Finding All Global Minimum Cuts in Practice

Monika Henzinger 🆔
University of Vienna, Faculty of Computer Science, Austria
monika.henzinger@univie.ac.at

Alexander Noe 🆔
University of Vienna, Faculty of Computer Science, Austria
alexander.noe@univie.ac.at

Christian Schulz 🆔
University of Vienna, Faculty of Computer Science, Austria
christian.schulz@univie.ac.at

Darren Strash 🆔
Hamilton College, Department of Computer Science, Clinton, NY, USA
dstrash@hamilton.edu

— Abstract —

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. In shared memory we are able to find all minimum cuts of graphs with up to billions of edges and millions of minimum cuts in a few minutes. We also give a new linear time algorithm to find the most balanced minimum cuts given as input the representation of all minimum cuts.

1 Introduction

We consider the problem of finding *all minimum cuts* of an undirected network where edges are weighted by positive integers. A *minimum cut* in a graph is a partition of the vertices into two sets so that the total weight of edges crossing the boundary between the blocks is minimized. The problem of finding all minimum cuts has applications in many fields. In particular, minimum cuts in similarity graphs can be used to find clusters [57, 25]. As the minimum cut is often highly skewed, a variety of techniques to find more balanced bipartitions with small cuts were developed [16, 24, 53]. However, these balanced variations of the problem are generally NP-complete. In contrast to that, we find all minimum cuts in practice in a similar timescale than finding an arbitrary minimum cut and can thus output the minimum cut that is least skewed. In community detection, the absence of a small cut inside a cluster can indicate a likely community in a social network [10]. Other applications for finding all minimum cuts can be found in network reliability [34, 51], where a minimum cut in a network

has the highest risk to disconnect the network; in VLSI design [40] and graph drawing [32], in which minimum cuts are used to separate the network; and finding all minimum cuts is an important subproblem for edge-connectivity augmentation algorithms [19, 47].

The problem of finding all minimum cuts is closely related to the *(global) minimum cut problem*, which aims to find *some* minimum cut in the graph. The classical algorithm of Gomory and Hu [23] solves the minimum cut problem by solving n minimum-s-t cut problems. Using the push-relabel algorithm [22] this results in a total running time of $\mathcal{O}\left(n^2 m \log \frac{n^2}{m}\right)$. Nagamochi et al. [43, 46] give an algorithm for the minimum cut problem, which is based on edge contractions instead of maximum flows. Their algorithm has a worst case running time of $\mathcal{O}\left(nm + n^2 \log n\right)$ but performs far better in practice on many graph classes [12, 31, 28]. Henzinger et al. [26] give a fast shared-memory parallel algorithm for the minimum cut problem based on their algorithm which finds a minimum cut very fast in practice.

Even though a graph can have up to $\binom{n}{2}$ minimum cuts [33], there is a compact representation of all minimum cuts of a graph called *cactus graph* with $\mathcal{O}(n)$ vertices and edges. A cactus graph is a graph in which each edge belongs to at most one simple cycle. Karzanov and Timofeev [36] give the first polynomial time algorithm to construct the cactus representation for all minimum cuts. Picard and Queyranne [50] show that all minimum cuts separating two specified vertices can be found from a maximum flow between them. Thus, similar to the classical algorithm of Gomory and Hu [23] for the minimum cut problem, we can find all minimum cuts in $n-1$ maximum flow computations. The algorithm of Karzanov and Timofeev [36] combines all those minimum cuts into a cactus graph representing all minimum cuts. Nagamochi and Kameda [44] 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. [45] give an algorithm that finds all minimum cuts and gives the minimum cut cactus in $\mathcal{O}\left(nm + n^2 \log n + n^* m \log n\right)$, where n^* is the number of vertices in the cactus.

Karger and Stein [35] give a randomized algorithm to find all minimum cuts in $\mathcal{O}\left(n^2 \log^3 n\right)$ time by contracting random edges. Based on the algorithm of Karzanov and Timofeev [36] and its parallel variant given by Naor and Vazirani [48] they show how to give the cactus representation of the graph in the same asymptotic time. Ghaffari et al. [20] recently gave an algorithm that finds all *non-trivial minimum cuts* 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 $\mathcal{O}\left(n^2 \log n\right)$. While there are multiple implementations of the algorithm of Karger and Stein [12, 21, 28] 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 (with or without data reduction techniques).

In the last two decades significant advances in FPT algorithms have been made: an NP-hard graph problem is fixed-parameter tractable (FPT) if large inputs can be solved efficiently and provably optimally, as long as some problem parameter is small. This has resulted in an algorithmic toolbox that are by now well-established. Few of the new techniques are implemented and tested on real datasets, and their practical potential is far from understood. However, recently the engineering part in area has gained some momentum. There are several experimental studies in the area that take up ideas from FPT or kernelization theory, e.g. for independent sets (or equivalently vertex cover) [2, 11, 14, 41, 29, 30], for cut tree construction[3], for treewidth computations [7, 54, 39], for the feedback vertex set problem [37, 18], for the dominating set problem [1], for the minimum cut [26, 28], for the multiterminal cut problem [27], for the maximum cut problem [17] and for the cluster editing problem [8]. Recently, this type of data reduction techniques is also applied for problem in P such as matching [38].

