Minimum Cut in $O(m \log^2 n)$ Time

Paweł Gawrychowski · Shay Mozes · Oren Weimann

Abstract We give a randomized algorithm that finds a minimum cut in an undirected weighted m-edge n-vertex graph G with high probability in $O(m \log^2 n)$ time. This is the first improvement to Karger's celebrated $O(m \log^3 n)$ time algorithm from 1996. Our main technical contribution is a deterministic $O(m \log n)$ time algorithm that, given a spanning tree T of G, finds a minimum cut of G that 2-respects (cuts two edges of) T.

1 Introduction

The minimum cut problem is one of the most fundamental and well-studied optimization problems in theoretical computer science. Given an undirected edge-weighted graph $G = (V, E)$, the problem asks to find a subset of vertices S such that the total weight of all edges between S and $V \setminus S$ is minimized. The vast literature on the minimum cut problem can be classified into three main approaches:

The maximum-flow approach. The minimum cut problem was originally solved by computing the maximum st-flow [6] for all pairs of vertices s and t. In 1961, Gomory and Hu [14] showed that only $O(n)$ maximum st-flow computations are required, and in 1994 Hao and Orlin [15] showed that in fact a single maximum st-flow computation suffices. A maximum st-flow can be found in $O(mn \log(n^2/m))$ time using the Goldberg-Tarjan algorithm [13], and the fastest algorithm to date takes $O(mn)$ time [27,37]. Faster maximum st-flow algorithms are known (see e.g [5,12,30,31,41] and references within) when the graph is unweighted or when the maximum edge weight W is not extremely large. However, even with a linear $O(m)$ time maximum st-flow algorithm, computing the maximum flow for all pairs of vertices takes $O(mn)$ time. Very recently [1], it was shown how to break this barrier and compute the maximum flow for all pairs of vertices in $\tilde{O}(n^2)$ time.

The edge-contraction approach. An alternative method is edge contraction. If we can identify an edge that does not cross the minimum cut, then we can contract this edge without affecting the minimum cut. Nagamochi and Ibaraki [34,35] showed how to deterministically find a contractible edge in $O(m)$ time, leading to an $O(mn + n^2 \log n)$-time minimum cut algorithm. Karger [19] showed that randomly choosing the edge to contract works well with high probability. In particular, Karger and Stein [25] showed that this leads to an improved $O(n^2 \log^3 n)$ Monte Carlo algorithm.

The tree-packing approach. In 1961, Nash-Williams [36] proved that, in unweighted graphs, any graph with minimum cut c contains a set of $c/2$ edge-disjoint spanning trees. Gabow's algorithm [7] can be used to find such a tree-packing with $c/2$ trees in $O(mc \log n)$ time. Karger [23] observed that the c edges of a minimum cut must be partitioned among these $c/2$ spanning trees, hence the minimum cut *1-* or *2-respects* some tree in the packing. That is, one of the trees is such that at most two of its edges cross the minimum cut (these edges are said to *determine* the cut). We can therefore find the minimum cut by examining each tree and finding the minimum cut that 1- or 2-respects it.

Paweł Gawrychowski
University of Wrocław, Poland
E-mail: gawry@cs.uni.wroc.pl

Shay Mozes
The Interdisciplinary Center Herzliya, Israel
E-mail: smozes@idc.ac.il
Supported in part by Israel Science Foundation grant 810/21.

Oren Weimann
University of Haifa, Israel
E-mail: oren@cs.haifa.ac.il
Supported in part by Israel Science Foundation grant 810/21.

Several obstacles need to be overcome in order to translate this idea into an efficient minimum cut algorithm for weighted graphs: (1) we need a weighted version of tree-packing, (2) finding the packing (even in unweighted graphs) takes time proportional to c (and c may be large), (3) checking all trees takes time proportional to c, and (4) one needs an efficient algorithm that, given a spanning tree T of G, finds the minimum cut in G that 2-respects T (finding a minimum cut that 1-respects T can be easily done in $O(m+n)$ time, see e.g [23, Lemma 5.1]).

In a seminal work, Karger [23] overcame all four obstacles: First, he converts G into an unweighted graph by conceptually replacing an edge of weight w by w parallel edges. Then, he uses his random sampling from [19,21] combined with Gabow's algorithm [7] to reduce the packing time to $O(m+n\log^3 n)$ and the number of trees in the packing to $O(\log n)$. Finally, he designs a deterministic $O(m\log^2 n)$ time algorithm that given a spanning tree T of G finds the minimum cut in G that 2-respects T. Together, this gives an $O(m\log^3 n)$ time randomized algorithm for minimum cut. Karger also showed how to improve the running time by a $\log\log n$ factor, at the cost of an additive $O(n\log^6 n)$ factor by observing that 1-respecting cuts can be found faster than 2-respecting cuts, and tweaking the parameters of the tree packing.

Karger's $O(m\log^2 n)$ algorithm for the 2-respecting problem finds, for each edge $e \in T$, the edge $e' \in T$ that minimizes the cut determined by $\{e, e'\}$. He used link-cut trees [39] to efficiently keep track of the sizes of cuts as the candidate edges e of T are processed in a certain order (bough decomposition), consisting of $O(\log n)$ iterations, and guarantees that the number of dynamic tree operations is $O(m)$ per iteration. Since each link-cut tree operation takes $O(\log n)$ time, the total running time for solving the 2-respecting problem is $O(m\log^2 n)$.

In a very recent paper, Lovett and Sandlund [3] proposed a simplified version of Karger's algorithm. Their algorithm has the same $O(m\log^3 n)$ running time as Karger's. To solve the 2-respecting problem they use top trees [2] rather than link-cut trees, and use heavy path decomposition [16,39] to guide the order in which edges of T are processed. A property of heavy path decomposition is that, for every edge $(u,v) \notin T$, the u-to-v path in T intersects $O(\log n)$ paths of the decomposition. This property implies that the contribution of each non-tree edge to the cut changes $O(\log n)$ times along the entire process. See also [10] who use the fact that the bough decomposition, implicitly used by Karger, also satisfies the above property. The idea of traversing a tree according to a heavy path decomposition, i.e., by first processing a smaller subtree and then processing the larger subtree has been used quite a few times in similar problems on trees. See e.g., [4]. While the ideas of Lovett and Sandlund [3] do not improve on Karger's bound, their paper has drawn our attention to this problem.

1.1 Our result and techniques

In this paper, we present a deterministic $O(m\log n)$ time algorithm that, given a spanning tree T of G, finds the minimum cut in G that 2-respects T. Using Karger's framework, this implies an $O(m\log^2 n)$ time randomized algorithm for minimum cut in weighted graphs. We also show (see Section 4) that Karger's $\log\log n$ speedup approach can be used to obtain an $O(m\log^2 n/\log\log n + n\log^{3+\delta} n)$ time algorithm for any $\delta > 0$.

Like prior algorithms for this problem, our algorithm finds, for each edge $e \in T$ the edge $e' \in T$ that minimizes the cut determined by $\{e, e'\}$. The difficult case to handle is when e and e' are such that neither of them is an ancestor of the other. In Karger's solution, handling each edge $e = (u,v) \in T$ was done using amortized $O(d\log n)$ operations on Sleator and Tarjan's link-cut trees [39] where d is the number of non-tree edges incident to u. Since operations on link-cut trees require $O(\log n)$ amortized time, the time to handle all edges is $O(m\log^2 n)$ (implying an $O(m\log^3 n)$ time algorithm for the minimum cut problem). As an open problem, Karger [23] asked (more than 20 years ago) whether the required link-cut tree operations can be done in constant amortized time per operation (implying an $O(m\log^2 n)$ time algorithm for the minimum cut problem). Karger even pointed out why one could perhaps achieve this: "*We are not using the full power of dynamic trees (in particular, the tree we are operating on is static, and the sequence of operations is known in advance).*" In this paper, we manage to achieve exactly that. We show how to order the link cut tree operations so that they can be handled efficiently in batches. We call such a batch a *bipartite problem* (see Definition 4).

