A Simple Algorithm for Minimum Cuts in Near-Linear Time

Nalin Bhardwaj
University of California San Diego, CA, USA
nalinbhardwaj@nibnalin.me

Antonio J. Molina Lovett
Department of Computer Science, Princeton University, NJ, USA
antonio@amolina.ca

Bryce Sandlund
Cheriton School of Computer Science, University of Waterloo, Canada
bcsandlund@gmail.com

Abstract

We consider the minimum cut problem in undirected, weighted graphs. We give a simple algorithm to find a minimum cut that 2-respects (cuts two edges of) a spanning tree T of a graph G. This procedure can be used in place of the complicated subroutine given in Karger's near-linear time minimum cut algorithm [23]. We give a self-contained version of Karger's algorithm with the new procedure, which is easy to state and relatively simple to implement. It produces a minimum cut on an m-edge, n-vertex graph in $O(m \log^3 n)$ time with high probability, matching the complexity of Karger's approach.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis

Keywords and phrases minimum cut, sparsification, near-linear time, packing

Digital Object Identifier 10.4230/LIPIcs.SWAT.2020.12

Supplementary Material Our implementation is available at: `https://github.com/nalinbhardwaj/min-cut-paper`.

1 Introduction

The minimum cut problem on an undirected (weighted) graph G asks for a vertex subset S such that the total number (weight) of edges from S to $V \setminus S$ is minimized. The minimum cut problem is a fundamental problem in graph optimization and has received vast attention by the research community across a number of different computation models [23, 8, 36, 20, 25, 15, 41, 19, 5, 17, 4, 22, 11, 27, 18, 6, 10, 34, 39, 12]. Its applications include network reliability [37, 21], cluster analysis [3], and a critical subroutine in cutting-plane algorithms for the traveling salesman problem [2].

A seminal result in weighted minimum cut algorithms is an algorithm by Karger [23] which produces a minimum cut on an m-edge, n-vertex graph in $O(m \log^3 n)$ time with high probability[1]. This algorithm stood as the fastest minimum cut algorithm for the past two decades, until very recently, work published on arXiv shaved a log factor in Karger's approach [31, 9]. The main component of Karger's algorithm is a subroutine that finds a minimum cut that 2-respects (cuts two edges of) a given spanning tree T of a graph G. In other words, the cut found is minimal amongst all cuts of G that cut exactly two edges of T. Despite the number of pairs of spanning tree edges totaling $\Omega(n^2)$, Karger shows

[1] Probability $1 - 1/n^c$ for some constant c.

this can be accomplished in $O(m \log^2 n)$ time. Unfortunately, the procedure developed is particularly complex, a detail Karger admits when comparing the algorithm to a simpler $O(n^2 \log n)$ algorithm he develops to find *all* minimum cuts [23]. Indeed, perhaps for this reason, implementation of the asymptotically fastest minimum cut algorithm has been avoided in practical performance analyses [5, 19].

In this paper, we give a simple algorithm to find a minimum cut that 2-respects a spanning tree T of a graph G. Our procedure runs in $O(m \log^2 n)$ time, matching the performance of Karger's more-complicated subroutine. We achieve the simplification via a clever use of the heavy-light decomposition. Although our procedure requires the top tree data structure [1] to achieve optimal performance, at the cost of an extra $O(\log n)$ factor, heavy-light decomposition can be used a second time so that only augmented binary search trees are required. We also give a self-contained version of Karger's algorithm [23] with this new procedure and implement it, avoiding issues associated with previous implementations [23, 5].

Karger's algorithm [23], as well as the edge-sampling technique it is based on [22], has been extended and adapted to achieve results in a number of different settings [12, 6, 41, 11, 10, 34]. In particular, in the fully-dynamic setting, Thorup [41] uses the tree-packing technique developed by Karger [23], but maintains a larger set of trees so that the minimum cut 1-respects at least one of them. In the parallel setting, Geissmann and Gianinazzi [11] are able to parallelize both the dynamic tree data structure and the necessary computation required by Karger's algorithm [23]. This work is based off prior work in the cache-oblivious model [10], also based on Karger's algorithm [23]. In the distributed setting, Ghaffari and Kuhn [12] achieve a $(2 + \epsilon)$-approximation to the minimum cut based on Karger's sampling technique [22]. This is improved to a $(1+\epsilon)$-approximation with similar runtime by Nanongkai and Su [34]. Nanongkai and Su develop their algorithm from Thorup's fully-dynamic min-cut algorithm [41], Karger's sampling technique [22], and Karger's dynamic program to find the minimum cut that 1-respects a tree [23]. Finally, Daga et al. [6] achieve a sublinear time distributed algorithm to compute the exact minimum cut in an unweighted undirected graph. This algorithm builds off a more recent development in minimum cut algorithms [27], combined again with the tree-packing technique introduced by Karger [23]. Specifically, a tree packing is found in an efficient number of distributed rounds, then Karger's more-complicated algorithm to find a minimum 2-respecting cut is applied in the distributed setting.

This vast amount of work based on Karger's original near-linear time algorithm suggests that simplifying it may yield additional techniques that can be applied both sequentially and in alternative settings. Indeed, the very recent improvements to Karger's algorithm [31, 9] were published on arXiv two months after our paper was first made available online [28], one of which [9] cites our paper as what drew the authors to the problem. Indeed, their procedure for "descendent edges", given in Section 3.1, is similar to our procedure given in Section 5. We have further found use of the approach given in this paper to achieve new results in dynamic higher connectivity algorithms [30].

This paper is organized as follows. In Section 2, we state the history of the minimum cut problem, in particular discussing other simple algorithms. In Section 3, we give an overview of Karger's algorithm to pack spanning trees, leaving the details of the approach to Appendix A. Our main contribution is given in Sections 4 and 5. In Section 4, we show how to find minimum cuts that 1-respect (cut one edge of) a tree using our new procedure. In Section 5, we extend the approach to find minimum cuts that 2-respect (cut two edges of) a tree. We discuss our implementation in Section 6 and give concluding remarks in Section 7.

2 Related Work

Before we begin, we give a brief history of the minimum cut problem. The minimum cut problem was originally perceived as a harder variant of the maximum s-t flow problem and was solved by $\binom{n}{2}$ flow computations. Gomory and Hu [14] showed how to compute all pairwise max flows in $n-1$ flow computations, thus reducing the complexity of the minimum cut problem by a $\Theta(n)$ factor. Hao and Orlin [16] further showed that the minimum cut in a directed graph can be reduced to a single flow computation.

Nagamochi and Ibaraki [33, 32] developed a deterministic algorithm that is not based on computing maximum s-t flows. They achieve $O(nm + n^2 \log n)$ time on a capacitated, undirected graph. This procedure was simplified by Stoer and Wagner [39], achieving the same runtime. The Stoer-Wagner algorithm gives a simple procedure to find an *arbitrary* minimum s-t cut. Vertices s and t are then merged, and the procedure repeats. Although the $O(nm + n^2 \log n)$ time complexity requires an efficient priority queue such as a Fibonacci heap [7], a binary heap can be used to achieve runtime $O(nm \log n)$.

Two algorithms based on *edge contraction* have been devised. The first is an algorithm of Karger [20] and is incredibly simple. The algorithm randomly contracts edges until only two vertices remain. Repeated $O(n^2 \log n)$ times, the algorithm finds all minimum cuts on an undirected, weighted graph in $O(n^2 m \log n)$ time with high probability. This technique was improved by Karger and Stein [25] by observing an edge of the minimum cut is more likely to be contracted later in the contraction procedure. Their improvement branches the contraction procedure after a certain threshold has been reached, spending more time to avoid contracting an edge of the minimum cut when fewer edges remain. The Karger-Stein algorithm achieves runtime $O(n^2 \log^3 n)$, finding the minimum cut with high probability.