Our Results

We give an algorithm that finds all minimum cuts on very large graphs. Our algorithm is based on the recursive algorithm of Nagamochi et al. [45]. We combine the algorithm with a multitude of techniques to find edges that can be contracted without affecting any minimum cut in the graph. Some of these techniques are adapted from techniques for the global minimum cut problem [49, 46]. 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. [45] 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 (and all others too) of graphs with billions of edges in mere minutes, this allows the use of minimum cuts as a subroutine in sophisticated data mining and graph analysis tools.

2   Basic Concepts

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 $E[A] := \{(u, v) \in E \mid u \in A, v \in V \setminus A\}$; that is, the set of edges in E that start in A and end in its complement. A cut $(A, V \setminus A)$ is a partitioning of the vertex set V into two non-empty *partitions* A and $V \setminus A$, each being called a *side* of the cut. The *capacity* or *weight* of a cut $(A, V \setminus A)$ is $c(A) = \sum_{(u,v) \in E[A]} c(u, v)$. A *minimum cut* is a cut $(A, V \setminus A)$ that has smallest capacity $c(A)$ among all cuts in G. We use $\lambda(G)$ (or simply λ, when its meaning is clear) to denote the value of the minimum cut over all $A \subset V$. 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. If all edges have weight 1, $\lambda(G, s, t)$ 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, $\hat{\lambda}(G)$ (or simply $\hat{\lambda}$) denotes the smallest upper bound of the minimum cut that the algorithm discovered until that point. For a vertex $u \in V$ with minimum vertex degree, the size of the *trivial cut* $(\{u\}, V \setminus \{u\})$ is equal to the vertex degree of u. Many algorithms tackling 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 $c(u, w) + c(v, w)$.

A graph with n vertices can have up to $\Omega(n^2)$ minimum cuts [33]. 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 can be represented by a cactus graph C_G with up to $2n$ vertices and $\mathcal{O}(n)$ edges [45]. 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 it 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.

3   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 a minimum cut. Thus, we first show that an edge e that is not part of any minimum cut in graph G can be contracted and all minimum cuts of G remain in the resulting graph G/e.

▶ **Lemma 1** ([35]). *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 1 is very useful to reduce the size of the graph by a multitude of techniques to identify such edges. We first give a short overview of our algorithm and then explain the techniques in more detail. First, we use the shared-memory parallel heuristic minimum cut algorithm VieCut [28] in order to find an upper bound $\hat{\lambda}$ for the minimum cut which is very likely to be the correct value. VieCut is a multilevel algorithm that uses the label propagation algorithm to contract heavily connected clusters. 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. [43, 46] and Padberg et al. [49] to the problem of finding all minimum cuts. Section 3.1 explains these contraction routines. On the resulting graph we find all minimum cuts using an optimized variant of the algorithm of Nagamochi, Nakao and Ibaraki [45] 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 3.2. Afterwards, in Section 3.3 we show how we combine the parts into a fast algorithm to find all minimum cuts of large networks.

3.1   Edge Contraction

As shown in Lemma 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 [5]. After each run of a contraction technique that finds contractible edges, we create the contracted
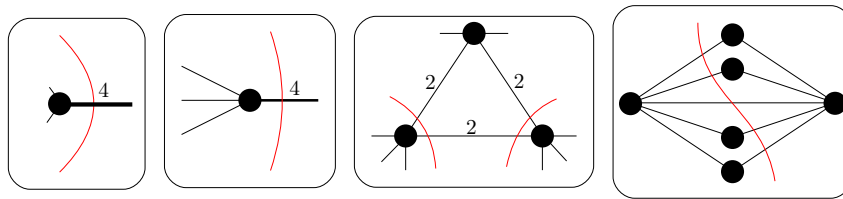
Figure 1 Contraction: (1) `HeavyEdge`, (2) `ImbalancedVertex`, (3) `ImbalancedTriangle`, (4) `HeavyNeighborhood`.

graph using a shared-memory parallel hash table [42]. 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.

3.1.1 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 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. [43, 46] give an algorithm that computes a lower bound $q(e)$ for the connectivity of every edge e of G in a total running time of $\mathcal{O}(m + n \log n)$. Each of the edges whose connectivity lower bound is already larger than $\hat{\lambda}$ can be contracted as it cannot be part of any minimum cut. Their algorithm builds *edge-disjoint maximum spanning forests* and contracts all edges that are not in the first $\lambda - 1$ spanning forests, as those connect vertices that have connectivity at least λ [26]. This is possible as the incident vertices of any such edge e are connected in each of the first $\lambda - 1$ spanning forests and by e and thus have a connectivity of at least λ. In other words, there can not be any cut smaller than λ which contains e.