Perhaps a reason that the running time of Karger's algorithm has not been improved in more than two decades is that it is not at all apparent that these bipartite problems can indeed be solved more

efficiently. Coming up with an efficient solution to the bipartite problem requires a combination of several additional ideas. Like [3], we use heavy path decomposition, but in a different way. We develop a new decomposition of a tree that combines heavy path decomposition with biased divide and conquer, and use this decomposition in conjunction with a compact representation which we call topologically induced subtrees (see Definition 3). This compact representation turns out to be crucial not only for solving the bipartite problem, but also to the reduction from the original problem to a collection of bipartite problems.

1.2 Application to unweighted graphs

Karger's method is inherently randomized and obtaining a deterministic (or at least Las Vegas) near-linear time algorithm for the minimum cut in undirected weighted graphs is an interesting problem that has been solved very recently [17] (the exact exponent in the polylog is not stated in [17]). For *simple* unweighted undirected graphs, a deterministic near-linear time algorithm was provided by Kawarabayashi and Thorup [26]. Later, Henzinger, Rao, and Wang [18] designed a faster $O(m \log^2 n (\log \log n)^2)$ time algorithm. Ghaffari, Nowicki and Thorup [11] introduced a new technique of random 2-out contractions and applied it to design an $O(\min\{m+n \log^3 n, m \log n\})$ time randomized algorithm that finds a minimum cut with high probability. We stress that the faster algorithms of Henzinger et al. and Ghaffari et al. work only for simple unweighted graphs, that is, for edge connectivity. Interestingly, the latter uses Karger's $O(m \log^3 n)$ time algorithm as a black box, and by plugging in our faster method one immediately obtains an improved running time of $O(\min\{m + n \log^2 n, m \log n\})$ for simple unweighted graphs.

1.3 Independent work and summary of state-of-the-art

Independently to our work[1], Mukhopadhyay and Nanongkai [33] came up with an $O(m \log n + n \log^4 n)$ time algorithm for finding a minimum 2-respecting cut (see also a simplified and faster $O(m \log n + n \log^2 n)$ implementation in [9]). While this improves Karger's bound for sufficiently dense graphs, it does not improve it for all graphs, and is randomized. Our algorithm uses a different (deterministic and simple) approach. There are however benefits to the approach of [33]; It can be improved [9] to obtain an $O(m \log n + n^{1+\epsilon})$ time min-cut algorithm in weighted graphs for any fixed $\epsilon > 0$, which is the state of the art for dense graphs with $m = \Omega(n^{1+\epsilon})$. Furthermore, it can be used to obtain an algorithm that requires $\tilde{O}(n)$ cut queries to compute the min-cut, and a streaming algorithm that requires $\tilde{O}(n)$ space and $O(\log n)$ passes to compute the min-cut.

The following table summarizes the state-of-the-art for randomized algorithms for minimum cut in undirected weighted graphs.

$O(m \log^2 n)$	Here
$O(m \log^2 n / \log \log n + n \log^{3+\delta} n)$	Here
$O(m \log n + n^{1+\epsilon})$	[9]

2 Preliminaries

2.1 Karger's algorithm

At a high level, Karger's algorithm [23] has two main steps. The input is a weighted undirected graph G. The first step produces a set $\{T_1, \ldots, T_s\}$ of $s = O(\log n)$ spanning trees such that, with high probability, the minimum cut of G 1- or 2-respects at least one of them. The second step deterministically computes for each T_i the minimum cut in G that 1-respects T_i and the minimum cut in G that 2-respects T_i. The minimum cut in G is the minimum among the cuts found in the second step.

Karger shows [23, Theorem 4.1] that producing the trees $\{T_1, \ldots, T_s\}$ in the first step can be done in $O(m + n \log^3 n)$ time, and that finding the minimum 2-respecting cut for all the T_i's in the second step can be done in $O(m \log^3 n)$ time. We will show that each of the steps can be implemented in $O(m \log^2 n)$

[1] To be accurate, their work appeared on arXiv one day after ours.

time. Showing this for the second step is the main result of the paper, and is presented in Section 3. For the first step, we essentially use Karger's proof. Since the first step was not the bottleneck in Karger's paper, proving a bound of $O(m + n \log^3 n)$ was sufficient for his purposes. Karger's concluding remarks suggest that he knew that the first step could be implemented in $O(m \log^2 n)$ time. For completeness, we prove the $O(m \log^2 n)$ bound by slightly modifying Karger's arguments and addressing a few issues that were not important in his proof. Readers proficient with Karger's algorithm can safely skip the proof.

Definition 1 (2-respecting and 2-constraining) Given a spanning tree T and a cut $(S, \bar{S})$, we say that the cut 2-respects T and that T 2-constrains the cut if at most 2 edges of T cross the cut.

Definition 2 (weighted tree packing) Let G be an unweighted undirected graph. Let $\mathcal{T}$ be a set of spanning trees of G, where each tree $T \in \mathcal{T}$ is assigned a weight $w(T)$. We say that the *load* of an edge e of G (w.r.t. $\mathcal{T}$) is $\ell(e) = \sum_{T \in \mathcal{T}: e \in T} w(T)$. We say that $\mathcal{T}$ is a *weighted tree packing* if no edge has load exceeding 1. The *weight* of the packing $\mathcal{T}$ is $\tau = \sum_{T \in \mathcal{T}} w(T)$.

Theorem 1 *Given a weighted undirected graph G, in $O(m \log^2 n)$ time, we can construct a set $\mathcal{T}$ of $O(\log n)$ spanning trees such that, with high probability, the minimum cut 2-respects at least one of the trees in $\mathcal{T}$.*

Proof Let $(S, \bar{S})$ be the partition of the vertices of G that forms a minimum cut, and let c be the weight of the minimum cut in G. We precompute a constant factor approximation of c in $O(m \log^2 n)$ time using Matula's algorithm [32]. See Appendix A for details.

We assume that all weights are integers, each fitting in a single memory word. Since edges with weight greater than c never cross the minimum cut, we contract all edges with weight greater than our estimate for c, so that now the total weight of edges of G is $O(mc)$.

For the sake of presentation we think of an unweighted graph $\tilde{G}$, obtained from G by replacing an edge of weight w by w parallel edges. We stress that $\tilde{G}$ is never actually constructed by the algorithm. Let $\tilde{m}$ denote the number of edges of $\tilde{G}$. By the argument above, $\tilde{m} = O(mc)$. Let $p = \Theta(\log n / c)$. Let H be the unweighted multigraph obtained by sampling $\lceil p\tilde{m} \rceil$ edges of $\tilde{G}$ ($H = \tilde{G}$ if $c < \log n$). Clearly, the expected value of every cut in H is p times the value of the same cut in G. By [21, Lemma 5.1], choosing the appropriate constants in the sampling probability p guarantees that, with high probability, the value of every cut in H is at least 64/65 times its expected value, and no more than 66/65 times its expected value. It follows that, with high probability, (i) the minimum cut in H has value $c' = \Theta(\log n)$, and that (ii) the value of the cut in H defined by $(S, \bar{S})$ is at most $33c'/32$.

