we also store the ID of the last problem which accessed the flow on this edge. When the flow of an edge is read or updated in a flow problem, we check whether the ID of the last access equals the ID of the current problem. If they are equal, we simply return or update the flow value, as the edge has already been accessed in this flow problem and does not need to be reset. Otherwise, we need to reset the edge flow to 0 and set the problem ID to the ID of the current problem and then perform the operation on the updated edge. Thus, we implicitly reset the edge flow on first access in the current problem. As we increment the flow problem ID after every flow problem, no two flow problems share the same ID.

Using this implicit reset of the edge flows saves $\mathcal{O}(m)$ overhead but introduces a constant amount of work on each access and update of the edge flow. It is therefore useful in practice if the problem terminates with significantly fewer than m flow updates due to early termination. It does not affect the worst-case running time of the algorithm, as we only perform a constant amount of work on each edge

update. The running time of the initialization of the implementation is improved from $\mathcal{O}(n+m)$ to $\mathcal{O}(n)$, as we do not explicitly reset the flow on each edge.

6.3 Fully Dynamic Minimum Cut

Based on the incremental and decremental algorithm described in the preceding sections, we now describe our fully dynamic algorithm. As the operations in the previous section each output the minimum cut $\lambda(G)$ and a corresponding cut cactus $\mathcal{C}$ that stores a set of minimum cuts for G, the algorithm gives correct results on all operations. However, there are update sequences in which every insertion or deletion changes the minimum cut value and, thus, triggers a recomputation of the minimum cut cactus $\mathcal{C}$. One such example is the repeated deletion and reinsertion of an edge that belongs to a minimum cut. In the following paragraphs we describe a technique that is used to mitigate such worst-case instances. Nevertheless, it is still possible to construct update sequences in which the minimum cut cactus $\mathcal{C}$ needs to be recomputed every $\mathcal{O}(1)$ edge updates and thus the worst-case asymptotic running time per update is equal to the running time of the static algorithm.

6.3.1 Cactus Cache

Computing the minimum cut cactus $\mathcal{C}$ is expensive if there is a large set of minimum cuts and the cactus is therefore large. Thus, it is beneficial to reduce the amount of recomputations to speed up the process. On some fully dynamic workloads, the minimum cut often jumps between values λ_1 and λ_2 with $\lambda_1 > \lambda_2$, where the minimum cut cactus for cut value λ_1 is large and thus expensive to recompute whenever the cut value changes.

A simple example workload is a large unweighted cycle, which has a minimum cut of 2. If we delete any edge, the minimum cut value changes to 1, as the incident vertices have a degree of 1. By reinserting the just-deleted edge, the minimum cut changes to a value of 2 again and the minimum cut cactus is equal to the cactus prior to the edge deletion. Thus we can save a significant amount of work by caching and reusing the previous cactus graph when the minimum cut is increased to 2 again.

Reuse Cactus Graph from Cactus Cache

Whenever the deletion of an edge e from graph G decreases the minimum cut value from λ_1 to λ_2, we cache the previous cactus $\mathcal{C}$. After this point, we also remember all edge insertions, as these can invalidate minimum cuts in $\mathcal{C}$. If at a later point the minimum cut is again increased from λ_2 to λ_1 and the number of edge insertions divided by the number of vertices in $\mathcal{C}$ is smaller than a parameter δ, we recreate the cactus graph from the cactus cache instead of recomputing it. The default value for δ is 2. The algorithm does not store the intermediate edge deletion, as there can only lower connectivities and by computing the minimum cut value we know that there is no cut of value $< \lambda_1$ and thus all cuts of value λ_1 are global minimum cuts.

For each edge insertion since caching the cactus we perform the edge insertion operation from Section 6.1 to eliminate all cuts that are invalidated by the edge insertion. All cuts that remain in $\mathcal{C}$ are still minimum cuts. If there are only a small amount of edge insertions since the cactus was cached, this is significantly faster than recomputing the cactus from scratch. As we do not remember edge deletions, the cactus might not contain all minimum cuts and thus require slightly earlier recomputation.

6.4 Experiments and Results

We now perform an experimental evaluation of the proposed algorithms. This is done in the following order. We use the static and dynamic graph instances detailed in Section 2.5 and Table 2.1. In Section 6.4.1, we analyze the impact of local relabeling on the static preflow-push algorithm to determine with value of the relabeling depth to use in the experiments on dynamic graphs. Then, in Sections 6.4.2 and 6.4.3, we evaluate our dynamic algorithms on a wide variety of instances. In Section 6.4.4, we generate a set of worst-case problems and use these to evaluate the performance of our algorithm on instances that were specifically created to be difficult.

Experimental Setup and Methodology

We implemented the algorithms using C++-17 and compiled all code using g++ version 8.3.0 with full optimization (`-O3`). Our experiments are conducted on a machine with two Intel Xeon Gold 6130 processors with 2.1GHz with 16 CPU cores each and 256 GB RAM in total. In this section, we first describe our 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 significantly in cut size and time.

6.4.1 Local Relabeling

In order to examine the effects of local relabeling with different values of relabeling depth γ, we run experiments using all static graph instances (Graph Family A and Graph Family B) from Table 2.1, in which we delete 1000 random edges in random order. We report the total time spent executing delete operations. We compare a total of 5 variants, one that does not run initial relabeling, three variants with relabeling depth $\gamma = 0, 1, 2$ and one variant which performs global relabeling in the initialization process, i.e. local relabeling with depth $\gamma = (n - 1)$. Local relabeling with $\gamma = 0$ is very similar to no relabeling, however the distance value of non-sink vertices are set to $(\gamma + 1) = 1$ and not to 0.

In Figure 6.2, we report the slowdown to the fastest variant for all static graph instances from Table 2.1. The x-axis shows the average vertex degree for the instances. On most instances, the fastest variant is local relabeling with $\gamma = 1$. Depending on

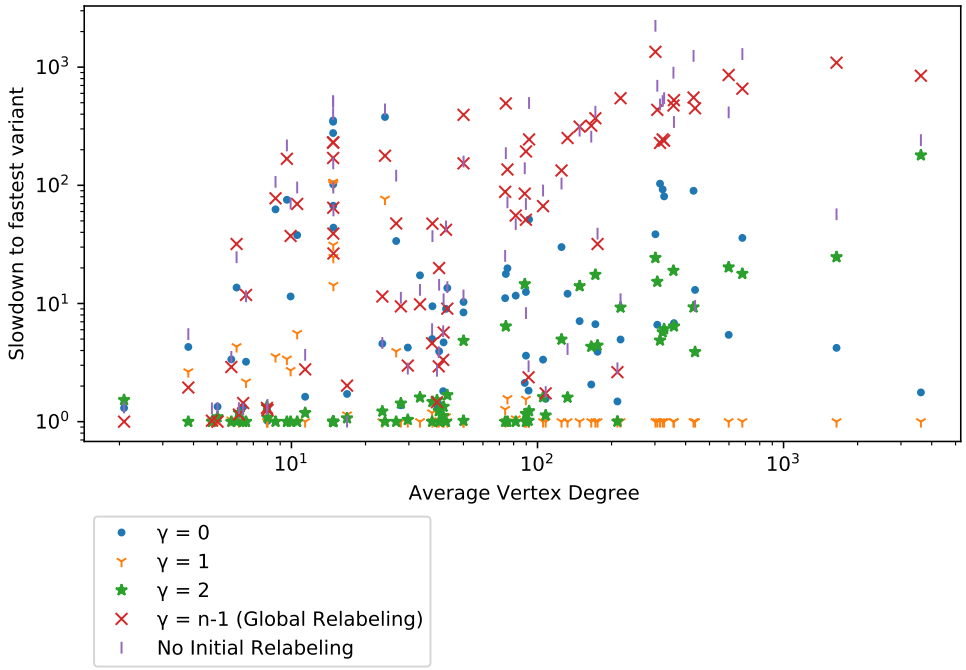


Figure 6.2: Effect of local relabeling depth on running time of delete operations.

the graph instance, this variant spends $25 - 90\%$ of the deletion time in the initialization (including initial relabeling). An increase in labeling depth increases the initialization running time, but decreases the subsequent algorithm running time. Thus we aim to find a labeling depth value that maintains some balance between initial labeling and the subsequent algorithm execution. On some instances, it is outperformed by local relabeling with $\gamma = 2$, which is slower by a factor of $3 - 10$x on most instances, with $90 - 99\%$ of the total running time spent in the initialization of the algorithm. We can see that in instances with a higher average degree, local relabeling with $\gamma = 1$ performs better. This is an expected result, as the larger local relabeling is more expensive in higher-average-degree graphs, as the 2-neighborhood of a vertex is much larger. Local relabeling with $\gamma = 2$ spends $90 - 99\%$ of the total running time in initialization and initial relabeling. The same effect is even more pronounced for the variant which performs global relabeling in initialization. On vertices with a low average degree, we can perform global relabeling in reasonable time, which makes the variant competitive with the local relabeling variants. However, in high average degree instances, the excessive running time of a global relabeling step causes the variant to have slowdowns of up to 1000x compared to the fastest variant. On all instances, the vast majority of running time is spent in initialization including initial global relabeling.

One graph family where local relabeling with $\gamma = 1$ performs badly are the graph instances based on `auto` [108], a 3D finite element mesh graph. These graphs

are rather sparse (average degree 15) and planar. On these graphs, the value of the minimum cut divided by the average degree is very large, as they do not contain any vertices of degree $1, 2, 3$. Thus, the variants which perform only minor local relabeling do not guide the flow enough and therefore the push-relabel algorithm takes a long time. On most other instances in our test set, local relabeling with $\gamma = 1$ is enough to guide at least λ flow to the sink quickly.

Local relabeling with a relabeling depth $\gamma = 0$ (i.e. we set the distance of the sink to 0, the source to n and all other vertices to 1) has a slowdown factor of $10 - 100$x with only $1 - 10\%$ of the running time spent in the initialization. The slowdown factor is generally increasing for larger values of the minimum cut λ and average degree, which indicates that "the lack of guidance towards the sink" causes the algorithm to send flow to regions of the graph that are far away from the source. For graphs with large minimum cut value λ, the algorithm does not terminate early and needs to perform a significant amount of push and relabel steps. In variants that perform more relabeling at initialization, the flow is guided towards the sink by the distance labels and the termination trigger is reached faster. The variant which does not include any relabeling in the initialization phase has similar issues with an even larger slowdown factor of $10 - 2000$x, as even flow that is already incident to the sink does not necessarily flow straight to the sink.

On most instances, local relabeling with depth $\gamma = 1$ performed best, as it helps guide the flow towards the sink with additional work (compared to no relabeling) only equal to the degree of the sink. While performing more relabeling can increase this guidance even further, it comes with a trade-off in additional time spent in the initialization. Note that this is not a general observation for the push-relabel algorithm and can only be applied to our application, in which the push-relabel algorithm is terminated early as soon as λ units of flow reach the sink vertex. Based on these experiments, we use local relabeling with $\gamma = 1$ for edge deletions in all following experiments.

6.4.2 Dynamic Graphs

Figure 6.3 shows experimental results on the dynamic graph instances from Graph Family C in Table 2.1. These graph instances are mostly incremental with some being fully dynamic and most instances have multiple connected components, i.e. a minimum cut value $\lambda = 0$, even after all insertions. On these incremental graphs with multiple connected components, our algorithm behaves similar to a simple union-find based connected components algorithm that for edge insertion checks whether the incident vertices already belong to the same connected component and merges their connected components if they are different.

In this section we compare our dynamic minimum cut algorithm to the static algorithm of Nagamochi et al. [147], which has been shown to be one of the fastest sequential algorithms for the minimum cut problem [37, 94]. The static algorithm performs the updates batch-wise, i.e. the static algorithm is not called inbetween

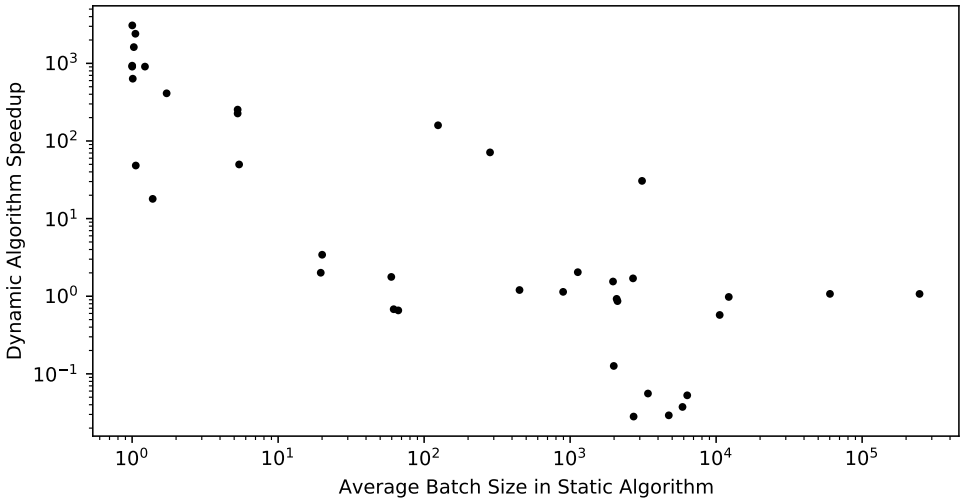


Figure 6.3: Speedup of Dynamic Algorithm.

multiple edge updates with equal timestamp. In Figure 6.3, we show the dynamic speedup in comparison to the average batch size. As expected, there is a large speedup factor of up to 1000x for graphs with small batch sizes; and the speedup decreases for increasing batch sizes. The family of instances in which the dynamic algorithm is outperformed by the static algorithm is the `insecta-ant-colony` graph family [137]. These graphs have a very high minimum cut value and fewer batches than changes in the minimum cut value. Therefore, the dynamic algorithm which updates on every edge insertion needs to recompute the minimum cut cactus more often than the static algorithm is run and, thus, takes a longer time.

As these dynamic instances do not have sufficient diversity, we also perform experiments on static graphs in graph family B in which a subset of edges is inserted or removed dynamically. We report on this experiment in the following section.

6.4.3 Random Insertions and Deletions from Static Graphs

Figure 6.4 shows results for dynamic edge insertions and deletions from all graphs in Graph Family A and B from Table 2.1. These graphs are static, we create a dynamic problem from graph $G = (V, E, c)$ as follows: let $\alpha_{ins} \in (0, 1)$ and $\alpha_{del} \in (0, 1)$ with $\alpha_{ins} + \alpha_{del} < 1$ be the edge insertion and deletion rate. We randomly select edge lists E_{ins} and E_{del} with $|E_{ins}| = \alpha_{ins} \cdot |E|$, $|E_{del}| = \alpha_{del} \cdot |E|$ and $E_{ins} \cap E_{del} = \emptyset$. For every vertex $v \in V$, we make sure that at least one edge incident to v is neither in E_{ins} nor in E_{del}, so that the minimum degree of $(V, E \backslash (E_{ins} \cap E_{del}), c)$ is strictly greater than 0 at any point in the update sequence.

We initialize the graph as $(V, E \backslash E_{ins}, c)$ and create a sequence of edge updates E_u by concatenating E_{ins} and E_{del} and randomly shuffling the combined list. Then we perform edge updates one after another and compute the minimum

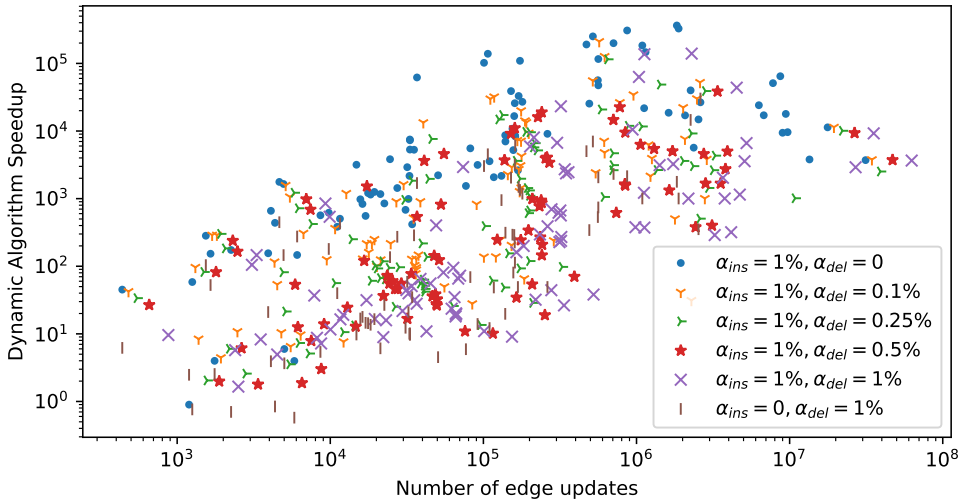


Figure 6.4: Speedup of Dynamic Algorithm on Random Insertions and Deletions from Static Graphs.

cut - either statically using our efficient implementation of the algorithm of Nagamochi et al. [147] or by performing an update in the dynamic algorithm - after every update. Note that all of these algorithms are sequential. We report the total running time of either variant and give the speedup of the dynamic algorithm over the static algorithm as a function of the number of edge updates performed. For each graph we create problems with $\alpha_{ins} = 1\%$ and $\alpha_{del} \in \{0, 0.1\%, 0.25\%, 0.5\%, 1\%\}$; and additionally a decremental problem with $\alpha_{ins} = 0$ and $\alpha_{del} = 1\%$. We set the timeout for the static algorithm to 1 hour, if the algorithm does not finish before timeout, we approximate the total running time of the static algorithm by performing 100 or 1000 updates in batch.

Dynamic edge insertions are generally much faster than edge deletions, as most real-world graphs have large sets that are not separated by any global minimum cut. When inserting an edge where both incident vertices are in the same set in $\mathcal{C}$, the edge insertion only requires two array accesses; if they are in different sets, it requires a breadth-first search on the relatively small cactus graph $\mathcal{C}$ and only if there are no minimum cuts remaining, an edge insertion requires a recomputation. In contrast to that, every edge deletion requires solving of a flow problem and therefore takes significantly more time in average. Therefore, the average speedup is larger on problems with a higher rate of edge insertions.

Generally, the speedup of the dynamic algorithm increases with larger problems and more edge updates. For larger graphs with $\geq 10^6$ edge updates, the average speedup is more than four orders of magnitude for instances with $\alpha_{del} = 0$ and still more than two orders of magnitude for large instances when $\alpha_{del} = \alpha_{ins} = 1\%$. Note that in this experiment, the number of edge

updates is a function of the number of edges, thus instances with more updates directly correspond to graphs with more edges.

For decremental instances with $\alpha_{ins} = 0$, the speedup is generally lower, but still reaches multiple orders of magnitude in larger instances.

Most Balanced Minimum Cut

In Section 5.2 we show that given the cactus graph $\mathcal{C}$ we can compute the most balanced minimum cut, i.e. the minimum cut which has the highest number of vertices in the smaller partition, in $\mathcal{O}(n^*)$ time. In our algorithm for the dynamic minimum cut problem we also compute a cactus graph of minimum cuts, however this cactus graph does not necessarily contain all minimum cuts in G, as we do not introduce new minimum cuts added by edge deletions.

We use the algorithm given in Chapter 5 to find the most balanced minimum cut for all instances of Graph Family B every 1000 edge updates and compare it to the most balanced minimum cut found by our algorithm. In instances that are not just decremental, in 97.3% of all cases where there is a nontrivial minimum cut (i.e. smaller side contains multiple vertices), both algorithms give the same result, i.e. our algorithm can almost always output the most balanced minimum cut. In the instances that are purely decremental, i.e. $|E_{ins}| = 0$, we only find the most balanced minimum cut in 25.4% of cases where there is a non-trivial minimum cut. This is the case because an increase of the minimum cut prompts a full recomputation of a cactus graph that represents all (potentially many) minimum cuts, thus also the most balanced minimum cut. Only if this cut in particular is affected by an edge update, the dynamic algorithm "loses" it. In the purely decremental case, the minimum cut value only decreases. Thus, the dynamic algorithm only knows one or a few minimum cuts. All cuts that reach the same value λ in later edge deletions are not in $\mathcal{C}$, as we do not add cuts of the same value to it. As these decremental instances do not have any edge insertions that can increase the value of these cuts, there is eventually a large set of minimum cuts of which the algorithm only knows a few. If maintaining a balanced minimum cut is a requirement, this can easily be achieved by occasionally recomputing the entire cactus graph $\mathcal{C}$ from scratch.

6.4.4 Worst-case Instances

On random edge insertions, there is a high chance that the vertices incident to the newly inserted edge were not separated by a minimum cut and therefore require no update of the cactus graph $\mathcal{C}$. In this experiment we aim to generate instances that aim to maximize the work performed by the dynamic algorithm. We initialize the graph as $G = (V, E, c)$ and add random unit-weight edges $e = (u, v)$ where $\Pi(u) \neq \Pi(v)$ for every newly added edge. Then we randomly select $|E_{ins}| = 1000$ edges to add so that for each such edge (u, v), $\Pi(u) \neq \Pi(v)$ before inserting (u, v), and select a subset $E_{del} \subseteq E_{ins}$ to delete. For each graph we create 5 problems, with $|E_{del}| \in \{0, 100, 250, 500, 1000\}$. We randomly shuffle the edge updates while

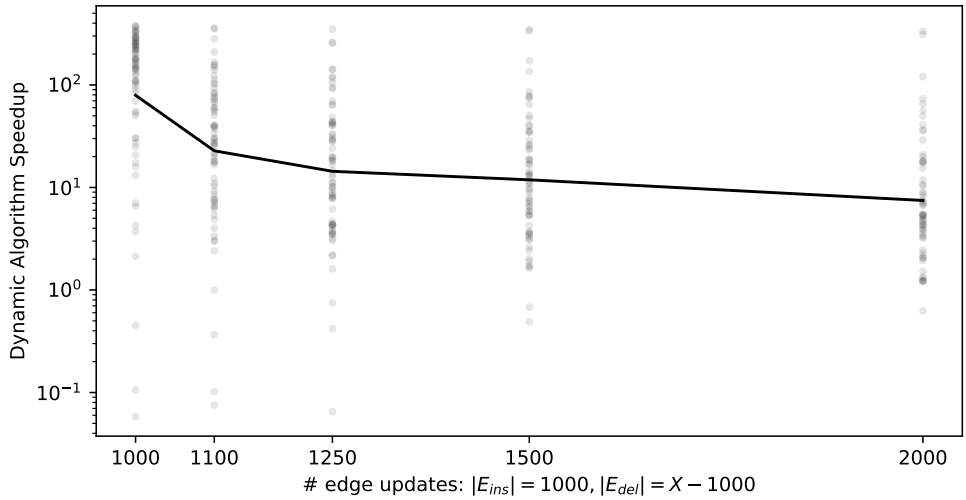


Figure 6.5: Speedup of Dynamic Algorithm on Worst-case Insertions and Deletions from Static Graphs.

making sure that an edge deletion is only performed after the respective edge has been added to the graph, but still interspersing edge insertions and deletions to create true worst-case instances for the dynamic algorithm, as each edge deletion or insertion affects one or multiple minimum cuts in the graph.

Figure 6.5 shows the results of this experiment. Each low-alpha dot shows the speedup of the dynamic algorithm on a single problem, the black line gives the geometric mean speedup. As indicated in previous experiments, we can see that the average speedup decreases when the ratio of deletions is increased. However, even on these worst-case instances, the mean speedup factor is still 7.46x for $|E_{ins}| = |E_{del}| = 1000$ up to 79.2x for the purely incremental instances on instances where both algorithms finished before timeout at one hour. Similar to previous experiments, the speedup factor increases with the graph size.

On these problem instances we can see interesting effects. Especially in instances with $|E_{del}| = 500$ we can see many instances where the minimum cut fluctuates between two different values in more than half of all edge updates. As the larger of the values usually has a large cactus graph $\mathcal{C}$, this would result in expensive recomputation on almost every update. However, using the cactus caching technique detailed in Section 6.3.1 we can save this overhead and simply reuse the almost unchanged previous cactus graph. In some cases, this reduces the number of calls to the static all-minimum-cut algorithm by more than a factor of 10.

We also find some instances where the static graph has few minimum cuts, but there is a large set of cuts slightly larger than lambda. One such example are planar graphs derived from Delaunay triangulation [125] that have a few vertices of minimal degree near the edges of the triangulated object, but a large number of vertices

with a slightly larger degree. If we now add edges to increase the degree of the minimum-degree vertices, the resulting graph has a huge number of minimum cuts and computing all minimum cuts is significantly more expensive than computing just a single minimum cut. In these instances the dynamic algorithm is actually slower than rerunning the static algorithm on every edge update. The dynamic algorithm is slower than the static algorithm in 3.9% of the worst-case instances.

6.5 Conclusion

In this chapter, we presented the first implementation of a fully-dynamic algorithm that maintains the minimum cut of a graph under both edge insertions and deletions. Our algorithm combines ideas from the theoretical foundation with efficient and fine-tuned implementations to give an algorithm that outperforms static approaches by up to five orders of magnitude on large graphs. In our experiments, we show the performance of our algorithm on a wide variety of graph instances.

Future work includes maintaining all global minimum cuts also under edge deletions and employing shared-memory or distributed parallelism to further increase the performance of our algorithm.

Part II

The Balanced Graph Partitioning Problem

CHAPTER 7

ILP-based Local Search for Graph Partitioning

Computing high-quality balanced graph partitions is a challenging problem with numerous applications. In this chapter, we present a novel meta-heuristic for the balanced graph partitioning problem. Our approach is based on integer linear programs that solve the partitioning problem to optimality. However, since those programs typically do not scale to large inputs, we adapt them to heuristically improve a given partition. We do so by defining a much smaller model that allows us to use symmetry breaking and other techniques that make the approach scalable. For example, in Walshaw's well-known benchmark tables, we are able to improve roughly half of all entries when the number of blocks is high. Additionally, we include our techniques in a memetic framework and develop a crossover operation based on the proposed techniques.