Henzinger et al. [26] give a fast shared-memory parallel variant of their algorithm. As both of these algorithms only aim to find a single minimum cut, they also contract edges that have connectivity equal to $\hat{\lambda}$, as they only want to see whether there is a cut 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 $\hat{\lambda}$. Nagamochi et al. could prove that at least one edge has value $\hat{\lambda}$ in their routine and can thus be contracted. We do not have such a guarantee when trying to find edges that have connectivity $> \hat{\lambda}$. Consider for example an unweighted tree, whose minimum cut has a value of 1 and each edge has connectivity 1 as well.

3.1.2 Local Contraction Criteria

Padberg and Rinaldi [49] give a set of *local reduction* routines which determine whether an edge can be contracted without affecting the minimum cut. Their reductions routines were shown to be very useful in order to find a minimum cut fast in practice [12, 31, 28]. 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 $\hat{\lambda}$, 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 1 depicts the contraction routines and Lemma 2 gives a more formal definition of them.

▶ **Lemma 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 $(v, V \setminus v)$, 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 weight $> \hat{\lambda}$ 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 [52]. Thus, we only perform linear-time runs as developed by Chekuri et al. [12] by marking the neighborhood of u and v while we check the conditions and do not perform the test on marked vertices.

3.1.3   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 $\hat{\lambda}$ to the minimum degree whenever we perform a bulk edge contraction, $c(e) \geq \hat{\lambda}$: for an edge whose weight is $> \hat{\lambda}$, condition `HeavyEdge` will contract it. For an edge whose weight is $\hat{\lambda}$, the edge represents a trivial minimum cut iff $\hat{\lambda} = \lambda$. 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 $\hat{\lambda}$ 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.

3.2   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 [45]. 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 [45], 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 details we refer the reader to [45]. 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. For details we refer the reader to [45].

As the contraction routines in Section 3.1 usually mark a large amount of edges that can be contracted in bulk, we represent the graph in the compressed sparse row format [56]. 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. This information is updated during each edge contraction. Inside this algorithm we re-run the contraction routines of Section 3.1. 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 steps.

3.2.1   Edge Selection

The recursive algorithm of Nagamochi, Nakao and Ibaraki [45] 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, afterwards 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.

3.2.2   Degree-two Reductions

Over the course of this recursive contraction-based algorithm, we routinely 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 $(\{v\}, V \setminus \{v\})$ 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 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 $(\{v\}, V \setminus \{v\})$ 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.

Algorithm 1 Algorithm to find all minimum cuts.

1: **procedure** FINDALLMINCUTS($G = (V, E)$)
2:    $\hat{\lambda} \leftarrow$ VieCut(G) [28]
3:    **while** not converged **do**
4:       $(G, D_1, \hat{\lambda}) \leftarrow$ contract degree-one vertices($G, \hat{\lambda}$)
5:       $(G, \hat{\lambda}) \leftarrow$ connectivity-based contraction($G, \hat{\lambda}$)
6:       $(G, \hat{\lambda}) \leftarrow$ local contraction($G, \hat{\lambda}$)
7:    $\lambda \leftarrow$ FindMinimumCutValue(G)
8:    $C \leftarrow$ RecursiveAllMincuts(G, λ) ([45])
9:    $C \leftarrow$ reinsert vertices(C, D_1)
10:    **return** (C, λ)

$c(e_0) = c(e_1)$ 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 $(\{v\}, V \setminus \{v\})$ 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 into a single vertex 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 $c(u_0, u_1)$.

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.

3.3 Putting it all together

Algorithm 1 gives an overview over our algorithm to find all minimum cuts. Over the course of the algorithm we keep an upper bound $\hat{\lambda}$ for the minimum cut, initially set to the result of the inexact variant of the `VieCut` minimum cut algorithm [28]. While the `VieCut` algorithm also offers an exact version [26], 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 3.1, 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 [43, 46] 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. [45], as sketched in Section 3.2, and afterwards re-insert all minimum cut edges that were

previously deleted. Before each recursive call of the algorithm of Nagamochi et al. [45], we contract edges incident to degree-one and eligible degree-two vertices. Every 10 recursion levels we additionally check for connectivity-based edge contractions and local contractions.

3.4   Shared-Memory Parallelism

Algorithm 1 employs shared-memory parallelism in every step. When we run the algorithm in parallel, we use the parallel variant of VieCut [28]. Local contraction and marking of degree one vertices are parallelized using OpenMP [13]. For the first round of connectivity-based contraction, we use the parallel connectivity certificate used in the shared-memory parallel minimum cut algorithm by Henzinger et al. [26]. This connectivity certificate is essentially a parallel version of the connectivity certificate of Nagamochi et al. [43, 46], 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 the shared-memory parallel minimum cut algorithm of Henzinger et al. [26] to find the exact minimum cut of the graph. The algorithm of Nagamochi et al. [45] 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.