The conceptual process for constructing H can be carried out by randomly selecting $\lceil p\tilde{m} \rceil$ edges of G (with replacement) with probability proportional to their weights. Since the total edge weight of G is $O(mc)$, each selection can be easily performed in $O(\log(mc))$ time, assuming that our model allows generating a random word in constant time. Using a standard technique, each selection can actually be done in $O(\log m)$ time, even under a weaker assumption that we can only generate a random bit in constant time.

Lemma 1 *Given n non-negative integers $x_1, x_2, \ldots, x_n$, we can build in $O(n)$ time a structure that supports the following operation in $O(\log n)$ time, assuming that we can generate a random bit in constant time: choose $i \in \{1, 2, \ldots, n\}$ with probability $p_i = x_i / X$, where $X = \sum_{i=1}^{n} x_i$, and then with high probability return i or fail otherwise.*

Using the above lemma (see proof in Appendix B), the time to construct H is $O(p\tilde{m} \log m) = O(m \log^2 n)$. If any of the invocations fails, we simply terminate the whole algorithm. We emphasize that H is an unweighted multigraph with $m' = O(m \log n)$ edges, and note that we can assume no edge of H has multiplicity greater than c' (we can just delete extra copies).

Next, we apply the following specialized instantiation [40, Theorem 2] of Young's variant [44] of the Lagrangian packing technique of Plotkin, Shmoys, and Tardos [38]. It is shown [40, 44] that for an unweighted graph H with m' edges and minimum cut of size c', the following algorithm finds a weighted tree packing of weight $3c'/8 \leq \tau \leq c'$.

1: $\ell(e) := 0$ for all $e \in E(H)$
2: **while** there is no e with $\ell(e) \geq 1$ **do**
3: find a minimum spanning tree T w.r.t. $\ell(\cdot)$
4: $w(T) = w(T) + 1/(96 \ln m')$
5: $\ell(e) = \ell(e) + 1/(96 \ln m')$ for all $e \in T$
6: **end while**

Karger [23, Lemma 2.3] proves that for a graph H with minimum cut c', any tree packing of weight at least $3c'/8$, and any cut $(S, \bar{S})$ of H of value at most $33c'/32$, at least a $1/8$ fraction of the trees (by weight) 2-constrain the cut $(S, \bar{S})$. Thus, a tree chosen at random from the packing according to the weights 2-constrains the cut $(S, \bar{S})$ with probability at least $1/8$. Choosing $\theta(\log n)$ trees guarantees that, with high probability, one of them 2-constrains the cut $(S, \bar{S})$, which is the minimum cut in G.

It remains to bound the running time of the packing algorithm. Observe that the algorithm increases the weight of some tree by $1/(96 \ln m')$ at each iteration. Since the weight τ of the resulting packing is bounded by c', there are at most $96c' \ln m' = O(\log^2 n)$ iterations. The bottleneck in each iteration is the time to compute a minimum spanning tree in H. We argue that this can be done in $O(m)$ time even though $m' = O(n \log n)$. To see this, first note that since H is a subgraph of G, H has at most m edges (ignoring multiplicities of parallel edges). Next note that it suffices to invoke the MST algorithm on a subgraph of H that includes just the edge with minimum load among any set of parallel edges. Since the algorithm always increases the load of edges by a fixed amount, the edge with minimum load in each set of parallel edges can be easily maintained in $O(1)$ time per load increase by maintaining a cyclic ordered list of each set of parallel edges and moving to choose the next element in this cyclic list whenever the load of the current element is incremented. Hence, we can invoke the randomized linear time MST algorithm [24] on a simple subgraph of H of size $O(m)$. It follows that the running time of the packing algorithm, and hence of the entire procedure, is $O(m \log^2 n)$. □

2.2 Link-cut trees

In our algorithm we will repeatedly use a structure that maintains a rooted tree T with costs on the edges under the following operations:

1. T.ADD(u, Δ) adds Δ to the cost of every edge on the path from u to the root,
2. T.PATH(u) finds the minimum cost of an edge on the path from u to the root,
3. T.SUBTREE(u) finds the minimum cost of an edge in the subtree rooted at u.

All three operations can be supported with a link-cut tree [39] in amortized $O(\log |T|)$ time.[2] We note that we only require these three operations and do not actually use the link and cut functionality of link-cut trees. Other data structures might also be suitable. See, e.g., the use of top-trees in [3].

2.3 Topologically induced subtrees

For a rooted tree T and a node v we denote by T_v the subtree of T rooted at v. For an edge e of T we denote by T_e the subtree of T rooted at the lower endpoint of e.

Let T be a binary tree with edge-costs and n nodes, equipped with a data structure that can answer lowest common ancestor (LCA) queries on T in constant time [16]. Let $\Lambda = \{w_1, w_2, \ldots, w_s\}$ be a subset of nodes of T. We define a smaller tree T^Λ that is equivalent to T in the following sense:

Definition 3 (topologically induced tree) We say that a tree T^Λ is topologically induced on T by Λ if for every $S \subseteq \Lambda$, the minimum cost edge $f \in T^\Lambda$ with $T^\Lambda_f \cap \Lambda = S$ has the same cost as the minimum cost edge $e \in T$ with $T_e \cap \Lambda = S$.

To be clear, the above definition implies that, for any $S \subseteq \Lambda$, there is an edge $e \in T$ with $T_e \cap \Lambda = S$, if and only if there is an edge $f \in T^\Lambda$ with $T^\Lambda_f \cap \Lambda = S$. The term *topologically induced tree* will be justified by the construction in the following lemma.

[2] The original paper [39] did not include the third operation. However, as shown in [28, Appendix 17], it is not difficult to add it.

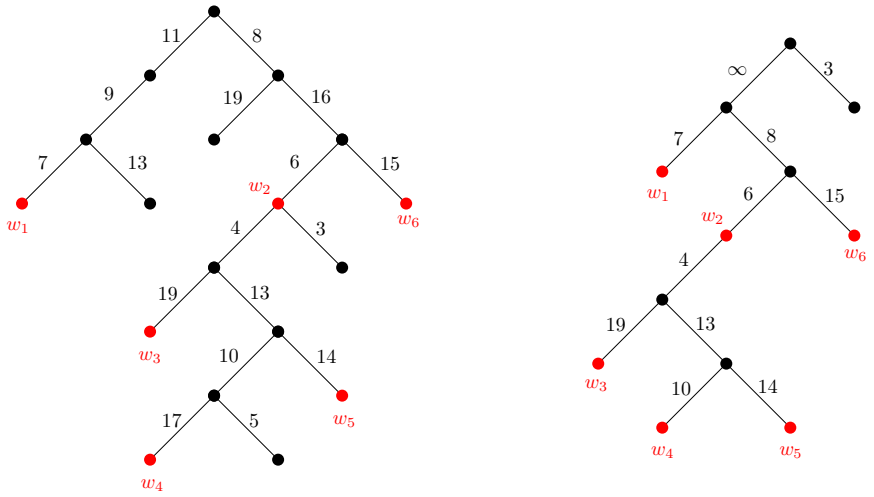

Fig. 1 On the left: a tree and (in red) a set $\Lambda = \{w_1, w_2, \ldots, w_6\}$ sorted according to their preorder numbers. On the right: the corresponding topologically induced tree.