In an unweighted graph, Gabow [8] showed how to compute the minimum cut in $O(cm \log(n^2/m))$ time, where c is the capacity of the minimum cut. Karger [22] improved Gabow's algorithm by applying random sampling, achieving runtime $\tilde{O}(m\sqrt{c})$ in expectation[2]. The sampling technique developed by Karger [22], combined with the tree-packing technique devised by Gabow [8], form the basis of Karger's near-linear time minimum cut algorithm [23]. As previously mentioned, this technique finds the minimum cut in an undirected, weighted graph in $O(m \log^3 n)$ time with high probability.

A recent development uses low-conductance cuts to find the minimum cut in an undirected unweighted graph. This technique was introduced by Kawarabayashi and Thorup [27], who achieve near-linear deterministic time (estimated to be $O(m \log^{12} n)$). This was improved by Henzinger, Rao, and Wang [18], who achieve deterministic runtime $O(m \log^2 n \, (\log \log n)^2)$. Although the algorithm of Henzinger et al. is more efficient than Karger's algorithm [23] on unweighted graphs, the procedure, as well as the one it was based on [27], are quite involved, thus making them largely impractical for implementation purposes.

Since an earlier version of this paper became available online [28], several important improvements in minimum cut algorithms have been discovered. Ghaffari et al. [13] devise a randomized unweighted minimum cut algorithm by using contraction based on sampling from each vertex, rather than standard uniform edge sampling. Their algorithm reduces unweighted minimum cuts to weighted minimum cuts on a graph with $O(n)$ edges, achieving $O(\min(m + n \log^3 n, m \log n))$ time complexity. Gawrychowski et al. [9] improve Karger's procedure for finding the minimum cut that 2-respects a tree to $O(m \log n)$ time. This improves the state-of-the-art for weighted minimum cuts to $O(m \log^2 n)$ time and, by Ghaffari et al. [13],

[2] The $\tilde{O}(f)$ notation hides $O(\log f)$ factors.

improves the complexity of unweighted minimum cuts to $O(\min(m + n\log^2 n, m\log n))$ time. Mukhopadhyay and Nanongkai [31] also study Karger's procedure for finding the minimum cut that 2-respects a tree, arriving at an $O(m\frac{\log^2 n}{\log\log n} + n\log^6 n)$ time weighted minimum cut algorithm. Mukhopadhyay and Nanongkai further apply their new procedure to minimum cuts in the cut-query and streaming models.

3 Overview of Karger's Spanning Tree Packing

We first formalize the definition mentioned earlier in this paper and originally given by Karger.

▶ **Definition 1** (Karger [23]). *Let T be a spanning tree of G. We say that a cut in G k-respects T if it cuts at most k edges of T. We also say that T k-constrains the cut in G.*

We also define weighted tree packings.

▶ **Definition 2** (Karger [23]). *A weighted tree packing is a set of spanning trees, each with an assigned non-negative weight, such that the total weight of trees containing a given edge of G is no greater than the weight of that edge. The weight of the packing is the total weight of the trees in it.*

The first stage of Karger's algorithm is to sample edges independently and uniformly at random from graph G to form a graph H, and then pack spanning trees in H. If we sample a tree T from a packing with probability proportional to its weight, a minimum cut in G will cut at most two edges of T with constant probability. Thus, if we sample $O(\log n)$ trees from the weighted packing, a minimum cut in G 2-respects at least one of the sampled trees with high probability. The remainder of the algorithm is a procedure that, given a spanning tree T of a graph G, finds a minimal cut of G that 2-respects T. This procedure is applied to all $O(\log n)$ sampled spanning trees.

We leave the intuition behind Karger's approach and the relevant mathematics to Appendix A. We will use Algorithm 1 to pack spanning trees, credited to Thorup and Karger [42], Plotkin-Shmoys-Tardos [36], and Young [43]. The procedure appears in Gawrychowski et al. [9].

Algorithm 1 Obtain a Packing of Weight at least $.4c$ from a Graph G.

Let G be a graph with m edges and n vertices.

1. Initialize $\ell(e) \leftarrow 0$ for all edges e of G. Initialize multiset $P \leftarrow \emptyset$. Initialize $W \leftarrow 0$.
2. Repeat the following:
   a. Find a minimum spanning tree T with respect to $\ell(\cdot)$.
   b. Set $\ell(e) \leftarrow \ell(e) + 1/(75\ln m)$ for all $e \in T$. If $\ell(e) > 1$, return W, P.
   c. Set $W \leftarrow W + 1/(75\ln m)$.
   d. Add T to P.

▶ **Lemma 3** ([36, 42, 43]). *Given an undirected unweighted graph G with m edges, n vertices, and minimum cut c, Algorithm 1 returns a weighted packing of weight at least $.4c$ in $O(mc\log n)$ time.*

Algorithm 1 and Lemma 3 are given in Appendix A with general epsilon and proven. To achieve $O(mc\log n)$ time in Algorithm 1, we may use a linear time minimum spanning tree routine [24] or the following implementation trick given by Gawrychowski et al. [9]. In the use of Algorithm 1 in Algorithm 2, the graph in Algorithm 1 has edges which may be

duplicated $O(\log n)$ times, while the number of distinct edges can be bounded as a factor $\Theta(\log n)$ fewer. It suffices to invoke the minimum spanning tree algorithm of Algorithm 1 with only the minimum of each set of parallel edges. We can easily maintain the minimum of each set of parallel edges in $O(\log n)$ time per edge per iteration, which suffices to shave a log factor in the runtime of Algorithm 1. Note that if we chose to avoid these optimizations and/or avoid the use of top trees in Section 5, the final runtime becomes $O(m \log^4 n)$.

We use Algorithm 1 in Algorithm 2 to obtain $\Theta(\log n)$ trees for the 2-respect algorithm given in Sections 4 and 5.

Algorithm 2 Obtain $\Theta(\log n)$ Spanning Trees for the 2-respect Algorithm.

Let d denote the exponent in the probability of success $1 - 1/n^d$. Let $b = 3 \cdot 6^2(d+2) \ln n$.

1. Form graph G' from G by first normalizing the edge weights of G so the smallest non-zero edge weight has weight 1, then multiplying each edge weight by 100 and rounding to the nearest integer. Let U be an upper bound for the size of the minimum cut of G'.
2. Initialize $c' \leftarrow U$. Repeat the following:
   a. Construct H in the following way: for each edge e of G', let e have weight in H drawn from the binomial distribution with probability $p = \min(b/c', 1)$ and number of trials the weight of e in G'. Cap the weight of any edge in H to at most $\lceil 7/6 \cdot 12b \rceil$.
   b. Run Algorithm 1 on H, considering an edge of weight w as w parallel edges. There are three cases:
      i. If $p = 1$, set P to the packing returned and skip to step 3.
      ii. If the returned packing is of weight $24b/70$ or greater, set $c' \leftarrow c'/6$ and repeat steps 2a and 2b, setting P to the packing returned and then proceeding to step 3.
      iii. Otherwise, repeat steps 2a and 2b with $c' \leftarrow c'/2$.
3. Return $\lceil 36.53d \ln n \rceil$ trees sampled uniformly at random proportional to their weights from P.

▶ **Lemma 4.** *Algorithm 2 returns a collection of $\Theta(\log n)$ spanning trees of G in time $O(m \log^3 n)$ such that the minimum cut of G 2-respects at least one tree in the collection with high probability.*

Algorithm 2 and Lemma 4 are given in Appendix A with general epsilon and proven.