4   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 $(A, V \setminus A)$ that maximizes $\min(|A|, |V \setminus A|)$. Note that this is not equal to the most balanced s-t-cut problem, which is NP hard [9]. 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 $\mathcal{O}\big((n^*)^3\big)$, as one can enumerate all $\mathcal{O}\big((n^*)^2\big)$ minimum cuts [33] 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 $(A, V \setminus A)$, we define the balance $b(A)$ (or $b(V \setminus A)$) 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 v; and $b(v)$ accordingly. For a cycle $C = \{c_1, \ldots, c_i\}$, we define $b(c_j, \ldots, c_{k \mod i})$ with $0 \geq j \geq k$ analogously as the balance of the minimum cut splitting the cycle so that the sub-cacti rooted in $c_j, \ldots, c_{k \mod i}$ are on one side of the cut and the rest are on the other side (see blue line in Figure 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.
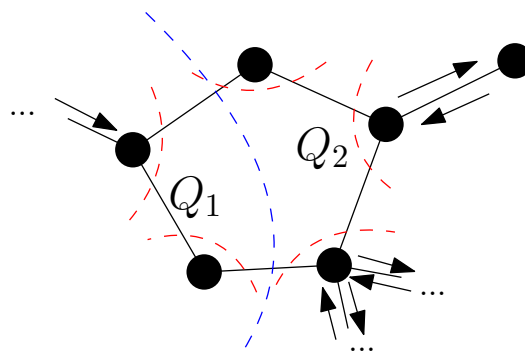
Figure 2 Cycle check in balanced cut algorithm.

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 $\mathcal{C} = \{c_0, \ldots, c_{i-1}\}$ 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 $c_1, \ldots, c_{i-1}$ 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 2. 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.

Algorithm 2 Algorithm to find most balanced cut in cycle $\{c_0, \ldots, c_{i-1}\}$.

1: **procedure** BALANCEINCYCLE($G = (V, E), C = \{c_1, \ldots, c_i\}$)
2:     $b_{OPT} \leftarrow 0$
3:     $Q_1 = \text{Queue}(\{\})$
4:     $Q_2 = \text{Queue}(\{c_0, c_1, \ldots, c_{i-1}\})$
5:     **while** c_0 not Q_1.head() for second time **do**
6:         $b_{OPT} \leftarrow \text{checkBalance}(Q_1, Q_2)$
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:     **return** b_{OPT}

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

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 3.** *Algorithm 2 terminates after $O(i)$ steps.*

Proof. In each step of Algorithm 2, 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 2, we only check the balance of a subset of cuts represented by edges in the cycle C. Lemma 5 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 2, we first show the following Lemma:

▶ **Lemma 4.** *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.** *Algorithm 2 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 $c_{(l-1) \mod i}$ 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.
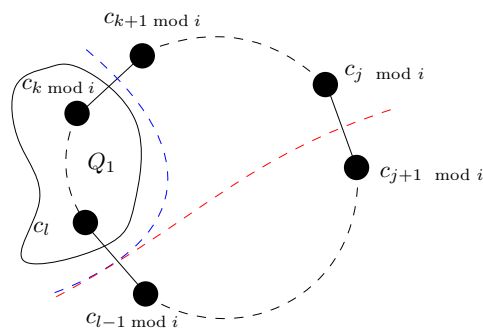
Figure 3 State of Q_1 at time t_l (cut in blue). Cut in red denotes cut considered at time t^*.

Let t_l be the time that c_l becomes the head of Q_1 for the first time. Figure 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. $Q_1 = \{c_{l-1 \bmod i}, \ldots, c_{k \bmod i}\}$ 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 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 $c_l, \ldots, c_{j \bmod i}$ 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 $(\{c_l, \ldots, c_{j \bmod i}\}, \mathcal{C} \backslash \{c_l, \ldots, c_{j \bmod i}\})$.

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 $(\{c_l, \ldots, c_{k \bmod i}\}, \mathcal{C} \backslash \{c_l, \ldots, c_{k \bmod i}\})$, 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 $(\{c_l, \ldots, c_{k \bmod i}\}, \mathcal{C} \backslash \{c_l, \ldots, c_{k \bmod i}\})$ 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 $c_{l-1 \bmod i}, \ldots, c_{k-1 \bmod i}$ 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, $(\{c_l, \ldots, c_{k-1 \bmod i}\}, \mathcal{C} \backslash \{c_l, \ldots, c_{k-1 \bmod i}\})$ 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 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 [4], 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   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. Our code is freely available under the permissive MIT license [1].