Lemma 2 *There exists an algorithm that, given a binary tree T with edge costs, equipped with a link-cut data structure, and a list $\Lambda = \{w_1, w_2, \ldots, w_s\}$ of nodes of T, ordered according to their visit time in a preorder traversal of T, constructs in $O(\min\{|T|, s \log |T|\})$ time, a tree T^Λ of size $O(s)$ that is topologically induced on T by Λ.*

Proof We define the tree T^Λ to be a tree over all nodes $w_i \in \Lambda$, together with the root and the lowest common ancestor in T of every pair of nodes w_i and w_j. For any two nodes $u, v \in T^\Lambda$, u is an ancestor of v in T^Λ if and only if u is an ancestor of v in T. Thus, each edge (u, v) of T^Λ corresponds to the u-to-v path in T. The edges on this path in T are exactly the edges e of T with $T_e \cap \Lambda = T^\Lambda_{(u,v)} \cap \Lambda$. Hence, the paths of T corresponding to distinct edges of T^Λ are edge disjoint. We define the cost of the edge (u, v) of T^Λ to be the minimum cost of an edge on the corresponding path in T. It follows that for every $\emptyset \neq S \subseteq \Lambda$, the minimum cost edge $f \in T^\Lambda$ with $T^\Lambda_f \cap \Lambda = S$ has the same cost as the minimum cost edge $e \in T$ with $T_e \cap \Lambda = S$. To guarantee that this condition holds for $S = \emptyset$ as well, we choose the edge e of T with the minimum cost such that $T_e \cap \Lambda = \emptyset$ and proceed as follows if such an edge exists. We create a new node v and change the root of T^Λ to v by making the old root of T^Λ a child of v via an edge with infinite cost. We then add a new edge incident to v, whose cost is set to the cost of e. This transformation does not change $T_e \cap \Lambda$ for any edge e of T^Λ, but now that condition with $S = \emptyset$ is satisfied for the new edge incident to the root.

We now turn to proving the construction time. We first prove that T^Λ consists of at most $2s$ nodes. This is because T^Λ consists only of the root, the nodes w_i and $\mathrm{LCA}(w_i, w_{i+1})$. To see this, consider two nodes w_i and w_j with $i < j$ such that their lowest common ancestor u is different than w_i and w_j. Let u_ℓ (u_r) be the left (right) child of u. Then, w_i is a descendant of u_ℓ and w_j a descendant of u_r. Let i' be the largest index such that $w_{i'}$ is in the subtree rooted at u_ℓ. Then $u = \mathrm{LCA}(w_{i'}, w_{i'+1})$.

We next prove that T^Λ (without its edge weights) can be constructed edge weights) in $O(s)$ time. We use a method similar to constructing the Cartesian tree [8,42] of a sequence: we scan $w_1, w_2, \ldots, w_s$ from the left to right while maintaining the subtree of T^Λ induced by $w_0 = \mathrm{LCA}(w_1, w_s)$, and $w_1, w_2, \ldots, w_i$. initially, the subtree of T^Λ induced by w_0 and w_1 is just a single edge (w_0, w_1). We keep the rightmost path of the subtree of T^Λ induced by $w_0, w_1, \ldots, w_i$ on a stack, with the bottommost edge on the top. To process w_{i+1}, we first find $x = \mathrm{LCA}(w_i, w_{i+1})$. Then, we pop from the stack all edges (u, v) such that u and v are both below (or equal to) x in T. Finally, we possibly split the edge on the top of the stack into two and push a new edge onto the stack. The amortized complexity of every step is constant, so the total time is $O(s)$.

Once T^Λ is constructed, we set the cost of every edge (u, v) in T^Λ to be the minimum cost of an edge on the u-to-v path in T. This can be done in $O(\log |T|)$ time per edge of T^Λ by first calling $T.\textsc{add}(u)(\infty)$, then $T.\textsc{path}(v)$ to retrieve the answer, and finally $T.\textsc{add}(u)(-\infty)$, for a total of $O(s \log |T|)$ time. Alternatively,

we can explicitly go over the edges of the corresponding paths of T for every edge of T^{Λ}. We had argued above that these paths are disjoint so this takes $O(|T|)$ in total.

We also need to compute the cost of the edge e of T with the minimum cost such that $T_e \cap \Lambda = \emptyset$. To this end, for each $v \in \Lambda$ we add ∞ to the cost of all edges on the path from v to the root of T. This takes $O(\min(|T|, s \log |T|)$ by either a bottom-up computation on T, or using $T.\text{ADD}(u, \infty)$ for every $v \in \Lambda$. We then retrieve the edge with minimum cost in the entire tree in $O(\log |T|)$ time by a call to SUBTREE for the root of T, and then subtract ∞ from the cost of all edges on the path from v to the root of T for every $v \in \Lambda$. □

We will use the fact that the operation of taking the topologically induced subtree is composable in the following sense.

Proposition 1 *Let T be a binary tree with edge-costs. Let $\Lambda_2 \subseteq \Lambda_1$ be subsets of nodes of T. Let T_1 be topologically induced on T by Λ_1 and T_2 be topologically induced on T_1 by Λ_2. Then T_2 is topologically induced on T by Λ_2.*

3 Finding a Minimum 2-respecting Cut

Given a graph G and a spanning tree T of G, a cut in G is said to *2-respect* the tree T if at most two edges e, e' of T cross the cut (these edges are said to *determine* the cut). In this section we prove the main theorem of this paper:

Theorem 2 *Given an edge-weighted graph G with n vertices and m edges and a spanning tree T, the minimum (weighted) cut in G that 2-respects T can be found in $O(m \log n)$ time.*

The minimum cut determined by every single edge can be easily found in $O(m + n)$ time [23, Lemma 5.1]. We therefore focus on finding the minimum cut determined by two edges. Observe that the cut determined by $\{e, e'\}$ is unique and consists of all edges $(u, v) \in G$ such that the u-to-v path in T contains exactly one of $\{e, e'\}$.

We begin by transforming T (in linear time) into a binary tree. This is standard and is done by replacing every node of degree d with a binary tree of size $O(d)$ where internal edges have weight ∞ and edges incident to leaves have their original weight. We also add an artificial root to T and connect it to the original root with an edge of weight ∞. From now we will be working with binary trees only.

3.1 Descendant edges

We first describe an $O(m \log n)$ time algorithm for finding the minimum cut determined by all pairs of edges $\{e, e'\}$ where e' is a descendant of e in T (i.e. e' is in the subtree of T rooted at the lower endpoint of e). To this end, we shall efficiently find, for each edge e of T, the descendant edge e' that minimizes the weight of the cut determined by $\{e, e'\}$, and return the pair minimizing the weight of the cut.

For a given edge e of T, let T_e denote the subtree of T rooted at the lower endpoint of e. We associate with every node x a list of all edges (u, v) such that x is the lowest common ancestor of u and v. Note that all these lists can be computed in linear time, and form a partition of the edges of G. We compute for every node x the total weight of edges incident to x, and the total weight of edges (u, v) such that $\text{LCA}(u, v)$ is x. This also takes $O(m)$ time. We then compute using an $O(m)$-time bottom-up computation on T, for every edge e of T, the total weight $A(e)$ of all edges with exactly one endpoint in T_e (in fact, this is the information computed by Karger's algorithm for the 1-respecting case). To see how this is done, observe that for an edge e whose lower endpoint is v and whose children in T are e_ℓ and e_r, $A(e)$ is given by $A(e_\ell) + A(e_r)$ minus twice the total weight of edges whose lowest common ancestor is v plus the total weight of edges incident to v. Note that $A(e)$ includes the weight of e.

Using a link-cut tree we maintain a *score* for every edge e of T. The scores will be such that, when an edge e is first encountered by a depth first scan of T, the weight of the cut determined by e and any descendant e' of e is $A(e)$ plus the score of e'. This will allow us to find the best e' for e using a single call to $T.\text{SUBTREE}$. We next explain the details.

