Practical Minimum Cut Algorithms*

Monika Henzinger† Alexander Noe † Christian Schulz‡ Darren Strash§

Abstract

The minimum cut problem for an undirected edge-weighted graph asks us to divide its set of nodes into two blocks while minimizing the weight sum of the cut edges. Here, we introduce a linear-time algorithm to compute near-minimum cuts. Our algorithm is based on cluster contraction using label propagation and Padberg and Rinaldi's contraction heuristics [SIAM Review, 1991]. We give both sequential and shared-memory parallel implementations of our algorithm. Extensive experiments on both real-world and generated instances show that our algorithm finds the optimal cut on nearly all instances significantly faster than other state-of-the-art exact algorithms, and our error rate is lower than that of other heuristic algorithms. In addition, our parallel algorithm shows good scalability.

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. A minimum cut is often also referred to as the *edge connectivity* of a graph [14, 26]. The problem has applications in many fields. In particular, for network reliability [17, 32], assuming equal failure chance on edges, the smallest edge cut in the network has the highest chance to disconnect the network; in VLSI design [23], 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 [29].

For minimum cut algorithms to be viable for these (and other) applications they must be fast on small data sets and scale to large data sets. Thus, an algorithm should have either linear or near-linear running time, or have an efficient parallelization. Note that *all* existing exact algorithms have non-linear running time [13, 14, 18], where the fastest of these is the deterministic algorithm of Henzinger et al. [14] with running time $\mathcal{O}(m \log^2 n \log\log^2 n)$. Although this is arguably near-linear theoretical running time, it is not known how the algorithm performs in practice. Even the randomized algorithm of Karger and Stein [18] which finds a minimum cut only with high probability, has $\mathcal{O}(n^2 \log^3 n)$ running time, although this was later improved by Karger [16] to $\mathcal{O}(m \log^3 n)$. There is a linear time approximation algorithm, namely the $(2+\varepsilon)$-approximation algorithm by Matula [25]. However, the quality of Matula's algorithm in practice is currently unknown—no experiments have been published, although Chekuri et al. provide an implementation [7, 37].

To the best of our knowledge, there exists only one parallel algorithm for the minimum cut problem: Karger and Stein present a parallel variant for their random contraction algorithm [18] which computes a minimum cut with high probability in polylogarithmic time on n^2 processors. However, we are unaware of any implementation of this algorithm, or any variant that may be suitable for running in a shared-memory setting. Furthermore, no experiments have been published with any parallelizations. This is not altogether surprising, as previous experimental studies of the minimum cut problem only include instances of up to 85 900 vertices and 1 024 000 edges [24, Table 4.2, Table A.22], which can be solved in milliseconds without parallelization.

1.1 Our Results. In this paper, we give the first *practical* shared-memory parallel algorithm for the minimum cut problem. Our algorithm is heuristic (i.e. , it does not give guarantees on solution quality), randomized, and has running time $\mathcal{O}(n+m)$ 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

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

†University Vienna, Vienna, Austria. {monika.henzinger, alexander.noe}@univie.ac.at

‡Karlsruhe Institute of Technology, Karlsruhe, Germany and University of Vienna, Vienna, Austria. christian.schulz@{kit.edu, univie.ac.at}

§Colgate University, Hamilton, NY, USA dstrash@cs.colgate.edu

a minimum cut using label propagation introduced by Raghavan et al. [31] and contracts them in bulk. We further combine this technique with the exact reduction routines from Padberg and Rinaldi [28]. 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 on up to 70 million vertices and 5 billion edges—the largest graphs ever used for experiments for the minimum cut problem. Results indicate that our algorithm finds optimal cuts on almost all instances and also that the empirically observed error rate is lower than competing heuristic algorithms 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.

2 Related Work

We 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 [9] proved that minimum s-t-cut is equal to maximum s-t-flow. Gomory and Hu [12] 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 [18]. One of the fastest known maximum flow algorithms is the push-relabel algorithm [11] by Goldberg and Tarjan.

Hao and Orlin [13] adapt the push-relabel algorithm to pass information to future flow computations. When a push-relabel iteration is finished, they implicitly merge the source and sink to form a new sink and find a new source. Vertex heights are maintained over multiple iterations of push-relabel. With these techniques they achieve a total running time of $O(mn \log \frac{n^2}{m})$ 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 [28] give a set of heuristics for edge contraction. Chekuri et al. [7] 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. If their algorithm does not find an edge to contract, it performs a maximum flow computation, giving the algorithm worst case running time $O(n^4)$. However, the heuristics can also be used to improve the expected running time of other algorithms by applying them on interim graphs [7].