The content of this chapter is based on [86] and [87].

7.1 Introduction

Balanced graph partitioning is an important problem in computer science and engineering with an abundant amount of application domains, such as VLSI circuit design, data mining and distributed systems [168]. It is well known that this problem is NP-complete [30] and that no approximation algorithm with a constant ratio factor exists for general graphs unless P=NP [30]. Still, there is a large amount of literature on methods (with worst-case exponential time) that solve the graph partitioning problem to optimality. This includes methods dedicated to the bipartitioning case [13, 14, 48, 49, 55, 56, 80, 107, 130, 172] and some methods that solve the general graph partitioning problem [58, 173]. Most of these methods rely on the branch-and-bound framework [123]. However, these methods can typically solve only very

small problems as their running time grows exponentially, or if they can solve large bipartitioning instances using a moderate amount of time [48, 49], the running time highly depends on the bisection width of the graph. Methods that solve the general graph partitioning problem [58, 173] have huge running times for graphs with up to a few hundred vertices. Thus in practice mostly heuristic algorithms are used.

Typically the graph partitioning problem asks for a partition of a graph into k blocks of about equal size such that there are few edges between them. Here, we focus on the case when the bounds on the size are very strict, including the case of *perfect balance* when the maximal block size has to equal the average block size.

Our focus here is on solution quality, i.e. minimize the number of edges that run between blocks. During the past two decades there have been numerous researchers trying to improve the best graph partitions in Walshaw's well-known partitioning benchmark [174, 190]. Overall there have been more than forty different approaches that participated in this benchmark. Indeed, high solution quality is of major importance in applications such as VLSI Design [7, 8] where even minor improvements in the objective can have a large impact on the production costs and quality of a chip. High-quality solutions are also favorable in applications where the graph needs to be partitioned only once and then the partition is used over and over again, implying that the running time of the graph partitioning algorithms is of a minor concern [47, 60, 124, 140, 169, 170]. Thirdly, high-quality solutions are even important in areas in which the running time overhead is paramount [174], such as finite element computations [167] or the direct solution of sparse linear systems [72]. Here, high-quality graph partitions can be useful for benchmarking purposes, i.e. measuring how much more running time can be saved by higher quality solutions.

In order to compute high-quality solutions, state-of-the-art local search algorithms exchange vertices between blocks of the partition trying to decrease the cut size while also maintaining balance. This highly restricts the set of possible improvements. Sanders and Schulz introduced new techniques that relax the balance constraint for vertex movements but globally maintain balance by combining multiple local searches [163]. This was done by reducing this combination problem to finding negative cycles in a graph. Here, we extend the neighborhood of the combination problem by employing integer linear programming. This enables us to find even more complex combinations and hence to further improve solutions. More precisely, our approach is based on integer linear programs that solve the partitioning problem to optimality. However, these programs typically do not scale to large inputs, in particular because the graph partitioning problem has a very large amount of symmetry – given a partition of the graph, each permutation of the block IDs gives a solution having the same objective and balance. Hence, we adapt the integer linear program to improve a given input partition. We do so by defining a much smaller graph, called *model*, and solve the graph partitioning problem on the model to optimality by the integer linear program. More specifically, we select vertices close to the cut of the given input partition for potential movement and

contract all remaining vertices of a block into a single vertex. A feasible partition of this model corresponds to a partition of the input graph having the same balance and objective. Moreover, this model enables us to use symmetry breaking, which allows us to scale to much larger inputs. To make the approach even faster, we combine it with initial bounds on the objective provided by the input partition, as well as providing the input partition to the integer linear program solver. Overall, we arrive at a system that is able to improve more than half of all entries in Walshaw's benchmark when the number of blocks is high. We include our integer linear program-based operation into the memetic graph partitioner KaBaPE [163]. Additionally, we develop a crossover operation which is also based on our linear program. This crossover operation contracts blocks of vertices, which all partitions place in the same block. The extended memetic algorithm computes graph partitions from scratch and manages to improve 17% of the entries in Walshaw's benchmark on the instances with $8, 16, 32$ or 64 partitions and and a maximum allowed imbalance of 3% or 5%. In roughly half of all problems considered, KaBaPE+ILP either reproduces or improves the previous best solution.

In Section 7.2 we first introduce basic concepts. After presenting some related work in Section 7.3 we outline the integer linear program as well as our novel local search algorithm in Section 7.4. Here, we start by explaining the technique we use to find combinations of simple vertex movements. We then explain our strategies to improve the running time of the solver and vertex selection strategies. In Section 7.5 we detail how the algorithm can be used in the context of memetic graph partitioning. A summary of extensive experiments done to evaluate the performance of our algorithms is presented in Section 7.6. We conclude in Section 7.7.

7.2 Preliminaries

Let $G = (V = \{0, \dots, n-1\}, E)$ be an undirected graph. We consider positive, real-valued edge and vertex weight functions c resp. ω and extend them to sets, i.e., $c(E') := \sum_{x \in E'} c(x)$ and $\omega(V') := \sum_{x \in V'} \omega(x)$. We use the same terminology to describe graphs as in Part I of this dissertation. A vertex is a *boundary vertex* if it is incident to at least one vertex in a different block. We are looking for disjoint *blocks* of vertices V_1,...,V_k that partition V; i.e., $V_1 \cup \dots \cup V_k = V$. The *balancing constraint* demands that each block has weight $\omega(V_i) \leq (1+\epsilon)\lceil \frac{\omega(V)}{k} \rceil =: L_{\max}$ for some imbalance parameter ϵ. We call a block V_i *overloaded* if its weight exceeds $L_{\max}$. The objective of the problem is to minimize the total *cut* $c(E \cap \bigcup_{i<j} V_i \times V_j)$ subject to the balancing constraint.

7.3 Related Work

There has been a *huge* amount of research on graph partitioning and we refer the reader to the surveys given in [21, 31, 167, 191] for most of the material. Here,

we focus on issues closely related to our main contributions. All general-purpose methods that are able to obtain good partitions for large real-world graphs are based on the multi-level principle. Well-known software packages based on this approach include Jostle [191], KaHIP [162], Metis [108] and Scotch [154].

Walshaw's well-known benchmark archive for the balanced graph partitioning problem has been established in 2001 [174, 190]. Overall it contains 816 instances (34 graphs, 4 values of imbalance, and 6 values of k). In this benchmark, the running time of the participating algorithms is not measured or reported. Submitted partitions will be validated and added to the archive if they improve on a particular result. This can either be an improvement in the number of cut edges or, if they match the current best cut size, an improvement in the weight of the largest block. Most entries in the benchmark have as of Jan. 2021 been obtained by Galinier et al. [69] (more precisely an implementation of that approach by Schneider), Hein and Seitzer [85], the Karlsruhe High-Quality Graph Partitioning (KaHIP) framework [163] and the local search techniques described in this work. More precisely, Galinier et al. [69] use a memetic algorithm that is combined with tabu search to compute solutions and Hein and Seitzer [85] solve the graph partitioning problem by providing tight relaxations of a semi-definite program into a continuous problem.

Bisseling et al. [22] use integer linear programming to solve the graph partitioning problem in directed graphs. In contrast to our work, they aim to minimize the number of vertices that have incoming edges from a different block. Miyauchi et al. [139] use integer linear programming to solve the graph partitioning problem on fully connected edge-weighted graphs.

The Karlsruhe High-Quality Graph Partitioning (*KaHIP*) framework implements many different algorithms, for example flow-based methods and local searches, as well as several coarse-grained parallel and sequential meta-heuristics. KaBaPE [163] is a coarse-grained parallel memetic algorithm, i.e. each processor has its own population (set of partitions) and a copy of the graph. After initially creating the local population, each processor performs multi-level combine and mutation operations on the local population. This is combined with a meta-heuristic that combines local searches that individually violate the balance constraint into a more global feasible improvement. For more details, we refer the reader to [163].

7.4 Local Search based on Integer Linear Programming

We now explain our algorithm that combines integer linear programming and local search. We start by explaining the integer linear program that can solve the graph partitioning problem to optimality. However, out-of-the-box this program does not scale to large inputs, in particular because the graph partitioning problem has a very large amount of symmetry. Thus, we reduce the size of the graph by first computing a partition using an existing heuristic and based on it collapsing parts of the graph. Roughly speaking, we compute a small graph, called *model*, in which we

only keep a small number of selected vertices for potential movement and perform graph contractions on the remaining ones. A partition of the model corresponds to a partition of the input network having the same objective and balance. The computed model is then solved to optimality using the integer linear program. As we will see this process enables us to use symmetry breaking in the linear program, which in turn drastically speeds up computation times.

7.4.1 Integer Linear Program for the Graph Partitioning Problem

We now introduce a generalization of an integer linear program formulation for balanced bipartitioning [28] to the general graph partitioning problem. First, we introduce binary decision variables for all edges and vertices of the graph. More precisely, for each edge $e = \{u, v\} \in E$, we introduce the variable $e_{uv} \in \{0, 1\}$ which is one if e is a cut edge and zero otherwise. Moreover, for each $v \in V$ and block k, we introduce the variable $x_{v,k} \in \{0, 1\}$ which is one if v is in block k and zero otherwise. Hence, we have a total of $|E| + k|V|$ variables. We use the following constraints to ensure that the result is a valid k-partition:

$$\forall \{u, v\} \in E, \forall k : e_{uv} \geq x_{u,k} - x_{v,k} \tag{7.1}$$

$$\forall \{u, v\} \in E, \forall k : e_{uv} \geq x_{v,k} - x_{u,k} \tag{7.2}$$

$$\forall k : \sum_{v \in V} x_{v,k} \omega(v) \leq L_{\max} \tag{7.3}$$

$$\forall v \in V : \sum_{k} x_{v,k} = 1 \tag{7.4}$$

The first two constraints ensure that e_{uv} is set to one if the vertices u and v are in different blocks. For an edge $\{u, v\} \in E$ and a block k, the right-hand side in this equation is one if one of the vertices u and v is in block k and the other one is not. If both vertices are in the same block then the right-hand side is zero for all values of k. Hence, the variable can either be zero or one in this case. However, since the variable participates in the objective function and the problem is a minimization problem, it will be zero in an optimum solution.

The third constraint ensures that the balance constraint is satisfied for each partition. And finally, the last constraint ensures that each vertex is assigned to exactly one block. To sum up, our program has $2k|E| + k + |V|$ constraints and $k \cdot (6|E| + 2|V|)$ non-zeros. Since we want to minimize the weight of cut edges, the objective function of our program is written as:

$$\min \sum_{\{u,v\} \in E} e_{uv} \cdot c(\{u, v\}) \tag{7.5}$$

7.4.2 Local Search

The graph partitioning problem has a large amount of symmetry – each permutation of the block IDs gives a solution with equal objective and balance. Hence, the

integer linear program described above will scan many branches that contain essentially the same solutions so that the program does not scale to large instances. Moreover, it is not immediately clear how to improve the scalability of the program by using symmetry breaking or other techniques. For the closely related problem of vertex partitioning, Bisseling et al. [22] report that using symmetry breaking is highly important in order to get optimal solutions in reasonable time.

Our goal in this section is to develop a local search algorithm using the integer linear program above. Given a partition as input to be improved, our *main idea* is to contract vertices "that are far away" from the cut of the partition. In other words, we want to keep vertices close to the cut and contract all remaining vertices into one vertex for each block of the input partition. This ensures that a partition of the contracted graph yields a partition of the input graph with the same objective and balance. Hence, we apply the integer linear program to the model and solve the partitioning problem on it to optimality. Note, however, that due to the performed contractions this does not imply an optimal solution on the input graph.

We now outline the details of the algorithm. Our local algorithm has two inputs, a graph G and a partition $V_1, \dots, V_k$ of its vertices. For now assume that we have a set of vertices $\mathcal{K} \subset V$ which we want to keep in the coarse model, i.e. a set of vertices which we do not want to contract. We outline in Section 7.4.4 which strategies we have to select the vertices $\mathcal{K}$. For the purpose of contraction we define k sets $\mathcal{V}_i := V_i \backslash \mathcal{K}$. We obtain our coarse model by contracting each of these vertex sets. The contraction of a vertex set $\mathcal{V}_i$ works by iteratively contracting all pairs of vertices in that set until only one node is left. After all contractions have been performed the coarse model contains $k + |\mathcal{K}|$ vertices, and potentially much fewer edges than the input graph. Figure 7.1 gives an abstract example of our model.

There are two things that are important to see: first, due to the way we perform contraction, the given partition of the input network yields a partition of our coarse model that has the same objective and balance simply by putting μ_i into block i and keeping the block of the input for the vertices in $\mathcal{K}$. Moreover, if we compute a new partition of our coarse model, we can build a partition in the original graph with the same properties by putting the vertices $\mathcal{V}_i$ into the block of their coarse representative μ_i together with the vertices of $\mathcal{K}$ that are in this block. Hence, we can solve the integer linear program on the coarse model to compute a partition for the input graph. After the solver terminates, i.e. found an optimum solution of our mode or has reached a predefined time limit $\mathcal{T}$, we transfer the best solution to the original graph. Note that the latter is possible since an integer linear program solver typically computes intermediate solutions that may not be optimal.

7.4.3 Optimizations

Independent of the vertices $\mathcal{K}$ that are selected to be kept in the coarse model, the approach above allows us to define optimizations to solve our integer linear program faster. We apply four strategies: (i) symmetry breaking, (ii) providing a

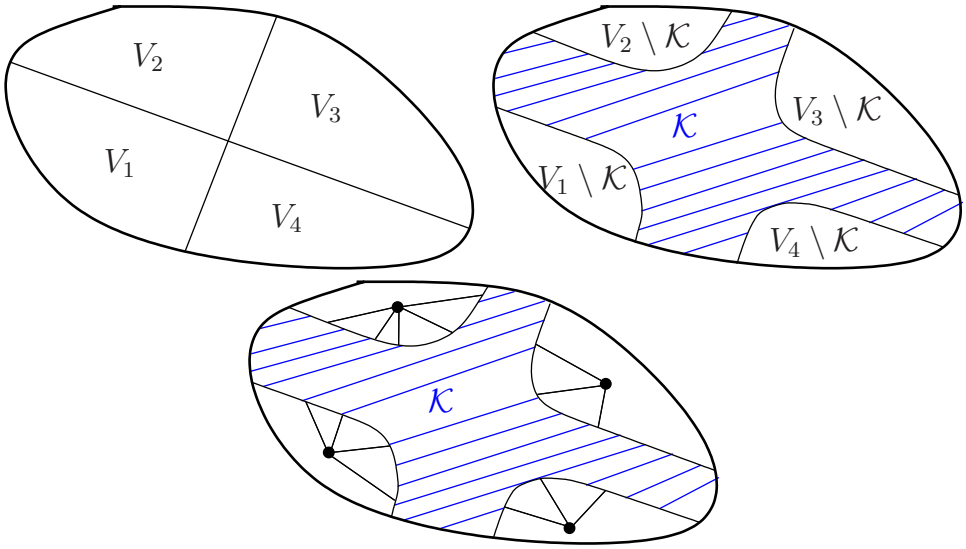


Figure 7.1: Top left: a graph that is partitioned into four blocks, top right: the set $\mathcal{K}$ close to the boundary that will stay in the model, bottom: the model in which the sets $V_i \setminus \mathcal{K}$ have been contracted.

start solution to the solver, (iii) add the objective of the input as a constraint as well as (iv) using the parallel solving facilities of the underlying solver. We outline the first three strategies in greater detail:

Symmetry Breaking

If the set $\mathcal{K}$ is small, then the solver will find a solution much faster. Typically, our algorithms selects the vertices $\mathcal{K}$ such that $\omega(\mu_i) + \omega(\mu_j) > L_{\max}$. In other words, no two contracted vertices can be clustered in one block. We can use this to break symmetry in our integer linear programming by adding constraints that fix the block of μ_i to block i, i.e. we set $x_{\mu_i,i} = 1$ and $x_{\mu_i,j} = 0$ for $i \neq j$. Moreover, for those vertices we can remove the constraint which ensures that the vertex is assigned to a single unique block—since we assigned those vertices to a block using the new additional constraints. Note that we perform symmetry breaking even if it is mathematically possible that multiple μ_i could be in the same block.

Providing a Start Solution to the Solver

The integer linear program performs a significant amount of work in branches which correspond to solutions that are worse than the input partition. Only very few - if any - solutions are better than the given partition. However, we already know a fairly good partition (the given partition from the input) and give this partition to the solver by setting according initial values for all variables. This ensures

that the integer linear program solver can omit many branches and hence speeds up the time needed to solve the integer linear program.

Solution Quality as a Constraint

Since we are only interested in improved partitions, we can add an additional constraint that disallows solutions which have a worse objective than the input partition. Indeed, the objective function of the linear program is linear, and hence the additional constraint is also linear. Depending on the objective value, this reduces the number of branches that the linear program solver needs to look at. However, note that this comes at the cost of an additional constraint that needs to be evaluated. Also note that if we provide a start solution to the solver, the solver already knows a solution of said quality. Thus, the solver is then able to prune worse solutions by itself.

Row Generation

Equation 7.3 ensures that the balancing constraints in the graph partitioning problem are adhered to. However, checking these constraints comes with a computational cost. The idea of row generation is to initially omit these constraints and lazily introduce balance constraints when a given solution violates them. For each solution found by the ILP solver, we check whether any block is heavier than $L_{\max}$. If none is, the solution is valid. For each block V_k heavier than $L_{\max}$ we introduce a new constraint which makes sure that a subset of V_k with a total weight of $> L_{\max}$ is not in block k and thus reject the solution, as it violates the new constraint.

In preliminary experiments this yields mixed results for $k = \{2, 4\}$, but slowed down the ILP for $k \geq 8$, as most solutions without balancing constraints are too heavy in multiple blocks and thus the row generation introduces a large amount of balancing constraints over the course of the solving process. We therefore do not employ row generation in our experiments.

7.4.4 Vertex Selection Strategies

The algorithm above works for different vertex sets $\mathcal{K}$ that should be kept in the coarse model. There is an obvious trade-off: on the one hand, the set $\mathcal{K}$ should not be too large, otherwise the coarse model would be large and hence the linear programming solver needs a large amount of time to find a solution. On the other hand, the set should also not be too small, since this restricts the amount of possible vertex movements, and hence the approach is unlikely to find an improved solution. We now explain different strategies to select the vertex set $\mathcal{K}$. In any case, while we add vertices to the set $\mathcal{K}$, we compute the number of non-zeros in the corresponding ILP. We stop to add vertices when the number of non-zeros in the corresponding ILP is larger than a parameter $\mathcal{N}$.

Vertices Close to Input Cut

The intuition of the first strategy, Boundary, is that changes or improvements of the partition will occur reasonable close to the input partition. In this simple strategy our algorithm tries to use all *boundary vertices* as the set $\mathcal{K}$. In order to adhere to the constraint on the number of non-zeros in the ILP, we add the vertices of the boundary uniformly at random and stop if the number of non-zeros $\mathcal{N}$ is reached. If the algorithm managed to add all boundary vertices whilst not exceeding the specified number of non-zeros, we do the following extension: we perform a breadth-first search that is initialized with a random permutation of the boundary vertices. All additional vertices that are reached by the BFS are added to $\mathcal{K}$. As soon as the number of non-zeros $\mathcal{N}$ is reached, the algorithm stops.

Start at Promising Vertices

Especially for high values of k the boundary contains many vertices. The Boundary strategy quickly adds a lot of random vertices while ignoring vertices that have high gain. Note that even in good partitions it is possible that vertices with positive gain exist but cannot be moved due to the balance constraint.

Hence, our second strategy, $\texttt{Gain}_\rho$, tries to fix this issue by starting a breadth-first search initialized with only high gain vertices. More precisely, we initialize the BFS with each vertex having gain $\geq \rho$ where ρ is a tuning parameter. Our last strategy, $\texttt{TopVertices}_\delta$, starts by sorting the boundary vertices by their gain. We break ties uniformly at random. Vertices are then traversed in decreasing order (highest gain vertices first) and for each start vertex v our algorithm adds all vertices with distance $\leq \delta$ to the model. The algorithm stops as soon as the number of non-zeros exceeds $\mathcal{N}$.

Early gain-based local search heuristics for the ϵ-balanced graph partitioning problem searched for pairwise swaps with positive gain [59, 112]. More recent algorithms generalized this idea to also search for cycles or paths with positive total gain [163]. An important advantage of our new approach is that we solve the combination problem to optimality, i.e. our algorithm finds the best combination of vertex movements of the vertices in $\mathcal{K}$ with respect to the input partition of the original graph. Therefore we can also find more complex optimizations that cannot be reduced to positive gain cycles and paths.

7.5 Integer Linear Programming based Crossover

A memetic algorithm is a population-based metaheuristic algorithm for an optimization problem. The general outline of a memetic algorithm is such that we first create a population of solutions and then use crossover and mutation operations to generate new individuals out of existing ones. Generally, a mutation operation has a single input partition and a cross operation has multiple input partitions. If those new

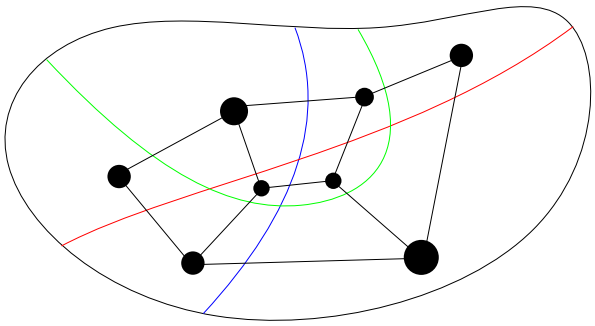


Figure 7.2: Example overlap graph for a graph with 3 partitions and 2 blocks each.

individuals are sufficiently fit, they evict the lowest fitness individual from the population. *KaBaPE* [163] is a distributed parallel memetic algorithm for the graph partitioning problem that provides multiple cross and mutation operations. Based on the optimization techniques in this work, we now describe new mutation and cross operations. These operations are added to the existing portfolio of operations of KaBaPE.

More precisely, the ILP-based local search algorithm described in Section 7.4 can be used as a mutation operation directly. For this, we take an individual from the population and run ILP-based local search on the individual. If this results in an improved cut value, the new individual is added to the population.

7.5.1 ILP on Overlap Graph

Our new cross operation builds and solves an integer linear program from multiple individuals. For this operation, we take l individuals and build an overlap graph $G_O = (V_O, E_O)$ out of G by contracting regions that are in the same block in every partition. An example overlap graph can be found in Figure 7.2. In the literature this concept is also called overlap clustering [180].

The weight of a vertex $v_O \in V_O$ is equal to the weight sum of all vertices that are contracted into v_O. For vertices u_O and v_O in V_O, $e_O = (u_O, v_O)$ exists if there is an edge from any vertex in u_O to any vertex in v_O. If there are multiple edges, the edge weight $c(e_O)$ is equal to the sum of their weights. The fundamental idea behind that contraction is such that if a region of vertices that is in the same block in all partitions, most good partitions will have them in the same block. It is therefore more valuable to model regions in which the partitions 'disagree' on the vertex placement to make the ILP tractable.

In order to break symmetries, we select a subset of vertices I where no two vertices in I are in the same block for any individual used to create the overlap graph G_O. We choose the first vertex $v \in V_O$ in I at random and add vertices that share no block with any vertex in I in any individual until we can't find such a vertex in V_O any more. In this way we break many symmetries and only dis-

allow solutions that aim to place vertices in the same block that were placed in different blocks in every individual used to create G_O.

We model the overlap graph as an ILP and initialize the block affiliations in the ILP according to the partition that has the lowest cut value. When multiple partitions have the same cut value, we choose any of them at random. For each vertex $v \in V_O$, the block affiliation of v is set to the block ID of the vertices merged into v. Thus, we already have a solution that has value equal to the best partition used for the overlap. If the ILP finds a better solution, we insert the individual into the population.

As there is a very high variability in running times of the ILP operations, we do not give a fixed ratio of ILP operation calls. Instead, we limit the total running time fraction used in the ILP operations, so that they never take up more than a third of the total running time.

Each process in KaBaPE+ILP keeps two timers, one for each ILP operation, to count the total running time used for all calls of the operation. If the sum of them is at least 33% of the total time used for the algorithm, we will not choose them. If no time has been spent in the ILP operations yet, we choose one of them with a probability of 75%. In between, we perform linear interpolation, i.e. the probability of performing an ILP operation is $(0.33-\alpha)\cdot 0.75$, where α is the fraction of the total runtime due to ILP operations. Thus, in graphs where the ILP operations are very fast, we use them often. However, if the ILP operations are slow compared to other operations they will not use up the majority of the running time. We also use linear interpolation over the total running times to determine fairly which ILP operation is used. As the solution quality improves more rapidly in the start of the memetic algorithm, we gradually increase the time limit given to the ILP solver in an ILP operation. The time limit is equal to 10% of the current total running time. These parameters were obtained from preliminary experiments, however, in general, the algorithm is not very susceptible to those parameters within reasonable limits.

We denote the extended memetic algorithm as KaBaPE+ILP.

7.5.2 Post-processing

We also employ a similar strategy to find the overlap graph for all high-quality partitions. After the memetic algorithm is terminated, we collect all unique partitions. We then build the overlap graph $G_{\mathcal{O}}$ on the best κ partitions, where κ is a tuning parameter. In this graph, vertices are merged if every high-quality partition in the population places them in the same block. Thus, if the diversity of the population is large enough, it is highly likely that the vertices will be placed in the same block in any good partition. We run the memetic algorithm KaBaPE+ILP again, this time on $G_{\mathcal{O}}$. As $G_{\mathcal{O}}$ has significantly fewer vertices and edges than G, all operations perform faster and convergence is faster. However, this also limits the solution space, as partitions that place merged vertices into different blocks can not be found on $G_{\mathcal{O}}$. Thus, we might be converging to a local optimum.