All scores are first initialized to zero. Then, for every edge (u, v) of G, we increase the score of all edges on the u-to-v path in T by the weight $w(u, v)$ of (u, v). This takes $O(\log n)$ time per edge (u, v) by

calling $T.\text{ADD}(u, w(u,v))$, $T.\text{ADD}(v, w(u,v))$ and $T.\text{ADD}(\text{LCA}(u,v), -2w(u,v))$. This initialization takes $O(m \log n)$ time. We then perform an Euler tour of T. When the tour first descends below a node x, for every edge (u,v) in the list of x, we decrease the score of all edges on the u-to-v path in T by $2w(u,v)$. Then, when the tour ascends above x, for every edge (u,v) in the list of x, we increase the score of all edges on the u-to-v path in T by $2w(u,v)$. Note that, at any point during this traversal, each edge (u,v) either contributes $w(u,v)$ or $-w(u,v)$ to the score of every edge on the u-to-v path in T, depending on whether the tour is yet to descend below $\text{LCA}(u,v)$ or has already done so. As above, each update can be implemented in $O(\log n)$ time. Since every edge appears in exactly one list, the total time to perform all the updates is $O(m \log n)$.

Lemma 3 *Consider the point in time when the Euler tour had just encountered an edge e for the first time. At that time, for every descendant edge e' of e, the weight of the cut determined by $\{e, e'\}$ is $A(e)$ plus the score of e'.*

Proof Observe that the weight of the cut determined by $\{e, e'\}$ is the sum of weights of all edges with (1) one endpoint in $T_e - T_{e'}$ and the other not in T_e, or (2) one endpoint in $T_e - T_{e'}$ and the other in $T_{e'}$. Note that the edges satisfying (1) have exactly one endpoint in T_e, and hence their weight is accounted for in $A(e)$. However, $A(e)$ also counts the weight of edges (u,v) with one endpoint in $T_{e'}$ and the other not in T_e (see Fig. 2). Such edges do not cross the cut. Note that for such edges both e and e' are on the u-to-v path in T. The fact that e is on the u-to-v path implies that the traversal has already descended below $\text{LCA}(u,v)$. Hence, (u,v) currently contributes $-w(u,v)$ to the score of e', offsetting its contribution to $A(e)$. Next note that the edges satisfying (2) are edges (u,v) with both u and v in T_e, which means that they are not accounted for in $A(e)$, and that the traversal did not yet descend below $\text{LCA}(u,v)$. Hence the contribution of such an edge (u,v) to the score of e' is indeed the weight of (u,v). □

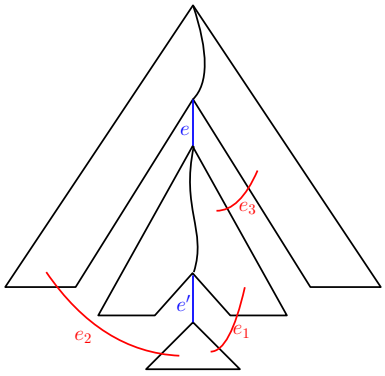

Fig. 2 A tree T and two descendant edges e, e'. Edges e_1 and e_3 cross the cut determined by $\{e, e'\}$. Edge e_2 does not cross the cut. The edge e_1 is not counted in $A(e)$, while the edges e_2, e_3 are. The edge e_1 contributes $w(e_1)$ to the score of e', the edge e_2 contributes $-w(e_2)$ to the score of e', and the edge e_3 does not contribute to the score of e'.

By the lemma, the descendant edge e' of e that minimizes the weight of the cut determined by $\{e, e'\}$ is the edge with minimum score in the subtree of e at that time. The score of this edge e' can be found in $O(\log n)$ time by calling $T.\text{SUBTREE}(x)$, where x is the lower endpoint of e.

3.2 Independent edges

We now describe an $O(m \log n)$ time algorithm for finding the minimum cut determined by all pairs of edges $\{e, e'\}$ where e is *independent* of e' in T (i.e. e is not a descendant of e' and e' is not a descendant of e). We begin by showing that the problem can be reduced to the following *bipartite* problem:

Definition 4 (The bipartite problem) Given two trees T_1 and T_2 with costs on the edges and a list of non-tree edges $L = \{(u,v) : u \in T_1, v \in T_2\}$ where each non-tree edge has a cost, find a pair of edges

$e \in T_1$ and $e' \in T_2$ that minimize the sum of costs of e, of e', and of all non-tree edges $(u, v) \in L$ where u is in T_{1e}, and v is in $T_{2e'}$. The size of such a problem is defined as the number of non-tree edges in L plus the sizes of T_1 and T_2.

Lemma 4 *Given an edge-weighted graph G with n vertices and m edges and a spanning tree T, finding the minimum cut among those determined by a pair of independent edges $\{e, e'\}$ can be reduced in $O(m \log n)$ time to multiple instances of the bipartite problem of total size $O(m)$.*

Proof Recall that every node w of T has at most two children. We create a separate bipartite problem for every node w of T that has exactly two children (x and y). This bipartite problem will be responsible for finding the minimum cut determined by all pairs of independent edges $\{e, e'\}$ where e is in T_x (or possibly (w, x)) and e' is in T_y (or possibly (w, y)).

Throughout our description, note the distinction between edge weights and edge costs. The input graph G has edge weights, and the goal is to find the cut with minimum weight. The bipartite problems we define have edge costs, which are derived from the weights of edges in the input graph.

We initialize the cost of every edge of G to be zero. Then, for every edge $f = (u, v)$ of G, we add the weight of f to the cost of every edge on the u-to-v path. We maintain the costs in a link-cut tree so each f is handled in $O(\log n)$ time. Now consider any node w with exactly two children x and y, and any pair of independent edges $\{e, e'\}$ where e is in T_x and e' is in T_y. Observe that the edges crossing the cut determined by $\{e, e'\}$ are exactly the edges $f = (u, v)$ with one endpoint in T_e or in $T_{e'}$, and the other endpoint not in T_e nor in $T_{e'}$. Hence, the weight of the cut determined by $\{e, e'\}$ equals the sum of the cost of e plus the cost of e' minus twice the total weight of all non-tree edges $f = (u, v)$ such that u is in T_e and v is in $T_{e'}$.

We therefore define the bipartite problem for w as follows: (1) T_1 is composed of the edge (w, x) and the subtree rooted at x with costs as described above, (2) T_2 is composed of the edge (w, y) and the subtree rooted at y with the costs as described above, and (3) for every non-tree edge $f = (u, v)$ with weight c such that $\mathrm{LCA}(u, v) = w$ the list of non-tree edges L includes (u, v) with cost $-2c$. By construction, the solution to this bipartite problem is the pair of independent edges e, e' with $e \in T_x$ and $e' \in T_y$ that minimize the weight of the cut in G defined by e and e'.

The only issue with the above bipartite problem is that the overall size of all bipartite problems (over all nodes w) might not be $O(m)$. This is because the edges of T might appear in the bipartite problems defined for more than a single node w. In order to guarantee that the overall size of all bipartite problems is $O(m)$, we construct a compact bipartite problem using the topologically induced trees of Lemma 2.

We construct in $O(m)$ time a constant-time LCA data structure [16] for T. In overall $O(m \log n)$ time, we construct, for each node $w \in T$ with exactly two children x and y:

1. A list L_w of all non-tree edges (u, v) with $\mathrm{LCA}(u, v) = w$.
2. A list $\Lambda_x = \{w, x\} \cup \{u : (u, v) \in L_w \text{ and } u \in T_x\}$, sorted according to their visit time in a preorder traversal of T.
3. A list $\Lambda_y = \{w, y\} \cup \{v : (u, v) \in L_w \text{ and } v \in T_y\}$, sorted according to their visit time in a preorder traversal of T.