Nagamochi et al. [26, 27] 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 $O(mn + n^2 \log n)$. Wagner and Stoer [38] give a simpler variant of the algorithm of Nagamochi, Ono and Ibaraki [27], which has a 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 in experiments conducted by Jünger et al. [15]. Both the algorithms of Hao and Orlin or Nagamochi, Ono and Ibaraki achieve close to linear running time on most benchmark instances [7, 15]. There are no parallel implementation of either algorithm known to us.

Kawarabayashi and Thorup [20] give a deterministic near-linear time algorithm for the minimum cut problem, which runs in $O(m \log^{12} n)$. Their algorithm works by growing contractible regions using a variant of PageRank [30]. It was later improved by Henzinger et al. [14] to run in $O(m \log^2 n \log\log^2 n)$ time.

Based on the algorithm of Nagamochi, Ono and Ibaraki, Matula [25] 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. Karger and Stein [18] give a randomized Monte Carlo algorithm based on random edge contractions. This algorithm returns the minimum cut with high probability and a larger cut otherwise.

3 Preliminaries

Basic Concepts. Let $G = (V, E, c)$ 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 extend c to a set of edges $E' \subseteq E$ by summing the weights of the edges; that is, $c(E') := \sum_{e=\{u,v\} \in E'} c(u, v)$. We apply the same notation for single nodes and sets of nodes. 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 weight 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* 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 weight $c(A)$ 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 $A \subset V$. At any point in the execution of a minimum cut algorithm, $\hat{\lambda}(G)$ (or simply $\hat{\lambda}$) denotes the lowest upper bound of the minimum cut that an 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. Hence, the minimum vertex degree $\delta(G)$ can serve as initial bound.

When *clustering* a graph, we are looking for *blocks* of nodes $V_1,\ldots,V_k$ that partition V, i.e., $V_1 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. The parameter k is usually not given in advance. Many algorithms tackling the minimum cut problem use *graph contraction*. Given an edge $(u, v) \in E$, we define $G/(u, v)$ 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 $c(u, w) + c(v, w)$.

The Minimum Cut Algorithm of Nagamochi, Ono and Ibaraki. We discuss the algorithms by Nagamochi, Ono and Ibaraki [26, 27] in greater detail since our work makes use of the tools proposed by those authors. The minimum cut algorithm of Nagamochi, Ono and Ibaraki works on graphs with positive integer weights and computes a minimum cut by building *edge-disjoint maximum spanning forests* and contracting all edges that are not in one of the $\hat{\lambda} - 1$ first spanning forests. There is at least one minimum cut that contains no contracted edges. The algorithm has worst case running time $\mathcal{O}\big(mn + n^2 \log n\big)$. In experimental evaluations [7, 15] it is one of the fastest minimum cut algorithms on real-world instances.

The algorithm uses a modified *breadth-first graph traversal* (BFS) algorithm [26, 27] to find edges that can be contracted without increasing the minimum cut. The algorithm starts at an arbitrary vertex. In each step, the algorithm then visits the vertex that is most strongly connected to the already visited vertices.

Using the modified BFS routine, the algorithm computes a lower bound $q(e)$ for each edge $e = (v, w)$ for the smallest cut $\lambda(G, v, w)$, which places v and w on different sides of the cut. After finishing the BFS, all edges with $q(e) \geq \hat{\lambda}$ are contracted as there exists at least one minimum cut without these edges. Afterwards, the algorithm continues on the contracted graph. A single iteration of the subroutine can be performed in $\mathcal{O}(m + n \log n)$. The authors show that in each BFS run, at least one edge of the graph can be contracted [26]. 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 $n-1$, often scaling proportional to $\log n$.

4 `VieCut`: A Parallel Heuristic Minimum-Cut Algorithm

In this section we explain 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.

The rest of this section is organized as follows: first we give a general overview of our algorithm. Then we introduce the label propagation algorithm [31], which we use to find clusters in the input graph. Note that cluster contraction is an aggressive contraction technique. In contrast to previous approaches, it enables us to drastically shrink the size of the input graph. The intuition behind this technique is that each cluster consists of a set of nodes that all belong to the same side of the cut; as there are usually many edges inside the clusters and only a few between clusters. Thus we contract each cluster. We continue this section with the description of the Padberg-Rinaldi heuristics [28]. These are a set of conditions that can be used to find edges that can be contracted without increasing the value of the minimum cut. We use these contractions to further contract the graph after the label propagation step. Both contractions are repeated until the graph has a constant number of vertices. Then, we apply an exact minimum cut algorithm. We finish the section with a discussion of our parallelization.