4 Minimum Cuts that 1-Respect a Tree

We now give our algorithm for finding a minimum cut that 1-respects a spanning tree T of a graph G. We present it here only to build intuition for the idea used to find 2-respecting cuts in the following section, which also finds 1-respecting cuts.

We use the following lemma, a consequence of Sleator and Tarjan's heavy-light decomposition [38].

▶ **Lemma 5** (Sleator and Tarjan [38]). *Given a tree T, there is an ordering of the edges of T such that the edges of the path between any two vertices in T consist of the union of up to $2 \log n$ contiguous subsequences of the order. The order can be found in $O(n)$ time.*

Proof. We use heavy-light decomposition, credited to Sleator and Tarjan [38]. Note that the algorithm assumes T is rooted. We can root T arbitrarily. We then take the heavy paths given from the usual construction and concatenate them in any order. ◀

Our algorithm begins by labeling the edges of T in heavy-light decomposition order $e_1, \ldots, e_{n-1}$ as given by Lemma 5. Consider the cut of G induced by the vertex partition resulting from cutting a single edge of T. We iterate index i through heavy-light decomposition order and keep up-to-date the total weight of all edges of G that cross the cut induced by e_i. The minimum weight found is then returned.

Call the edges of G in T *tree edges* and edges of G not in T *non-tree edges*. Critical to our approach is the following proposition.

▶ **Proposition 6.** *For any cut of G that 2-respects T, the non-tree edge uv crosses the cut if and only if exactly one tree edge from the uv-path in T crosses the cut.*

Proof. Recall that for any edge of T crossing the cut, the components of each of its endpoints must fall on opposite sides of the cut. Therefore if the number of tree edges in the cut on the uv-path in T is odd, the non-tree edge uv crosses the cut. Since we are only considering cuts that cut at most 2 edges of T, the proposition follows. ◀

We now give our algorithm explicitly.

Algorithm 3 Minimum Cuts that 1-Respect T.

1. Arrange the edges of T in the order of Lemma 5; label them $e_1, \ldots, e_{n-1}$.
2. For each non-tree edge uv, mark every i such that e_i is on the uv-path in T and e_{i+1} is not on the uv-path in T, or vice versa. Indicate whether edge e_1 is on the uv-path in T.
3. Iterate index i from 1 to $n-1$, in each iteration keeping track of the total weight of all non-tree edges uv such that e_i lies on the uv-path in T, added together with the weight of edge e_i.
4. Return the minimum total weight found in step 3.

▶ **Lemma 7.** *Algorithm 3 finds the value of the minimum cut that 1-respects a spanning tree T of a graph G in $O(m \log n)$ time.*

Proof. Via Proposition 6, in a 1-respecting cut including only e_i from T, a non-tree edge uv is cut if and only if the edge e_i lies on the uv-path in T. Algorithm 3 keeps track of all such non-tree edges for each possible e_i that is cut, therefore it finds the minimum cut of G that cuts a single edge of T.

The time complexity can be determined as follows. Finding the heavy-light decomposition for step 1 takes $O(n)$ time. In doing so, we can label each edge and each heavy path so that every edge knows its index in the order as well as the heavy path to which it belongs. Each heavy path can store its starting and ending index in the order. With this information, step 2 can be completed by walking up from u and v in T towards the root of T. We spend $O(1)$ work per heavy path from root to vertex, which is bounded by $O(\log n)$ via the heavy-light decomposition. In total this step takes $O(m \log n)$ time.

In step 3, we spend $O(n)$ total work plus $O(1)$ work for each transition of the current edge e_i on or off the uv path for all non-tree edges uv. Each non-tree edge transitions on or off $O(\log n)$ times as guaranteed by Lemma 5, therefore the time complexity of this step is $O(m \log n)$. Overall, Algorithm 3 takes $O(m \log n)$ time. ◀

Note that if we wish to find the edges in the minimum cut, we can keep track of the minimum-achieving index i so we know the vertex separation of the minimum cut. With the vertex separation, it is easy to find in $O(m \log n)$ time which non-tree edges cross the cut.

Further note that we need not know the identity of the non-tree edge uv as e_i falls on or off the uv-path. Thus the space required for step 2 need only be $O(m)$, since at each transition point we can just keep track of the total weight added or subtracted from the minimum cut.

5 Minimum Cuts that 2-Respect a Tree

We now discuss an extension of Algorithm 3 to find a minimum cut that 2-respects a tree. We still iterate i through heavy-light decomposition order, but in addition to cutting e_i, we find the best j so that the cut resulting from cutting e_i and e_j is minimal. To find the best j efficiently we use a clever data structure.

▶ **Lemma 8** (Alstrup et al. [1]). *There is a data structure that supports the following operations on a weighted tree T in $O(\log n)$ time:*

- *`PathAdd(u, v, x)` := Add weight x to all edges on the unique uv-path in T.*
- *`NonPathAdd(u, v, x)` := Add weight x to all edges not on the unique uv-path in T.*
- *`QueryMinimum()` := Query for the minimum weight edge in T.*

Proof. Operations `PathAdd()` and `QueryMinimum()` are just Theorems 3 and 4 of [1]. Operation `NonPathAdd(u, v, x)` can be achieved by keeping a counter of global weight added to (subtracted from) T and executing `PathAdd(u, v, -x)` to undo this action on the uv-path. See also [40]. ◀

Note that the weight x can be positive or negative.

If we seek to avoid implementing any sophisticated data structures, we can instead use heavy-light decomposition again and support the above two operations in $O(\log^2 n)$ time. To see how, by Lemma 5 each path of T represents at most $O(\log n)$ contiguous segments of the total order of edges. Range add and a global minimum query can be supported in $O(\log n)$ time via an augmented binary search tree. Thus the total time complexity per operation is $O(\log^2 n)$.

We use the range operations as follows. As we iterate index i through the order of Lemma 5, we keep up to date the cost of the cut resulting from cutting any other edge e_j via the data structure of Lemma 8. Instead of querying each other edge e_j directly, however, we just use a global minimum query to find the best choice of j. The procedure is given in Algorithm 4. The first two steps are the same as Algorithm 3.

▶ **Lemma 9.** *Algorithm 4 finds the value of the minimum cut that 2-respects a spanning tree T of a graph G in $O(m \log^2 n)$ time.*

Proof. By Proposition 6, in a 2-respecting cut including e_i and e_j of T, a non-tree edge uv is cut if and only if exactly one of e_i or e_j lies on the uv-path in T. Observe that the invariants enforced in step 4 guarantee that in each iteration the total weight of edges from the cut resulting from cutting any other edge e_j along with e_i is kept up-to-date in the data structure of Lemma 8. Since the minimum such j is found for every i, it follows that step 4 finds the weight of the minimum cut of G that cuts exactly two edges of T. In step 5, we return the minimum of this weight with a single call to `QueryMinimum()` where we assume edge e_i to be off the path of all non-tree edges uv. Observe that this computes the minimum cut of G that cuts exactly one edge of T. Thus, the minimum cut of G that 2-respects T is returned in step 5.

The time complexity follows similarly to Algorithm 3. Steps 1 and 2 take $O(m \log n)$ total time. However, step 4 requires $O(\log n)$ time for non-tree edge uv whenever edge e_i

Algorithm 4 Minimum Cuts that 2-Respect T.