These lists require total $O(m)$ space and can be easily computed in $O(m \log n)$ time by going over the non-tree edges, because each non-tree edge is in the list L_w of a unique node w.

The list L for the compact bipartite problem is identical to the list L for the non-compact problem. The tree T_1° (T_2°) for the compact bipartite problem of w is the topologically tree induced on $(w, x) \cup T_x$ ($(w, y) \cup T_y$) by Λ_x (Λ_y). This is done in $O(|\Lambda_x| \log n)$ ($O(|\Lambda_y| \log n)$) time by invoking Lemma 2. It follows that the total time for constructing all compact bipartite problems is $O(m \log n)$ and that their total space is $O(m)$.

It remains to argue that the solution to the compact bipartite problem is identical to the solution to the non-compact one. Observe that the cost of a solution e, e' for the non-compact bipartite problem is the cost of e plus the cost of e' plus the cost of all edges in L with one endpoint in $T_{1e} \cap \Lambda$ and the other endpoint in $T_{2e'} \cap \Lambda$.

Consider now any pair of edges e and e' for the non-compact bipartite problem. By definition of topologically induced trees, the minimum cost edge $f \in T_1^\circ$ with $T_{1f}^\circ \cap \Lambda = T_{1e} \cap \Lambda$, has cost not exceeding that of e. An analogous argument holds for e' and an edge f' of T_2°. Hence, the cost of the optimal solution for the compact problem is not greater than that of the non-compact problem. Conversely, for any edge f in T_1°, there exists an edge e in T_1 with cost not exceeding that of f and $T_{1f}^\circ \cap \Lambda = T_{1e} \cap \Lambda$.

Hence, the cost of the optimal solution for the compact problem is not less than that of the non-compact problem. It follows that the two solutions are the same. □

The proof of Theorem 2 follows from the above reduction and the following solution to the bipartite problem:

Lemma 5 *A bipartite problem of size m can be solved in $O(m \log m)$ time.*

Proof Recall that in the bipartite problem we are given two trees T_1 and T_2 with edge-costs and a list of non-tree edges $L = \{(u, v) : u \in T_1, v \in T_2\}$ where each non-tree edge has a cost. To prove the lemma, we describe a recursive $O(m \log m)$ time algorithm that finds, for every edge $e \in T_1$, the best edge $e' \in T_2$ (i.e. the edge e' that minimizes the sum of costs of e' and of all non-tree edges $(u, v) \in L$ where u in T_{1e} and $v \in T_{2e'}$).

We begin by applying a standard heavy path decomposition [16] to T_1, guided by the number of non-tree edges: The *heavy* edge of a node of T_1 is the edge leading to the child (called the *heavy* child) whose subtree has the largest number of incident non-tree edges in L (breaking ties arbitrarily). The other edges are called *light*. The maximal sets of connected heavy edges define a decomposition of the nodes into *heavy paths*.

We define a *fragment* of the tree T_1 to be a subtree of T_1 formed by a contiguous subpath $u_1 - u_2 - \cdots - u_k$ of some heavy path of T_1, together with all subtrees hanging from this subpath via light edges. Given a fragment f, let $L(f) = \{(x_1, y_1), (x_2, y_2) \ldots, (x_\ell, y_\ell)\}$ be the set of edges (x, y) of L with $x \in f$. We define the induced subtree $T_2(f)$ to be the tree topologically induced on T_2 by the root of T_2 and $\{y_1, y_2, \ldots, y_\ell\}$. The size of $T_2(f)$ is $|T_2(f)| = O(|L(f)|)$. We also define a modified induced subtree $T_2'(f)$ as follows. Let $L_\downarrow(f)$ be the set of edges $(x, y) \in L$ with x in the subtree rooted at the heavy child of the last node u_k of the fragment f (if such a heavy child exists). Consider the tree T_2, where the cost of each edge e' of T_2 is increased by the total cost of all edges $(x, y) \in L_\downarrow(f)$, where y is in $T_{2e'}$. The modified induced subtree $T_2'(f)$ is defined as the tree topologically induced by the root of T_2 and $\{y_1, y_2, \ldots, y_\ell\}$ on this modified T_2.

We are now ready to describe the recursion. The input to a recursive call is a fragment f of T_1 and the list $(x_1, y_1), (x_2, y_2), \ldots, (x_\ell, y_\ell)$ of all non-tree edges in L with x_i in f, together with $T_2(f)$ and $T_2'(f)$. A fragment f is specified by the *top* node (u_1) and the *bottom* node (u_k) of the corresponding subpath of a heavy path of T_1. In the first call, f is specified by the root of T_1 and the leaf ending the heavy path of T_1 that contains the root. That is, in the first call f is the entire tree T_1. The list of non-tree edges for the first call is the entire list L. The recursion works by selecting the *middle* node of the subpath, defined as follows: We define the *light size* s_i of node u_i as the number of non-tree edges $(x, y) \in L$ where either $x = u_i$ or x is in the subtree rooted at the light child of u_i. Note that $s_1 + s_2 + \ldots + s_k = |L(f)|$. If $s_1 > |L(f)|/2$ then the middle node is defined as u_1. Otherwise, the middle node is defined as the node u_i such that $s_1 + \ldots + s_{i-1} \leq |L(f)|/2$ but $s_1 + \ldots + s_i > |L(f)|/2$. We keep, for every heavy path P of T_1 a list of the nodes of P with non-zero light size, ordered according to their order on P. We find the middle node u_i in $O(|L(f)|)$ time by going over the nodes in this list one after the other until we encounter the middle node.

After identifying the middle node u_i we apply recursion on the following three fragments: the fragment defined by subpath $u_1 - \cdots - u_{i-1}$, the fragment defined by subpath $u_{i+1} - \cdots - u_k$, and the fragment consisting of the entire subtree rooted at the light child of u_i. Before a recursive call to fragment g, we construct the appropriate $T_2(g)$ and $T_2'(g)$. The induced tree $T_2(g)$ can be computed from $T_2(f)$ in $O(|T_2(f)|) = O(|L(f)|)$ time by invoking Lemma 2 on $T_2(f)$ with $\Lambda_g = r \cup \{y : (x, y) \in L(g)\}$, where r is the root of T_2. Note that we had defined $T_2(g)$ as the topologically induced tree on T_2 by Λ_g, not on $T_2(f)$ by Λ_g. However, since $\Lambda_g \subseteq \Lambda_f$, by Proposition 1, the two definitions are equivalent.

For constructing $T_2'(g)$ from $T_2'(f)$, we first need to increase the cost of each edge $\tilde{e}$ of $T_2'(f)$ by the total cost of edges in $L_\downarrow(g) \setminus L_\downarrow(f)$ that are incident to $T_2'(f)_{\tilde{e}}$. This can be done in a single bottom-up traversal of $T_2'(f)$ in $O(|T_2'(f)|)$ time. Then, we invoke Lemma 2 on $T_2'(f)$ with Λ_g to obtain $T_2'(g)$. To summarize, constructing the trees $T_2(g)$ and $T_2'(g)$ for all three recursive subproblems takes $O(|L(f)|)$ time.

The three recursive calls will take care of finding the best edge $e' \in T_2$ for every edge $e \in T_1$ included in one of the recursive problems. It only remains to handle the three edges that do not belong to any of the recursive problems; the edge between u_i and its light child, the edge (u_{i-1}, u_i), and the edge (u_i, u_{i+1}). For each such edge e we describe a procedure that finds its best $e' \in T_2(f)$ in time $O(|T_2(f)|)$.