4.1 Fast Minimum Cuts

The algorithm of Karger and Stein 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 un-

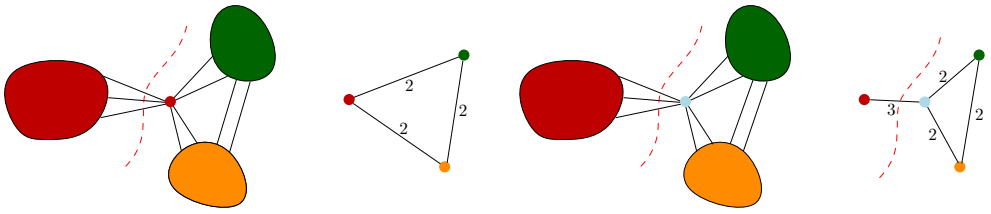

Figure 1: A case in which label propagation misplaces vertices. Left: label propagation assigns the centered vertex correctly to the left (red) cluster. However, this results in a situation in which the contracted graph does not contain the minimum cut anymore. Setting the centered vertex to be a singleton fixes this problem.

desirable: 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 a label propagation algorithm 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 [28]. 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 [27] to find the minimum cut of the contracted graph. Finally, we transfer the resulting cut into a cut in the original graph. Pseudocode can be found in Algorithm 1.

The *label propagation algorithm* (LPA) was proposed by Raghavan et al. [31] 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 [31], the process is repeated until the process converges and no vertices change their labels in a round. Kothapalli et al. [21] 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 a 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 exception are connected components consisting of only a single vertex, i.e. *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 terminates immediately. In practice we do not use the modification, as the label propagation usually returns 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 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 final solution.

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 corre-

Algorithm 1: `VieCut`

```
1  input graph G = (V, E, c : V → ℕ_{>0}), n_0 : bound for exact algorithm
2
3  G ← G
4  while |V_G| > n_0 do
       // compute inexact kernel
5      C ← computeClustering(G)                    // label propagation clustering
6      C' ← fixMisplacedVertices(G, C)             // fix misplaced vertices
7      G_{C'} ← contractClustering(G, C')          // perform contraction
8
       // further apply exact reductions
9      E ← findContractableEdges(G_{C'})           // find contractable edges with Padberg-Rinaldi
10     G ← contractEdges(G_{C'}, E)                // perform contraction
11 end
12 (A, B) ← NagamochiOnoIbaraki(G)                 // solve minimum cut problem on final kernel
13 (A', B') ← solutionTransfer(A, B)               // transfer solution to input network
14
15 return (A', B')
```

sponding 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 4.1. *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, $\lambda(G) = \lambda(G/e)$ [18]. 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. We now briefly discuss the exact reductions by Padberg and Rinaldi [28]:

LEMMA 4.2. (PADBERG-RINALDI) *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) $c(v, w) + \sum_{u \in V} \min\{c(v, u), c(w, u)\} \geq \hat{\lambda}$.

We use this lemma to find contractible edges in the graph. In our implementation, we perform so-called "runs" of the Padberg-Rinaldi heuristics similar to the implementation of Chekuri et al. [7], where each run takes linear time and is split into two passes. In the

first pass of a run, 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 [10], 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 on all triangles in G in linear time, as a graph might have as many as $\Theta(m^{\frac{3}{2}})$ triangles [33]. In the beginning of a pass, we mark each vertex as unscanned. When scanning two adjacent unscanned vertices v and w, we check condition 3 for all vertices u in the common neighborhood $N(v) \cap N(w)$. As we iterate over all vertices in the common neighborhood, we can compute the sum in condition 4 by adding up the smaller of the two edge weights for each vertex in the common neighborhood. Afterwards we 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 for all possible edges (v, w) it is tested whether the vertices v and w can be contracted.

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 described in Section 3.

LEMMA 4.3. *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 4-11) 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.

When 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, our algorithm terminates, as the graph certainly has a minimum cut of 0.

4.2 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 [35] using the `parallel for` directive from the OpenMP [8] 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 a cluster level, i.e. 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 randomness and graph locality by randomly shuffling small blocks of vertex IDs but traversing these shuffled blocks successively. If $v \in C$, 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 every thread builds a temporary graph data structure. In this contraction step, the vertices are assigned to thread at runtime. These temporary graphs are then combined by a single thread.

Parallel Padberg-Rinaldi. In parallel, we run the Padberg-Rinaldi reductions 1, 3 and 4 on the contracted graph. As these criteria are local and independent, they can be parallelized trivially. Updates to the union-find data structure are inside of a `critical` section. Performing reduction 2 in parallel would entail in additional locks, as the vertex weights need to be updated on edge contraction.