7.6 Experiments

7.6.1 Experimental Setup and Methodology

We implemented the algorithms described in the previous sections using C++-17 and compiled all codes with full optimization enabled (-O3). We use Gurobi as an ILP solver and use its shared-memory parallel version. The experiments in Sections 7.6.2, 7.6.3 and 7.6.4 were conducted on a machine with two Haswell Xeon E5-2697 v3 processors, using g++-7.2.0 and Gurobi 7.5.2. The machine has 28 cores at 2.6GHz as well as 64GB of main memory and runs the SUSE Linux Enterprise Server (SLES) operating system. Unless otherwise mentioned, our approach uses the shared-memory parallel variant of Gurobi using all 28 cores. The experiments in Section 7.6.5 use g++-8.3.0 and Gurobi 8.1.1 and were conducted on a machine with two Intel Xeon E5-2643 v4 with 3.4GHz with 6 CPU cores each and 1.5 TB RAM in total. As the memetic algorithm in this section has multiple parallel threads that perform cross and mutation operations independent from each other, KaBaPE+ILP uses the sequential variant of Gurobi. In general, we perform five repetitions per instance and report the average running time as well as cut. Unless otherwise mentioned, we use a time limit for the integer linear program. When the time limit is passed, the integer linear program solver outputs the best solution that has currently been discovered. This solution does not have to be optimal. Note that we do not perform experiments with Metis [108] and Scotch [154], since previous papers, e.g. [162, 163], have already shown that solution quality obtained is much worse than results achieved in the Walshaw benchmark. When averaging over multiple instances, we use the geometric mean in order to give every instance the same influence on the *final score*. We use performance plots to compare the performance of different algorithm configurations on a per-instance basis. For an explanation of these performance plots, we refer the reader to Section 4.4.1.

Instances. We perform experiments on two sets of instances. Set A is used to determine the performance of the integer linear programming optimizations and to tune the algorithm. We obtained these instances from the Florida Sparse Matrix collection [46] and the 10th DIMACS Implementation Challenge [15] to test our algorithm. Set B are all graphs from Chris Walshaw's graph partitioning benchmark archive [174, 190]. This archive is a collection of instances from finite-element applications, VLSI design and is one of the default benchmarking sets for graph partitioning.

Table 7.1 gives basic properties of the graphs from both benchmark sets. We ran the unoptimized integer linear program that solves the graph partitioning problem to optimality from Section 7.4.1 on the five smallest instances from the Walshaw benchmark set. With a time limit of 30 minutes, the solver has only been able to compute a solution for the graphs uk and add32 with $k = 2$. For higher values of k the solver was unable to find any solution in the time limit. Even giving a start-

Table 7.1: Basic properties of the benchmark instances.

Graph	n	m	Graph	n	m
Walshaw Graphs (Set B)			Walshaw Graphs (Set B)		
add20	2 395	7 462	wing	62 032	$\approx$ 121K
data	2 851	15 093	brack2	62 631	$\approx$ 366K
3elt	4 720	13 722	finan512	74 752	$\approx$ 261K
uk	4 824	6 837	fe_tooth	78 136	$\approx$ 452K
add32	4 960	9 462	fe_rotor	99 617	$\approx$ 662K
bcsstk33	8 738	$\approx$ 291K	598a	110 971	$\approx$ 741K
whitaker3	9 800	28 989	fe_ocean	143 437	$\approx$ 409K
crack	10 240	30 380	144	144 649	$\approx$ 1.1M
wing_nodal	10 937	75 488	wave	156 317	$\approx$ 1.1M
fe_4elt2	11 143	32 818	m14b	214 765	$\approx$ 1.7M
vibrobox	12 328	$\approx$ 165K	auto	448 695	$\approx$ 3.3M
bcsstk29	13 992	$\approx$ 302K			
4elt	15 606	45 878	Parameter Tuning (Set A)		
fe_sphere	16 386	49 152	delaunay_n15	32 768	98 274
cti	16 840	48 232	rgg_15	32 768	$\approx$ 160K
memplus	17 758	54 196	2cubes_sphere	101 492	$\approx$ 772K
cs4	22 499	43 858	cfd2	123 440	$\approx$ 1.5M
bcsstk30	28 924	$\approx$ 1.0M	boneS01	127 224	$\approx$ 3.3M
bcsstk31	35 588	$\approx$ 572K	Dubcova3	146 689	$\approx$ 1.7M
fe_pwt	36 519	$\approx$ 144K	G2_circuit	150 102	$\approx$ 288K
bcsstk32	44 609	$\approx$ 985K	thermal2	1 227 087	$\approx$ 3.7M
fe_body	45 087	$\approx$ 163K	as365	3 799 275	$\approx$ 11.4M
t60k	60 005	89 440	adaptive	6 815 744	$\approx$ 13.6M

ing solution does not increase the number of ILPs solved. Hence, we omit further experiments in which we run an ILP solver on the full graph.

7.6.2 Impact of Optimizations

We now evaluate the impact of the optimization strategies for the ILP that we presented in Section 7.4.3. In this section, we use the variant of our local search algorithm in which $\mathcal{K}$ is obtained by starting depth-one breadth-first search at the 25 highest gain vertices, and set the limit on the non-zeros in the ILP to $\mathcal{N} = \infty$. However, due to preliminary experiments we expect the results in terms of speedup to be similar for different vertex selection strategies. To evaluate the ILP performance, we run KaFFPa using the strong preconfiguration on each of the graphs from set A using $\epsilon = 0$ and $k \in \{2, 4, 8, 16, 32, 64\}$ and then use the computed partition as input to each ILP (with the different optimizations). As the optimizations do not change the objective value achieved in the ILP and we only look at ILP formulations solved to optimality in this subsection, we only report running times of our different approaches. We set the time limit of the ILP solver to 30 minutes.

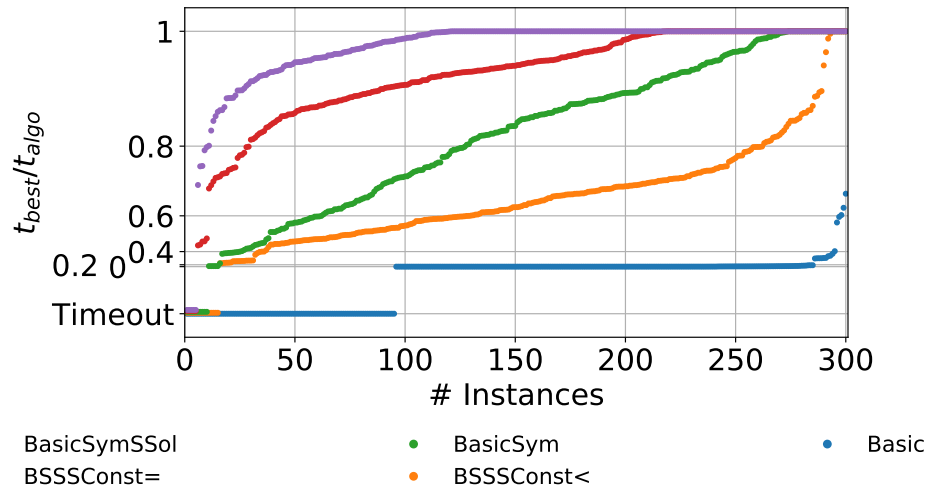


Figure 7.3: Performance plot for five variants of our algorithm: `Basic` does not contain any optimizations; `BasicSym` enables symmetry breaking; `BasicSymSSol` additionally gives the input partition to the ILP solver. The two variants `BSSSConst=` and `BSSSConst<` are the same as `BasicSymSSol` with additional constraints: `BSSSConst=` has the additional constraint that the objective has to be smaller or equal to the start solution, `BSSSConst<` has the constraint that the solution must be better than the start solution.

We use five variants of our algorithm in this experiment: `Basic` does not contain any optimizations; `BasicSym` enables symmetry breaking; `BasicSymSSol` additionally gives the input partition to the ILP solver. The two variants `BSSSConst=` and `BSSSConst<` are the same as `BasicSymSSol` with additional constraints to the solution quality: `BSSSConst=` has the additional constraint that the objective has to be smaller or equal to the start solution, `BSSSConst<` has the constraint that the objective value of a solution must be better than the objective value of the start solution. Figure 7.3 summarises the results.

In our experiments, which are detailed in Figure 7.3, the basic configuration reaches the time limit in 95 out of the 300 runs. Overall, enabling symmetry breaking drastically speeds up computations. On all of the instances which the `Basic` configuration could solve within the time limit, each other configuration is faster than the `Basic` configuration. Symmetry breaking speeds up computations by a factor of 41 in the geometric mean on those instances. The largest obtained speedup on those instances was a factor of 5663 on the graph `adaptive` for $k = 32$. The configuration solves all but the two instances (`boneS01`, $k = 32$) and (`Dubcova3`, $k = 16$) within the time limit. Providing the start solution (`BasicSymSSol`) gives an additional speedup of 22% on average. Over the `Basic` configuration, the average speedup is 50 with the largest speedup being 6495 and the smallest speedup being 1.47. This configuration can solve all instances within the time limit except the in-

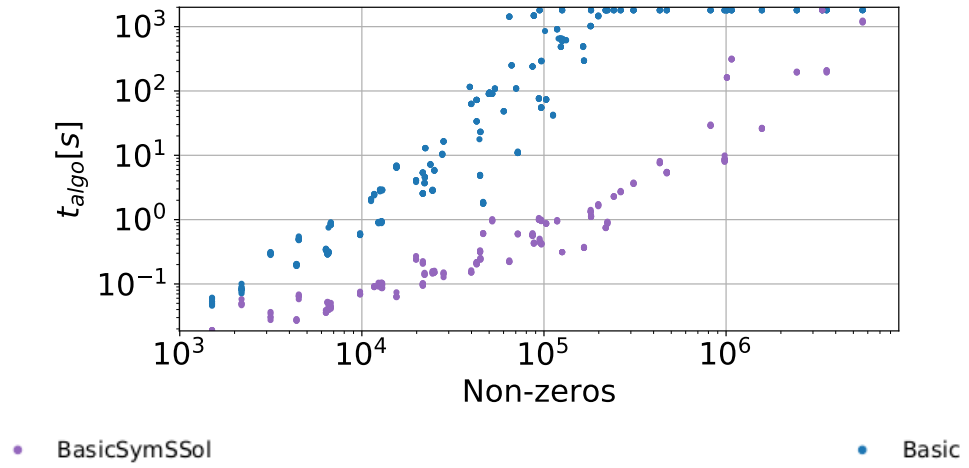


Figure 7.4: Performance of the slowest (`Basic`) and fastest ILPs (`BasicSymSSol`) depending on the number of non-zeros in the ILP.

stance `boneS01` for $k = 32$. Providing the objective function as a constraint (or strictly smaller constraint) does not further reduce the running time of the solver. Instead, the additional constraints even increase the running time. We attribute this to the fact that the solver has to do additional work to evaluate the constraint. We conclude that `BasicSymSSol` is the fastest configuration of the ILP. Hence, we use this configuration in all the following experiments. Moreover, from Figure 7.4 we can see that this configuration can solve most of the instances within the time limit if the number of non-zeros in the ILP is below 10^6. Hence, we set the parameter $\mathcal{N}$ to 10^6 in the following section.

7.6.3 Vertex Selection Rules

We now evaluate the vertex selection strategies to find the set of vertices $\mathcal{K}$ that model the ILP. We look at all strategies described in Section 7.4.4, i.e. `Boundary`, $\texttt{Gain}_\rho$ with the parameter $\rho \in \{-2, -1, 0\}$ as well as $\texttt{TopVertices}_\delta$ for $\delta \in \{1, 2, 3\}$. To evaluate the different selection strategies, we use the best of five runs of KaFFPa-strong on each of the graphs from set A using imbalance $\epsilon = 0$ and number of partitions $k \in \{2, 4, 8, 16, 32, 64\}$ and then use the computed partition as input to the ILP (with different sets $\mathcal{K}$). Table 7.2 summarizes the results of the experiment, i.e. the number of cases in which our algorithm was able to improve the result, the average running time in seconds for these selection strategies as well as the number of cases in which the strategy computed the best result (the partition having the lowest cut). We set the time limit to 2 days to be able to finish almost all runs without running into timeout. For the average running time we exclude all graphs in which at least one algorithm did not finish in 2 days (`rgg_`-

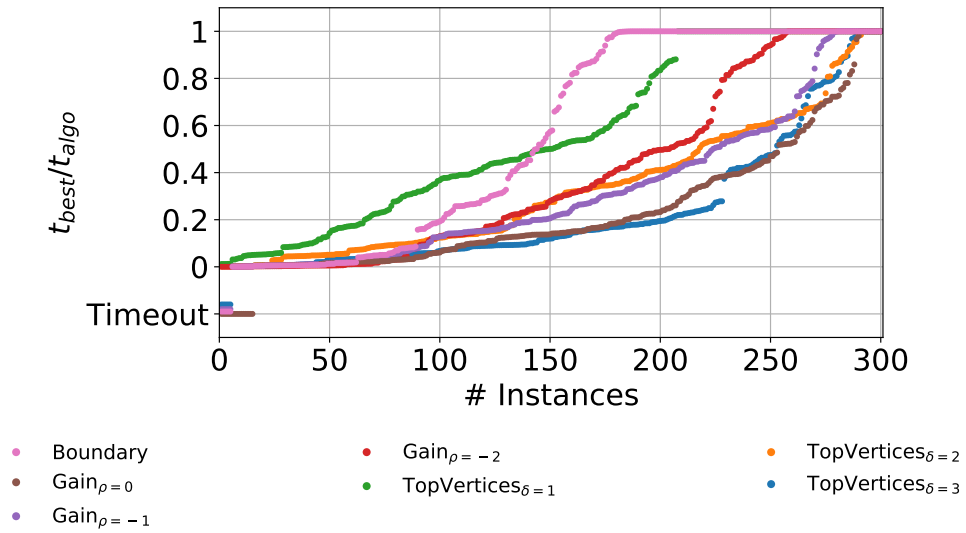


Figure 7.5: Performance plot for all vertex selection strategies.

15 $k = 16$, delaunay_n15 $k = 4$, G2_circuit $k = 4, 8$). If multiple runs share the best result, they are all counted. However, when no algorithm improves the input partition on a graph, we do not count them.

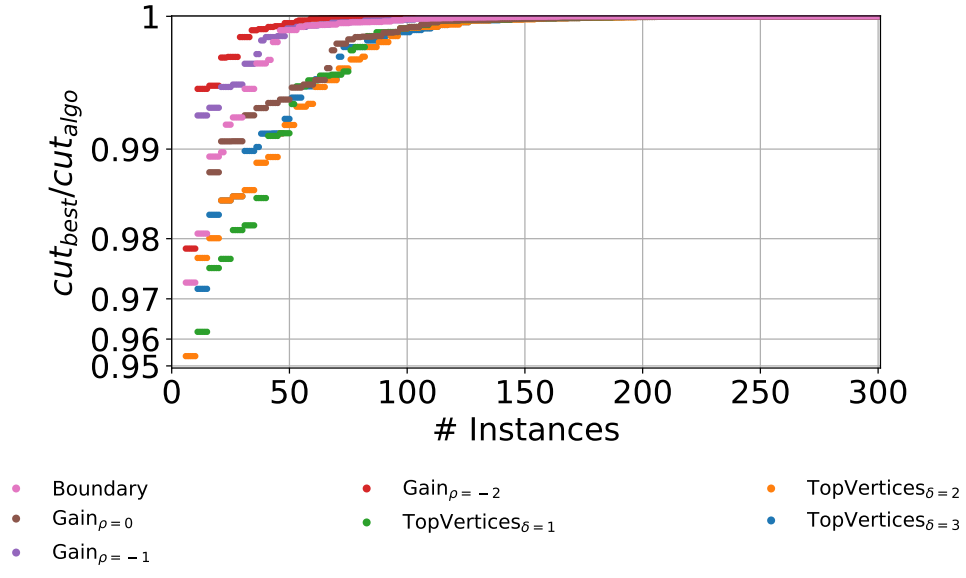


Figure 7.6: Cut value of vertex selection strategies in comparison to the best result given by any strategy.

Table 7.2: From top to bottom: Number of improvements found by different vertex selection rules relative to the total number of instances, average running time of the strategy on the subset of instances (graph, k) in which all strategies finished within the time limit, and the relative number of instances in which the strategy computed the lowest cut. Best values are highlighted in bold.

	Gain			TopVertices			Boundary
k	$\rho = 0$	$\rho = -1$	$\rho = -2$	$\delta = 1$	$\delta = 2$	$\delta = 3$	
	Relative Number of Improvements						
2	**70%**	**70%**	**70%**	50%	**70%**	**70%**	**70%**
4	50%	60%	**80%**	70%	70%	70%	**80%**
8	50%	60%	**78%**	60%	60%	60%	48%
16	30%	50%	**70%**	40%	30%	30%	40%
32	**60%**	**60%**	46%	50%	50%	20%	20%
64	**70%**	**70%**	50%	30%	20%	20%	0%
	Average Running Time						
2	189.943s	292.573s	357.145s	**34.045s**	61.152s	92.452s	684.198s
4	996.934s	628.950s	428.353s	**87.357s**	255.223s	558.578s	1 467.595s
8	552.183s	244.470s	244.046s	105.737s	167.164s	340.900s	**96.763s**
16	118.532s	52.547s	90.363s	53.385s	141.814s	243.957s	**34.790s**
32	40.300s	24.607s	94.146s	27.156s	80.252s	116.023s	**7.596s**
64	15.866s	21.908s	24.253s	14.627s	30.558s	44.813s	**4.187s**
	Relative Number Best Algorithm						
2	20%	**60%**	50%	10%	10%	0%	**60%**
4	10%	0%	**50%**	10%	0%	0%	30%
8	0%	20%	**30%**	10%	10%	10%	26%
16	0%	10%	**54%**	10%	0%	10%	20%
32	0%	8%	**38%**	0%	0%	0%	4%
64	0%	16%	**36%**	0%	0%	0%	0%

Looking at the number of improvements, the Boundary strategy is able to improve the input for small values of k, but with increasing number of blocks k improvements decrease to no improvement in all runs with $k = 64$. Because of the limit on the number of non-zeros, the ILP contains only random boundary vertices for large values of k in this case. Hence, there are not sufficiently many high gain vertices in the model and fewer improvements for large values of k are expected. For small values of $k \in \{2, 4\}$, the Boundary strategy can improve as many as the $\texttt{Gain}_{\rho=-2}$ strategy but the average running times are higher.

For $k = \{2, 4, 8, 16\}$, the strategy $\texttt{Gain}_{\rho=-2}$ has the highest number of improvements, for $k = \{32, 64\}$ it is surpassed by the strategy $\texttt{Gain}_{\rho=-1}$. However, the strategy $\texttt{Gain}_{\rho=-2}$ finds the best cuts in most cases among all tested strategies. Due to the way these strategies are designed, they are able to put a lot of high gain vertices into the model as well as vertices that can be used to balance vertex movements. The TopVertices strategies are overall also able to find a large number of improvements. However, the improvements are typically smaller than for the Gain

strategies. This is due to the fact that the TopVertices strategies grow BFS balls with a predefined depth around high gain vertices first, and later on are not able to include vertices that could be used to balance their movement. Hence, there are less potential vertex movements that could yield an improvement.

For almost all strategies, we can see that the average running time decreases as the number of blocks k increases. This happens because we limit the number of non-zeros $\mathcal{N}$ in our ILP. As the number of non-zeros grows linearly with the underlying model size, the models are far smaller for higher values of k. Using symmetry breaking, we already fixed the block of the k vertices μ_i which represent the vertices not part of $\mathcal{K}$. Thus the ILP solver can quickly prune branches which would place vertices connected heavily to one of these vertices in a different block. Additionally, our data indicates that a large number of small areas in our model results faster in solve times than when the model contains few large areas. The performance plot in Figure 7.5 shows that the strategies Boundary, $\texttt{TopVertices}_{\delta=1}$ and $\texttt{Gain}_{\rho=-2}$ have lower running times than other strategies. These strategies all select a large number of vertices to initialize the breadth-first search. Therefore they output a vertex set $\mathcal{K}$ that is the union of many small areas around these vertices. Variants that initialize the breadth-first search with fewer vertices have fewer areas, however each area is larger. Figure 7.6 shows that for almost all instances the variants $\texttt{Gain}_{\rho=-1}$ and $\texttt{Gain}_{\rho=-2}$ give very good solutions, even if they are not the best variant on that particular instance.

7.6.4 Walshaw Benchmark

In this section, we present the results when running our best configuration on all graphs from Walshaw's benchmark archive. Note that the rules of the benchmark imply that running time is not an issue, but algorithms should achieve the smallest possible cut value while satisfying the balance constraint. We run our algorithm in the following setting: We take existing partitions from the archive and use those as input to our algorithm. As indicated by the experiments in Section 7.6.3, the vertex selection strategies $\texttt{Gain}_{\rho\in\{-1,-2\}}$ perform best for different values of k. Thus we use the variant $\texttt{Gain}_{\rho=-2}$ for $k \leq 16$ and both $\texttt{Gain}_{\rho=-2}$ and $\texttt{Gain}_{\rho=-1}$ otherwise in this section. We repeat the experiment once for each instance (graph, k) and run

Table 7.3: Relative number of improved instances by performing an ILP-based local search in the Walshaw Benchmark starting from current entries reported in the Walshaw benchmark.

$\epsilon\backslash k$	2	4	8	16	32	64	overall
0%	6%	18%	26%	50%	62%	68%	38%
1%	12%	9%	24%	26%	47%	59%	29%
3%	6%	6%	12%	29%	47%	71%	28%
5%	6%	18%	15%	29%	53%	76%	33%

our algorithm for $k = \{2, 4, 8, 16, 32, 64\}$ and $\epsilon \in \{0, 1\%, 3\%, 5\%\}$. For larger values of $k \in \{32, 64\}$, we strengthen our strategy and use $\mathcal{N} = 5 \cdot 10^6$ as a bound for the number of non-zeros. We set the time limit to two hours. Table 7.3 summarizes the results. Detailed per-instance results are given in Section 7.8.

When running our algorithm using the currently best partitions provided in the benchmark, we are able to improve 38% of the currently reported perfectly balanced results. We are able to improve a larger number of results for larger values of k, more specifically, out of the partitions with $k \geq 16$, we can improve 60% of all perfectly balanced partitions. There is a wide range of improvements with the smallest improvement being 0.0008% for graph auto with $k = 32$ and $\epsilon = 3\%$ and with the largest improvement that we found being 1.72% for fe_body for $k = 32$ and $\epsilon = 0\%$. The largest absolute improvement we found is 117 for bcsstk32 with $k = 64$ and $\epsilon = 0\%$. In general, the total number of improvements is lower if some imbalance is allowed. This is also expected since traditional local search methods have a larger amount of freedom to move vertices. However, the number of improvements still shows that the method is also able to improve many partitions even if some imbalance is allowed. We submitted the improved partitionings of our ILP-based local search algorithm to the Walshaw graph partitioning archive [190], where it is denoted by *-ILP.

7.6.5 Integration into KaBaPE

Section 7.5 shows how we integrate our approach into the memetic graph partitioning algorithm KaBaPE. We detail the two new operations that we introduce to KaBaPE. In KaBaPE, we use the standard parameters given by the original authors [163].

We run experiments on the small and medium sized graphs of the Walshaw graph partitioning benchmark archive [190] as shown in Tables 7.9 and 7.10, which are the graphs also used in the original KaBaPE paper [163]. Similar to their experiments, we also give 2 hours for each problem. Afterwards, we perform post-processing by running the algorithm for 1 hour on the overlap graph given by the best 100 unique partitions. Note, that even though we have a total running time of 3 hours instead of 2 hours in the results of KaBaPE [163], all problems in which KaBaPE+ILP outperforms the current best solution in the Walshaw archive, the solution was already better before post processing.

We run experiments on the problems that have $k \in \{8, 16, 32, 64\}$ and $\epsilon \in \{3\%, 5\%\}$. These are the hard instances of the benchmark, in which algorithms do not just reproduce the same solution as previous approaches. Figures 7.7 and 7.8 show the development of the fittest individual over the course of the algorithm for a variety of graphs. A summary of the results is shown in Table 7.4, complete results for KaBaPE+ILP on all problems are given in Tables 7.9 and 7.10 in the appendix.