1. Arrange the edges of T in the order of Lemma 5; label them $e_1, \ldots, e_{n-1}$.
2. For each non-tree edge uv, mark every i such that e_i is on the uv-path in T and e_{i+1} is not on the uv-path in T, or vice versa. Indicate whether edge e_1 is on the uv-path in T.
3. Initialize the data structure of Lemma 8 on T so that the weight of edge e_j is equal to its weight in T.
4. Iterate index i from 1 to $n-1$. Via the computation done in step 2, maintain the following invariants in the data structure of Lemma 8 as i is iterated.
   a. When edge e_i is on the uv-path in T, add the weight of non-tree edge uv to all edges off the uv-path in T.
   b. When edge e_i is off the uv-path in T, add the weight of non-tree edge uv to all edges on the uv-path in T.

   Each time i is incremented, after updating weights in Lemma 8 as per 4a and 4b, add ∞ to edge e_i, execute `QueryMinimum()`, then subtract ∞ from edge e_i. The value of the minimum cut found in each iteration is the result of `QueryMinimum()` plus the weight of e_i.
5. Return the minimum of the smallest cut found in step 4 with the result of `QueryMinimum()` when we consider edge e_i to be off the path of all non-tree edges uv in the data structure of Lemma 8.

falls on or off the uv-path in T, since the data structure of Lemma 8 takes $O(\log n)$ time per operation. For a given non-tree edge uv, edge e_i falls on or off the uv-path in T a total of $O(\log n)$ times by Lemma 5; thus step 4 takes $O(m \log^2 n)$ time. The final `QueryMinimum()` call in step 5 takes $O(m \log n)$ time. The total time taken is $O(m \log^2 n)$. ◀

We make a few further remarks about Algorithm 4. To determine the edges of the minimum cut, the data structure of Lemma 8 can be augmented to return the index j of the edge that achieves the minimum given in operation `QueryMinimum()`. With e_i and e_j, we can determine the vertex partition in G of the minimum cut and, as stated in Section 4, and from this we can find which non-tree edges cross the minimum cut easily in $O(m \log n)$ time.

The space complexity of Algorithm 3 was easily linear. In Algorithm 4, we must know the identity of each non-tree edge uv in every transition point where edge e_i falls on or off the uv-path. Naively this costs $O(m \log n)$ space. This can be improved to $O(m)$ space by performing step 2 incrementally while executing step 4. That is, we only need to know the next transition point where the non-tree edge uv falls on or off the uv-path, and from the current transition point this can be determined in constant time via the heavy-light decomposition.

Recall that while Algorithm 3 helped demonstrate the approach of Algorithm 4, we need only implement Algorithm 4, since Algorithm 4 finds the minimum cut of G that cuts either 1 or 2 edges of T.

From this we get our final theorem, equivalent to the result of Karger [23].

▶ **Theorem 10.** *The minimum cut in a weighted undirected graph can be found in $O(m \log^3 n)$ time with high probability.*

Proof. We first find $\Theta(\log n)$ spanning trees by Algorithm 2. We then find the minimum cuts that 2-respect each of these trees by Algorithm 4. By Lemmas 4 and 9, this finds the minimum cut with high probability in $O(m \log^3 n)$ time. ◀

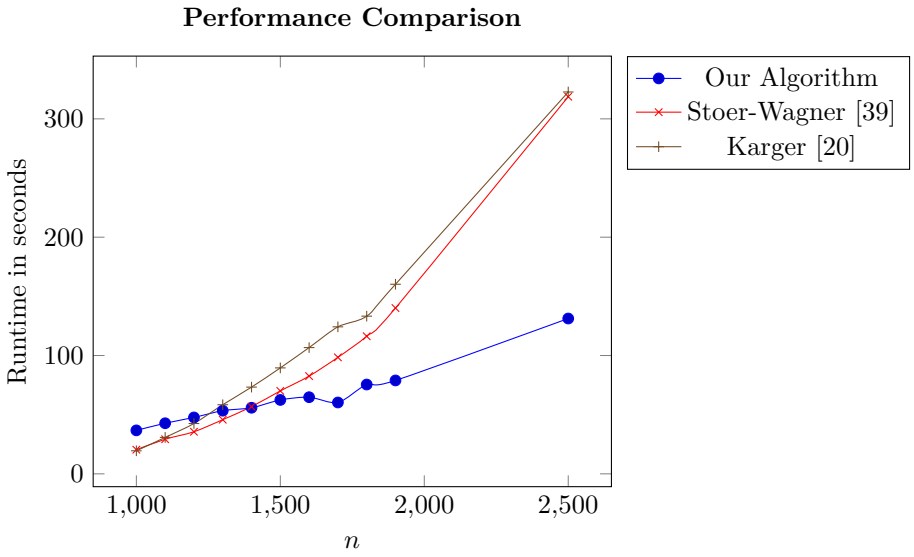

Figure 1 Performance comparison of an $O(m \log^4 n)$ implementation of our algorithm with an $O(n^3)$ Stoer-Wagner [39] and $O(n^3 \log n)$ Karger [20].

6 Implementation

We have implemented an $O(m \log^4 n)$ version of our algorithm in C++[3]. Algorithm 1 together with an $O(m \log n)$ minimum spanning tree routine take about 100 lines of code, Algorithm 2 takes about 200 lines, Algorithm 4 takes about 200 lines, and using an augmented binary search tree as the data structure for Lemma 8 takes about 200 lines. To the best of our knowledge, our implementation is the first to achieve near-linear time complexity. We have tested it against an $O(n^3)$ implementation of the Stoer-Wagner algorithm [39] and an $O(n^3 \log n)$ implementation of Karger's randomized contraction algorithm [20]. Under favorable inputs, the runtime compares as in Figure 1.

Figure 1 demonstrates the near-linear growth in the running time of our algorithm. Unfortunately, it does not appear our implementation is competitive compared to existing implementations [5]. The bottleneck is in obtaining the $O(\log n)$ spanning trees for Algorithm 4, even when Algorithm 2 runs in $O(m \log^3 n)$ time and Algorithm 4 runs in $O(m \log^4 n)$ time. The issue is the large constant factors due to the quadratic dependencies on epsilons, seen in Algorithms 5 and 6. We have calculated that the number of calls to the minimum spanning tree routine in our implementation can be as much as $8100 \ln n \ln m$, and that changing the choices of epsilons for Algorithms 1 and 2 does not yield significant improvement.

If we replace Algorithm 1 with the more-complicated Gabow's algorithm [8], we can likely improve our implementation's runtime. Further, a factor of about two can be saved by finding c' via an approximation algorithm [26]. However, a large constant factor will remain due to the sampling procedure in Lemma 14, discussed in Appendix A. All known algorithms to compute weighted tree packings have dependence on c, the value of the minimum cut, and Lemma 14 reduces the value of the minimum cut to at least $3(d+2)(\ln n)/\epsilon^2$, which in our algorithms manifests as a factor of $108(d+2) \ln n$. It appears that for Karger's approach to be made practical, this large constant factor will likely need to be improved or heuristic approaches would need to be considered [5].

[3] Our implementation is available at: `https://github.com/nalinbhardwaj/min-cut-paper`.

7 Conclusion

In this paper, we have discussed a simplification to Karger's original near-linear time minimum cut algorithm [23]. In contrast to Karger's original algorithm [23], finding spanning trees that have a constant probability of 2-respecting the minimum cut is now the more-complicated part of the algorithm and finding minimum cuts that 2-respect a tree is relatively simpler. In actuality, both were complicated in Karger's original algorithm, however the work to find the tree packing was largely abstracted to previous publications. The same can be said for many statements of Karger's near-linear time algorithm [9, 31]. Our version, on the other hand, is self-contained: the only procedures outside of Algorithms 1, 2, and 4 required to implement the full algorithm are a minimum spanning tree subroutine and (optionally) a top tree data structure.