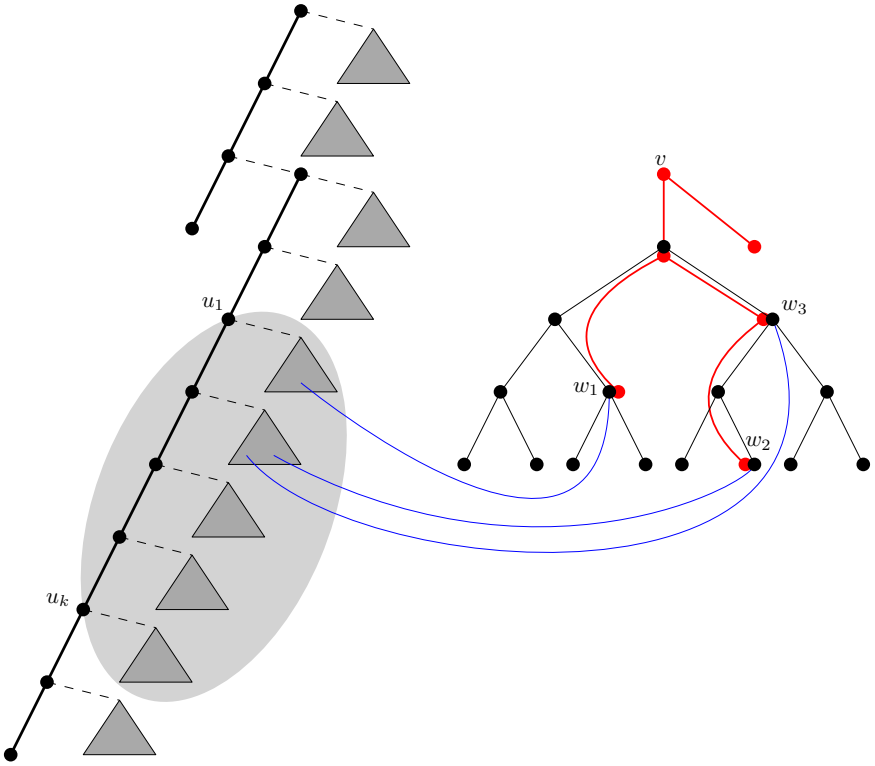

Fig. 3 On the left: A fragment (in light gray) in the tree T_1, defined by the top node u_1 and the bottom node u_k, both laying on the same heavy path (solid edges). The triangles (in dark gray) are the subtrees hanging from the heavy paths via light edges (dashed). On the right: The tree T_2 (black) connected to the fragment via three non-tree edges (blue). The endpoints w_1, w_2, w_3 of these edges define the topologically induced tree (in red).

Recall that, by definition of the bipartite problem, the best edge e' for e is the edge e' of T_2 minimizing the cost of e' plus the cost of all non-tree edges $(x, y) \in L$ with $x \in T_{1e}$, and $y \in T_{2e'}$. For the case where e is the edge between u_i and its light child, $T_{1e} = T_1(f)_e$. We therefore mark all non-tree edges $(x, y) \in L(f)$, where x is in $T_1(f)_e$. A non-efficient solution would work directly on T_2 by propagating, in a bottom-up traversal of T_2, the cost of all marked edges so that, after the propagation, the cost of every edge e' in T_2 has been increased by the total cost of all non-tree edges $(x, y) \in L$ with $x \in T_1(f)_e$, and with $y \in T_{2e'}$. Then we can take the edge $e' \in T_2$ with the minimum cost. However, this would take $O(|T_2|) = O(m)$, which is too slow. Instead, we perform the propagation in $T_2(f)$. Namely, in a bottom-up traversal of $T_2(f)$, we propagate the cost of all marked edges so that, after the propagation, the cost of every edge e' in $T_2(f)$ has been increased by the total cost of all non-tree edges $(x, y) \in L(f)$ with $x \in T_1(f)_e$, and with $y \in T_2(f)_{e'}$. This takes $O(|T_2(f)|) = O(|L(f)|)$ time. Since the propagation process affects all the edges $\tilde{e}$ with the same $T_{2\tilde{e}} \cap \Lambda_f$ in the same way, the definition of topologically induced tree guarantees that the edges with the minimum cost in T_2 and in $T_2(f)$ have the same cost, so using $T_2(f)$ instead of T_2 is correct.

The procedure for the cases where e is the edge (u_{i-1}, u_i) or (u_i, u_{i+1}) is identical, except that we apply it with $T_2'(f)$ instead of $T_2(f)$. This difference stems from the fact that applying the above procedure on $T_2(f)$ only considers the costs of the non-tree edges in $L(f)$, but not the costs of the non-tree edges in $L_\downarrow(f)$, which might also cross cuts involving the edges (u_{i-1}, u_i) or (u_i, u_{i+1}). The definition of the costs of edges in $T_2'(f)$ takes into account the contribution of costs of non-tree edges in $L_\downarrow(f)$. The rest of the propagation procedure and the proof of its correctness remain unchanged.

To analyze the overall running time, let $T(m)$ be the time to handle a fragment f corresponding to a whole heavy path, and $T'(m)$ be the time to handle a fragment f corresponding to a proper subpath of some heavy path, where $m = |L(f)|$. Then $T(m) = T'(m_1) + T(m_2) + T'(m_3)$ for some m_1, m_2, m_3, where $m_1 + m_2 + m_3 = m$ since the subproblems are disjoint, $m_1, m_3 \leq m/2$ by the choice of the middle node, and $m_2 \leq m/2$ by the definition of a heavy path. This is since the light child of u_i does not have more

incident non-tree edges in its subtree than the number of non-tree edges incident to the subtree of the heavy child u_{i+1}. When f corresponds to a whole heavy path the number of non-tree edges incident to the subtree of u_{i+1} is exactly m_3. Similarly, for the case where f does not correspond to a whole heavy path, $T'(m) = T'(m_1) + T(m_2) + T'(m_3)$ for some m_1, m_2, m_3, where $m_1 + m_2 + m_3 = m$ and $m_1, m_3 \leq m/2$ (but now we cannot guarantee that $m_2 \leq m/2$). Considering the tree describing the recursive calls, on any path from the root (corresponding to the fragment consisting of the whole T_1) to a leaf, we have the property that the value of m decreases by at least a factor of 2 every two steps. Hence, the depth of the recursion is $O(\log m)$. It follows that the total time to handle a bipartite problem of size m is $O(m \log m)$. □

To conclude, we have shown how to find the minimum cut determined by two descendant edges and the minimum cut determined by two independent edges. If we wish to find for every edge e of T the edge e' that minimizes the weight of the cut determined by $\{e, e'\}$, then we need to run every bipartite problem twice (switching the roles of T_1 and T_2). This does not incur any asymptotic overhead.

4 A $\log \log n$ Speedup

Karger modified his $O(m \log^3 n)$ time minimum cut algorithm to work in $O(m \log^3 n / \log \log n + n \log^6 n)$ time by observing that 1-respecting cuts can be found faster than 2-respecting cuts, and tweaking the parameters of the tree packing. In this section we explain how to apply this idea, together with our new $O(m \log n)$ time algorithm for the 2-respecting problem, to derive a new $O(m \log^2 n / \log \log n + n \log^{3+\delta} n)$ time minimum cut algorithm, for any $\delta > 0$.