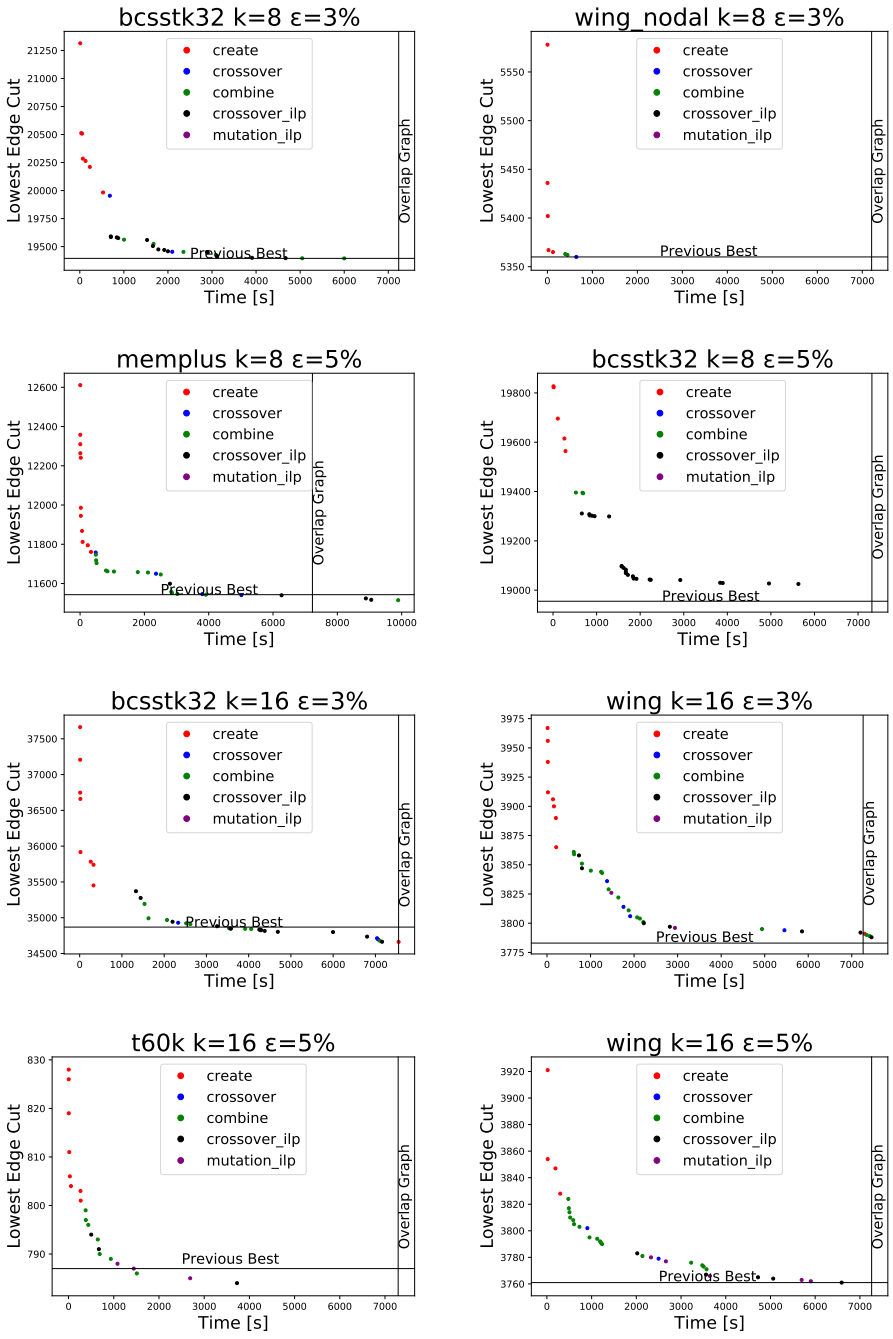


Figure 7.7: Improvement of best partition over time, compared to previously best solution.

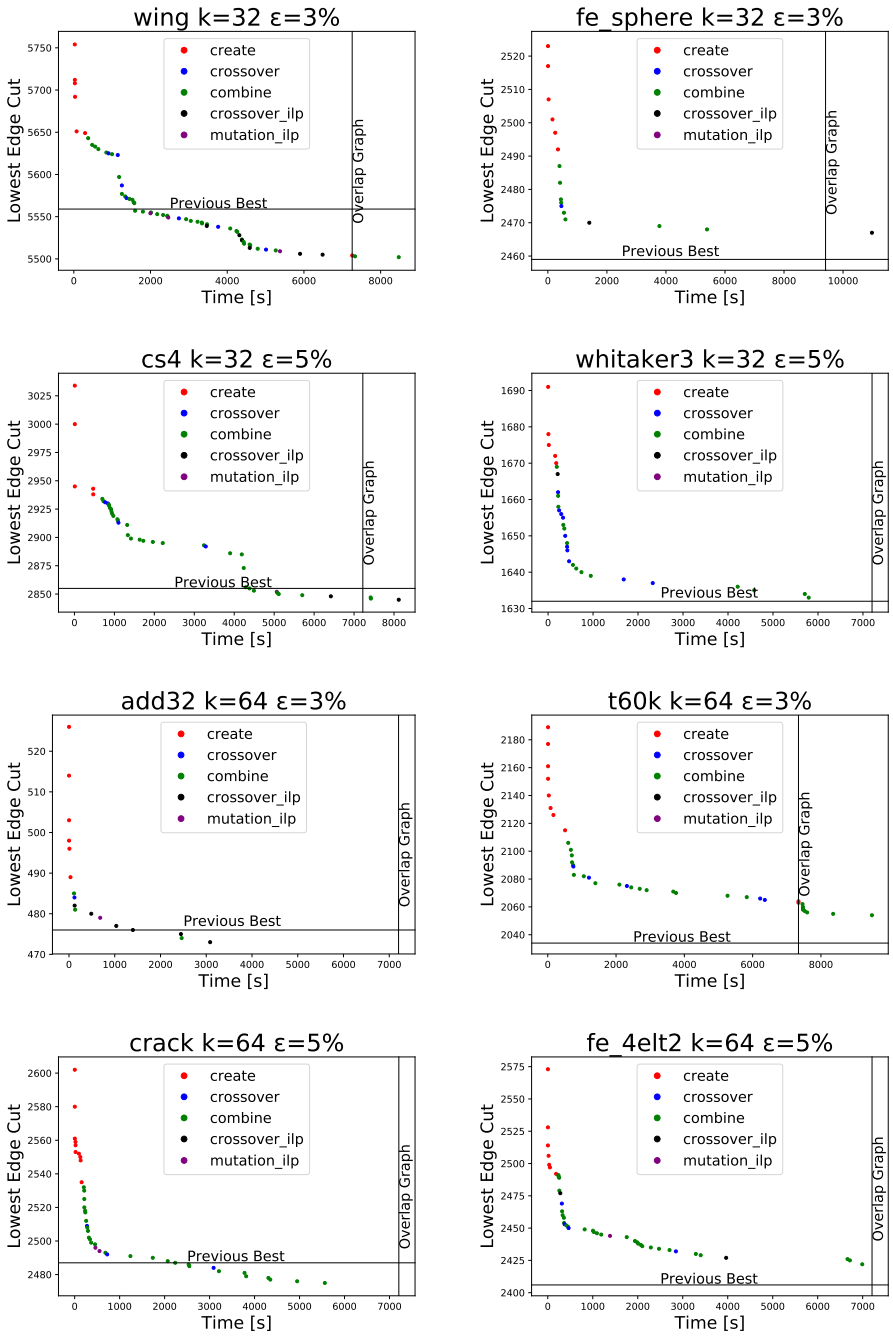


Figure 7.8: Improvement of best partition over time, compared to previously best solution.

Table 7.4: Relative number of improved instances by KaBaPE in the Walshaw Benchmark starting from scratch.

$\epsilon \backslash k$	8	16	32	64	overall
3%	16%	4%	20%	4%	11%
5%	8%	32%	28%	24%	23%

On those 200 problems we manage to improve the best known solution in 35 cases. The previously best results hereby include the improvements given in the previous experiments. In 62 of the problems, KaBaPE+ILP reproduces the best known cut. The highest improvement can be found on graph `bcsstk32`, $k = 32$, $\epsilon = 5\%$, where we improve the best known edge cut by a value of more than 800.

Note that feeding the best known solution from the Walshaw archive into the population as a seed partition does not increase the quality of the solution. For all 35 instances in which KaBaPE+ILP outperforms the best solution from the Walshaw archive, a larger improvement is only seen in 3 instances when additionally using a seed partition.

7.7 Conclusion

We presented a novel meta-heuristic for the balanced graph partitioning problem. Our approach is based on an integer linear program that solves a model to combine unconstrained vertex movements into a global feasible improvement. Through a given input partition, we were able to use symmetry breaking and other techniques that make the approach scale to large inputs. In Walshaw's benchmark, we were able to improve a large number of partitions.

We also integrated the algorithm into the KaHIP framework by adding new crossover operations based on integer linear programs into the evolutionary algorithm KaBaPE [163]. This extended evolutionary algorithm produces high quality partitions from scratch. On half of the hard problems from Walshaw's benchmark, our new algorithm produces a result that is at least as good as the previously best result. On 17%, the solution given is better than the previous best solution.

We would like to look at other objective functions as long as they can be modelled linearly. Moreover, we want to investigate whether this kind of contractions can be useful for other ILPs. Besides using other exact techniques like branch-and-bound to solve the model, it may also be worthwhile to use a heuristic algorithm instead. In the Walshaw graph partitioning benchmark [190], the results given by this algorithm are denoted by KaBaPE+ILP.

7.8 Additional Tables

Table 7.5: Improvement of existing partitions from the Walshaw benchmark with $\epsilon = 0\%$ using our ILP approach. In each k-column the results computed by our approach are on the left and the current Walshaw cuts are on the right. Results achieved by $\texttt{Gain}_{\rho=-1}$ are marked with ˆand results achieved by $\texttt{Gain}_{\rho=-2}$ are marked with *.

Graph / k	2		4		8		16		32		64	
add20	596	596	1 151	1 151	1 681	1 681	2 040	2 040	*2 360	2 361	^2 947	2 949
data	189	189	382	382	668	668	1 127	1 127	1 799	1 799	2 839	2 839
3elt	90	90	201	201	345	345	573	573	960	960	1 532	1 532
uk	19	19	41	41	83	83	145	145	*^246	247	408	408
add32	11	11	34	34	67	67	118	118	213	213	485	485
bcsstk33	10 171	10 171	21 717	21 717	34 437	34 437	54 680	54 680	77 414	77 414	107 185	107 185
whitaker3	127	127	381	381	656	656	1 085	1 085	1 668	1 668	2 491	2 491
crack	184	184	366	366	679	679	1 088	1 088	*1 678	1 679	2 535	2 535
wing_nodal	1 707	1 707	3 575	3 575	5 435	5 435	*8 333	8 334	11 768	11 768	*^15 774	15 775
fe_4elt2	130	130	349	349	607	607	1 007	1 007	1 614	1 614	2 475	2 478
vibrobox	10 343	10 343	18 976	18 976	24 484	24 484	*^31 848	31 850	*39 474	39 477	*46 568	46 571
bcsstk29	2 843	2 843	8 035	8 035	13 975	13 975	21 905	21 905	*34 733	34 737	55 241	55 241
4elt	139	139	326	326	545	545	*^933	934	1 551	1 551	^2 564	2 565
fe_sphere	386	386	768	768	1 156	1 156	1 714	1 714	2 488	2 488	3 543	3 543
cti	334	334	954	954	1 788	1 788	2 793	2 793	4 046	4 046	5 629	5 629
memplus	*5 499	5 513	*9 442	9 448	*^11 710	11 712	^12 893	12 895	*^13 947	13 953	^16 188	16 223
cs4	369	369	932	932	1 440	1 440	2 075	2 075	*2 907	2 928	^4 025	4 027
bcsstk30	6 394	6 394	16 651	16 651	34 846	34 846	*^70 407	70 408	113 336	113 336	*171 148	171 153
bcsstk31	2 762	2 762	7 351	7 351	*13 280	13 283	*23 857	23 869	*37 143	37 158	*57 354	57 402
fe_pwt	340	340	705	705	1 447	1 447	2 830	2 830	*^5 574	5 575	^8 177	8 180
bcsstk32	4 667	4 667	9 311	9 311	*^20 008	20 009	*^36 249	36 250	*60 013	60 038	*90 778	90 895
fe_body	262	262	599	599	1 033	1 033	*1 722	1 736	^2 797	2 846	*4 728	4 730
t60k	79	79	209	209	456	456	^812	813	1 323	1 323	*^2 074	2 077
wing	789	789	1 623	1 623	2 504	2 504	^3 870	3 876	^5 592	5 594	^7 622	7 625
brack2	731	731	3 084	3 084	7 140	7 140	11 570	11 570	^17 382	17 387	*25 805	25 808
finan512	162	162	324	324	648	648	1 296	1 296	2 592	2 592	10 560	10 560
fe_tooth	3 816	3 816	*6 888	6 889	*11 414	11 418	*^17 352	17 355	*24 879	24 885	*34 234	34 240
fe_rotor	2 098	2 098	7 222	7 222	^12 838	12 841	*20 389	20 391	*31 132	31 141	*45 677	45 687
598a	2 398	2 398	8 001	8 001	*15 921	15 922	*25 694	25 702	*38 576	38 581	*^56 094	56 097
fe_ocean	464	464	1 882	1 882	4 188	4 188	7 713	7 713	^12 667	12 684	^20 061	20 069
144	6 486	6 486	^15 194	15 196	25 273	25 273	*37 566	37 571	*55 467	55 475	*77 391	77 402
wave	8 677	8 677	*17 193	17 198	*29 188	29 198	*42 639	42 646	*61 100	61 108	^83 987	83 994
m14b	3 836	3 836	*13 061	13 062	*25 834	25 838	*42 161	42 172	*65 469	65 529	^96 446	96 452
auto	*^10 101	10 103	*27 092	27 094	*45 991	46 014	^77 391	77 418	*121 911	121 944	^172 966	172 973

Table 7.6: Improvement of existing partitions from the Walshaw benchmark with $\epsilon = 1\%$ using our ILP approach. In each k-column the results computed by our approach are on the left and the current Walshaw cuts are on the right. Results achieved by $\texttt{Gain}_{\rho=-1}$ are marked with ˆand results achieved by $\texttt{Gain}_{\rho=-2}$ are marked with *.

Graph / k	2		4		8		16		32		64	
add20	585	585	1 147	1 147	*^**1 680**	1 681	2 040	2 040	2 361	2 361	2 949	2 949
data	188	188	376	376	656	656	1 121	1 121	1 799	1 799	2 839	2 839
3elt	89	89	199	199	340	340	568	568	953	953	1 532	1 532
uk	19	19	40	40	80	80	142	142	246	246	408	408
add32	10	10	33	33	66	66	117	117	212	212	485	485
bcsstk33	10 097	10 097	21 338	21 338	34 175	34 175	54 505	54 505	77 195	77 195	106 902	106 902
whitaker3	126	126	380	380	654	654	1 083	1 083	1 664	1 664	2 480	2 480
crack	183	183	362	362	676	676	1 081	1 081	1 669	1 669	2 523	2 523
wing_nodal	1 695	1 695	3 559	3 559	5 401	5 401	8 302	8 302	*^**11 731**	11 733	*^**15 734**	15 736
fe_4elt2	130	130	349	349	603	603	1 000	1 000	1 608	1 608	^**2 470**	2 472
vibrobox	10 310	10 310	18 943	18 943	24 422	24 422	*^**31 710**	31 712	*^**39 396**	39 400	*^**46 529**	46 541
bcsstk29	2 818	2 818	8 029	8 029	13 891	13 891	21 694	21 694	34 606	34 606	*^**54 950**	54 951
4elt	138	138	320	320	532	532	927	927	1 535	1 535	2 546	2 546
fe_sphere	386	386	766	766	1 152	1 152	1 708	1 708	2 479	2 479	3 534	3 534
cti	318	318	944	944	1 746	1 746	2 759	2 759	3 993	3 993	5 594	5 594
memplus	*^**5 452**	5 457	9 385	9 385	11 672	11 672	12 873	12 873	^**13 931**	13 933	^**16 091**	16 110
cs4	366	366	925	925	1 434	1 434	2 061	2 061	2 903	2 903	^**3 981**	3 982
bcsstk30	6 335	6 335	16 583	16 583	34 565	34 565	69 912	69 912	112 365	112 365	170 059	170 059
bcsstk31	2 699	2 699	7 272	7 272	*^**13 134**	13 137	*^**23 333**	23 339	*^**37 057**	37 061	*^**57 000**	57 025
fe_pwt	340	340	704	704	1 432	1 432	2 797	2 797	5 514	5 514	^**8 128**	8 130
bcsstk32	4 667	4 667	9 180	9 180	*^**19 612**	19 624	35 617	35 617	*^**59 501**	59 504	*^**89 893**	89 905
fe_body	262	262	598	598	1 023	1 023	1 714	1 714	^**2 748**	2 756	*^**4 664**	4 674
t60k	75	75	208	208	454	454	805	805	1 313	1 313	2 062	2 062
wing	784	784	1 610	1 610	2 474	2 474	3 857	3 857	^**5 576**	5 577	^**7 585**	7 586
brack2	708	708	3 013	3 013	7 029	7 029	11 492	11 492	*^**17 120**	17 128	^**25 604**	25 607
finan512	162	162	324	324	648	648	1 296	1 296	2 592	2 592	10 560	10 560
fe_tooth	3 814	3 814	*^**6 843**	6 844	11 358	11 358	*^**17 264**	17 265	*^**24 799**	24 804	^**34 159**	34 170
fe_rotor	2 031	2 031	7 158	7 158	12 616	12 616	^**20 146**	20 152	*^**30 975**	30 982	*^**45 304**	45 321
598a	2 388	2 388	7 948	7 948	15 831	15 831	*^**25 620**	25 624	^**38 410**	38 422	*^**55 867**	55 882
fe_ocean	^**385**	387	1 813	1 813	*^**4 060**	4 063	7 616	7 616	^**12 523**	12 524	*^**19 851**	19 852
144	*^**6 476**	6 478	15 140	15 140	*^**25 225**	25 232	*^**37 341**	37 347	*^**55 258**	55 277	*^**76 964**	76 980
wave	*^**8 656**	8 657	^**16 745**	16 747	*^**28 749**	28 758	*^**42 349**	42 354	*^**60 617**	60 625	^**83 451**	83 466
m14b	3 826	3 826	12 973	12 973	*^**25 626**	25 627	*^**42 067**	42 080	*^**64 684**	64 697	^**96 145**	96 169
auto	9 949	9 949	*^**26 611**	26 614	*^**45 424**	45 429	*^**76 533**	76 539	*^**120 470**	120 489	^**171 866**	171 880

Table 7.7: Improvement of existing partitions from the Walshaw benchmark with $\epsilon = 3\%$ using our ILP approach. In each k-column the results computed by our approach are on the left and the current Walshaw cuts are on the right. Results achieved by $\texttt{Gain}_{\rho=-1}$ are marked with ˆand results achieved by $\texttt{Gain}_{\rho=-2}$ are marked with *.

Graph / k	2		4		8		16		32		64	
add20	560	560	1 134	1 134	1 673	1 673	2 030	2 030	2 346	2 346	2 920	2 920
data	185	185	369	369	638	638	1 088	1 088	1 768	1 768	***2 781**	2 783
3elt	87	87	198	198	334	334	561	561	944	944	1 512	1 512
uk	18	18	39	39	78	78	139	139	240	240	397	397
add32	10	10	33	33	66	66	117	117	212	212	476	476
bcsstk33	10 064	10 064	20 762	20 762	34 065	34 065	54 354	54 354	76 749	76 749	***105 737**	105 742
whitaker3	126	126	378	378	649	649	1 073	1 073	1 647	1 647	***2 456**	2 459
crack	182	182	360	360	671	671	1 070	1 070	1 655	1 655	*ˆ**2 487**	2 489
wing_nodal	1 678	1 678	3 534	3 534	5 360	5 360	8 244	8 244	***11 630**	11 632	*ˆ**15 612**	15 613
fe_4elt2	130	130	341	341	595	595	990	990	1 593	1 593	ˆ**2 431**	2 435
vibrobox	10 310	10 310	18 736	18 736	24 153	24 153	*ˆ**31 440**	31 443	***39 197**	39 201	***46 231**	46 235
bcsstk29	2 818	2 818	7 971	7 971	13 710	13 710	21 258	21 258	33 807	33 807	54 382	54 382
4elt	137	137	319	319	522	522	901	901	1 519	1 519	2 512	2 512
fe_sphere	384	384	764	764	1 152	1 152	1 696	1 696	2 459	2 459	*ˆ**3 503**	3 505
cti	318	318	916	916	1 714	1 714	2 727	2 727	3 941	3 941	***5 522**	5 524
memplus	*ˆ**5 352**	5 353	9 309	9 309	*ˆ**11 584**	11 586	12 834	12 834	***13 887**	13 895	***15 950**	15 953
cs4	360	360	917	917	*ˆ**1 423**	1 424	2 043	2 043	***2 884**	2 885	ˆ**3 979**	3 980
bcsstk30	6 251	6 251	16 372	16 372	34 137	34 137	69 357	69 357	110 334	110 334	***168 271**	168 274
bcsstk31	2 676	2 676	7 148	7 148	12 962	12 962	***22 949**	22 956	***36 567**	36 587	***56 025**	56 038
fe_pwt	340	340	700	700	1 410	1 410	2 754	2 754	5 403	5 403	8 036	8 036
bcsstk32	4 667	4 667	8 725	8 725	19 485	19 485	*ˆ**34 869**	34 875	ˆ**58 739**	58 740	***89 478**	89 479
fe_body	262	262	598	598	1 016	1 016	1 693	1 693	*ˆ**2 708**	2 709	*ˆ**4 522**	4 523
t60k	71	71	203	203	449	449	792	792	1 302	1 302	*ˆ**2 034**	2 036
wing	773	773	1 593	1 593	2 451	2 451	ˆ**3 783**	3 784	5 559	5 559	7 560	7 560
brack2	684	684	2 834	2 834	6 778	6 778	***11 253**	11 256	*ˆ**16 981**	16 982	*ˆ**25 362**	25 363
finan512	162	162	324	324	648	648	1 296	1 296	2 592	2 592	10 560	10 560
fe_tooth	3 788	3 788	6 756	6 756	11 241	11 241	***17 107**	17 108	***24 623**	24 625	***33 779**	33 795
fe_rotor	1 959	1 959	*ˆ**7 049**	7 050	12 445	12 445	***19 863**	19 867	***30 579**	30 587	***44 811**	44 822
598a	2 367	2 367	7 816	7 816	15 613	15 613	*ˆ**25 379**	25 380	***38 093**	38 105	***55 358**	55 364
fe_ocean	311	311	1 693	1 693	3 920	3 920	7 405	7 405	ˆ**12 283**	12 288	19 518	19 518
144	*ˆ**6 430**	6 432	15 064	15 064	***24 901**	24 905	*ˆ**36 999**	37 003	***54 800**	54 806	***76 548**	76 557
wave	8 591	8 591	ˆ**16 633**	16 638	28 494	28 494	42 139	42 139	***60 334**	60 356	***82 809**	82 811
m14b	3 823	3 823	12 948	12 948	25 390	25 390	41 778	41 778	ˆ**64 354**	64 364	*ˆ**95 575**	95 587
auto	9 673	9 673	25 789	25 789	*ˆ**44 724**	44 732	*ˆ**75 665**	75 679	ˆ**119 131**	119 132	ˆ**170 295**	170 314

Table 7.8: Improvement of existing partitions from the Walshaw benchmark with $\epsilon = 5\%$ using our ILP approach. In each k-column the results computed by our approach are on the left and the current Walshaw cuts are on the right. Results achieved by $\texttt{Gain}_{\rho=-1}$ are marked with ˆand results achieved by $\texttt{Gain}_{\rho=-2}$ are marked with *.

Graph / k	2		4		8		16		32		64	
add20	536	536	1 120	1 120	1 657	1 657	2 027	2 027	2 341	2 341	2 920	2 920
data	181	181	363	363	628	628	1 076	1 076	1 743	1 743	2 747	2 747
3elt	87	87	197	197	329	329	557	557	930	930	1 498	1 498
uk	18	18	39	39	75	75	137	137	236	236	394	394
add32	10	10	33	33	63	63	117	117	212	212	476	476
bcsstk33	9 914	9 914	20 158	20 158	33 908	33 908	54 119	54 119	**ˆ76 070**	76 079	***105 297**	105 309
whitaker3	126	126	376	376	644	644	1 068	1 068	1 632	1 632	***ˆ2 425**	2 429
crack	182	182	360	360	666	666	1 063	1 063	1 655	1 655	***ˆ2 487**	2 489
wing_nodal	1 668	1 668	3 520	3 520	5 339	5 339	8 160	8 160	***11 533**	11 536	***ˆ15 514**	15 515
fe_4elt2	130	130	335	335	578	578	979	979	1 571	1 571	**ˆ2 406**	2 412
vibrobox	10 310	10 310	18 690	18 690	23 924	23 924	**ˆ31 216**	31 218	***ˆ38 823**	38 826	***45 987**	45 994
bcsstk29	2 818	2 818	7 925	7 925	13 540	13 540	20 924	20 924	33 450	33 450	53 703	53 703
4elt	137	137	315	315	515	515	887	887	1 493	1 493	**ˆ2 478**	2 482
fe_sphere	384	384	762	762	1 152	1 152	1 678	1 678	2 427	2 427	3 456	3 456
cti	318	318	889	889	1 684	1 684	2 701	2 701	3 904	3 904	**ˆ5 460**	5 462
memplus	***ˆ5 253**	5 263	***9 281**	9 292	***ˆ11 540**	11 543	12 799	12 799	***13 857**	13 867	***15 875**	15 877
cs4	353	353	908	908	1 420	1 420	**ˆ2 042**	2 043	***2 855**	2 859	***ˆ3 959**	3 962
bcsstk30	6 251	6 251	16 165	16 165	34 068	34 068	68 323	68 323	109 368	109 368	***166 787**	166 790
bcsstk31	***ˆ2 660**	2 662	7 065	7 065	***ˆ12 823**	12 825	***22 718**	22 724	***36 354**	36 358	***55 250**	55 258
fe_pwt	340	340	700	700	1 405	1 405	2 737	2 737	**ˆ5 305**	5 306	**ˆ7 956**	7 959
bcsstk32	4 622	4 622	8 441	8 441	18 955	18 955	34 374	34 374	58 352	58 352	***88 595**	88 598
fe_body	262	262	588	588	1 012	1 012	1 683	1 683	***ˆ2 677**	2 678	**ˆ4 500**	4 501
t60k	65	65	195	195	441	441	787	787	***1 289**	1 291	***ˆ2 013**	2 015
wing	770	770	***1 589**	1 590	2 440	2 440	3 775	3 775	***ˆ5 512**	5 513	**ˆ7 529**	7 534
brack2	660	660	2 731	2 731	6 592	6 592	***11 052**	11 055	16 765	16 765	***25 100**	25 108
finan512	162	162	324	324	648	648	1 296	1 296	2 592	2 592	10 560	10 560
fe_tooth	3 773	3 773	6 687	6 687	***ˆ11 147**	11 151	***16 983**	16 985	**ˆ24 270**	24 274	***33 387**	33 403
fe_rotor	1 940	1 940	6 779	6 779	***12 308**	12 309	***19 677**	19 680	***30 355**	30 356	***44 368**	44 381
598a	2 336	2 336	***7 722**	7 724	15 413	15 413	25 198	25 198	**ˆ37 632**	37 644	***54 677**	54 684
fe_ocean	311	311	1 686	1 686	3 886	3 886	7 338	7 338	**ˆ12 033**	12 034	***ˆ19 391**	19 394
144	6 345	6 345	**ˆ14 978**	14 981	***24 174**	24 179	***ˆ36 608**	36 608	***54 160**	54 168	***75 753**	75 777
wave	8 524	8 524	***16 528**	16 531	28 489	28 489	***ˆ42 024**	42 025	***ˆ59 608**	59 611	***81 989**	82 006
m14b	3 802	3 802	***ˆ12 858**	12 859	25 126	25 126	***41 097**	41 098	***63 397**	63 411	***94 123**	94 140
auto	9 450	9 450	25 271	25 271	44 206	44 206	***74 266**	74 272	***118 998**	119 004	**ˆ169 260**	169 290

Table 7.9: Complete results for KaBaPE+ILP compared to best of previous approaches on Walshaw's benchmark with $\epsilon = 3\%$.