Instances

We use a variety of graphs from the 10th DIMACS Implementation challenge [6] and the SuiteSparse Matrix Collection [15]. These are social graphs, web graphs, co-purchase matrices, cooperation networks and some generated instances. Table 2 shows a set of smaller instances and Table 3 shows a set of larger and harder to solve instances. All instances are undirected. If the original graph is directed, we generate an undirected graph by removing edge directions and then removing duplicate edges. If a network has multiple connected components, we run on the largest.

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 [55]. However, usually there is usually one huge block that is connected by minimum cuts to a set of small and medium size blocks. Thus, we use our algorithm 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.

5.1   Edge Selection

Figure 4 shows the results for the graphs in Table 2. We compute the cactus graph representing all minimum cuts using the edge selection variants `Random`, `Central`, `Heavy` and `HeavyWeighted`, as detailed in Section 3.2. As we want a majority of the running time in the recursive algorithm of Nagamochi et al. [45], where we actually select edges, we run a variant of our algorithm that only contracts edges using connectivity-based contraction and then runs the algorithm of Nagamochi et al. [45].

---

[1] `https://github.com/VieCut/VieCut`
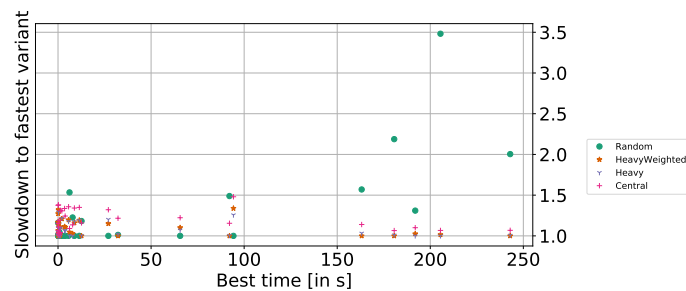
Figure 4 Effect of edge selection strategies.

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.

5.2 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 [45] on the input graph. `+Connectivity` additionally runs VieCut [28] to find an upper bound for the minimum cut and uses this to contract high-connectivity edges as described in Section 3.1.1. In addition to this, `+LocalContract` also contracts edges whose neighborhood guarantees that they are not part of any minimum cut, as described in Section 3.1.2 and Lemma 2. `+DegreeOne` runs also the last remaining contraction routine from Algorithm 1, contraction and re-insertion of degree-one vertices as described in Section 3.1.3.

`+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 3.2.2. We compare these variants on the graphs in Tables 2 and 3. We use a timeout of 30 minutes on these graphs. 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 5a shows the speedup of all variants to the baseline variant `BasicCactus` on all graphs in Table 2. 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
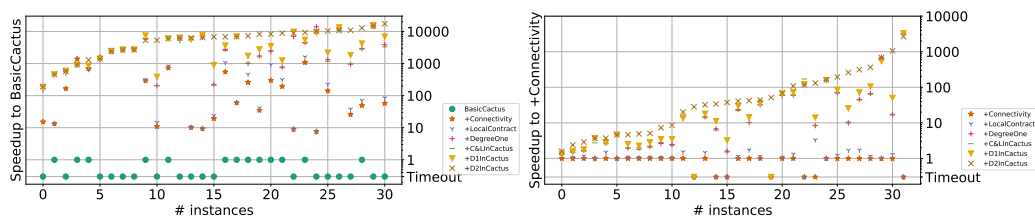
(a) Speedup to `BasicC` on small graphs (Table 2)   **(b)** Speedup to `+Conn` on large graphs (Table 3)

Figure 5 Speedup to respective basic version.

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. [45] 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 5b shows the speedup of all variants. As variant `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 5a, 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   Shared-memory Parallelism

Table 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.

Table 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

6 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 [45] 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 near-minimum cuts.

─── **References** ───

1 Faisal N. Abu-Khzam, Shaowei Cai, Judith Egan, Peter Shaw, and Kai Wang. Turbo-charging dominating set with an FPT subroutine: Further improvements and experimental analysis. In *Proc. 14th Annual Conference on Theory and Applications of Models of Computation (TAMC 2017)*, volume 10185 of *LNCS*, pages 59–70. Springer, 2017. `doi:10.1007/978-3-319-55911-7_5`.

2 Takuya Akiba and Yoichi Iwata. Branch-and-reduce exponential/FPT algorithms in practice: A case study of vertex cover. *Theor. Comput. Sci.*, 609, Part 1:211–225, 2016. `doi:10.1016/j.tcs.2015.09.023`.

3 Takuya Akiba, Yoichi Iwata, Yosuke Sameshima, Naoto Mizuno, and Yosuke Yano. Cut tree construction from massive graphs. In *Proc. 16th International Conference on Data Mining (ICDM 2016)*, pages 775–780, 2016. `doi:10.1109/ICDM.2016.0089`.