The main contribution of our algorithm is a new, simple procedure to find a minimum cut that 2-respects a tree T in $O(m \log^2 n)$ time. Karger advertises that the complexity of his near-linear time algorithm is $O(m \log^3 n)$ and thus his routine to find a minimum cut that 2-respects a tree also takes $O(m \log^2 n)$ time. However, he gives two small improvements to the algorithm to reduce the overall runtime to $O(m \log^2 n \log(n^2/m)/ \log\log n + n \log^6 n)$. The first uses the fact that finding a 1-respecting cut can be done in linear time, and the other is an improvement which reduces an $O(\log n)$ factor to an $O(\log(n^2/m))$ factor in the 2-respect routine. For our algorithm, the first improvement can be applied by substituting our 1-respect algorithm with his. The second improvement can not be applied. Thus, when $m = \Theta(n^2)$, his algorithm is faster by an $O(\log n)$ factor. However, for this case, Karger gives a different, simpler algorithm [23] which finds the global minimum cut in $O(n^2 \log n)$ time anyway.

There are three algorithms that are referred to as simple min-cut algorithms: the Stoer-Wagner algorithm [39] which runs in $O(nm \log n)$ time or $O(nm + n^2 \log n)$ time with a Fibonacci heap [7], Karger's randomized contraction algorithm [20] which runs in $O(n^2 m \log n)$ time, and the improvement to Karger's algorithm by Karger and Stein [25] which runs in $O(n^2 \log^3 n)$ time. In comparison to these, our approach is the least simple. However, our $O(m \log^3 n)$ runtime is significantly better. While the large constant factors in our approach make this only relevant at large values of n, we hope the procedure developed in this paper can be used in conjunction with an optimized version of Karger's sampling technique to produce an asymptotically fast, practical minimum cut algorithm.

References

1 Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Trans. Algorithms*, 1(2):243–264, 2005.

2 David L. Applegate, Robert E. Bixby, Vasek Chvatal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Princeton University Press, Princeton, NJ, USA, 2007.

3 Rodrigo A. Botafogo. Cluster analysis for hypertext systems. In *Proceedings of the 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '93, pages 116–125. ACM, 1993.

4 Parinya Chalermsook, Jittat Fakcharoenphol, and Danupon Nanongkai. A deterministic near-linear time algorithm for finding minimum cuts in planar graphs. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 828–829, 2004.

5 Chandra S. Chekuri, Andrew V. Goldberg, David R. Karger, Matthew S. Levine, and Cliff Stein. Experimental study of minimum cut algorithms. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 324–333, 1997.

6 Mohit Daga, Monika Henzinger, Danupon Nanongkai, and Thatchaphol Saranurak. Distributed edge connectivity in sublinear time. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 343–354, 2019.

7 Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

8 H.N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. *Journal of Computer and System Sciences*, 50(2):259–273, 1995.

9 Paweł Gawrychowski, Shay Mozes, and Oren Weimann. Minimum cut in $O(m \log^2 n)$ time. *CoRR*, abs/1911.01145, 2019.

10 B. Geissmann and L. Gianinazzi. Cache oblivious minimum cut. In *International Conference on Algorithms and Complexity*, 2017.

11 Barbara Geissmann and Lukas Gianinazzi. Parallel minimum cuts in near-linear work and low depth. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 1–11, 2018.

12 Mohsen Ghaffari and Fabian Kuhn. Distributed minimum cut approximation. In *International Symposium on Distributed Computing*, pages 1–15, 2013.

13 Mohsen Ghaffari, Krzysztof Nowicki, and Mikkel Thorup. Faster algorithms for edge connectivity via random 2-out contractions. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms*, 2020.

14 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.

15 Gramoz Goranci, Monika Henzinger, and Mikkel Thorup. Incremental exact min-cut in polylogarithmic amortized update. *ACM Transactions on Algorithms*, 14(2), 2018.

16 J.X. Hao and J.B. Orlin. A faster algorithm for finding the minimum cut in a directed graph. *Journal of Algorithms*, 17(3):424–446, 1994.

17 Monika Henzinger, Alexander Noe, Christian Schulz, and Darren Strash. Practical minimum cut algorithms. *J. Exp. Algorithmics*, 23:1.8:1–1.8:22, 2018.

18 Monika Henzinger, Satish Rao, and Di Wang. Local flow partitioning for faster edge connectivity. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1919–1938, 2017.

19 M. Jünger, G. Rinaldi, and S. Thienel. Practical performance of efficient minimum cut algorithms. *Algorithmica*, 26:172, 2000.

20 David R. Karger. Global min-cuts in RNC, and other ramifications of a simple min-out algorithm. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, pages 21–30, 1993.

21 David R. Karger. A randomized fully polynomial time approximation scheme for the all terminal network reliability problem. In *Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing*, pages 11–17, 1995.

22 David R. Karger. Random sampling in cut, flow, and network design problems. *Mathematics of Operations Research*, 24(2), 1999.

23 David R. Karger. Minimum cuts in near-linear time. *J. ACM*, 47(1):46–76, January 2000.

24 David R. Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–328, 1995.

25 David R. Karger and Clifford Stein. A new approach to the minimum cut problem. *J. ACM*, 43(4):601–640, 1996.

26 David Ron Karger. *Random Sampling in Graph Optimization Problems*. PhD thesis, Stanford University, Stanford, CA, USA, 1995. UMI Order No. GAX95-16851.

27 Ken-Ichi Kawarabayashi and Mikkel Thorup. Deterministic edge connectivity in near-linear time. *J. ACM*, 66:4:1–4:50, 2018.

28 Antonio Molina Lovett and Bryce Sandlund. A simple algorithm for minimum cuts in near-linear time. *CoRR*, abs/1908.11829, 2019.

29 David W. Matula. A linear time $2 + \epsilon$ approximation algorithm for edge connectivity. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 500–504, 1993.

30 Antonio J. Molina Lovett and Bryce Sandlund. personal communication.

31 Sagnik Mukhopadhyay and Danupon Nanongkai. Weighted min-cut: Sequential, cut-query and streaming algorithms. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (to appear)*, 2020.

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

33 Hiroshi Nagamochi and Toshihide Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7:583–596, 1992.

34 Danupon Nanongkai and Hsin-Hao Su. Almost-tight distributed minimum cut algorithms. In *International Symposium on Distributed Computing*, 2014.

35 C. St.J. A. Nash-Williams. Edge-Disjoint Spanning Trees of Finite Graphs. *Journal of the London Mathematical Society*, s1-36(1):445–450, 1961.

36 S. A. Plotkin, D. B. Shmoys, and E. Tardos. Fast approximation algorithms for fractional packing and covering problems. In *Proceedings 32nd Annual Symposium of Foundations of Computer Science*, pages 495–504, 1991.

37 Aparna Ramanathan and Charles J. Colbourn. Counting almost minimum cutsets with reliability applications. *Mathematical Programming*, 39:253–261, 1987.

38 Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

39 Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *J. ACM*, 44(4):585–591, 1997.

40 Robert E. Tarjan and Renato F. Werneck. Self-adjusting top trees. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 813–822, 2005.

41 Mikkel Thorup. Fully-dynamic min-cut*. *Combinatorica*, 27(1):91–127, 2007.

42 Mikkel Thorup and David R. Karger. Dynamic graph algorithms with applications. In *Proceedings of the 7th annual scandanavian symposium and workshops on algorithm theory*, 2000.

43 Neal E. Young. Randomized rounding without solving the linear program. In *Proceedings of the 6th Annual Symposium on Discrete Algorithms.*, 1995.

A Karger's Algorithm for Packing Spanning Trees

In this section we give the intuition and mathematics behind the spanning tree packing of Karger's algorithm.