Graph / k,ϵ	$k = 8, \epsilon = 3\%$		$k = 16, \epsilon = 3\%$		$k = 32, \epsilon = 3\%$		$k = 64, \epsilon = 3\%$	
add20	**1 664**	1 673	2 030	2 030	2 350	2 346	2 932	2 920
data	638	638	1 088	1 088	1 768	1 768	2 791	2 781
3elt	334	334	561	561	944	944	1 521	1 512
uk	**77**	78	139	139	**239**	240	401	397
add32	66	66	117	117	212	212	**471**	476
bcsstk33	34 065	34 065	54 354	54 354	76 879	76 749	106 263	105 737
whitaker3	649	649	1 077	1 073	1 653	1 647	2 468	2 456
crack	671	671	1 070	1 070	1 655	1 655	2 497	2 487
wing_nodal	5 360	5 360	8 255	8 244	11 721	11 630	15 637	15 612
fe_4elt2	**594**	595	990	990	**1 592**	1 593	2 452	2 431
vibrobox	24 209	24 153	32 475	31 440	39 376	39 197	46 792	46 231
bcsstk29	13 710	13 710	21 271	21 258	33 831	33 807	54 501	54 382
4elt	522	522	901	901	1 519	1 519	2 523	2 512
fe_sphere	1 152	1 152	1 696	1 696	2 467	2 459	3 509	3 503
cti	1 714	1 714	2 728	2 727	3 948	3 941	5 561	5 522
memplus	11 589	11 584	13 015	12 834	14 109	13 887	16 371	15 950
cs4	1 423	1 423	2 057	2 043	**2 876**	2 884	4 015	3 979
bcsstk30	34 137	34 137	69 399	69 357	112 124	110 334	170 796	168 271
bcsstk31	12 967	12 962	22 949	22 949	37 069	36 567	56 634	56 025
fe_pwt	1 410	1 410	2 756	2 754	5 436	5 403	8 076	8 036
bcsstk32	**19 395**	19 485	**34 662**	34 869	**58 060**	58 739	90 997	89 478
fe_body	1 016	1 016	1 697	1 693	2 754	2 708	4 596	4 522
t60k	449	449	793	792	1 305	1 302	2 054	2 034
wing	**2 449**	2 451	3 788	3 783	**5 502**	5 559	7 620	7 560
brack2	6 779	6 778	11 388	11 253	17 012	16 981	25 671	25 362

Table 7.10: Complete results for KaBaPE+ILP compared to best of previous approaches on Walshaw's benchmark with $\epsilon = 5\%$.

Graph / k,ϵ	$k = 8, \epsilon = 5\%$		$k = 16, \epsilon = 5\%$		$k = 32, \epsilon = 5\%$		$k = 64, \epsilon = 5\%$	
add20	**1 651**	1 657	**2 024**	2 027	2 341	2 341	2 925	2 920
data	628	628	1 076	1 076	1 747	1 743	2 761	2 747
3elt	329	329	557	557	931	930	1 499	1 498
uk	75	75	137	137	**235**	236	**392**	394
add32	63	63	117	117	212	212	**471**	476
bcsstk33	33 908	33 908	54 137	54 119	76 213	76 070	105 746	105 297
whitaker3	644	644	1 068	1 068	1 633	1 632	2 439	2 425
crack	666	666	**1 062**	1 063	**1 641**	1 655	**2 470**	2 487
wing_nodal	5 339	5 339	8 170	8 160	11 608	11 533	15 563	15 514
fe_4elt2	578	578	981	979	**1 566**	1 571	2 420	2 406
vibrobox	23 924	23 924	32 277	31 216	39 350	38 823	46 365	45 987
bcsstk29	13 540	13 540	20 924	20 924	33 451	33 450	54 136	53 703
4elt	515	515	887	887	1 494	1 493	2 493	2 478
fe_sphere	1 152	1 152	1 679	1 678	2 427	2 427	3 456	3 456
cti	1 684	1 684	2 701	2 701	3 913	3 904	5 470	5 460
memplus	**11 515**	11 543	12 954	12 799	14 053	13 857	16 174	15 875
cs4	1 421	1 420	2 043	2 042	**2 845**	2 855	**3 949**	3 959
bcsstk30	34 069	34 068	68 996	68 323	110 680	109 368	169 824	166 787
bcsstk31	12 851	12 823	**22 626**	22 718	**36 339**	36 354	55 864	55 250
fe_pwt	1 405	1 405	2 743	2 737	5 329	5 305	7 998	7 956
bcsstk32	19 025	18 955	**34 163**	34 374	**57 529**	58 352	89 460	88 595
fe_body	1 012	1 012	**1 682**	1 683	2 677	2 677	**4 485**	4 500
t60k	441	441	**784**	787	1 290	1 289	2 028	2 013
wing	2 441	2 440	**3 761**	3 775	**5 464**	5 512	**7 493**	7 529
brack2	6 592	6 592	**11 046**	11 052	16 981	16 765	25 397	25 100

Part III

The Multiterminal Cut Problem

CHAPTER 8

Shared-memory Branch-and-Reduce for Multiterminal Cut

We introduce the fastest known exact algorithm for the multiterminal cut problem with k terminals. In particular, we engineer existing as well as new highly effective data reduction rules to transform the graph into a smaller equivalent instance. We use these rules within a branch-and-reduce framework as well as to boost the performance of an ILP formulation. In addition, we present a local search algorithm that can significantly improve a given solution to the multiterminal cut problem. 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; and give better solutions for problems that are too large to be solved to optimality. Furthermore, we give an inexact heuristic algorithm that computes high-quality solutions for very hard instances in reasonable time.

The content of this chapter is based on [88] and [91].

8.1 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 blocks 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 [179], clustering [155] and bioinformatics [102, 142, 189]. It

is a fundamental combinatorial optimization problem which was first formulated by Dahlhaus et al. [43] and Cunningham [41]. It is NP-hard for $k \geq 3$ [43], 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. [43] 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. [29] 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 $\mathcal{W}(G)$. Marx [135] proves that the multiterminal cut problem is FPT and Chen et al. [38] give the first FPT algorithm with a running time of $4^{\mathcal{W}(G)} \cdot n^{\mathcal{O}(1)}$, later improved by Xiao [195] to $2^{\mathcal{W}(G)} \cdot n^{\mathcal{O}(1)}$ and by Cao et al. [33] 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 minimum s-t-cut problem and its equivalent counterpart, the maximum s-t-flow problem [63] were first formulated by Harris et al. [82]. Ford and Fulkerson [63] gave the first algorithm for the problem with a running time of $\mathcal{O}(mn\mathcal{W})$. One of the fastest known algorithms in practice is the push-relabel algorithm of Goldberg and Tarjan [77] 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 [11, 40, 132, 138]. PageRank [153] is reported to be well suited for the problem [115] and there are multiple approaches that aim to make PageRank perform even better [9, 20, 126]. Another approach is to use machine learning methods such as geometric [197] or relational [133] neighborhood classifiers.

Closely related to the problem is also the global minimum cut problem, which is discussed in Part I of this work. In this chapter, we adapt some of the reductions discussed there that are applicable to the minimum multiterminal cut problem and use them to reduce the size of the problem.

Our work on the multiterminal cut problem 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. Using this observation we combine the branch-and-reduce framework with an ILP formulation and solve subproblems using the solver better suited to the subproblem in question. In addition, we present a local search algorithm that can significantly improve a given solution to the multiterminal cut problem. 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. Additionally, we give an inexact algorithm that gives high-quality solutions to hard problems in reasonable time, but does not give an optimality guarantee.

8.2 Preliminaries

8.2.1 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 $c : E \to \mathbb{N}$. We use the same terminology to describe graphs as in Parts I and II of this dissertation. 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. $c(E \cap \bigcup_{i<j} V_i \times V_j)$.

8.2.2 Multiterminal Cuts

A *multiterminal cut* for a graph $G = (V, E)$ with 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 $\mathcal{V} = \{\mathcal{V}_1, \dots, \mathcal{V}_k\}$. $\mathcal{C}$ 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 $\mathcal{W}(G) = c(\mathcal{C}(G))$. 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 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 $\lambda(G, s, T)$.

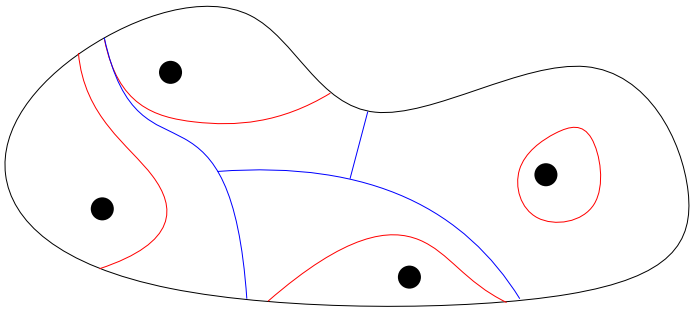


Figure 8.1: Graph with 4 terminals. Minimum s-T-cut for each terminal shown in red, minimum multiterminal cut $\mathcal{C}$ shown in blue.

The examples in Figures 8.1 and 8.2 show graphs with 4 terminals each. 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 $c(u, x) = c(v, x)$ to G or, if the edge already exists, we give it the edge weight $c(u, x) + c(v, x)$. 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 $G \backslash S$ 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 $G \backslash S$ containing t_i and $\delta(t_i) = |E(R(t_i), V \backslash R(t_i))|$ be the edges in S incident to t_i.

8.3 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.

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.

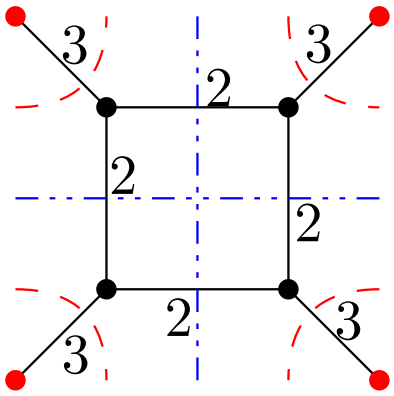


Figure 8.2: Simple graph with 4 terminals, in which minimum s-T-cuts (red) are different from minimum multiterminal cut (blue).

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 [43]. 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, Dahlhaus et al. [43] show that the sum of minimum s-T-cut weights minus the heaviest of them is an upper bound $\widehat{\mathcal{W}}$ of the weight of the minimum multiterminal cut, as denoted in Equation 8.1.

$$\mathcal{W}(G) \leq \widehat{\mathcal{W}}(G) = \sum_{s \in T} \lambda(G, s, T \backslash \{s\}) - \arg\, max_{s \in T} \lambda(G, s, T \backslash \{s\}) \tag{8.1}$$

The intuition behind Equation 8.1 is that any set of $t-1$ s-T-cuts pairwisely separates all terminals and is thus a valid multiterminal cut of weight $\widehat{\mathcal{W}}(G)$. However, $\widehat{\mathcal{W}}(G)$ is not necessarily the value of the minimum multiterminal cut $\mathcal{C}(G)$, as the minimum s-T-cuts might share edges – which then do not need to be counted twice – and the minimum multiterminal cut might be smaller. For a simple example where the minimum multiterminal cut is smaller than any set of $t-1$ minimum multiterminal cuts, see Figure 8.2, where any set of $t-1$ minimum s-T-cuts result in a multiterminal cut of weight 9 whereas the minimum multiterminal cut has a weight of 8.

Dahlhaus et al. [43] also give a lower bound for the minimum multiterminal cut: as $\lambda(G, s, T \backslash \{s\})$ is by definition minimal, $\mathcal{C}$ has at least as many edges incident to terminal s as $\lambda(G, s, T \backslash \{s\})$. As this is true for every terminal (and every edge is only incident to two vertices), $\mathcal{C}(G) \cdot 2 \geq \sum_{s \in T} \lambda(G, s, T \backslash \{s\})$, so that $\mathcal{C}(G) \geq \sum_{s \in T} \lambda(G, s, T \backslash \{s\})/2$.

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 $\widehat{\mathcal{W}(G)}$ 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 8.3.1. 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 8.3.1 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 8.3.2 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 multiterminal cut on either. We also give an enhanced branching scheme that aims to increase performance by creating more than two subproblems. Further details on the branching and edge selection are given in Section 8.3.2.

We compute upper and lower bounds for each of the problems and follow the branches whose lower bounds are lower than $\widehat{\mathcal{W}}$, the best cut weight previously found. In Section 8.4.3 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 $\mathcal{Q}$.

In Section 8.3.5 we describe our local search algorithm which can improve a given solution by iteratively moving vertices on the original graph until the solution reaches a local optimum. This allows us to significantly lower $\widehat{\mathcal{W}}$ and therefore improve performance by pruning subproblems whose lower bound is $\geq \widehat{\mathcal{W}}$.

We then give a variant of our algorithm in Section 8.3.6 that does not guarantee optimality but is able to solve significantly larger instances. This variant aggressively prunes problems that are unlikely to improve the solution quality and performs additional data reductions that do not have an optimality guarantee but can significantly shrink the graph while maintaining the most promising regions therein.

8.3.1 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

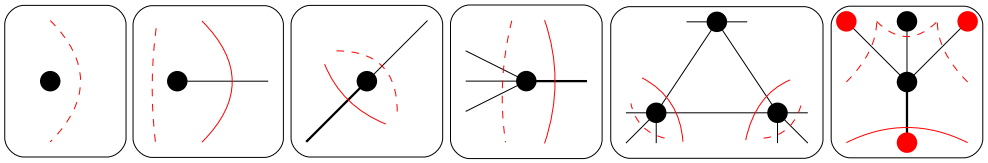


Figure 8.3: Reductions. Solid line cannot be minimal as dashed line has smaller weight: (1) `IsolatedVertex`, (2) `DegreeOne`, (3) `DegreeTwo` , (4) `HeavyEdge`, (5) `HeavyTriangle` and (6) `SemiEnclosedVertex`.

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 8.3 are used to identify such edges.

Lemma 8.3.1. *[33] 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. As $e \notin \mathcal{C}(G)$, $\mathcal{C}(G/e)$ is equal to $\mathcal{C}(G)$ and thus still has weight equal to $c(\mathcal{C}(G)) = \mathcal{W}(G)$. 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 8.3.1 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 8.3.2. *[33] 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. Let $\mathcal{W}(G)$ be the weight of the minimum multiterminal cut $\mathcal{C}(G)$. We show that for an edge $e \in \mathcal{C}(G)$, $\mathcal{W}(G - e) = \mathcal{W}(G) - c(e)$. 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 $\mathcal{W}(G)-c(e)$, 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 $c(e)$ (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 $c(e)$, $G-e$ cannot have a minimum multiterminal cut with weight $< \mathcal{W}(G) - c(e)$. 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. [43] 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 8.3.1, we can contract all vertices of the largest isolating cut into a single vertex. In Figure 8.1 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 8.3.2 they can therefore be deleted.

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 [68]. 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 [33]

Figures 8.3.(1), 8.3.(2) and 8.3.(3) show examples of non-terminal vertices with degree ≤ 2 that can be contracted while maintaining a minimum multiterminal cut. 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. The reductions `HeavyEdge` (8.3.(4)) and `HeavyTriangle` (8.3.(5)) were originally used for the minimum cut problem [37, 94,

152] and are described in Part I (Section 2.4.2) of this work. 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 8.3.(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 8.3.1.

The condition SemiEnclosed, shown in Figure 8.3.(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 $c(v, t_1) > c(v, t_2) + \sum_{u \in V \setminus T} c(v, u)$, 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 $deg(v) - c(v, t_1)$ 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 $c(v, t_2) + \sum_{u \in V \setminus T} c(v, u)$ 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 $\geq \widehat{\mathcal{W}}(G)$, 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 $< \widehat{\mathcal{W}}(G)$. We can therefore contract u and v. We now show how to improve the bound.

Lemma 8.3.3. *If for a graph G with best known multiterminal cut $\widehat{\mathcal{C}}(G)$, 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.*

In order to prove Lemma 8.3.3 we first prove the following useful claim:

Claim 8.3.4. *For any two nodes u and v, if u and v belong to different connected components of $G\backslash\mathcal{C}(G)$, 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. 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 $\lambda(u,v)$, $\lambda(u,v) \leq |S(u,v)|$.

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

Proof. Using Claim 8.3.4 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 8.3.4 and 8.3.5 to prove Lemma 8.3.3.

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

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

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

We can use Lemma 8.3.3 to contract edges whose high connectivity ensures that they are not in a minimum multiterminal cut. For any edge $e = (u,v)$, if $\lambda(u,v)+\frac{\sum_{i\in\{1,\dots,k\}\backslash\max_2}\lambda(G,t_i,T\backslash\{t_i\})}{4} > |\mathcal{W}(G)| > |\widehat{\mathcal{W}}(G)|$, 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 8.3.1. This condition is denoted as `HighConnectivity`.

As it is very expensive to compute the connectivity for every edge, we use the CAPFOREST algorithm of Nagamochi et al. [143, 147] (see Section 2.4.1 for a description of the CAPFOREST algorithm) 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 8.2, we can use Lemma 8.3.3 to contract u and v.

$$\gamma(u, v) > |\hat{\mathcal{W}}| - \frac{\sum_{i \in \{1,\dots,k\} \setminus \max_2} \lambda(G, t_i, T \setminus \{t_i\})}{4} \tag{8.2}$$

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 8.3.6. *For an articulation point ϕ whose removal disconnects the graph G into multiple connected components $(G_1, \dots, G_p)$ and a component G_i with $i \in \{1, \dots, p\}$ 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 $c(e)$. 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 8.3.1 and 8.3.6 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 [185] 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. This reduction rule is denoted as `ArticulationPoints`.

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 8.3.1 and 8.3.7 to contract them into a single vertex.

Lemma 8.3.7. *For two vertices v_1 and v_2 with $\{N(v_1)\backslash v_2\} = \{N(v_2)\backslash v_1\}$ where for all $v \in \{N(v_1)\backslash v_2\}$, $c(v_1, v) = c(v_2, v)$, 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 $cc(v)$ 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 $cc(v_2) \geq cc(v_1)$. 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 $cc(v_2)$ and then all edges incident to v_2 in its new location are added to ζ, which is exactly $cc(v_1)$ by the conditions of the lemma. Thus the weight of ζ changes by $cc(v_1) - cc(v_2) \leq 0$. If the edge $e_{12} = (v_1, v_2)$ exists, both $cc(v_1)$ and $cc(v_2)$ are furthermore decreased by $c(e_{12})$, 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. This reduction rule is denoted as `EqualNeighborhoods`

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 that separates v from the

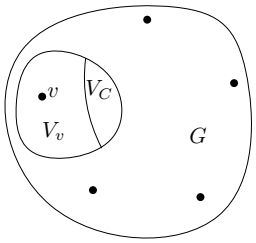


Figure 8.4: Illustration of vertex sets in Lemma 8.3.8.

set of terminal vertices T. Lemma 8.3.8 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.

Lemma 8.3.8. *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 8.4 gives an illustration of the vertex sets defined here.

Due to the minimality of the minimum isolating cut $(V_v, V\backslash V_v)$, we know that $c(V_C, V_v\backslash V_C) \geq c(V_C, V\backslash V_v)$ (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 $c(V_C, V_v\backslash V_C) \geq c(V_C, V\backslash V_v)$, 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 solve a maximum s-T-flow problem for an arbitrary non-terminal vertex s and the set of all terminals T and contract the source side of the largest minimum isolating cut into a single vertex, using Lemmas 8.3.1 and 8.3.8. 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 weight 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. This reduction rule is denoted as `NonTerminalFlows`.

Other Reductions

We now briefly present other reductions that we tried, but have been unsuccessful since they are either subsumed by other reductions or have excessive running time overheads in comparison to how many contractions are found.

Bridges. A *bridge* is an edge whose removal disconnects a graph G into two blocks G_1 and G_2. For every bridge, if one block has no terminals, we can contract this block into a single vertex, similar to the articulation point reduction in Section 8.3.1. As the two incident vertices of a bridge are always articulation points, the articulation point reduction already finds these contractions and finding bridges is not faster than finding articulation. If both blocks contain terminals, branching on this bridge allows the disconnection of the problem in one of the subproblems. However, we found that even if bridges like this exist in the original graph, generally they are already added to the multiterminal cut by other routines and thus all contractions that the bridge reduction finds are already found by other reductions.

Semi-isolated Clique. If a graph contains a clique C that has only a weak connection to the rest of the graph, no minimum multiterminal cut can cut C and we can thus contract it into a single vertex. We employed the maximal clique search algorithm of Eppstein et al. [54] with aggressive pruning of cliques that have a strong connection to non-clique vertices. However, as maximal clique detection is an NP-complete problem [54], even aggressive pruning still entails excessive running time. Also, as the instances contracted with all reductions usually have increased average degree and decreased diameter, almost all cliques in them have a large amount of edges to other vertices and thus there are only few semi-isolated cliques to be found.

8.3.2 Branching Tree Search

If our reductions detailed in Section 8.3.1 are unable to contract any edges in G, we branch on an edge adjacent to a terminal. Figure 8.5 shows an example in which we chose an edge to branch on. For each edge, there are two options: either the edge

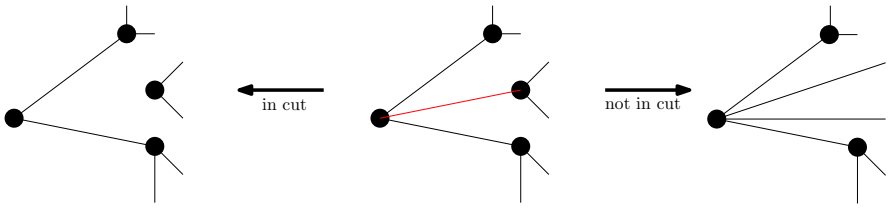


Figure 8.5: Branch on marked edge e in G, adjacent to a terminal - create two subproblems, (1) G/e and (2) $G - e$.

is part of the minimum multiterminal cut $\mathcal{C}(G)$ or it is not. Lemmas 8.3.1 and 8.3.2 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 $\mathcal{Q}$. This branching scheme for the multiterminal cut problem was introduced by Chen et al. [38] in their FPT algorithm for the problem.

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 $\lambda(G, s, T\backslash\{s\})$. 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 $c(e) - \frac{c(e)}{2} = \frac{c(e)}{2}$.

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 $\lambda(G/e, s, T\backslash\{s\}) > \lambda(G, s, T\backslash\{s\})$ 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 $\mathcal{W}(G)$.

Vertex Branching

When our multiterminal cut 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 comes from the embarrassingly parallel solving of different problems on separate threads. When branching, we select 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 now give 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, $\{t_1, \ldots, t_i\}$ 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 $c(x, t_j) + c(x, V \backslash T) > w_M$ 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 $c(x, t_j) + c(x, V \backslash T) \leq w_M$, 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 8.3.9 proves the correctness of this pruning step.

Lemma 8.3.9. *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 $c(x, t_j) + c(x, V \backslash T) \geq w_M$, 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 $\geq E(x) - (c(x, V \backslash T) + c(x, t_j))$, as all edges connecting x with other terminals than t_j are guaranteed to be cut edges. As $c(x, t_j) + c(x, V \backslash T) \geq w_M$, 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 $c(x, V \backslash T) > w_M$ 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.

Edge Selection

In Section 8.4.2 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 8.4.3 we evaluate the following comparators for the priority queue 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 subbranches. 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`.

8.3.3 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 embarassingly 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.