4 Reid Andersen and Kevin J Lang. An algorithm for improving graph partitions. In *Proc. 19th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 651–660. SIAM, 2008. URL: `https://dl.acm.org/doi/10.5555/1347082.1347154`.

5 Richard J Anderson and Heather Woll. Wait-free parallel algorithms for the union-find problem. In *Proc. 23rd ACM Symposium on Theory of Computing*, STOC '91, pages 370–380. ACM, 1991. `doi:10.1145/103418.103458`.

6 David A Bader, Henning Meyerhenke, Peter Sanders, Christian Schulz, Andrea Kappes, and Dorothea Wagner. Benchmarking for graph clustering and partitioning. *Encyclopedia of Social Network Analysis and Mining*, pages 73–82, 2014. `doi:10.1007/978-1-4614-7163-9_23-1`.

7 Max Bannach and Sebastian Berndt. Practical access to dynamic programming on tree decompositions. In *Proc. 26th European Symposium on Algorithms (ESA'18)*, volume 112 of *LIPIcs*, pages 6:1–6:13. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018. `doi:10.4230/LIPIcs.ESA.2018.6`.

8 Sebastian Böcker, Sebastian Briesemeister, and Gunnar W. Klau. Exact algorithms for cluster editing: Evaluation and experiments. *Algorithmica*, 60(2):316–334, 2011. `doi:10.1007/s00453-009-9339-7`.

9 Paul Bonsma. Most balanced minimum cuts. *Discrete Appl. Math.*, 158(4):261–276, February 2010. `doi:10.1016/j.dam.2009.09.010`.

10 Deng Cai, Zheng Shao, Xiaofei He, Xifeng Yan, and Jiawei Han. Mining hidden community in heterogeneous social networks. In *Proc. 3rd International Workshop on Link Discovery (LinkKDD '05)*, pages 58–65. ACM, 2005. `doi:10.1145/1134271.1134280`.

11 Lijun Chang, Wei Li, and Wenjie Zhang. Computing a near-maximum independent set in linear time by reducing-peeling. In *Proc. 2017 ACM International Conference on Management of Data (SIGMOD '17)*, pages 1181–1196. ACM, 2017. `doi:10.1145/3035918.3035939`.

12 Chandra S. Chekuri, Andrew V. Goldberg, David R. Karger, Matthew S. Levine, and Cliff Stein. Experimental study of minimum cut algorithms. In *Proc. 8th Symposium on Discrete Algorithms (SODA '97)*, pages 324–333. SIAM, 1997. URL: `https://dl.acm.org/doi/10.5555/314161.314315`.

13   Leonardo Dagum and Ramesh Menon. OpenMP: An industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998. `doi:10.1109/99.660313`.

14   Jakob Dahlum, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. Accelerating local search for the maximum independent set problem. In *Proc. 15th International Symposium on Experimental Algorithms (SEA 2016)*, volume 9685 of *LNCS*, pages 118–133. Springer, 2016. `doi:10.1007/978-3-319-38851-9_9`.

15   Timothy A Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Trans. Mathematical Software (TOMS)*, 38(1):1, 2011. `doi:10.1145/2049662.2049663`.

16   Chris HQ Ding, Xiaofeng He, Hongyuan Zha, Ming Gu, and Horst D Simon. A min-max cut algorithm for graph partitioning and data clustering. In *Proc. 2001 IEEE International Conference on Data Mining (ICDM 2001)*, pages 107–114. IEEE, 2001. `doi:10.1109/ICDM.2001.989507`.

17   Damir Ferizovic, Demian Hespe, Sebastian Lamm, Matthias Mnich, Christian Schulz, and Darren Strash. Engineering kernelization for maximum cut. In *Proc. 22nd Symposium on Algorithm Engineering and Experiments (ALENEX 2020)*, pages 27–41. SIAM, 2020. `doi:10.1137/1.9781611976007.3`.

18   Rudolf Fleischer, Xi Wu, and Liwei Yuan. Experimental study of FPT algorithms for the directed feedback vertex set problem. In *Proc. 17th European Symposium on Algorithms (ESA 2009)*, volume 5757 of *LNCS*, pages 611–622. Springer, 2009. `doi:10.1007/978-3-642-04128-0_55`.

19   Harold N Gabow. Applications of a poset representation to edge connectivity and graph rigidity. In *Proc. 32nd Symposium of Foundations of Computer Science (FOCS '91)*, pages 812–821. IEEE, 1991. `doi:10.1109/SFCS.1991.185453`.

20   Mohsen Ghaffari, Krzysztof Nowicki, and Mikkel Thorup. Faster algorithms for edge connectivity via random 2-out contractions. In *Proc. 2020 ACM-SIAM Symposium on Discrete Algorithms (SODA 2020)*, pages 1260–1279. SIAM, 2020. `doi:10.1137/1.9781611975994.77`.