4.3 Further Details The label propagation algorithm by Raghavan et al. [31] traverses the graph vertices in random order. Other implementations of the algorithm [35] omit this explicit 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 [40] the label propagation as well as the contraction algorithm can be implemented in external memory [1] 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.

5 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.

Experimental Setup and Methodology. We implemented the algorithms using C++-17 and compiled all codes using g++-7.1.0 with full optimization (`-O3`). All of our experiments are 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. In general, we perform five repetitions per instance and report the average running time as well as cut.

Algorithms. We compare our algorithm with our implementations of the algorithm of Nagamochi, Ono and Ibaraki (`NOI`) [27] and the $(2+\varepsilon)$-approximation algorithm of Matula (`Matula`) [25]. In addition, we compare against the algorithm of Hao and Orlin (`HO`) [13] by using the implementation of Chekuri et al. [7]. 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 [7], 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 [7, 18, 37] (`KS`) without Padberg-Rinaldi tests. We only perform a single iteration of their 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 to for edge contractions. We do not include the algorithm by Stoer and Wagner [38], as it is far slower than `NOI` and `HO` [7, 15] and was also slower in preliminary experiments we conducted. We also do not include the near-linear algorithm of Henzinger et al. [14], as the other algorithms are quasi linear in most instances examined and the algorithm of Henzinger et al. has large constants. We performed preliminary experiments with the core of the algorithm, which indicate that the algorithm is slower in practice.

Instances. We perform experiments on clustered Erdős-Rényi graphs that are generated using the generator from Chekuri et al. [7], which are commonly used in the literature [7, 15, 27, 28]. We also perform experiments on random hyperbolic graphs [22, 39] and on large undirected real-world graphs taken from the 10th DIMACS Implementation Challenge [2] as well as the Laboratory for Web Algorithmics [4, 5]. As these graphs contain vertices with low degree (and therefore trivial cuts), we use the k-core decomposition [3], 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. Appendix A shows basic properties of all graph instances in more detail. The graphs used in our experiments have up to 70 million vertices and up to 5 billion edges. To the best of our knowledge, these graphs are the largest instances reported in literature to be used for experiments on global minimum cuts.

Configuring the Algorithm. In preliminary experiments we tuned the parameters of our algorithm on smaller random hyperbolic graphs. These experiments have been performed on instances not used for the evaluation here. We detail these experiments in Appendix B, and omit them here due to space constraints. In further experiments conducted in this section, we use the configuration of `VieCut` given by the parameter tuning in Appendix B, which performs two iterations of LPA and randomly shuffles blocks of 128 vertices each. We set

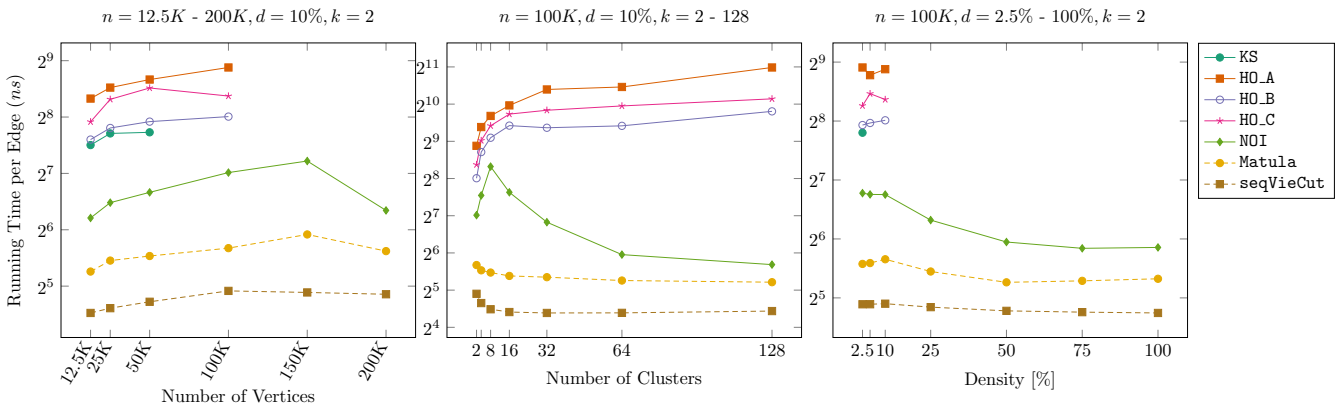

Figure 2: Total running time in nanoseconds per edge in clustered Erdős-Rényi graphs

the bound n_0 to 10 000 and did not encounter a single instance with more than a single bulk contraction step.