8.3.4 Combining Kernelization with ILP

Multiterminal cut problems are generally solved in practice using integer linear programs [142]. The following ILP formulation is adapted from our implementation for the graph partitioning problem in Section 7.4.1 (without balance constraints) and implemented using Gurobi 8.1.1. It is functionally equal to [142].

$$\min \sum_{\{u,v\} \in E} e_{uv} \cdot c(\{u,v\}) \tag{8.3}$$

$$\forall \{u,v\} \in E, \forall k : e_{uv} \geq x_{u,k} - x_{v,k} \tag{8.4}$$

$$\forall \{u,v\} \in E, \forall k : e_{uv} \geq x_{v,k} - x_{u,k} \tag{8.5}$$

$$\forall v \in V : \sum_{k} x_{v,k} = 1 \tag{8.6}$$

$$\forall i,j \in \{1,\dots,|T|\} : x_{t_i,j} = [i = j] \tag{8.7}$$

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. We also integrate the ILP formulation directly into the branch-and-reduce solver as an alternative to a branching operation. We hereby 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 8.4.7 we study which subproblems to solve with an ILP first.

8.3.5 Local Search

Our algorithm for the multiterminal cut problem 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, the algorithm has reductions that depend on the value of $\widehat{W}(G)$ and can thus contract more edges if the cut value $\widehat{W}(G)$ 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 $c(\gamma)$ 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 $c(\gamma_{\text{IMP}}) < c(\gamma)$.

Kernighan-Lin Local Search

Kernighan and Lin [129] give a heuristic for the traveling-salesman problem that has been adapted to many hard optimization problems [52, 165, 188, 196], where each vertex $v \in V_G$ is assigned a gain $g(v) = \max_{i \in \{i,\dots,|T|\}, i \neq \gamma(v)} \sum c(v, \xi_i(V_G)) - c(v, \xi_{\gamma(v)}(V_G))$, 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 $g(w) + g(v) + 2 \cdot c(v, u) > 0$, i.e. moving both u and v at the same time is a positive gain move. If it is, we perform the move.

Pairwise Maximum Flow

For any pair of blocks $1 \leq i < j \leq |T|$ where $c(\xi_i(V_G), \xi_j(V_G)) > 0$, 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 a 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 $c(\xi_i(V_G), \xi_j(V_G)) > 0$ in random order. We continue on all pairs of blocks where $c(\xi_i(V_G), \xi_j(V_G))$ 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.

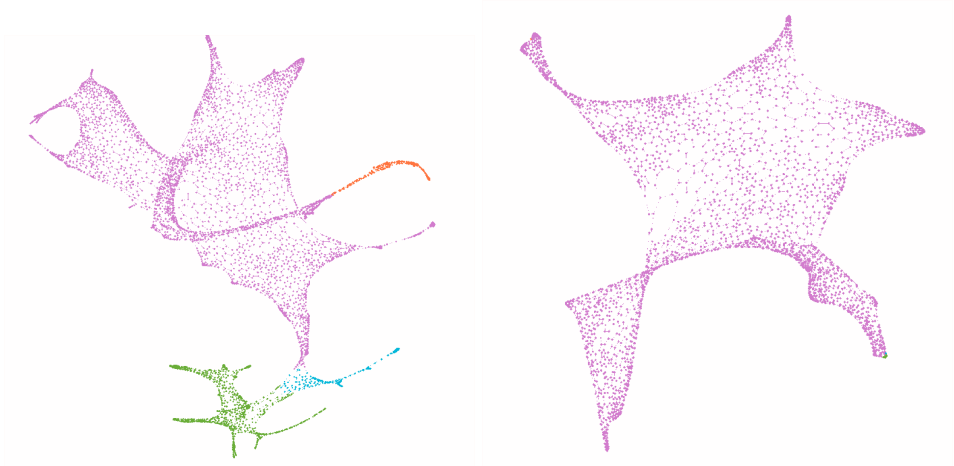


Figure 8.6: Minimum multiterminal cut for graph uk [174] and four terminals - on original graph (left) and remaining graph at time of first branch operation (right), visualized using Gephi-0.9.2 [17].

8.3.6 Fast Inexact Algorithm

Our algorithm for the multiterminal cut problem in an exact algorithm, i.e. when it terminates the output is guaranteed to be optimal. As the multiterminal cut problem is NP-complete [43], it is not feasible to expect termination in difficult instances of the problem. In fact, in difficult instances the 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 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 8.6 for an example with 4 terminals (selected with high distance to each other) on graph uk from the Walshaw Graph Partitioning Archive [174]. 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 $\beta = 5$.

8.4 Experiments and Results

We now perform an experimental evaluation of the multiterminal cut algorithms described in this chapter. 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 on a variety of graphs. Lastly, we perform experiments on protein-protein interaction networks and social, map and web graphs to compare the performance of different variants of our algorithm.

This section describes experiments performed for [91] and [88], where [91] introduces our first algorithm for the multiterminal cut problem and [88] enhances this algorithm by adding more reduction rules, improving the branching rule and including ILP and local search into the algorithm. The previous sections of this chapter give the full algorithm as described in both of our works. In the following we will use the terminology of [88], where the preliminary algorithm of [91] is denoted as `VieCut-MTC`, the full algorithm is denoted as `Exact-MTC` and the inexact algorithm described in Section 8.3.6 is denoted as `Inexact-MTC`.

`VieCut-MTC` is a shared-memory parallel branch-and-reduce algorithm that uses the reduction rules `Low`, `High`, `Triangle` and `HighConnectivity` to reduce the size of a graph instance and branches on an edge incident to a terminal when this is not possible anymore.

The `Exact-MTC` and `Inexact-MTC` algorithms additionally use the reduction rules `ArticulationPoints`, `EqualNeighborhoods` and `NonTerminalFlows`, create multiple subproblems when branching as described in Section 8.3.2 and integrate local search and ILP into the algorithm.

8.4.1 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 three machine types: Machine A is a machine with two Intel Xeon Gold 6130 with 2.1GHz with 16 CPU cores each and 256 GB RAM in total. Machine B is a machine with two Intel Xeon E5-2643v4 with 3.4 GHz with 6 CPU cores each and 1.5 TB RAM in

Table 8.1: Large Real-world Benchmark Instances.

Graph	n	m
Social, Web and Map Graphs (1A)		
bcsstk30 [174]	28 924	$1.01M$
ca-2010 [15]	$710K$	$1.74M$
ca-CondMat [46]	23 133	93 439
cit-HepPh [46]	34 546	$422K$
eu-2005 [25]	$862K$	$16.1M$
higgs-twitter [46]	$457K$	$14.9M$
in-2004 [25]	$1.38M$	$13.6M$
ny-2010 [15]	$350K$	$855K$
uk-2002 [25]	$18.5M$	$261M$
vibrobox [174]	12 328	$165K$
Social, Web and Map Graphs (1B)		
598a [174]	$111K$	$742K$
astro-ph [46]	16 706	$121K$
caidaRouterLevel [46]	$192K$	$609K$
citationCiteseer [46]	$268K$	$1.16K$
cnr-2000 [46]	$326K$	$2.74M$
coAuthorsCiteseer [46]	$227K$	$814K$
cond-mat-2005 [46]	40 421	$176K$
coPapersCiteseer [46]	$434K$	$16.0M$
cs4 [174]	22 499	43 858
fe_body [174]	45 087	$164K$
NACA0015 [46]	$1.04M$	$3.11M$
venturiLevel3 [46]	$4.03M$	$8.05M$

Graph	n	m
Protein-protein Interaction [181, 182] (2)		
Acidi. 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
S. cerevisiae	6 691	69 809
Toxoplasma gondii	7 988	11 779
Vitis vinifera	29 697	70 206
Map Graphs (3)		
ak2010 [15]	45 292	$109K$
ct2010 [15]	67 578	$168K$
de2010 [15]	24 115	58 028
hi2010 [15]	25 016	62 063
luxembourg.osm [46]	$115K$	$120K$
me2010 [15]	69 518	$168K$
netherlands.osm [46]	$2.22M$	$2.44M$
nh2010 [15]	48 837	$117K$
nv2010 [15]	84 538	$208K$
ri2010 [15]	25 181	62 875
sd2010 [15]	88 360	$205K$
vt2010 [15]	32 580	77 799

total. Machine C 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 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.

Instances

We use multiple sets of instances to avoid overtuning the branch-and-reduce algorithm. To analyze the impact of algorithmic components in Sections 8.4.2 and 8.4.3, we generate random hyperbolic graphs using the KaGen graph generator [65]. 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 10^{th} DIMACS implementa-

tion challenge [15]. 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 the 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.

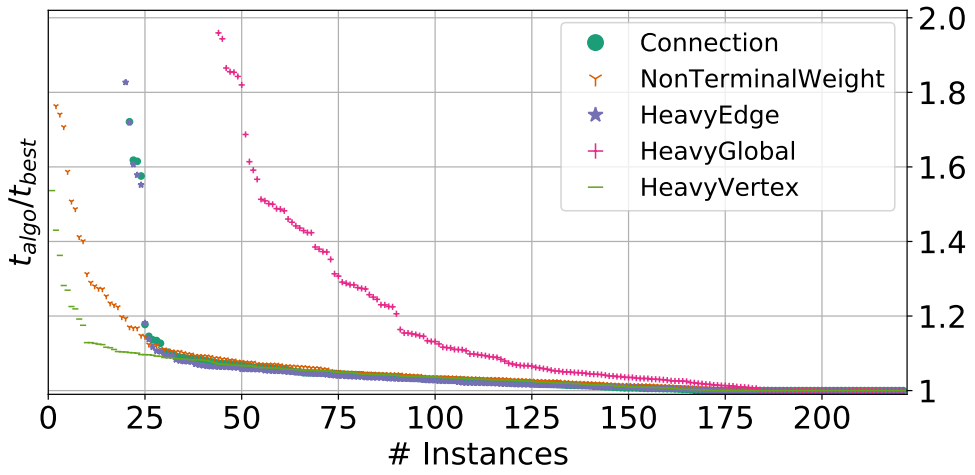
As the instances generally do not have any terminals, we find random vertices that have a high distance from each other in the following way: we start with a random vertex r, run a breadth-first search starting at r and select the vertex v encountered last as first terminal. While the number of terminals 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. The parameter $p \in (0, 1)$ hereby bounds the size of the terminal, i.e. only up to $\frac{p \cdot n}{k}$ vertices are added to each terminal. 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 inbetween.

When comparing `Kernel+ILP` with `VieCut-MTC` in Sections 8.4.6 and 8.4.8 on large instances, we use all 32 cores of machine A (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. 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.

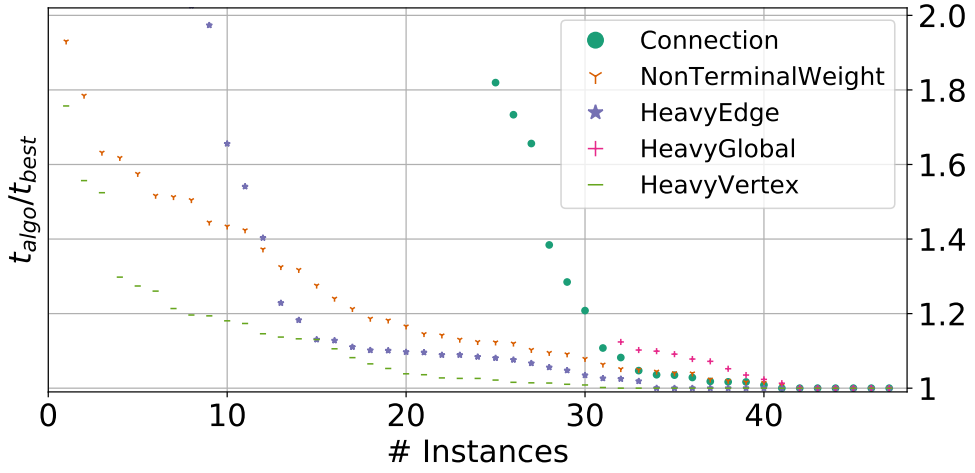
In Section 8.4.6 we perform experiments on protein-protein interaction networks (graph family 2) generated from the STRING protein interaction database [181, 182] by using all edges they predict with a high certainty. We use the protein description to assign functions (block terminal affiliations) to proteins (vertices). We use the first occurence of a set of pre-defined function classes. For each graph, we examine problems with the $4, 5, 6, 7, 8$ most often occuring functions and with all (up to 15, if all occuring in an organism) classes.

8.4.2 Branching Edge Selection

Figure 8.7 shows the results for the branching edge selection rules on machine A. In Subfigure 8.7a, we show performance plots for RHG graphs and in Subfigure 8.7b 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) RHG graphs with partition centers as terminals.



(b) Map graphs with partition centers as terminals.

Figure 8.7: Performance plots for branching edge selection variants.

a terminal. In this experiment we use the `BoundSum` comparator and enable `Low`, `High`, `Triangle` and `HighConnectivity` 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`.

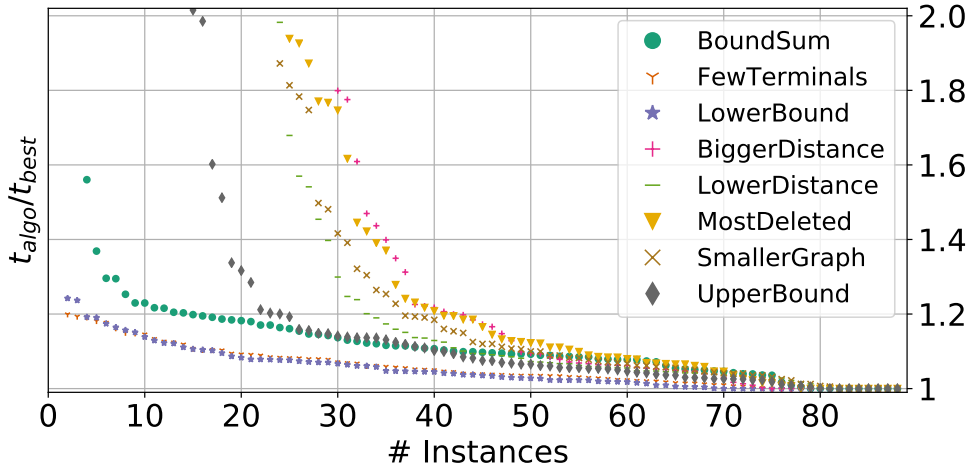
In all following experiments we use `HeavyVertex`, as it outperforms all other variants consistently.

8.4.3 Priority Queue Comparator

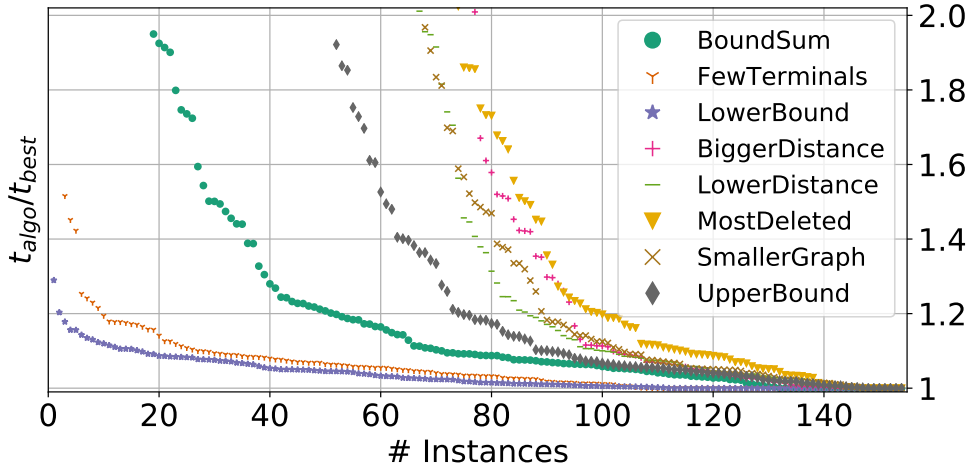
We now explore the effect of the comparator used in the priority queue $\mathcal{Q}$. This experiment was performed in machine A with algorithm `VieCut-MTC`. The choice of comparator decides which problems are highest priority and will be explored first. We want to first explore the problems and branches 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 might yield improved solutions, either directly on 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 multiterminal 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 8.8a shows results for 20% of vertices in the terminal on RHG graphs and Figure 8.8b show 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



(a) Graphs with 20% of vertices in terminal.



(b) Graphs with 80% of vertices in terminal.

Figure 8.8: Performance plots for priority queue comparator variants.

when their lower bound is not lower than the currently best known solution. None of the other variants have noteworthy performance.

8.4.4 Kernelization

We 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 social, web and map graphs (graph families (1A), (1B) and (3) in Table 8.1) with $k = \{4, 8, 10\}$ terminals and 10% of all vertices added to the terminals on machine C. For these instances, we run subsets of all contractions exhaustively and check how many vertices remain in the graph. Figure 8.9 gives results with 8 different variants, starting with a version

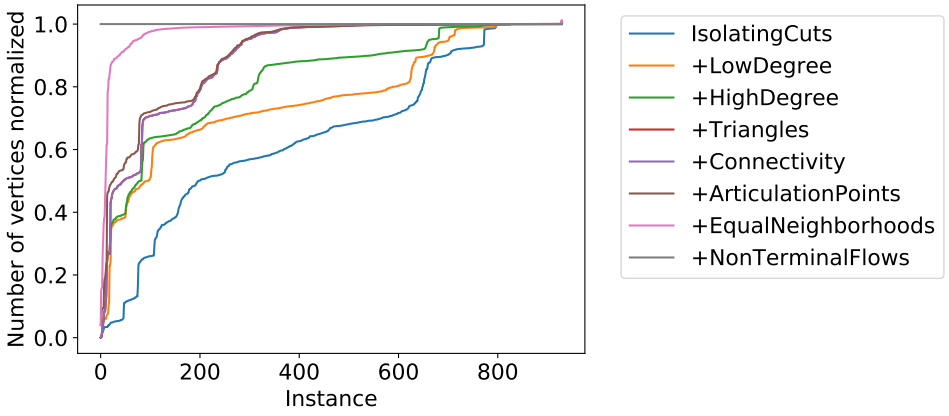


Figure 8.9: Number of vertices in graph after reductions are finished, normalization by (# vertices remaining with all reductions / # vertices remaining in variant) and sorted by normalized value.

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.

For each instance and variant we normalize by the number of vertices remaining with all reductions divided by the number of vertices remaining in a given variant. Thus, a value close to 1 indicates that this variant already performs most reductions that the full algorithm does and a value close to 0 indicates that the resulting graph is much larger than it is when using the full algorithm. The effectiveness of a reduction can therefore be read from the area between a line and the line below it.

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, which can be explained by the fact that 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.

Out of the new reductions that are not part of `VieCut-MTC`, all find a significant amount of contractible edges on the graphs already contracted by the reductions included therein. 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% and `NonTerminalFlows` reduces the number of vertices by 2.0%. However, there are some instances in which these reductions reduce the number of vertices remaining by more than 99%.

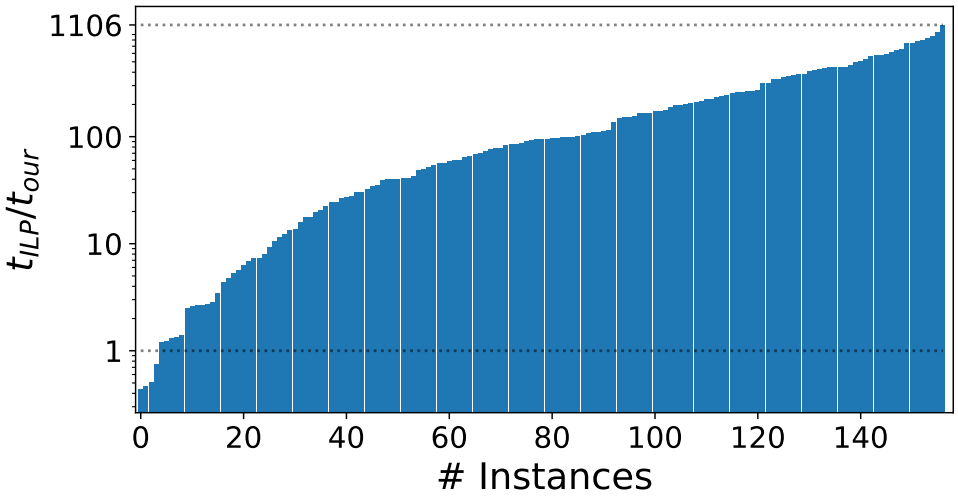


Figure 8.10: Speedup of opt. branch-and-reduce `VieCut-MTC` to ILP.

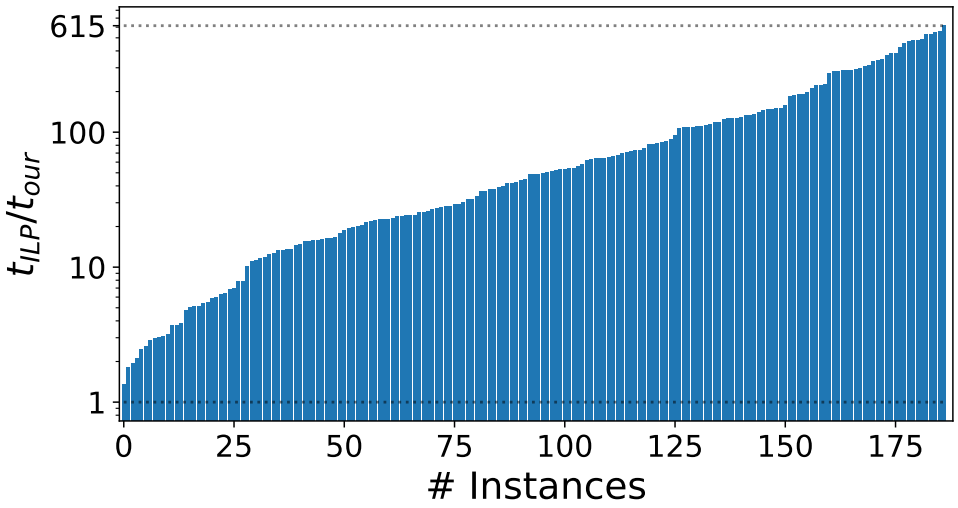


Figure 8.11: Speedup of `Kernel+ILP` to ILP.

8.4.5 Comparison between VieCut-MTC and ILP

Figure 8.10 shows the speedup of the engineered `VieCut-MTC` algorithm, using `HeavyVertex` edge selection, `LowerBound` priority queue comparator and all kernelization rules of `VieCut-MTC` enabled, to the ILP on all graphs from Sections 8.4.2 and 8.4.3 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, `VieCut-MTC` solves 61%; on the problems solved by both, the branch-

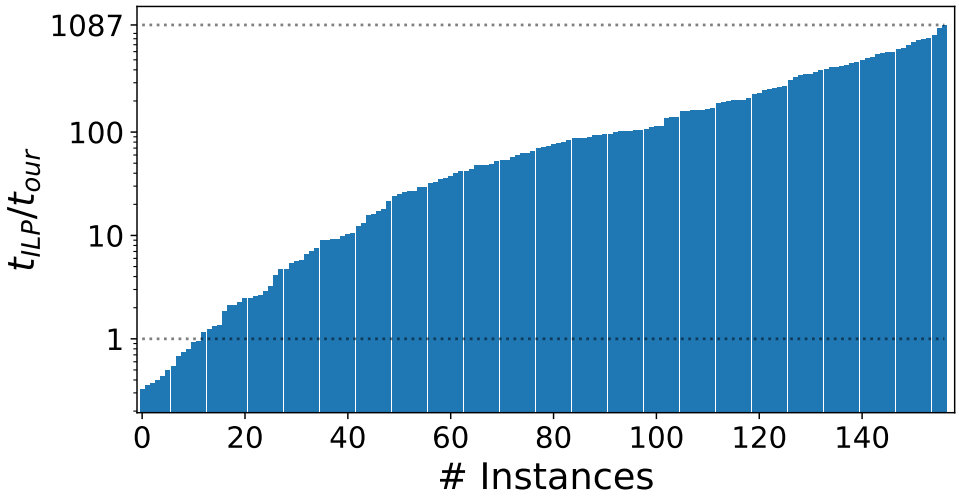


Figure 8.12: Speedup of avg. branch-and-reduce VieCut-MTC to ILP.

and-reduce algorithm has a mean speedup factor of 67, a median speedup factor of 95 and a maximum speedup factor of 1 106. The mean speedup factor of the average of all algorithm configurations compared to ILP is 43 with a median speedup factor of 71. The speedup can be seen in Figure 8.12. 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, as shown in Figure 8.11.

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.

8.4.6 VieCut-MTC on 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 [102, 142, 189]. Table 8.2 shows the results for these graphs. We can see that Kernel+ILP outperforms VieCut-MTC 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. Due to these results in [91], Exact-MTC integrates the ILP solving into the branch-and-reduce algorithm and solves some subproblems using an ILP solver. In the following section we examine which subproblems should be solved using branching and which should be solved using ILP.

Table 8.2: Result overview on protein-protein-interaction 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

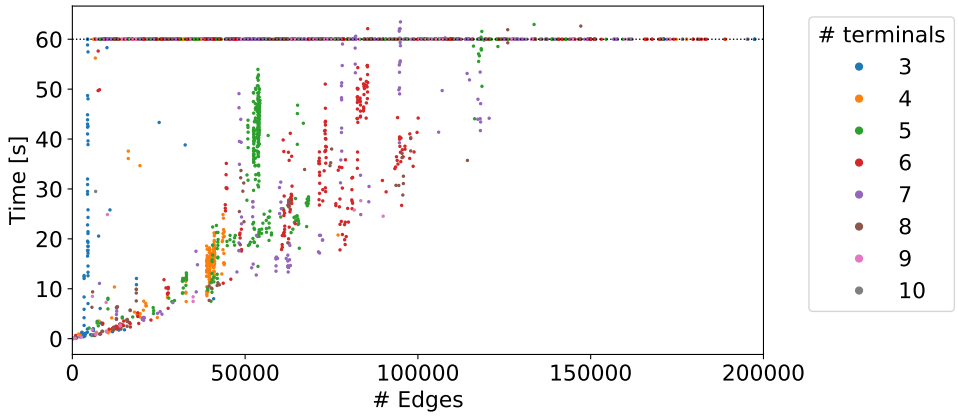


Figure 8.13: Running time of ILP subproblems in relation to $|E|$.