A.1 Tree Packing

The basic idea of Karger's near-linear time algorithm [23] is to exploit the following combinatorial result. Recall that a tree packing of an undirected unweighted graph G is a set of spanning trees such that each edge of G is contained in at most one spanning tree. The weight of a tree packing is the number of trees in it.

▶ **Theorem 11** (Nash-Williams [35]). *Any undirected unweighted multigraph with minimum cut c contains a tree packing of weight at least $c/2$.*

Now consider a minimum cut and a tree packing given by Theorem 11. Each edge of the minimum cut can only be present in at most one spanning tree. As there are c edges of the minimum cut, this implies that the average spanning tree contains at most $c/(c/2) = 2$ edges of the minimum cut. In other words, a spanning tree chosen at random from a packing of Theorem 11 will 2-constrain the minimum cut with probability at least $1/2$.

Suppose we are given a spanning tree T of G with each edge of T marked if it crosses the minimum cut. The endpoints of any marked edge must fall on opposite sides of the cut. Conversely, the endpoints of any unmarked edge must be on the same side of the cut. It follows that if we know the edges of T that cross the minimum cut, we can determine the vertex partition of the minimum cut and its total weight in G.

This gives the intuition behind Karger's algorithm [23]. We sample spanning trees from a tree packing of G and for each tree T, we find the minimum cut that 2-respects T. Unfortunately, several obstacles need be overcome before this can be made into an efficient algorithm. For one, all currently known approaches of determining a tree packing of Theorem 11 have runtime $\Omega(cm)$, which for large values of c is far more than the runtime we seek. Further, Theorem 11 must be generalized to weighted graphs.

We first address the latter concern. Recall the definition of weighted tree packings given in Section 3.

▶ **Lemma 12** (Karger [23]). *Any undirected weighted graph with minimum cut c contains a weighted tree packing of weight at least $c/2$.*

Proof. For contradiction, suppose some graph G with minimum cut c and $\epsilon > 0$ exist such that G does not contain a weighted packing of weight $(1-\epsilon)c/2$ or greater.

Take G and approximate each edge e_i of weight w_i by a rational number a_i/b_i such that $a_i/b_i < w_i$ and $w_i - a_i/b_i < \epsilon$. Multiply all edges by $d = \prod_i b_i$ and call the resulting graph G'. Then by Theorem 11, when viewed as an unweighted multigraph, G' has a tree packing of weight at least $(1-\epsilon)dc/2$. If we weight each tree of the packing by $1/d$, the packing becomes a weighted packing of G of weight at least $(1-\epsilon)c/2$, a contradiction. ◀

Note that for both Lemma 11 and Lemma 12, an upper bound of weight c also exists, because every spanning tree in the packing must cross the minimum cut at least once.

To effectively use Lemma 12, we formally state the relationship between weighted packings and trees that 2-constrain small cuts.

▶ **Lemma 13** (Karger [23]). *Consider a weighted graph G and a weighted tree packing of weight βc, where c is the weight of the minimum cut in G. Then given a cut of weight αc, a fraction at least $\frac{1}{2}(3-\alpha/\beta)$ of the trees (by weight) 2-constrain the cut.*

Proof. Note that every spanning tree must cross every cut. Let x denote the total weight of trees with at least three edges crossing the cut and y the total weight of trees with one or two edges crossing the cut. Then $x + y = \beta c$ and $3x + y \leq \alpha c$. Rearranging, we get $y \geq \frac{1}{2}(3\beta c - \alpha c)$. ◀

A.2 Random Sampling

In order to avoid the $\Omega(cm)$ complexity of finding a packing of weight $c/2$, we first apply random sampling to G. Specifically, we use the following from Karger's earlier work.

▶ **Lemma 14** (Karger [22]). *Let $p = 3(d+2)(\ln n)/(\epsilon^2\gamma c) \leq 1$, where c is the weight of the minimum cut of an unweighted multigraph G and $\gamma \leq 1, \gamma = \Theta(1)$. Then if we sample each edge of G independently with probability p, the resulting graph H has the following properties with probability $1 - 1/n^d$.*

1. *The minimum cut in H is of size within a $(1+\epsilon)$ factor of $cp = 3(d+2)(\ln n)/(\gamma\epsilon^2)$, which is $O(\epsilon^{-2}\log n)$.*

2. *A cut in G takes value within a factor $(1+\epsilon)$ of its expected value in H. In particular, the minimum cut in G corresponds (under the same vertex partition) to a $(1+\epsilon)$-times minimum cut of H.*

By picking ϵ to be a constant such as $1/6$, Lemma 14 will allow us to reduce the size of the minimum cut in H to $O(\log n)$. We can then run existing algorithms [36, 8] to pack trees in H in $\tilde{O}(m)$ time. Further, since the minimum cut of G corresponds to a $(1+\epsilon)$-times minimum cut of H, we can still apply Lemma 13 on the sampled graph H so that a tree randomly sampled from the packing has a constant probability of 2-constraining the minimum cut in G.

There are still several issues to resolve. Lemma 14 applies to unweighted multigraphs G, but our graph G can have non-negative real weights. The other issue is that the value γ needs to be known ahead of time in order to apply the lemma. We first address the latter issue.

Lemma 14 requires knowing a constant-factor underestimate $c' = \gamma c$ for the minimum cut c. In particular, without $\gamma \leq 1$, property 2 of Lemma 14 is not guaranteed with high probability, and if $\gamma = o(1)$, the minimum cut of H will be of size $\omega(\epsilon^{-2} \log n)$ with high probability. We may run a linear-time 3-approximation algorithm [29], with modifications to work on weighted graphs [26], to find this approximation. This is simple to state, but more difficult to implement.

A different approach is to start with a known upper bound U for c'. Karger states that we can then halve this upper bound until "our algorithms succeed" [22]. This approach is taken by the implementation of Chekuri et al. [5]. Unfortunately, it is not rigorous as stated. Lemma 14 indicates that with a constant-factor underestimate $c' = \gamma c$ for c, our algorithm can proceed. However, it does not give a process for rejecting a guess c' that is not a constant-factor underestimate for c. We could try all powers of 2 for c' within a known lower and upper bound of the value of the minimum cut, and run our algorithms for all possibilities. This is rigorous, but introduces an extra $O(\log n)$ factor in our runtimes, assuming the range of c' we try is polynomial in n. We instead show the following.

▶ **Lemma 15.** *Let $p = 3(d+2)(\ln n)/(\epsilon^2 \gamma c) \leq 1$ as in Lemma 14, but with $\gamma \geq 6$ and $\epsilon \leq 1/3$. Then if we sample each edge of the unweighted multigraph G uniformly at random with probability p, the resulting graph H has minimum cut of size less than $(d+2)(\ln n)/\epsilon^2$ with probability at least $1 - 1/n^{d+2}$.*

Proof. Consider the size of a minimum cut of G as a cut in H. Let X be a random variable denoting this size. Then $\mathbb{E}[X] = cp$. By a Chernoff bound, $\Pr[X \geq (1+\delta)cp] \leq e^{-\frac{1}{3}(cp\delta)}$ for $\delta \geq 1$. Let $(1+\delta) = \frac{\gamma}{3}$. Then

$$\begin{aligned}
\Pr\left[X \geq (d+2)(\ln n)/\epsilon^2\right] &\leq e^{-\frac{1}{3}(cp(\frac{\gamma}{3}-1))} \\
&= e^{-(d+2)(\ln n)\gamma^{-1}\epsilon^{-2}(\frac{\gamma}{3}-1))} \\
&= n^{-\frac{1}{3}(d+2)\epsilon^{-2}+(d+2)\gamma^{-1}\epsilon^{-2}} \\
&\leq n^{-\frac{1}{6}(d+2)\epsilon^{-2}} \\
&< n^{-(d+2)}.
\end{aligned}$$