5.1 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. [7] uses 32 bit integers to store vertices and edges. We could therefore not perform the experiments with $m \geq 2^{31}$ with `HO`. Figure 2 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.

`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 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 experiments by Chekuri et al. [7].

Random Hyperbolic Graphs. We now perform 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 yielding a total of 72 RHG graphs. Note that these graphs are hard instances for the inexact algorithms, as they contain few – 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

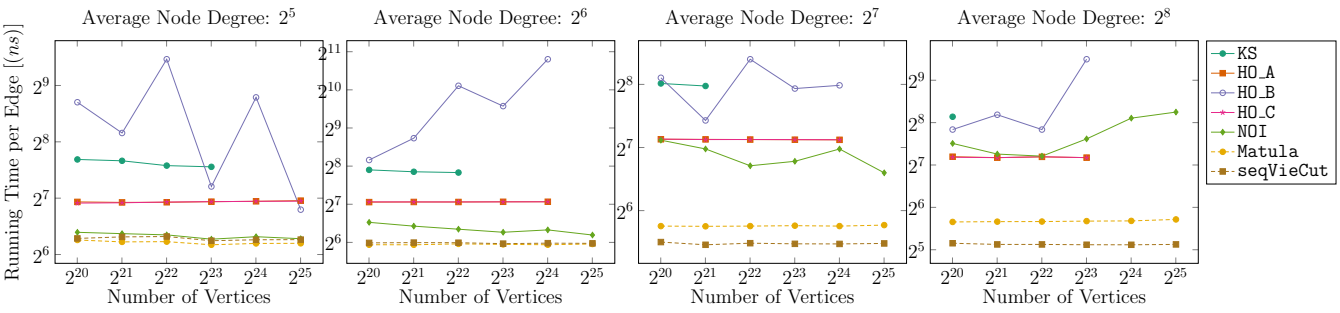

Figure 3: Total running time in nanoseconds per edge in RHG graphs

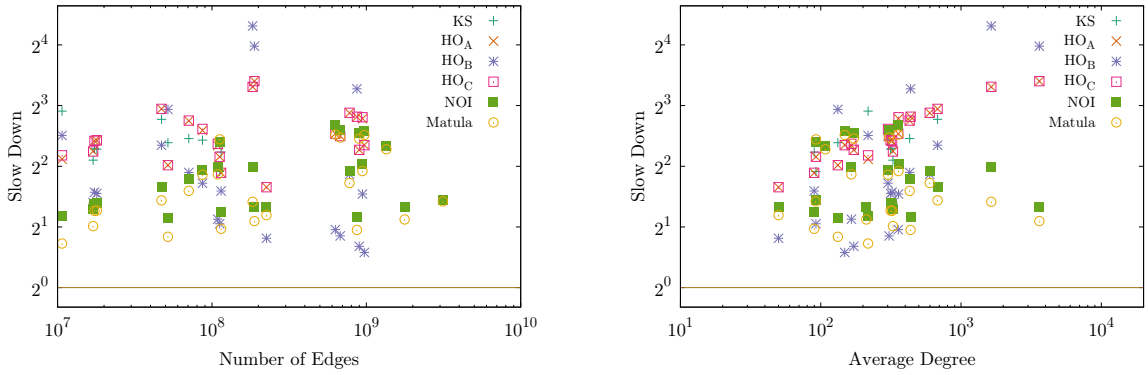

Figure 4: Slowdowns of competitors to `VieCut` in large real-world graphs. We display slowdowns based on the absolute number of edges (left), and by the average vertex degree (right) in the graph

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 shows the results for these experiments. On nearly all of these graphs, `NOI` is faster than `HO`. On the sparse graphs with an average degree of 2^5, `seqVieCut`, `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 5 shows a time breakdown for `seqVieCut` on large RHG graphs with $n = 2^{25}$. Around 80% 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.

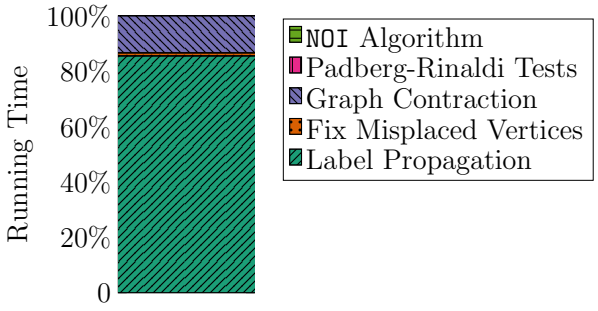

Figure 5: Running Time Breakdown for RHG Graphs with $n = 2^{25}$ and $m = 2^{32}$