8.4.7 Integer Linear Programming

In order to get all a wide variety of ILP problems, we run the `Inexact-MTC` algorithm on all instances in graph families (1A), (1B) and (3) of Table 8.1 with $k = 10$ terminals and 10% of vertices added to the terminals on machine C. 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 the algorithm chooses between branching and ILP on graph G, we select a random integer $r \in (1{,}200\,000)$. We use this random integer, as we want to have problems of all different sizes and using a hard limit would result in many instances just barely below that size limit. We select 200000 edges as the maximum, as we did not encounter any larger instances in which the ILP was solved to optimality 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 8.13 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.

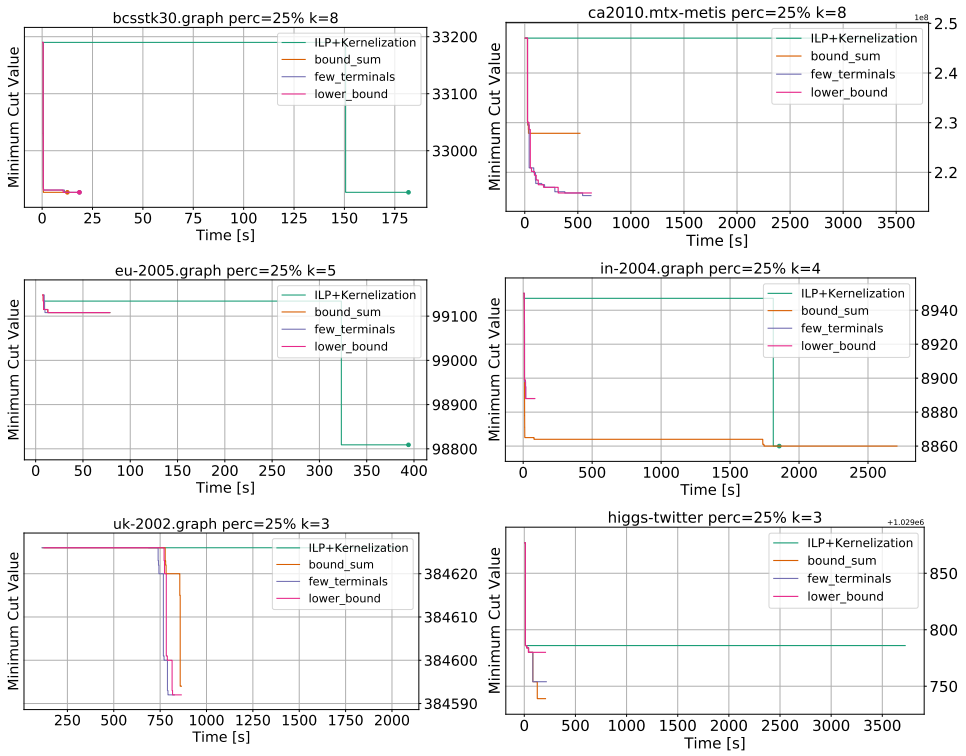


Figure 8.14: Progression of best result over time. Dot at end symbolizes that algorithm certifies optimality.

Table 8.3: Result overview of Section 8.4.8.

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

8.4.8 Large Real-World Networks

In this experiment we compare configurations of `VieCut-MTC` with `Kernel+ILP`. We use graph family (1A) of Table 8.1. For each graph, we solve the minimum multi-terminal cut problem for $k \in \{3, 4, 5, 8\}$ terminals and $p \in \{10\%, 15\%, 20\%, 25\%\}$ vertices in the terminal. We hereby use the priority queue configurations `BoundSum`, `LowerBound` and `FewTerminals`. Figure 8.14 shows the progression of the best result over time for a set of interesting problems. Table 8.3 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. In both Figure 8.14 and Table 8.3 we can see that the branch and reduce variants

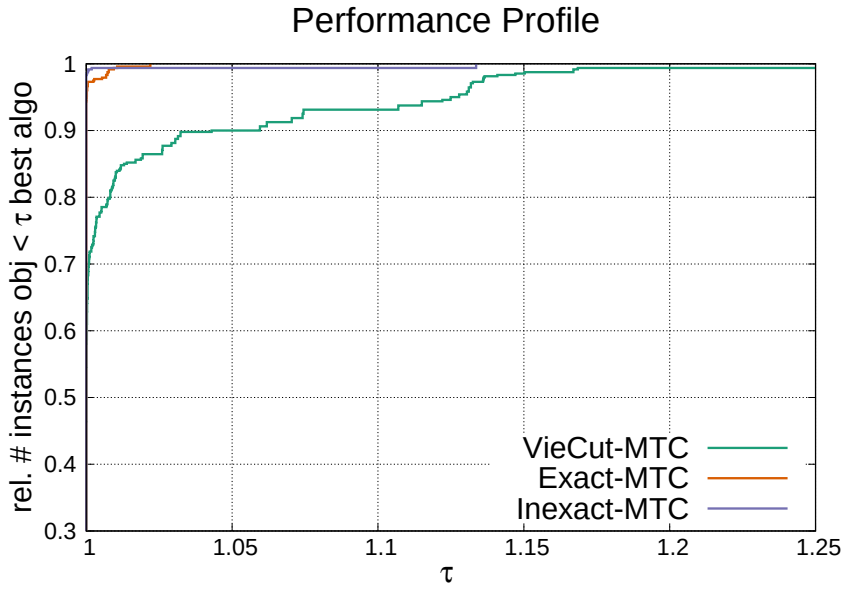


Figure 8.15: Performance profile for $k \in \{3, 4, 5, 8\}$ and graph family (1A).

find good solutions faster than `Kernel+ILP`. However, the variants often run out of memory in some of the largest instances. 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 on 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 for 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.

We use the same instances to compare `Exact-MTC` to `VieCut-MTC` (both using `BoundSum` as priority queue implementation), using machine C with all 12 cores and a 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. Of the 114 instances that were not solved to optimality by both algorithms, `Exact-MTC` gives a better result on 75 instances and the same result on all others. The geometric mean of results given by `Exact-MTC` and `Inexact-MTC` are both about 1.5% lower than `VieCut-MTC`. Note that in the first iteration of this experiment, 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

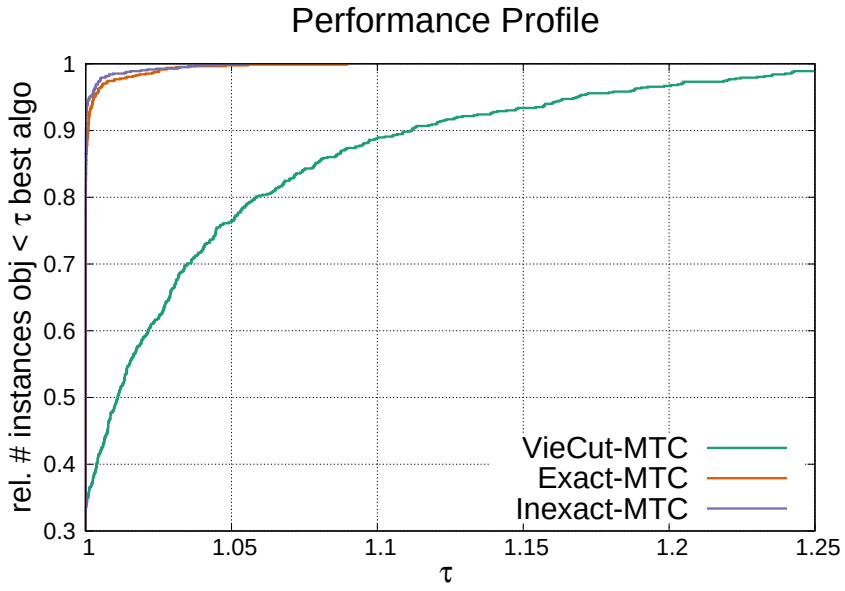


Figure 8.16: Performance Profiles for $k \in \{4, 5, 8, 10\}$ and graph families (1A), (1B) and (3).

comparison. The largest part of the improvement of `Exact-MTC` and `Inexact-MTC` over `VieCut-MTC` is gained by the local search algorithm detailed in Section 8.3.5.

Figure 8.15 shows the performance profile of this experiment. We can see that both `Exact-MTC` and `Inexact-MTC` are almost always equal to the best result on this instance 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.

Additionally, we compare `VieCut-MTC`, `Exact-MTC` and `Inexact-MTC` on a larger set of instances, all graphs from Table 8.1 graph families (1A), (1B) and (3) with $k = \{4, 5, 8, 10\}$ terminals and $p = \{10\%, 20\%\}$ 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 B 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.

Table 8.4 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

Table 8.4: Result overview for Section 8.4.8.

# 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**	—

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

Figure 8.16 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

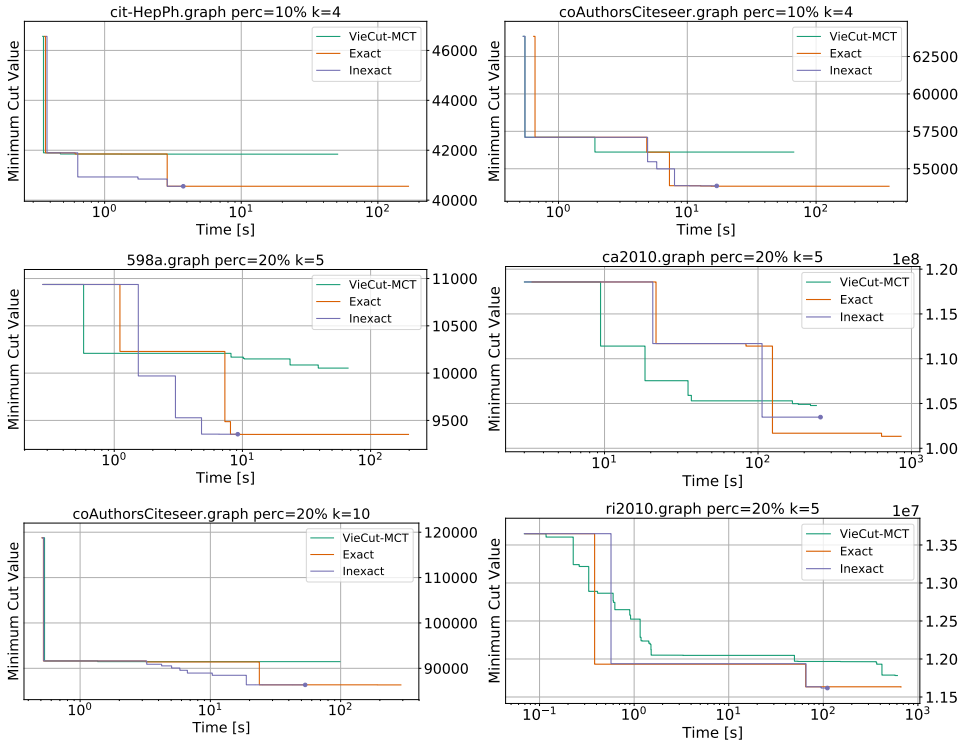


Figure 8.17: Progression of best result over time. Dot at end marks termination of algorithm.

differ significantly. In general, both of them outperform `VieCut-MTC` on most instances that are not solved to optimality by every algorithm.

8.5 Conclusion

In this chapter, 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. Additionally, we give an inexact algorithm for the multiterminal cut problem that aggressively shrinks the graph instances and is able to 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.

Bibliography

[1] F. Abu-Khzam, S. Lamm, M. Mnich, A. Noe, C. Schulz, and D. Strash. "Recent Advances in Practical Data Reduction." In: *arXiv preprint arXiv:2012.12594* (2020).

[2] F. N. Abu-Khzam, S. Cai, J. Egan, P. Shaw, and K. Wang. "Turbo-Charging Dominating Set with an FPT Subroutine: Further Improvements and Experimental Analysis." In: *Theory and Applications of Models of Computation*. Cham: Springer Intl. Publishing, 2017, pp. 59–70.

[3] Y. Akhremtsev, P. Sanders, and C. Schulz. "(Semi-) external algorithms for graph partitioning and clustering." In: *2015 Proc. of the Seventeenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2014, pp. 33–43.

[4] Y. Akhremtsev, P. Sanders, and C. Schulz. "High-quality shared-memory graph partitioning." In: *IEEE Transactions on Parallel and Distributed Systems* 31.11 (2020), pp. 2710–2722.

[5] T. Akiba, Y. Iwata, Y. Sameshima, N. Mizuno, and Y. Yano. "Cut Tree Construction from Massive Graphs." In: *16th Intl. Conf. on Data Mining, ICDM 2016*. 2016, pp. 775–780.

[6] T. Akiba, Y. Iwata, Y. Sameshima, N. Mizuno, and Y. Yano. "Cut tree construction from massive graphs." In: *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE. 2016, pp. 775–780.

[7] C. J. Alpert and A. B. Kahng. "Recent directions in netlist partitioning: a survey." In: *Integr.* 19.1-2 (1995), pp. 1–81.

[8] C. J. Alpert, A. B. Kahng, and S.-Z. Yao. "Spectral Partitioning with Multiple Eigenvectors." In: *Discrete Applied Mathematics* 90.1 (1999), pp. 3–26.

[9] R. Andersen, F. Chung, and K. Lang. "Local graph partitioning using pagerank vectors." In: *2006 47th IEEE Symp. on Foundations of Computer Science (FOCS'06)*. IEEE. 2006, pp. 475–486.

[10] R. Andersen and K. J. Lang. "An algorithm for improving graph partitions." In: *Proc. of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics. 2008, pp. 651–660.

[11] R. Andersen and K. J. Lang. "Communities from seed sets." In: *Proc. of the 15th international conference on World Wide Web*. ACM. 2006, pp. 223–232.

[12] R. J. Anderson and H. Woll. "Wait-free parallel algorithms for the union-find problem." In: *Proc. of the Twenty-Third ACM Symp. on Theory of Computing*. STOC '91. ACM. 1991, pp. 370–380.

[13] M. Armbruster. "Branch-and-Cut for a Semidefinite Relaxation of Large-Scale Minimum Bisection Problems." PhD thesis. 2007.

[14] M. Armbruster, M. Fügenschuh, C. Helmberg, and A. Martin. "A Comparative Study of Linear and Semidefinite Branch-and-Cut Methods for Solving the Minimum Graph Bisection Problem." In: *Proc. of the 13th International Conference on Integer Programming and Combinatorial Optimization*. Vol. 5035. LNCS. Springer, 2008, pp. 112–124.

[15] D. A. Bader, A. Kappes, H. Meyerhenke, P. Sanders, C. Schulz, and D. Wagner. "Benchmarking for Graph Clustering and Partitioning." In: *Encyclopedia of Social Network Analysis and Mining, 2nd Edition*. Springer, 2018.

[16] M. Bannach and S. Berndt. "Practical Access to Dynamic Programming on Tree Decompositions." In: *26th European Symp. on Algorithms (ESA'18)*. Vol. 112. LIPIcs. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 6:1–6:13.

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

[18] V. Batagelj and M. Zaversnik. "An O(m) Algorithm for Cores Decomposition of Networks." In: *CoRR* cs.DS/0310049 (2003).

[19] N. Bhardwaj, A. M. Lovett, and B. Sandlund. "A Simple Algorithm for Minimum Cuts in Near-Linear Time." In: *17th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2020, June 22-24, 2020, Tórshavn, Faroe Islands*. Vol. 162. LIPIcs, 12:1–12:18.

[20] 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 Intl. Conf. on Data Mining (ICDM)*. IEEE. 2017, pp. 21–30.

[21] C. Bichot and P. Siarry, eds. *Graph Partitioning*. Wiley, 2011.

[22] R. H. Bisseling, J. Byrka, S. Cerav-Erbas, N. Gvozdenovic, M. Lorenz, R. Pendavingh, C. Reeves, M. Roger, and A. Verhoeven. "Partitioning a call graph." In: *Proceedings 52nd European Study Group Mathematics with Industry Amsterdam 2005* (2006), pp. 95–107.

[23] S. Böcker, S. Briesemeister, and G. W. Klau. "Exact Algorithms for Cluster Editing: Evaluation and Experiments." In: *Algorithmica* 60.2 (June 2011), pp. 316–334.

[24] P. Boldi, M. Rosa, M. Santini, and S. Vigna. "Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks." In: *Proc. of the 20th Intl. Conf. on World Wide Web*. ACM Press, 2011, pp. 587–596.

[25] P. Boldi and S. Vigna. "The WebGraph Framework I: Compression Techniques." In: *Proc. of the Thirteenth Intl. World Wide Web Conf. (WWW 2004)*. Manhattan, USA: ACM Press, 2004, pp. 595–601.

[26] P. S. Bonsma. "Most balanced minimum cuts and partially ordered knapsack." In: *Sixth Cologne Twente Workshop on Graphs and Combinatorial Optimization, University of Twente, Enschede, The Netherlands, 29-31 May, 2007*. University of Twente, 2007, pp. 17–21.

[27] "Branch-and-reduce exponential/FPT algorithms in practice: A case study of vertex cover." In: *Theor. Comput. Sci.* 609, Part 1 (2016), pp. 211–225.

[28] R. Brillout. "A Multi-Level Framework for Bisection Heuristics." MA thesis. Karlsruhe Institute of Technology, 2009.

[29] N. Buchbinder, J. S. Naor, and R. Schwartz. "Simplex partitioning via exponential clocks and the multiway cut problem." In: *Proc. of the forty-fifth annual ACM symposium on Theory of computing*. ACM. 2013, pp. 535–544.

[30] T. N. Bui and C. Jones. "Finding Good Approximate Vertex and Edge Partitions is NP-Hard." In: *Information Processing Letters* 42.3 (1992), pp. 153–159.

[31] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. "Recent Advances in Graph Partitioning." In: *Algorithm Engineering*. Springer, 2016, pp. 117–158.

[32] D. Cai, Z. Shao, X. He, X. Yan, and J. Han. "Mining hidden community in heterogeneous social networks." In: *Proc. of the 3rd international workshop on Link discovery*. ACM. 2005, pp. 58–65.

[33] Y. Cao, J. Chen, and J.-H. Fan. "An $O(1.84^k)$ parameterized algorithm for the multiterminal cut problem." In: *Inf. Process. Lett.* 114.4 (2014), pp. 167–173.

[34] D. Chakrabarti and C. Faloutsos. "Graph mining: Laws, generators, and algorithms." In: *ACM Computing Surveys* 38.1 (2006), p. 2.

[35] L. Chang, W. Li, and W. Zhang. "Computing A Near-Maximum Independent Set in Linear Time by Reducing-Peeling." In: *2017 ACM Intl. Conf. on Management of Data, SIGMOD'17*. ACM, 2017, pp. 1181–1196.

[36] S. Chechik, T. D. Hansen, G. F. Italiano, V. Loitzenbauer, and N. Parotsidis. "Faster algorithms for computing maximal 2-connected subgraphs in sparse directed graphs." In: *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2017, pp. 1900–1918.

[37] C. S. Chekuri, A. V. Goldberg, D. R. Karger, M. S. Levine, and C. Stein. "Experimental Study of Minimum Cut Algorithms." In: *Proc. 8th Symp. on Discrete Algorithms (SODA '97)*. New Orleans, Louisiana, USA: SIAM, 1997, pp. 324–333.

[38] J. Chen, Y. Liu, and S. Lu. "An improved parameterized algorithm for the minimum node multiway cut problem." In: *Algorithmica* 55.1 (2009), pp. 1–13.

[39] B. V. Cherkassky and A. V. Goldberg. "On implementing the push—relabel method for the maximum flow problem." In: *Algorithmica* 19.4 (1997), pp. 390–410.

[40] A. Clauset. "Finding local community structure in networks." In: *Physical review E* 72.2 (2005), p. 026132.

[41] W. H. Cunningham. "The Optimal Multiterminal Cut Problem." In: *Reliability of computer and communication networks*. 1989, pp. 105–120.

[42] L. Dagum and R. Menon. "OpenMP: An industry standard API for shared-memory programming." In: *IEEE Computational Science and Engineering* 5.1 (1998), pp. 46–55.

[43] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. "The complexity of multiterminal cuts." In: *SIAM Journal on Computing* 23.4 (1994), pp. 864–894.

[44] J. Dahlum, S. Lamm, P. Sanders, C. Schulz, D. Strash, and R. F. Werneck. "Accelerating Local Search for the Maximum Independent Set Problem." In: *Intl. Symp. on Experimental Algorithms*. Springer. 2016, pp. 118–133.

[45] G. Dantzig and D. R. Fulkerson. "On the max flow min cut theorem of networks." In: *Linear inequalities and related systems* 38 (2003), pp. 225–231.

[46] T. A. Davis and Y. Hu. "The University of Florida sparse matrix collection." In: *ACM Trans. Mathematical Software (TOMS)* 38.1 (2011), p. 1.

[47] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. "Customizable Route Planning." In: *Proc. of the 10th International Symposium on Experimental Algorithms*. Vol. 6630. LCNS. Springer, 2011, pp. 376–387.

[48] D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. "Exact Combinatorial Branch-and-Bound for Graph Bisection." In: *Proc. of the 12th Workshop on Algorithm Engineering and Experimentation*. 2012, pp. 30–44.

[49] D. Delling and R. F. Werneck. "Better Bounds for Graph Bisection." In: *Proc. of the 20th European Symposium on Algorithms*. Vol. 7501. LNCS. 2012, pp. 407–418.

[50] E. A. Dinic. "Algorithm for solution of a problem of maximum flow in networks with power estimation." In: *Soviet Math. Doklady*. Vol. 11. 1970, pp. 1277–1280.

[51] Y. Dinitz. "Maintaining the 4-edge-connected components of a graph online." In: *[1993] The 2nd Israel Symposium on Theory and Computing Systems*. IEEE. 1993, pp. 88–97.

[52] M. Dorigo, M. Birattari, and T. Stutzle. "Ant colony optimization." In: *IEEE computational intelligence magazine* 1.4 (2006), pp. 28–39.

[53] D. Eppstein, Z. Galil, and G. F. Italiano. "Dynamic graph algorithms." In: *Algorithms and theory of computation handbook* 1 (1999), pp. 9–1.

[54] D. Eppstein, M. Löffler, and D. Strash. "Listing All Maximal Cliques in Large Sparse Real-World Graphs." In: *ACM J. Exp. Algorithmics* 18 (2013).

[55] A. Feldmann and P. Widmayer. "An $O(n^4)$ Time Algorithm to Compute the Bisection Width of Solid Grid Graphs." In: *Proc. of the 19th European Conference on Algorithms*. Vol. 6942. LNCS. Springer, 2011, pp. 143–154.

[56] A. Felner. "Finding Optimal Solutions to the Graph Partitioning Problem with Heuristic Search." In: *Annals of Mathematics and Artificial Intelligence* 45 (3-4 2005), pp. 293–322.

[57] D. Ferizovic, D. Hespe, S. Lamm, M. Mnich, C. Schulz, and D. Strash. "Engineering Kernelization for Maximum Cut." In: *Proc. of the Twenty-Second Workshop on Algorithm Engineering and Experiments, ALENEX 2020*. 2020. arXiv: 1905.10902.

[58] C. E. Ferreira, A. Martin, C. C. De Souza, R. Weismantel, and L. A. Wolsey. "The Node Capacitated Graph Partitioning Problem: A Computational Study." In: *Mathematical Programming* 81.2 (1998), pp. 229–256.

[59] C. M. Fiduccia and R. M. Mattheyses. "A Linear-Time Heuristic for Improving Network Partitions." In: *Proc. of the 19th Conference on Design Automation*. 1982, pp. 175–181.

[60] J. Fietz, M. Krause, C. Schulz, P. Sanders, and V. Heuveline. "Optimized Hybrid Parallel Lattice Boltzmann Fluid Flow Simulations on Complex Geometries." In: *Proc. of Euro-Par 2012 Parallel Processing*. Vol. 7484. LNCS. Springer, 2012, pp. 818–829.

[61] L. Fleischer. "Building chain and cactus representations of all minimum cuts from Hao–Orlin in the same asymptotic run time." In: *Journal of Algorithms* 33.1 (1999), pp. 51–72.

[62] R. Fleischer, X. Wu, and L. Yuan. "Experimental Study of FPT Algorithms for the Directed Feedback Vertex Set Problem." In: *Algorithms - ESA 2009, 17th European Symp., Copenhagen, Denmark, September 7-9, 2009. Proc.* Ed. by A. Fiat and P. Sanders. Vol. 5757. Lecture Notes in Computer Science. Springer, 2009, pp. 611–622.

[63] L. R. Ford and D. R. Fulkerson. "Maximal flow through a network." In: *Canadian Journal of Mathematics* 8.3 (1956), pp. 399–404.

[64] S. Forster, D. Nanongkai, L. Yang, T. Saranurak, and S. Yingchareonthawornchai. "Computing and testing small connectivity in near-linear time and queries via fast local cut algorithms." In: *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2020, pp. 2046–2065.

[65] D. Funke, S. Lamm, P. Sanders, C. Schulz, D. Strash, and M. von Looz. "Communication free massively distributed graph generation." In: *2018 IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS)*. IEEE. 2018, pp. 336–347.

[66] H. N. Gabow. "A matroid approach to finding edge connectivity and packing arborescences." In: *Journal of Computer and System Sciences* 50.2 (1995), pp. 259–273.

[67] H. N. Gabow. "Applications of a poset representation to edge connectivity and graph rigidity." In: *Proc. 32nd Symp. of Foundations of Computer Science*. IEEE. 1991, pp. 812–821.