Therefore, the minimum cut in H has size less than $(d+2)(\ln n)/\epsilon^{-2}$ with probability at least $1 - 1/n^{d+2}$. ◀

Lemma 15 states that if our estimate $c' = \gamma c$ satisfies $\gamma \geq 6$, the minimum cut will be at least a factor 3 smaller than $3(d+2)(\ln n)/\epsilon^2$ with high probability. Recall that with $\gamma = 1$ and therefore $c' = c$, we expect the minimum cut in H to be within a factor $(1+\epsilon)$ from

$3(d+2)(\ln n)/\epsilon^2$ with high probability. Lemma 15 gives us the necessary tool to reject c' that are not a constant factor underestimate of c. We try a value for c', and if the size of the minimum cut in H is greater than $(1+\epsilon)^{-1}3(d+2)(\ln n)/\epsilon^2$, we know $c' < 6c$. Therefore we can decrease c' by a factor of 6 and rerun the tree packing algorithm. The resulting graph H must satisfy the conditions of Lemma 14, therefore the algorithm may proceed. Since our tree packing algorithms determine the minimum cut up to constant factors, this approach avoids the need of a different (or recursive!) minimum cut algorithm to run on H.

We briefly remark on the choice of known upper bound U. If the edge weights are polynomially bounded by the number of vertices, n, a simple upper bound of the sum of weights of edges attached to any single vertex will do. If we do not consider this guarantee, Karger shows [22] that the minimum weight edge w in a maximum spanning tree has the property that the minimum cut must have weight between w and n^2w. Thus, setting $U = n^2w$ gives only $O(\log n)$ values of c' to try regardless of edge weights. The choice of an upper bound U is further discussed in [5].

We now return to the issue of real-value weights in Lemma 14. This was described as a complication in [5], to which they substituted a heuristic method in order to achieve practicality. The approach we have described thus far is amenable to small constant-factor approximations. Suppose we replace G with a graph G' such that each edge weight is first normalized so the smallest weight edge has weight 1, then all edge weights are multiplied by 100 and rounded to the nearest integer. Normalizing has no effect on the relative sizes of cuts in G'. Rounding to the nearest integer when the smallest weight edge has weight at least 100 has the effect that a cut of weight x will take on a new weight in range $[.995x, 1.005x]$. Then the original minimum cut of G corresponds to an at most 201/199-times minimum cut of G'. Now, G' can be represented as an unweighted multigraph and then sampled according to Lemma 14. In the resulting graph H, the minimum cut of G corresponds to an at most $201/199 \cdot 7/6$-times minimum cut of H with the choice $\epsilon = 1/6$. By adjusting constants throughout the rest of our approach, this shows we can treat real weighted graphs G correctly. The other issue is how to do so efficiently.

If we consider G' as an unweighted multigraph, the number of edges of G' is proportional to the weight of edges of G, which may be quite large. However, we may also consider G' as an integer-weighted graph, in which case we can sample each edge of G' by drawing from the binomial distribution with probability p and number of trials the weight of the respective edge. There are many methods to sample from the binomial distribution. One simple method that can be made efficient for our purposes is inverse transform sampling. Let X denote a random variable sampled from the binomial distribution as described. In inverse transform sampling, we draw a number u uniformly at random between 0 and 1, and then choose our sample x to be the largest such that $P(X < x) \leq u$. Instead of having to sample a number of times equal to the weight of an edge, we must only compute the probabilities of the cumulative distribution function for the binomial distribution for all possible values that may result in H. We can make this efficient with the following observation. Say the weight of the minimum cut in H is $\hat{c}$. Then a tree packing of H has value at most $\hat{c}$, and in particular for a given edge, any weight beyond $\hat{c}$ is excess capacity that cannot be used in the tree packing. It follows that capping the weight of any edge of H to the maximum size of the minimum cut in H, thus $O(\log n)$, will have no impact on the packing found. Thus, we must only compute $O(\log n)$ probabilities of the binomial distribution per edge, which can be done in total $O(\log n)$ time per edge.

The final choice is to pick a tree packing algorithm. Karger gives two options. The first is an algorithm by Gabow [8], which computes a $c/2$ packing. The second is a more general approach by Plotkin-Shmoys-Tardos [36], which can find a packing a factor $(1+\epsilon')$ from

the maximum packing, which has value in $[c/2, c]$. Karger describes the latter approach as simpler, using only minimum spanning tree computations. Although the paper [36] does not explicitly give a routine for packing spanning trees, such a procedure is explicitly given in Thorup and Karger [42], with credit given to Plotkin-Shmoys-Tardos [36] and Young [43]. This procedure also appears in Gawrychowski et al. [9]. We give the procedure in Algorithm 1 and state a version of Algorithm 1 with general epsilon in Algorithm 5.

Algorithm 5 Obtain a Packing of Weight at least $(1-\epsilon)c/2$ from a Graph G.

Let G be a graph with m edges and n vertices.

1. Initialize $\ell(e) \leftarrow 0$ for all edges e of G. Initialize multiset $P \leftarrow \emptyset$. Initialize $W \leftarrow 0$.
2. Repeat the following:
   a. Find a minimum spanning tree T with respect to $\ell(\cdot)$.
   b. Set $\ell(e) \leftarrow \ell(e) + \epsilon^2/(3 \ln m)$ for all $e \in T$. If $\ell(e) > 1$, return W, P.
   c. Set $W \leftarrow W + \epsilon^2/(3 \ln m)$.
   d. Add T to P.

We now give the general form of Lemma 3 with proof.

▶ **Lemma 16** ([36, 42, 43]). *Given $0 < \epsilon < 1$ and an undirected unweighted graph G with m edges, n vertices, and minimum cut c, Algorithm 5 returns a weighted packing of weight at least $(1-\epsilon)c/2$ in $O(mc \log n)$ time.*

Proof. On each iteration, the weight of some tree is increased by $\epsilon^2/(3 \ln m)$. Since the weight of the resulting packing is bounded by c, there are at most $3c \ln m/\epsilon^2 = O(c \log n)$ iterations. The bottleneck in each iteration is the time to compute a minimum spanning tree in G. With an $O(m)$ time minimum spanning tree algorithm [24] our final time complexity is $O(mc \log n)$; an alternative way to achieve this runtime when Algorithm 5 is used in Algorithm 6 was shown in Section 3. Correctness is given via Thorup and Karger [42], Young [43], and Plotkin-Shmoys-Tardos [36]. ◀

Our full procedure for obtaining $\Theta(\log n)$ spanning trees for the rest of the algorithm is given in Algorithm 2. We give a version of Algorithm 2 with general epsilons in Algorithm 6.

We give the generalization of Lemma 4 for Algorithm 6 below.

▶ **Lemma 17.** *Algorithm 6 returns a collection of $\Theta(\log n)$ spanning trees of G in time $O(m \log^3 n)$ such that the minimum cut of G 2-respects at least one tree in the collection with high probability.*

Proof. We first prove correctness. Consider general epsilons $\epsilon_1, \epsilon_2, \epsilon_3 > 0$, where in Algorithm 2, $\epsilon_1 = 1/100$ is the real-weight approximation, $\epsilon_2 = 1/6$ is the approximation for Lemmas 14 and 15, and $\epsilon_3 = 1/5$ is the approximation for Algorithm 1 to return a packing of size $(1-\epsilon_3)c/2$ or greater.