Real-World Graphs. We now run experiments on six large real-world social and web graphs. Most of these graphs have many small cuts. On these graphs, there are no non-optimal minimum cuts for any algorithm except for `KS`, which has 36% non-optimal results. Figure 4 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 BFS 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 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

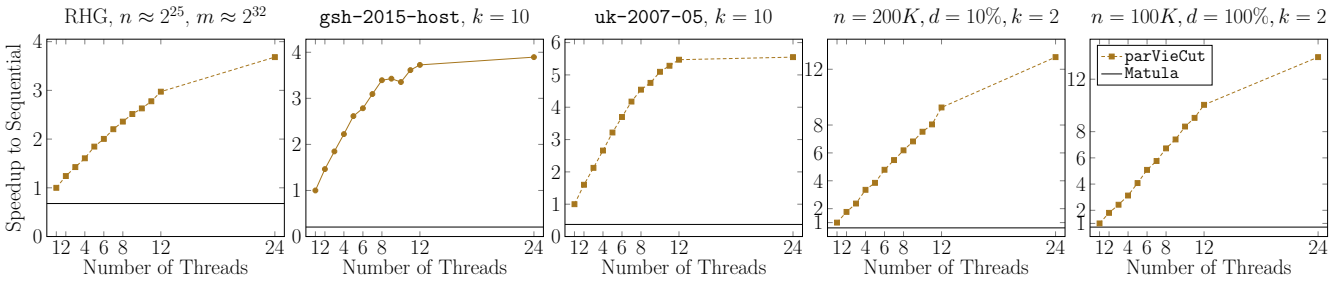

Figure 6: Speedup on large graphs over our algorithm using 1 thread.

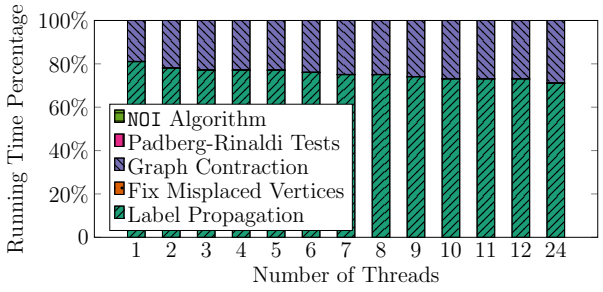

Figure 7: Parallel Running Time Breakdown

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 6 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. On average, `parVieCut` with 12 is 6.3 times faster than `seqVieCut` (24 threads: 7.9x faster), while still having a low error rate even on hard instances. On large RHG graphs we have 2 non-optimal results out of 195 runs. All other results of `parVieCut` on large graphs were optimal. Compared to the next fastest sequential algorithm `Matula`, this is an average speedup factor of 13.2 (24 threads: 15.8x faster). `parVieCut` scales better on the graph `uk-2007-05` ($k = 10$) and especially on the clustered Erdős-Rényi graphs, presumably as these graphs contain many vertices with high degree. Figure 7 shows average running time breakdowns averaged over all graphs. For this figure, the correcting algorithm is turned off for the two Erdős-Rényi graphs. We can see that label propagation scales better than graph contraction. With one thread, label propagation uses 81% of the total running time, with 24 threads, it uses 71% of the total running time.

6 Conclusions and Future Work

We presented the linear-time heuristic algorithm `VieCut` for the minimum cut problem. `VieCut` is based on the label propagation algorithm [31] and the Padberg-Rinaldi heuristics [28]. Both on real-world graphs and a varied family of generated graphs, `VieCut` is significantly faster than the state of art. Our algorithm has far higher solution quality than other heuristic algorithms while also being faster. Important future work includes checking whether using better clustering techniques affect the observed error probability. However, these clustering algorithms generally have higher running time.

References

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

[2] D. 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*. Springer, 2014.

[3] V. Batagelj and M. Zaversnik. An $O(m)$ algorithm for cores decomposition of networks. *arXiv preprint cs/0310049*, 2003.

[4] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, editors, *Proceedings of the 20th International Conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[5] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[6] D. Chakrabarti and C. Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM Computing Surveys*, 38(1):2, 2006.

[7] C. S. Chekuri, A. V. Goldberg, D. R. Karger, M. S. Levine, and C. Stein. Experimental study of minimum cut algorithms. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '97)*, pages 324–333. SIAM, 1997.

[8] L. Dagum and R. Menon. OpenMP: An industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[9] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.

[10] B. A. Galler and M. J. Fisher. An improved equivalence algorithm. *Communications of the ACM*, 7(5):301–303, 1964.

[11] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988.

[12] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961.

[13] J. Hao and J. B. Orlin. A faster algorithm for finding the minimum cut in a graph. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 165–174. Society for Industrial and Applied Mathematics, 1992.