21   Lukas Gianinazzi, Pavel Kalvoda, Alessandro De Palma, Maciej Besta, and Torsten Hoefler. Communication-avoiding parallel minimum cuts and connected components. In *Proc. 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 219–232. ACM, 2018. `doi:10.1145/3200691.3178504`.

22   Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988. `doi:10.1145/48014.61051`.

23   Ralph E. Gomory and Tien Chung Hu. Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961. `doi:10.1137/0109047`.

24   Lars Hagen and Andrew B Kahng. New spectral methods for ratio cut partitioning and clustering. *IEEE Trans. on Computer-aided Design of Integrated Circuits and Systems*, 11(9):1074–1085, 1992. `doi:10.1109/43.159993`.

25   Erez Hartuv and Ron Shamir. A clustering algorithm based on graph connectivity. *Information Processing Letters*, 76(4-6):175–181, 2000. `doi:10.1016/S0020-0190(00)00142-3`.

26   Monika Henzinger, Alexander Noe, and Christian Schulz. Shared-memory exact minimum cuts. In *Proc. 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2019)*, pages 13–22. IEEE, 2019. `doi:10.1109/IPDPS.2019.00013`.

27   Monika Henzinger, Alexander Noe, and Christian Schulz. Shared-memory branch-and-reduce for multiterminal cuts. In *Proc. 22nd Symposium on Algorithm Engineering and Experiments (ALENEX 2020)*, pages 42–55. SIAM, 2020. `doi:10.1137/1.9781611976007.4`.

28   Monika Henzinger, Alexander Noe, Christian Schulz, and Darren Strash. Practical minimum cut algorithms. *ACM Journal of Experimental Algorithmics*, 23, 2018. `doi:10.1145/3274662`.

29   Demian Hespe, Sebastian Lamm, Christian Schulz, and Darren Strash. We got you covered: The winning solver from the PACE 2019 challenge, vertex cover track. In *Proc. SIAM Workshop on Combinatorial Scientific Computing 2020 (CSC20)*, pages 1–11. SIAM, 2020. `doi:10.1137/1.9781611976229.1`.

30   Demian Hespe, Christian Schulz, and Darren Strash. Scalable kernelization for maximum independent sets. In *Proc. 20th Workshop on Algorithm Engineering and Experiments (ALENEX 2018)*, pages 223–237, 2018. `doi:10.1137/1.9781611975055.19`.

31   Michael Jünger, Giovanni Rinaldi, and Stefan Thienel. Practical performance of efficient minimum cut algorithms. *Algorithmica*, 26(1):172–195, 2000. `doi:10.1007/s004539910009`.

32   Goossen Kant. *Algorithms for drawing planar graphs*. PhD thesis, Utrecht University, 1993. URL: `https://www.persistent-identifier.nl/urn:nbn:nl:ui:10-1874-842`.

33   David R Karger. Minimum cuts in near-linear time. *Journal of the ACM*, 47(1):46–76, 2000. `doi:10.1145/331605.331608`.

34   David R. Karger. A randomized fully polynomial time approximation scheme for the all-terminal network reliability problem. *SIAM Review*, 43(3):499–522, 2001. `doi:10.1137/S0036144501387141`.

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

36   Alexander V Karzanov and Eugeniy A Timofeev. Efficient algorithm for finding all minimal edge cuts of a nonoriented graph. *Cybernetics and Systems Analysis*, 22(2):156–162, 1986. `doi:10.1007/BF01074775`.

37   Krzysztof Kiljan and Marcin Pilipczuk. Experimental evaluation of parameterized algorithms for feedback vertex set. In *Proc. 17th International Symposium on Experimental Algorithms (SEA 2018)*, volume 103 of *LIPIcs*, pages 12:1–12:12, 2018. `doi:10.4230/LIPIcs.SEA.2018.12`.

38   Viatcheslav Korenwein, André Nichterlein, Rolf Niedermeier, and Philipp Zschoche. Data reduction for maximum matching on real-world graphs: Theory and experiments. In *Proc. 26th European Symposium on Algorithms (ESA 2018)*, volume 112 of *LIPIcs*, pages 53:1–53:13, 2018. `doi:10.4230/LIPIcs.ESA.2018.53`.

39   Arie MCA Koster, Hans L Bodlaender, and Stan PM Van Hoesel. Treewidth: computational experiments. *Electronic Notes in Discrete Mathematics*, 8:54–57, 2001. `doi:10.1016/S1571-0653(05)80078-2`.

40   Balakrishnan Krishnamurthy. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Trans. on Computers*, 33(5):438–446, 1984. `doi:10.1109/TC.1984.1676460`.