[68] H. N. Gabow and R. E. Tarjan. "A linear-time algorithm for a special case of disjoint set union." In: *Journal of computer and system sciences* 30.2 (1985), pp. 209–221.

[69] P. Galinier, Z. Boujbel, and M. C. Fernandes. "An Efficient Memetic Algorithm for the Graph Partitioning Problem." In: *Annals of Operations Research* 191.1 (2011), pp. 1–22.

[70] B. A. Galler and M. J. Fisher. "An improved equivalence algorithm." In: *Communications of the ACM* 7.5 (1964), pp. 301–303.

[71] P. Gawrychowski, S. Mozes, and O. Weimann. "Minimum Cut in O (m $\log^2$ n) Time." In: *47th International Colloquium on Automata, Languages, and Programming (ICALP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2020.

[72] A. George. "Nested Dissection of a Regular Finite Element Mesh." In: *SIAM Journal on Numerical Analysis* 10.2 (1973), pp. 345–363.

[73] L. Georgiadis, D. Kefallinos, L. Laura, and N. Parotsidis. "An Experimental Study of Algorithms for Computing the Edge Connectivity of a Directed Graph*." In: *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2021, pp. 85–97.

[74] M. Ghaffari, K. Nowicki, and M. Thorup. "Faster algorithms for edge connectivity via random 2-out contractions." In: *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2020, pp. 1260–1279.

[75] L. Gianinazzi, P. Kalvoda, A. De Palma, M. Besta, and T. Hoefler. "Communication avoiding parallel minimum cuts and connected components." In: *Proc. of the 23rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*. ACM. 2018, pp. 219–232.

[76] A. V. Goldberg and K. Tsioutsiouliklis. "Cut tree algorithms: an experimental study." In: *Journal of Algorithms* 38.1 (2001), pp. 51–83.

[77] A. V. Goldberg and R. E. Tarjan. "A new approach to the maximum-flow problem." In: *Journal of the ACM* 35.4 (1988), pp. 921–940.

[78] R. E. Gomory and T. C. Hu. "Multi-terminal network flows." In: *Journal of the Society for Industrial and Applied Mathematics* 9.4 (1961), pp. 551–570.

[79] G. Goranci, M. Henzinger, and M. Thorup. "Incremental exact min-cut in polylogarithmic amortized update time." In: *ACM Transactions on Algorithms (TALG)* 14.2 (2018), pp. 1–21.

[80] W. W. Hager, D. T. Phan, and H. Zhang. "An Exact Algorithm for Graph Partitioning." In: *Mathematical Programming* 137.1-2 (2013), pp. 531–556.

[81] J. Hao and J. B. Orlin. "A faster algorithm for finding the minimum cut in a graph." In: *Proc. of the 3rd ACM-SIAM Symp. on Discrete Algorithms*. Society for Industrial and Applied Mathematics. 1992, pp. 165–174.

[82] T. Harris and F. Ross. *Fundamentals of a method for evaluating rail net capacities*. Tech. rep. RAND CORP SANTA MONICA CA, 1955.

[83] T. Hartmann and D. Wagner. "Fast and Simple Fully-Dynamic Cut Tree Construction." In: *Algorithms and Computation - 23rd International Symposium, ISAAC 2012, Taipei, Taiwan, December 19-21, 2012. Proceedings*. Vol. 7676. Lecture Notes in Computer Science. Springer, 2012, pp. 95–105.

[84] E. Hartuv and R. Shamir. "A clustering algorithm based on graph connectivity." In: *Information processing letters* 76.4-6 (2000), pp. 175–181.

[85] M. Hein and S. Setzer. "Beyond Spectral Clustering - Tight Relaxations of Balanced Graph Cuts." In: *Advances in Neural Information Processing Systems*. 2011, pp. 2366–2374.

[86] A. Henzinger, A. Noe, and C. Schulz. "ILP-Based Local Search for Graph Partitioning." In: *Journal of Experimental Algorithmics (JEA)* 25.1 (2020), pp. 1–26.

[87] A. Henzinger, A. Noe, and C. Schulz. "ILP-based Local Search for Graph Partitioning." In: *Proc. of the 17th Intl. Symp. on Experimental Algorithms (SEA 2018)* (2018).

[88] M. Henzinger, A. Noe, and C. Schulz. "Faster Parallel Multiterminal Cuts." In: *arXiv preprint arXiv:2004.11666* (2020).

[89] M. Henzinger, A. Noe, and C. Schulz. *Practical Fully Dynamic Minimum Cut Algorithms*. 2021.

[90] M. Henzinger, A. Noe, and C. Schulz. "Shared Memory Exact Minimum Cuts." In: *2019 IEEE Intl. Parallel and Distributed Processing Symp., IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*. IEEE, 2019, pp. 13–22.

[91] M. Henzinger, A. Noe, and C. Schulz. "Shared-Memory Branch-and-Reduce for Multiterminal Cuts." In: *Proc. of the Twenty-First Workshop on Algorithm Engineering and Experiments, ALENEX 2020*. SIAM, 2020.

[92] M. Henzinger, A. Noe, C. Schulz, and D. Strash. "Finding All Global Minimum Cuts in Practice." In: *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*.

[93] M. Henzinger, A. Noe, C. Schulz, and D. Strash. "Practical Minimum Cut Algorithms." In: *Proc. of the Twentieth Workshop on Algorithm Engineering and Experiments, ALENEX 2018, New Orleans, LA, USA, January 7-8, 2018*. 2018, pp. 48–61.

[94] M. Henzinger, A. Noe, C. Schulz, and D. Strash. "Practical Minimum Cut Algorithms." In: *ACM Journal of Experimental Algorithmics* 23 (2018).

[95] M. Henzinger, S. Rao, and D. Wang. "Local Flow Partitioning for Faster Edge Connectivity." In: *Proc. of the 28th ACM-SIAM Symp. on Discrete Algorithms*. SIAM. 2017, pp. 1919–1938.

[96] M. R. Henzinger. "Approximating minimum cuts under insertions." In: *International Colloquium on Automata, Languages, and Programming*. Springer. 1995, pp. 280–291.

[97] D. Hespe, S. Lamm, C. Schulz, and D. Strash. "We Got You Covered: The Winning Solver from the PACE 2019 Implementation Challenge, Vertex Cover Track." In: *SIAM Workshop on Combinatorial Scientific Computing 2020*. Vol. abs/1908.06795. SIAM, 2020. arXiv: 1908.06795.

[98] D. Hespe, C. Schulz, and D. Strash. "Scalable Kernelization for Maximum Independent Sets." In: *Proc. of the Twentieth Workshop on Algorithm Engineering and Experiments, ALENEX 2018, New Orleans, LA, USA, January 7-8, 2018*. 2018, pp. 223–237.

[99] P. Hu and W. C. Lau. "A survey and taxonomy of graph sampling." In: *arXiv preprint arXiv:1308.5865* (2013).

[100] M. Jünger, G. Rinaldi, and S. Thienel. "Practical performance of efficient minimum cut algorithms." In: *Algorithmica* 26.1 (2000), pp. 172–195.

[101] G. Kant. "Algorithms for drawing planar graphs." PhD thesis. 1993.

[102] 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." In: *Proc. of the National Academy of Sciences* 101.9 (2004), pp. 2888–2893.

[103] D. R. Karger. "Global Min-cuts in RNC, and Other Ramifications of a Simple Min-Cut Algorithm." In: *SODA*. Vol. 93. 1993, pp. 21–30.

[104] D. R. Karger. "Minimum cuts in near-linear time." In: *Journal of the ACM* 47.1 (2000), pp. 46–76.

[105] D. R. Karger and C. Stein. "A new approach to the minimum cut problem." In: *Journal of the ACM* 43.4 (1996), pp. 601–640.

[106] D. R. Karger. "A randomized fully polynomial time approximation scheme for the all-terminal network reliability problem." In: *SIAM Review* 43.3 (2001), pp. 499–522.

[107] S. E. Karisch, F. Rendl, and J. Clausen. "Solving Graph Bisection Problems with Semidefinite Programming." In: *INFORMS Journal on Computing* 12.3 (2000), pp. 177–191.

[108] G. Karypis and V. Kumar. "A fast and high quality multilevel scheme for partitioning irregular graphs." In: *SIAM Journal on scientific Computing* 20.1 (1998), pp. 359–392.

[109] G. Karypis and V. Kumar. *Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices, Version 4*. 1998.

[110] A. V. Karzanov and E. A. Timofeev. "Efficient algorithm for finding all minimal edge cuts of a nonoriented graph." In: *Cybernetics and Systems Analysis* 22.2 (1986), pp. 156–162.

[111] K.-i. Kawarabayashi and M. Thorup. "Deterministic global minimum cut of a simple graph in near-linear time." In: *Proc. of the 47th ACM Symp. on Theory of Computing*. ACM. 2015, pp. 665–674.

[112] B. W. Kernighan and S. Lin. "An Efficient Heuristic Procedure for Partitioning Graphs." In: *The Bell System Technical Journal* 49.1 (1970), pp. 291–307.

[113] F. Khorasani, R. Gupta, and L. N. Bhuyan. "Scalable SIMD-Efficient Graph Processing on GPUs." In: *Proc. of the 24th Intl. Conf. on Parallel Architectures and Compilation Techniques*. PACT '15. 2015, pp. 39–50.

[114] K. Kiljan and M. Pilipczuk. "Experimental Evaluation of Parameterized Algorithms for Feedback Vertex Set." In: *17th Intl. Symp. on Experimental Algorithms, SEA 2018*. Vol. 103. LIPIcs. 2018, 12:1–12:12.

[115] I. M. Kloumann and J. M. Kleinberg. "Community membership identification from small seed sets." In: *Proc. of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2014, pp. 1366–1375.

[116] V. Korenwein, A. Nichterlein, R. Niedermeier, and P. Zschoche. "Data Reduction for Maximum Matching on Real-World Graphs: Theory and Experiments." In: *26th European Symp. on Algorithms, ESA'18*. Vol. 112. LIPIcs. 2018, 53:1–53:13.

[117] A. M. Koster, H. L. Bodlaender, and S. P. Van Hoesel. "Treewidth: computational experiments." In: (2001).

[118] K. Kothapalli, S. V. Pemmaraju, and V. Sardeshmukh. "On the analysis of a label propagation algorithm for community detection." In: *Proc. of the 14th Intl. Conf. on Distributed Computing and Networking (ICDCN 2013)*. Vol. 7730. LNCS. Springer. 2013, pp. 255–269.

[119] D. Krioukov, F. Papadopoulos, M. Kitsak, A. Vahdat, and M. Boguná. "Hyperbolic geometry of complex networks." In: *Physical Review E* 82.3 (2010), p. 036106.

[120] B. Krishnamurthy. "An improved min-cut algorithm for partitioning VLSI networks." In: *IEEE Trans. on Computers* 33.5 (1984), pp. 438–446.

[121] J. Łącki and P. Sankowski. "Min-cuts and shortest cycles in planar graphs in o (n loglogn) time." In: *European Symposium on Algorithms*. Springer. 2011, pp. 155–166.

[122] S. Lamm, C. Schulz, D. Strash, R. Williger, and H. Zhang. "Exactly Solving the Maximum Weight Independent Set Problem on Large Real-World Graphs." In: *Proc. of the Twenty-First Workshop on Algorithm Engineering and Experiments, ALENEX 2019, San Diego, CA, USA, January 7-8, 2019*. Ed. by S. G. Kobourov and H. Meyerhenke. SIAM, 2019, pp. 144–158.

[123] A. H. Land and A. G. Doig. "An Automatic Method of Solving Discrete Programming Problems." In: *Econometrica* 28.3 (1960), pp. 497–520.

[124] U. Lauther. "An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background." In: *Proc. of the Münster GI-Days*. 2004.

[125] D.-T. Lee and B. J. Schachter. "Two algorithms for constructing a Delaunay triangulation." In: *International Journal of Computer & Information Sciences* 9.3 (1980), pp. 219–242.

[126] J. Leskovec, K. J. Lang, and M. Mahoney. "Empirical comparison of algorithms for network community detection." In: *Proc. of the 19th international conference on World wide web*. ACM. 2010, pp. 631–640.

[127] J. Li. "Deterministic Mincut in Almost-Linear Time." In: (2020).

[128] J. Li and D. Panigrahi. "Deterministic Min-cut in Poly-logarithmic Max-flows." In: (2020).

[129] S. Lin and B. W. Kernighan. "An effective heuristic algorithm for the traveling salesman problem." In: *Operations research* 21.2 (1973), pp. 498–516.

[130] A. Lisser and F. Rendl. "Graph Partitioning using Linear and Semidefinite Programming." In: *Mathematical Programming* 95.1 (2003), pp. 91–101.

[131] M. von Looz, H. Meyerhenke, and R. Prutkin. "Generating random hyperbolic graphs in subquadratic time." In: *Proc. of the 26th Intl. Symp. on Algorithms and Computation (ISAAC 2015)*. Vol. 9472. LNCS. Springer. 2015, pp. 467–478.

[132] F. Luo, J. Z. Wang, and E. Promislow. "Exploring local community structures in large networks." In: *Web Intelligence and Agent Systems: An Intl. Journal* 6.4 (2008), pp. 387–400.

[133] S. A. Macskassy and F. Provost. *A simple relational classifier*. Tech. rep. NEW YORK UNIV NY STERN SCHOOL OF BUSINESS, 2003.

[134] T. Maier, P. Sanders, and R. Dementiev. "Concurrent hash tables: Fast and general?(!)" In: *ACM SIGPLAN Notices*. Vol. 51. 8. ACM. 2016, p. 34.

[135] D. Marx. "Parameterized graph separation problems." In: *Theoretical Computer Science* 351.3 (2006), pp. 394–406.

[136] D. W. Matula. "A linear time $2+\varepsilon$ approximation algorithm for edge connectivity." In: *Proc. of the 4th annual ACM-SIAM Symp. on Discrete Algorithms*. SIAM. 1993, pp. 500–504.

[137] D. P. Mersch, A. Crespi, and L. Keller. "Tracking individuals shows spatial fidelity is a key regulator of ant social organization." In: *Science* 340.6136 (2013), pp. 1090–1093.

[138] A. Mislove, B. Viswanath, K. P. Gummadi, and P. Druschel. "You are who you know: inferring user profiles in online social networks." In: *Proc. of the third ACM international conference on Web search and data mining*. ACM. 2010, pp. 251–260.

[139] A. Miyauchi and N. Sukegawa. "Redundant constraints in the standard formulation for the clique partitioning problem." In: *Optimization Letters* 9.1 (2015), pp. 199–207.

[140] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. "Partitioning Graphs to Speedup Dijkstra's Algorithm." In: *Journal of Experimental Algorithmics (JEA)* 11.2006 (2007).

[141] S. Mukhopadhyay and D. Nanongkai. "Weighted min-cut: sequential, cut-query, and streaming algorithms." In: *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*. ACM, 2020, pp. 496–509.

[142] E. Nabieva, K. Jim, A. Agarwal, B. Chazelle, and M. Singh. "Whole-proteome prediction of protein function via graph-theoretic analysis of interaction maps." In: *Bioinformatics* 21.suppl_1 (2005), pp. i302–i310.

[143] H. Nagamochi and T. Ibaraki. "Computing edge-connectivity in multigraphs and capacitated graphs." In: *SIAM Journal on Discrete Mathematics* 5.1 (1992), pp. 54–66.

[144] H. Nagamochi and T. Kameda. "Canonical cactus representation for minimum cuts." In: *Japan Journal of Industrial and Applied Mathematics* 11.3 (1994), pp. 343–361.

[145] H. Nagamochi and T. Kameda. "Constructing cactus representation for all minimum cuts in an undirected network." In: *Journal of the Operations Research Society of Japan* 39.2 (1996), pp. 135–158.

[146] H. Nagamochi, Y. Nakao, and T. Ibaraki. "A fast algorithm for cactus representations of minimum cuts." In: *Japan journal of industrial and applied mathematics* 17.2 (2000), p. 245.

[147] H. Nagamochi, T. Ono, and T. Ibaraki. "Implementing an efficient minimum capacity cut algorithm." In: *Math. Prog.* 67.1 (1994), pp. 325–341.

[148] D. Naor, D. Gusfield, and C. Martel. "A fast algorithm for optimally increasing the edge connectivity." In: *J. on Comp.* 26.4 (1997), pp. 1139–1165.

[149] D. Naor and V. V. Vazirani. "Representing and enumerating edge connectivity cuts in RNC." In: *Workshop on Algorithms and Data Structures*. Springer. 1991, pp. 273–285.

[150] C. S. J. Nash-Williams. "Edge-disjoint spanning trees of finite graphs." In: *Journal of the London Mathematical Society* 1.1 (1961), pp. 445–450.

[151] M. Padberg and G. Rinaldi. "A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems." In: *SIAM Review* 33.1 (1991), pp. 60–100.

[152] M. Padberg and G. Rinaldi. "An efficient algorithm for the minimum capacity cut problem." In: *Mathematical Programming* 47.1 (1990), pp. 19–36.

[153] L. Page, S. Brin, R. Motwani, and T. Winograd. *The PageRank citation ranking: Bringing order to the web*. Tech. rep. Stanford InfoLab, 1999.

[154] F. Pellegrini and J. Roman. "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs." In: *International Conference on High-Performance Computing and Networking*. Springer. 1996, pp. 493–498.

[155] U. Pferschy, R. Rudolf, and G. J. Woeginger. "Some Geometric Clustering Problems." In: *Nord. J. Comput.* 1.2 (1994), pp. 246–263.

[156] J.-C. Picard and M. Queyranne. "On the structure of all minimum cuts in a network and applications." In: *Combinatorial Optimization II*. Springer, 1980, pp. 8–16.

[157] U. N. Raghavan, R. Albert, and S. Kumara. "Near linear time algorithm to detect community structures in large-scale networks." In: *Physical Review E* 76.3 (2007), p. 036106.

[158] A. Ramanathan and C. J. Colbourn. "Counting almost minimum cutsets with reliability applications." In: *Mathematical Programming* 39.3 (1987), pp. 253–261.

[159] R. A. Rossi and N. K. Ahmed. "An Interactive Data Repository with Visual Analytics." In: *SIGKDD Explor.* 17.2 (2016), pp. 37–41.

[160] R. A. Rossi and N. K. Ahmed. "The Network Data Repository with Interactive Graph Analytics and Visualization." In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*. 2015.

[161] P. Sanders and C. Schulz. "Distributed Evolutionary Graph Partitioning." In: *Proc. of the 12th Workshop on Algorithm Engineering and Experimentation (ALENEX'12)*. 2012, pp. 16–29.

[162] P. Sanders and C. Schulz. "Engineering Multilevel Graph Partitioning Algorithms." In: *Proc. of the 19th European Symp. on Algorithms*. Vol. 6942. LNCS. Springer, 2011, pp. 469–480.

[163] P. Sanders and C. Schulz. "Think Locally, Act Globally: Highly Balanced Graph Partitioning." In: *Proc. of the 12th Int. Symp. on Experimental Algorithms (SEA'13)*. LNCS. Springer, 2013.

[164] P. Sanders. "Algorithm engineering–an attempt at a definition." In: *Efficient Algorithms*. Springer, 2009, pp. 321–340.

[165] P. Sanders and C. Schulz. "Think Locally, Act Globally: Highly Balanced Graph Partitioning." In: *Proc. of the 12th Intl. Symp. on Experimental Algorithms (SEA 2013)*. Vol. 7933. LNCS. Springer, 2013, pp. 164–175.

[166] T. Schank and D. Wagner. "Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study." In: *Proc. of the 4th Intl. Workshop on Experimental and Efficient Algorithms (WEA 2005)*. Vol. 3503. LNCS. Springer. 2005, pp. 606–609.

[167] K. Schloegel, G. Karypis, and V. Kumar. "Graph Partitioning for High Performance Scientific Simulations." In: *The Sourcebook of Parallel Computing*. 2003, pp. 491–541.

[168] C. Schulz and D. Strash. "Graph Partitioning Formulations and Applications to Big Data." In: *Encyclopedia on Big Data Technologies* (2018).

[169] T. Kieritz, D. Luxen, P. Sanders, and C. Vetter. "Distributed Time-Dependent Contraction Hierarchies." In: *Proc. of the 9th International Symposium on Experimental Algorithms*. Vol. 6049. LNCS. Springer, 2010, pp. 83–93.

[170] D. Luxen and D. Schieferdecker. "Candidate Sets for Alternative Routes in Road Networks." In: *Proc. of the 11th International Symposium on Experimental Algorithms (SEA'12)*. Vol. 7276. LNCS. Springer, 2012, pp. 260–270.

[171] S. B. Seidman. "Network structure and minimum degree." In: *Social Networks* 5.3 (1983), pp. 269–287.

[172] M. Sellmann, N. Sensen, and L. Timajev. "Multicommodity Flow Approximation used for Exact Graph Partitioning." In: *Proc. of the 11th European Symposium on Algorithms*. Vol. 2832. LNCS. Springer, 2003, pp. 752–764.

[173] N. Sensen. "Lower Bounds and Exact Algorithms for the Graph Partitioning Problem Using Multicommodity Flows." In: *Proc. of the 9th European Symposium on Algorithms*. Vol. 2161. LNCS. Springer, 2001, pp. 391–403.

[174] A. J. Soper, C. Walshaw, and M. Cross. "A combined evolutionary search and multilevel optimisation approach to graph-partitioning." In: *Journal of Global Optimization* 29.2 (2004), pp. 225–241.

[175] C. L. Staudt and H. Meyerhenke. "Engineering high-performance community detection heuristics for massive graphs." In: *Proc. of the 42nd Intl. Conf. on Parallel Processing (ICPP 2013)*. IEEE. 2013, pp. 180–189.

[176] C. L. Staudt, A. Sazonovs, and H. Meyerhenke. "NetworKit: An interactive tool suite for high-performance network analysis." In: *CoRR, abs/1403.3005* (2014).

[177] C. Stein and M. Levine. *Minimum cut code*. `http://www.columbia.edu/~cs2035/code.html`. Accessed: 2017-06-09.

[178] M. Stoer and F. Wagner. "A simple min-cut algorithm." In: *Journal of the ACM* 44.4 (1997), pp. 585–591.

[179] H. S. Stone. "Multiprocessor Scheduling with the Aid of Network Flow Algorithms." In: *IEEE Trans. Software Eng.* 3.1 (1977), pp. 85–93.

[180] A. Strehl and J. Ghosh. "Cluster Ensembles — A Knowledge Reuse Framework for Combining Multiple Partitions." In: *J. Mach. Learn. Res.* 3 (2002), pp. 583–617.

[181] D. Szklarczyk, A. Franceschini, M. Kuhn, M. Simonovic, A. Roth, P. Minguez, T. Doerks, M. Stark, J. Muller, P. Bork, et al. "The STRING database in 2011: functional interaction networks of proteins, globally integrated and scored." In: *Nucleic acids research* 39.suppl_1 (2010), pp. D561–D568.

[182] 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." In: *Nucleic acids research* 47.D1 (2018), pp. D607–D613.

[183] H. Tamaki. "Positive-Instance Driven Dynamic Programming for Treewidth." In: *25th European Symp. on Algorithms, ESA'17*. Vol. 87. LIPIcs. 2017, 68:1–68:13.

[184] R. Tarjan. "Depth-first search and linear graph algorithms." In: *SIAM journal on computing* 1.2 (1972), pp. 146–160.

[185] R. E. Tarjan and U. Vishkin. "An efficient parallel biconnectivity algorithm." In: *SIAM Journal on Computing* 14.4 (1985), pp. 862–874.

[186] R. P. Tewarson. *Sparse matrices*. Academic Press, 1973.

[187] M. Thorup. "Fully-dynamic min-cut." In: *Combinatorica* 27.1 (2007), pp. 91–127.

[188] J. L. Träff. "Direct graph k-partitioning with a Kernighan–Lin like heuristic." In: *Operations Research Letters* 34.6 (2006), pp. 621–629.

[189] A. Vazquez, A. Flammini, A. Maritan, and A. Vespignani. "Global protein function prediction from protein-protein interaction networks." In: *Nature biotechnology* 21.6 (2003), p. 697.

[190] C. Walshaw. "Walshaw Partitioning Benchmark." `https://chriswalshaw.co.uk/partition/`. 2000.

[191] C. Walshaw and M. Cross. "JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview." In: *Mesh Partitioning Techniques and Domain Decomposition Techniques*. 2007, pp. 27–58.

[192] I. Wegener. "BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT beating, on an average, QUICKSORT (if n is not very small)." In: *Theoretical Computer Science* 118.1 (1993), pp. 81–98.

[193] J. W. J. Williams. "Heapsort." In: *Communications of the ACM* 7.6 (1964), pp. 347–348.

[194] Z. Wu and R. Leahy. "An optimal graph theoretic approach to data clustering: Theory and its application to image segmentation." In: *IEEE Trans. on Pattern Analysis & Machine Intelligence* 11 (1993), pp. 1101–1113.

[195] M. Xiao. "Simple and improved parameterized algorithms for multiterminal cuts." In: *Theory of Computing Systems* 46.4 (2010), pp. 723–736.

[196] R. Xu and D. Wunsch. "Survey of clustering algorithms." In: *IEEE Transactions on neural networks* 16.3 (2005), pp. 645–678.

[197] W. Ye, L. Zhou, D. Mautz, C. Plant, and C. Böhm. "Learning from labeled and unlabeled vertices in networks." In: *Proc. of the 23rd ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*. ACM. 2017, pp. 1265–1274.

[198] N. Zeh. "I/O-efficient graph algorithms." In: *EEF Summer School on Massive Data Sets* (2002).