[14] M. Henzinger, S. Rao, and D. Wang. Local flow partitioning for faster edge connectivity. In *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1919–1938. SIAM, 2017.

[15] M. Jünger, G. Rinaldi, and S. Thienel. Practical performance of efficient minimum cut algorithms. *Algorithmica*, 26(1):172–195, 2000.

[16] D. R. Karger. Minimum cuts in near-linear time. *Journal of the ACM*, 47(1):46–76, 2000.

[17] D. R. Karger. A randomized fully polynomial time approximation scheme for the all-terminal network reliability problem. *SIAM Review*, 43(3):499–522, 2001.

[18] D. R. Karger and C. Stein. A new approach to the minimum cut problem. *Journal of the ACM*, 43(4):601–640, 1996.

[19] 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.

[20] K.-i. Kawarabayashi and M. Thorup. Deterministic global minimum cut of a simple graph in near-linear time. In *Proceedings of the 47th Annual ACM Symposium on Theory of Computing*, pages 665–674. ACM, 2015.

[21] K. Kothapalli, S. V. Pemmaraju, and V. Sardeshmukh. On the analysis of a label propagation algorithm for community detection. In *Proceedings of the 14th International Conference on Distributed Computing and Networking (ICDCN 2013)*, volume 7730 of *LNCS*, pages 255–269. Springer, 2013.

[22] D. Krioukov, F. Papadopoulos, M. Kitsak, A. Vahdat, and M. Boguná. Hyperbolic geometry of complex networks. *Physical Review E*, 82(3):036106, 2010.

[23] B. Krishnamurthy. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Transactions on Computers*, 33(5):438–446, 1984.

[24] M. S. Levine. Experimental study of minimum cut algorithms. Master's thesis, Massachusetts Institute of Technology, 1997.

[25] D. W. Matula. A linear time $2 + \varepsilon$ approximation algorithm for edge connectivity. In *Proceedings of the 4th annual ACM-SIAM Symposium on Discrete Algorithms*, pages 500–504. SIAM, 1993.

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

[27] H. Nagamochi, T. Ono, and T. Ibaraki. Implementing an efficient minimum capacity cut algorithm. *Mathematical Programming*, 67(1):325–341, 1994.

[28] M. Padberg and G. Rinaldi. An efficient algorithm for the minimum capacity cut problem. *Mathematical Programming*, 47(1):19–36, 1990.

[29] M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33(1):60–100, 1991.

[30] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[31] U. N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3):036106, 2007.

[32] A. Ramanathan and C. J. Colbourn. Counting almost minimum cutsets with reliability applications. *Mathematical Programming*, 39(3):253–261, 1987.

[33] T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Proceedings of the 4th International Workshop on Experimental and Efficient Algorithms (WEA 2005)*, volume 3503 of *LNCS*, pages 606–609. Springer, 2005.

[34] S. B. Seidman. Network structure and minimum degree. *Social Networks*, 5(3):269–287, 1983.

[35] C. L. Staudt and H. Meyerhenke. Engineering high-performance community detection heuristics for massive graphs. In *Proceedings of the 42nd International Conference on Parallel Processing (ICPP 2013)*, pages 180–189. IEEE, 2013.

[36] C. L. Staudt, A. Sazonovs, and H. Meyerhenke. NetworKit: An interactive tool suite for high-performance network analysis. *CoRR, abs/1403.3005*, 2014.

[37] C. Stein and M. Levine. Minimum cut code. `http://www.columbia.edu/~cs2035/code.html`. Accessed: 2017-06-09.

[38] M. Stoer and F. Wagner. A simple min-cut algorithm. *Journal of the ACM*, 44(4):585–591, 1997.

[39] M. von Looz, H. Meyerhenke, and R. Prutkin. Generating random hyperbolic graphs in subquadratic time. In *Proceedings of the 26th International Symposium on Algorithms and Computation (ISAAC 2015)*, volume 9472 of *LNCS*, pages 467–478. Springer, 2015.

[40] N. Zeh. I/O-efficient graph algorithms. *EEF Summer School on Massive Data Sets*, 2002.

Appendices

Appendix A Instances

A.1 Clustered Erdős-Rényi Graphs. Many previous experimental studies of minimum cut algorithms used a family of clustered Erdős-Rényi graphs with $m = O(n^2)$ [7, 15, 27, 28]. 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 [7, 15, 27, 28] but scaled to larger graphs to account for improvements in machine hardware. We use the generator *noigen* of Andrew Goldberg [37] to generate the clustered Erdős-Rényi graphs for these experiments. This generator was also used in the study conducted by Chekuri et al. [7]. As our code uses the METIS [19] graph format, we use a script to translate the graph format. All experiments exclude I/O times.