Suppose for a particular c' that $c' \geq 6c$, where c is the size of the minimum cut in G'. Then by Lemma 15, H will have minimum cut of size less than $b/3 = (d+2) \ln n/{\epsilon_2}^2$ with high probability. A maximum tree packing of H will have weight at most $\hat{c}$, the weight of the minimum cut in H, and thus the weight of the tree packing found by Algorithm 5 will be at most $b/3 < \frac{1}{2}(1-\epsilon_3)(1+\epsilon_2)^{-1}b$ because $(1+\epsilon_2)^{-1}(1-\epsilon_3) > 2/3$. Therefore Algorithm 6 will proceed to the next iteration with $c' \leftarrow c'/2$. Note that the overall probability of failure from any of the $O(\log n)$ iterations of this step is at most $O(\log n \cdot n^{-(d+2)}) \leq n^{-d}$ for sufficiently large n.

Algorithm 6 Obtain $\Theta(\log n)$ Spanning Trees for the 2-respect Algorithm.

Let d denote the exponent in the probability of success $1 - 1/n^d$. Let $\epsilon_1, \epsilon_2, \epsilon_3 > 0$ be constants of approximation such that $f = 3/2 - (\frac{2+\epsilon_1}{2-\epsilon_1})(1+\epsilon_2)(1-\epsilon_3)^{-1} > 0$ and $(1+\epsilon_2)^{-1}(1-\epsilon_3) > 2/3$. Let $b = 3(d+2)\ln n/{\epsilon_2}^2$.

1. Form graph G' from G by first normalizing the edge weights of G so the smallest non-zero edge weight has weight 1, then multiplying each edge weight by ϵ_1^{-1} and rounding to the nearest integer. Let U be an upper bound for the size of the minimum cut of G'.
2. Initialize $c' \leftarrow U$. Repeat the following:
   a. Construct H in the following way: for each edge e of G', let e have weight in H drawn from the binomial distribution with probability $p = \min(b/c', 1)$ and number of trials the weight of e in G'. Cap the weight of any edge in H to at most $\lceil(1+\epsilon_2)12b\rceil$.
   b. Run Algorithm 5 on H with approximation ϵ_3, considering an edge of weight w as w parallel edges. There are three cases:
      i. If $p = 1$, set P to the packing returned and skip to step 3.
      ii. If the returned packing is of weight $\frac{1}{2}(1-\epsilon_3)(1+\epsilon_2)^{-1}b$ or greater, set $c' \leftarrow c'/6$ and repeat steps 2a and 2b, setting P to the packing returned and then proceeding to step 3.
      iii. Otherwise, repeat steps 2a and 2b with $c' \leftarrow c'/2$.
3. Return $\lceil -d\ln n/\ln(1-f)\rceil$ trees sampled uniformly at random proportional to their weights from P.

Now suppose Algorithm 5 returns a tree packing of weight $\frac{1}{2}(1-\epsilon_3)(1+\epsilon_2)^{-1}b$ or greater. By the above, $c' < 6c$ with high probability. If $c' \le c$, Lemma 14 says that the weight of the minimum cut is at least $(1+\epsilon_2)^{-1}b$ with high probability, unless $p > 1$. In the latter case, this implies the weight of the minimum cut is $O(\log n)$ and there is no need to apply sampling to G'. Consider the former case. The tree packing is of weight at least $(1-\epsilon_3)$ times half the minimum cut. It follows that the tree packing will be of weight at least $\frac{1}{2}(1-\epsilon_3)(1+\epsilon_2)^{-1}b$. The consequence of this is that if a tree packing of this weight or greater is found in step 2b, in addition to the bound $c' < 6c$, we also know $c' > c/2$ with high probability, since whenever $c' \le c$, Lemma 14 says the packing will have weight at least $\frac{1}{2}(1-\epsilon_3)(1+\epsilon_2)^{-1}b$, and we decrease c' by a factor of 2 in each iteration. Therefore, if we set $c' \leftarrow c'/6$, then in the next iteration we will have $c/12 < c' < c$.

Now consider the next iteration when the tree packing is returned. In sampling H, we only preserve weights in H up to $\lceil(1+\epsilon_2)\cdot 12b\rceil$. Since $c' > c/12$, the expected size of the minimum cut in H is at most $12b = 12\cdot 3(d+2)\ln n/{\epsilon_2}^2$. Thus, with high probability, by Lemma 14, the size of the minimum cut in H is at most $(1+\epsilon_2)12b$, and as explained previously, we can afford to remove the capacity of any edge beyond $(1+\epsilon_2)12b$ without impacting the returned packing. Now by Lemma 13 with $\alpha \le \frac{2+\epsilon_1}{2-\epsilon_1}(1+\epsilon_2)$ and $\beta \ge \frac{1}{2}(1-\epsilon_3)$, a fraction of at least $f = 3/2 - (\frac{2+\epsilon_1}{2-\epsilon_1})(1+\epsilon_2)(1-\epsilon_3)^{-1}$ of the trees in the packing found will 2-constrain the minimum cut of G. The probability that no tree in a sample of size t 2-constrains the minimum cut is $(1-f)^t$. Solving for t in $(1-f)^t = n^{-d}$ yields $t = -d\ln n/\ln(1-f)$. Therefore with probability at least $1 - 1/n^d$, at least one tree in the returned sample will 2-constrain the minimum cut.

Time complexity can be proven as follows. Sampling H can be done in $O(m\log n)$ time, as explained previously. Algorithm 5 runs in $O(m'\hat{c}\log^2 n)$ time using a textbook $O(m\log n)$ minimum spanning tree algorithm, where $\hat{c}$ is the value of the minimum cut in H and m' is

the number of edges in H, where weighted edges are considered parallel unit weight edges. Due to the sampling procedure, $m' = O(m \log n)$. To reduce this complexity, we can either use a linear time minimum spanning tree algorithm [24] or the implementation trick given in Section 3. If we use the latter, we reduce the effective m' needed in Algorithm 5 to $O(m)$. Further, in expectation, the value of the minimum cut $\hat{c}$ of H doubles in each iteration of Algorithm 6. A high probability statement can be made via an argument similar to Lemma 15. Therefore the cost of running Algorithm 5 doubles in each iteration, with the final cost being $O(m \log^3 n)$, since $\hat{c} = O(\log n)$ by Lemma 14. This is a geometric series, so the entire cost is $O(m \log^3 n)$, and so Algorithm 6 runs in $O(m \log^3 n)$ time with high probability. ◀

Since Algorithm 5 returns $O(\log n)$ trees, we could avoid sampling trees from the weighted packing and instead return all of them. We keep the sampling in Algorithm 6 because, depending on the constants, sampling may require less trees. Further, the above version of Algorithm 5 is more versatile in that the packing algorithm can be changed. Observe that the entire algorithm is still only correct with high probability, since we required sampling G' to construct graph H. Finally, returning all trees from Algorithm 5 does not actually allow us to relax ϵ_1, ϵ_2, or ϵ_3. The condition $f = 3/2 - (1 + \epsilon_1)(1 + \epsilon_2)(1 - \epsilon_3)^{-1} > 0$ is satisfied for all values of α and β that guarantee at least one tree in a weighted packing of weight βc 2-constrains a cut of weight αc given by Lemma 13.

Algorithm 6 is slightly different than the approach taken by Karger [23]. In particular, Karger sparsifies edges of H to have $m' = O(n \log n)$ and replaces an $O(m \log n)$ time minimum spanning tree computation in the tree packing algorithm with an $O(m)$ one, avoiding the implementation trick of Gawrychowski et al. [9]. This gives complexity $O(n \log^3 n)$ for finding the $\Theta(\log n)$ spanning trees. However, since the remaining part of the algorithm also takes $O(m \log^3 n)$ time, we avoid these optimizations to simplify our procedures.