41   Sebastian Lamm, Christian Schulz, Darren Strash, Robert Williger, and Huashuo Zhang. Exactly solving the maximum weight independent set problem on large real-world graphs. In *Proc. 21st Workshop on Algorithm Engineering and Experiments (ALENEX 2019)*, pages 144–158. SIAM, 2019. `doi:10.1137/1.9781611975499.12`.

42   Tobias Maier, Peter Sanders, and Roman Dementiev. Concurrent hash tables: Fast and general(?)! *ACM Trans. Parallel Comput.*, 5(4), 2019. `doi:10.1145/3309206`.

43   Hiroshi Nagamochi and Toshihide Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM Journal on Discrete Mathematics*, 5(1):54–66, 1992. `doi:10.1137/0405004`.

44   Hiroshi Nagamochi and Tiko Kameda. Canonical cactus representation for minimum cuts. *Japan Journal of Industrial and Applied Mathematics*, 11(3):343–361, 1994. `doi:10.1007/BF03167227`.

45   Hiroshi Nagamochi, Yoshitaka Nakao, and Toshihide Ibaraki. A fast algorithm for cactus representations of minimum cuts. *Japan Journal of Industrial and Applied Mathematics*, 17(2):245–264, 2000. `doi:10.1007/BF03167346`.

46   Hiroshi Nagamochi, Tadashi Ono, and Toshihide Ibaraki. Implementing an efficient minimum capacity cut algorithm. *Math. Prog.*, 67(1):325–341, 1994. `doi:10.1007/BF01582226`.

47   Dalit Naor, Dan Gusfield, and Charles Martel. A fast algorithm for optimally increasing the edge connectivity. *SIAM Journal on Computing*, 26(4):1139–1165, 1997. `doi:10.1137/S0097539792234226`.

48   Dalit Naor and Vijay V Vazirani. Representing and enumerating edge connectivity cuts in RNC. In *Proc. 2nd Workshop on Algorithms and Data Structures (WADS 1991)*, volume 519 of *LNCS*, pages 273–285. Springer, 1991. `doi:10.1007/BFb0028269`.

49   Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33(1):60–100, 1991. `doi:10.1137/1033004`.

50   Jean-Claude Picard and Maurice Queyranne. On the structure of all minimum cuts in a network and applications. In *Combinatorial Optimization II*, pages 8–16. Springer, 1980. `doi:10.1007/BFb0120902`.

51   Aparna Ramanathan and Charles J Colbourn. Counting almost minimum cutsets with reliability applications. *Mathematical Programming*, 39(3):253–261, 1987. `doi:10.1007/BF02592076`.

52   Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Proc. 4th International Workshop on Experimental and Efficient Algorithms (WEA 2005)*, volume 3503 of *LNCS*, pages 606–609. Springer, 2005. `doi:10.1007/11427186_5`.

53   Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000. `doi:10.1109/34.868688`.

54   Hisao Tamaki. Positive-instance driven dynamic programming for treewidth. In *Proc. 25th European Symposium on Algorithms (ESA 2017)*, volume 87 of *LIPIcs*, pages 68:1–68:13, 2017. `doi:10.4230/LIPIcs.ESA.2017.68`.

55   Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. `doi:10.1137/0201010`.

56   Reginald P Tewarson. *Sparse matrices.* Academic Press, 1973.

57   Zhenyu Wu and Richard Leahy. An optimal graph theoretic approach to data clustering: Theory and its application to image segmentation. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, pages 1101–1113, 1993. `doi:10.1109/34.244673`.

A   Graph Instances

Table 2 Set of small graphs.

Name	n	m	λ	n^*
amazon	64 813	153 973	1	10 068
auto	448 695	3.31M	4	43
	448 529	3.31M	5	102
	448 037	3.31M	6	557
	444 947	3.29M	7	1 128
	437 975	3.24M	8	2 792
	418 547	3.10M	9	5 814
caidaRouterLevel	190 914	607 610	1	49 940
cfd2	123 440	1.48M	7	15
citationCiteseer	268 495	1.16M	1	43 031
	223 587	1.11M	2	33 423
	162 464	862 237	3	23 373
	109 522	435 571	4	16 670
	73 595	225 089	5	11 878
	50 145	125 580	6	8 770
cnr-2000	325 557	2.74M	1	87 720
	192 573	2.25M	2	33 745
	130 710	1.94M	3	11 604
	110 109	1.83M	4	9 256
	94 664	1.77M	5	4 262
	87 113	1.70M	6	5 796
	78 142	1.62M	7	3 213
	73 070	1.57M	8	2 449
coAuthorsDBLP	299 067	977 676	1	45 242
cs4	22 499	43 858	2	2
delaunay_n17	131 072	393 176	3	1 484
fe_ocean	143 437	409 593	1	40
kron-logn16	55 319	2.46M	1	6 325
luxembourg	114 599	239 332	1	23 077
vibrobox	12 328	165 250	8	625
wikipedia	35 579	495 357	1	2 172

Table 3 Set of large graphs.

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