A.2 Random Hyperbolic Graphs (RHG) [22]. Random hyperbolic graphs replicate many features of real-world networks [6]: 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. [39], which is a part of NetworKit [36], 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 the minimum cut has two partitions with similar sizes.

A.3 Real-world Graphs. We use large real-world web graphs and social networks from [2, 4, 5], detailed in Table 1. The minimum cut problem on these web and social graphs can be seen as a network reliability problem. As these graphs are generally disconnected and contain vertices with very low degree, we use a k-core decomposition [3, 34] to generate versions of the graphs with a minimum degree of k. The k-core of

graph	n	m	k	n	m	λ	δ
hollywood-2011	2.2M	114M	20	1.3M	109M	1	20
[4, 5]			60	576K	87M	6	60
			100	328K	71M	77	100
			200	139K	47M	27	200
com-orkut	3.1M	117M	16	2.4M	112M	14	16
[4, 5]			95	114K	18M	89	95
			98	107K	17M	76	98
			100	103K	17M	70	100
uk-2002	18M	262M	10	9M	226M	1	10
[2, 4, 5]			30	2.5M	115M	1	30
			50	783K	51M	1	50
			100	98K	11M	1	100
twitter-2010	42M	1.2B	25	13M	958M	1	25
[4, 5]			30	10M	884M	1	30
			50	4.3M	672M	3	50
			60	3.5M	625M	3	60
gsh-2015-host	69M	1.8B	10	25M	1.3B	1	10
[4, 5]			50	5.3M	944M	1	50
			100	2.6M	778M	1	100
			1000	104K	188M	1	1000
uk-2007-05	106M	3.3B	10	68M	3.1B	1	10
[2, 4, 5]			50	16M	1.7B	1	50
			100	3.9M	862M	1	100
			1000	222K	183M	1	1000

Table 1: Statistics of real-world web graphs used in experiments. Original graph size and k-cores used in experiments with their respective minimum cuts

a graph $G = (V, E)$ is the maximum 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. On 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 only had few cores where the minimum cut is not trivial. Therefore we used those cores. As hollywood-2011 is very dense, we multiplied the k value of all cores by a factor of 2.

Appendix B Algorithm Configuration

We performed experiments to tune the number of label propagation iterations and to find an appropriate amount of randomness for our algorithm. We conducted these experiments with different configurations on generated hyperbolic graphs (see Section A.2) 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 one used in the main text.

Table 2 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 4.3. The variant VieCut25 performs up to 25 iterations or until the label propagation converges so

	VieCut1	VieCut2	VieCut3	VieCut4	VieCut5	VieCut10	VieCut25
# of non optimal cuts	29	14	15	16	19	19	18
average distance to opt.	16.2%	2.44%	2.46%	3.30%	3.80%	3.37%	3.14%

Table 2: Error rate for configurations of VieCut in RHG graphs (out of 300 instances)

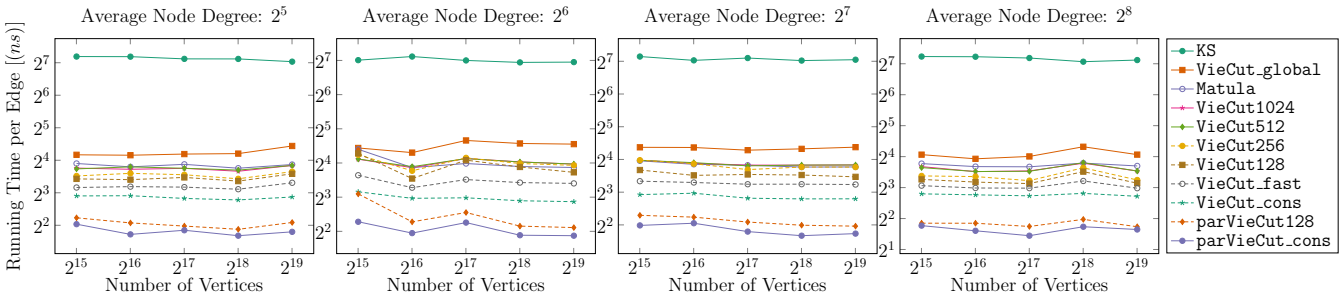

Figure 8: Total running time in nanoseconds per edge in RHG graphs

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 of our experiments.

To compare the effect of graph traversal strategies, we compared different configurations of our algorithm. Configuration `VieCut_cons` does not randomize the traversal order, i.e. it traverses vertices consecutively by their 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 8 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 main text 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 in the main text (named `parVieCut`).