As in Karger's implementation [23, Section 9.1], we start with an initial sampling step, except that instead of $\epsilon = \frac{1}{4 \log n}$ we use $\epsilon = \frac{1}{4 \log^\gamma n}$, for some $\gamma \in (0, 1)$ to be fixed later. This produces in linear time a graph H with $m' = O(n/\epsilon^2 \log n)$ edges and minimum cut $c' = O(\epsilon^{-2} \log n)$ such that the minimum cut in G corresponds to a $(1+\epsilon)$-times minimum cut in H. We find a packing in H of weight $c'/2$ with Gabow's algorithm [7] in $O(m'c' \log n) = O(n/\epsilon^4 \log^3 n)$ time. We choose $4 \log^{1+\gamma} n$ trees at random from the packing and for each tree we find the minimum cut that 1-respects it. This takes total $O(m \log^{1+\gamma} n)$ time. Then, we choose $\log n / \log(\log^\gamma n) = O(\log n / \log \log n)$ trees at random from the packing and for each tree we find the minimum cut that 2-respects it. This takes total in $O(m \log^2 n / \log \log n)$ time using our new algorithm.

Let $\rho \geq c'/2$ be the weight of the packing, let $\alpha\rho$ be the total weight of trees that 1-respect the minimum cut, and let $\beta\rho$ be the total weight of trees that 2-respect the minimum cut. As observed by Karger, $\beta \geq 1 - 2\epsilon - 2\alpha$. Thus, either $\alpha > \frac{1}{4 \log^\gamma n}$ and choosing $4 \log^{1+\gamma} n$ trees guarantees that none of them 1-respects the minimum cut with probability at most

$$(1-\alpha)^{4 \log^{1+\gamma} n} \leq \left(1 - \frac{1}{4 \log^\gamma n}\right)^{4 \log^\gamma n \cdot \log n} < 1/n,$$

or $\beta \geq 1 - 1/\log^\gamma n$ and choosing $\log n / \log(\log^\gamma n)$ trees guarantees that none of them 2-respects the minimum cut with probability at most

$$(1-\beta)^{\log n / \log(\log^\gamma n)} \leq \left(\frac{1}{\log^\gamma n}\right)^{\log n / \log(\log^\gamma n)} = 1/n.$$

The overall complexity is $O(n/\epsilon^4 \log^3 n + m \log^{1+\gamma} n + m \log^2 n / \log \log n) = O(m \log^2 n / \log \log n + n \log^{3+4\gamma} n)$. By adjusting $\gamma = \delta/4$ we obtain the claimed complexity of $O(m \log^2 n / \log \log n + n \log^{3+\delta} n)$.

Acknowledgements

We thank Daniel Anderson and Guy Blelloch for drawing our attention to an inaccuracy in a prior version of Section 3.1.

References

1. A. Abboud, R. Krauthgamer, J. Li, D. Panigrahi, T. Saranurak, and O. Trabelsi. Breaking the cubic barrier for all-pairs max-flow: Gomory-hu tree in nearly quadratic time. In *63rd FOCS*, pages 884–895, 2022.
2. S. Alstrup, J. Holm, K. D. Lichtenberg, and M. Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Trans. Algorithms*, 1(2):243–264, 2005.
3. N. Bhardwaj, A. M. Lovett, and B. Sandlund. A simple algorithm for minimum cuts in near-linear time. In *17th SWAT*, volume 162, pages 12:1–12:18, 2020.
4. G. S. Brodal, R. Fagerberg, and C. N. S. Pedersen. Computing the quartet distance between evolutionary trees in time $O(n \log^2 n)$. In *12th ISAAC*, pages 731–742, 2001.
5. L. Chen, R. Kyng, Y. P. Liu, R. Peng, M. P. Gutenberg, and S. Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. In *63rd FOCS*, pages 612–623, 2022.
6. L. R. Ford and D. R. Fulkerson. Flows in network. *Princeton Univ. Press*, 1962.
7. H. N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. *J. Comput. Syst. Sci.*, 50(2):259–273, 1995. Announced at STOC 1991.
8. H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *16th STOC*, pages 135–143, 1984.
9. P. Gawrychowski, S. Mozes, and O. Weimann. A note on a recent algorithm for minimum cut. In *4th SOSA*, pages 74–79, 2021.
10. B. Geissmann and L. Gianinazzi. Parallel minimum cuts in near-linear work and low depth. In *30th SPAA*, pages 1–11, 2018.
11. M. Ghaffari, K. Nowicki, and M. Thorup. Faster algorithms for edge connectivity via random 2-out contractions. *CoRR*, abs/1909.00844, 2019.
12. A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. *J. ACM*, 45(5):783–797, 1998.
13. A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.
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. J. Hao and J. B. Orlin. A faster algorithm for finding the minimum cut in a directed graph. *J. Algorithms*, 17(3):424–446, 1994.
16. D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.
17. M. Henzinger, J. Li, S. Rao, and D. Wang. Deterministic near-linear time minimum cut in weighted graphs. In *28th SODA*, page To appear, 2024.
18. M. Henzinger, S. Rao, and D. Wang. Local flow partitioning for faster edge connectivity. In *28th SODA*, pages 1919–1938, 2017.
19. D. R. Karger. Global min-cuts in RNC, and other ramifications of a simple min-cut algorithm. In *4th SODA*, pages 21–30, 1993.
20. D. R. Karger. *Random Sampling in Graph Optimization Problems*. PhD thesis, Stanford University, Stanford, CA 94305, 1994.
21. D. R. Karger. Random sampling in cut, flow, and network design problems. *Math. Oper. Res.*, 24(2):383–413, 1999. Announced at STOC 1994.
22. D. R. Karger. Random sampling in cut, flow, and network design problems. *Math. Oper. Res.*, 24(2):383–413, 1999.
23. D. R. Karger. Minimum cuts in near-linear time. *J. ACM*, 47(1):46–76, 2000. Announced at STOC 1996.
24. D. R. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–328, 1995.
25. D. R. Karger and C. Stein. A new approach to the minimum cut problem. *J. ACM*, 43(4):601–640, 1996.
26. K. Kawarabayashi and M. Thorup. Deterministic edge connectivity in near-linear time. *J. ACM*, 66(1):4:1–4:50, 2019.
27. V. King, S. Rao, and R. E. Tarjan. A faster deterministic maximum flow algorithm. *J. Algorithms*, 17(3):447–474, 1994. Announced at SODA 1992.
28. P. N. Klein and S. Mozes. Optimization algorithms for planar graphs. `http://planarity.org`. Book draft.
29. R. A. Kronmal and A. V. Peterson. On the alias method for generating random variables from a discrete distribution. *The American Statistician*, 33(4):214–218, 1979.
30. Y. T. Lee and A. Sidford. Path finding methods for linear programming: Solving linear programs in $\tilde{O}(\sqrt{rank})$ iterations and faster algorithms for maximum flow. In *55th FOCS*, pages 424–433, 2014.
31. A. Madry. Computing maximum flow with augmenting electrical flows. In *57th FOCS*, pages 593–602, 2016.
32. D. W. Matula. A linear time $2 + \epsilon$ approximation algorithm for edge connectivity. In *4th SODA*, pages 500–504, 1993.
33. S. Mukhopadhyay and D. Nanongkai. Weighted min-cut: sequential, cut-query, and streaming algorithms. In *52nd STOC*, pages 496–509, 2020.
34. H. Nagamochi and T. Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM J. Discrete Math.*, 5(1):54–66, 1992.
35. H. Nagamochi and T. Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7(5&6):583–596, 1992.
36. C. S. J. A. Nash-Williams. Edge disjoint spanning trees of finite graphs. *Journal of the London Mathematical Society*, 36:445–450, 1961.
37. J. B. Orlin. Max flows in $O(nm)$ time, or better. In *45th STOC*, pages 765–774, 2013.
38. S. A. Plotkin, D. B. Shmoys, and É. Tardos. Fast approximation algorithms for fractional packing and covering problems. *Math. Oper. Res.*, 20(2):257–301, 1995.
39. D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.
40. M. Thorup and D. R. Karger. Dynamic graph algorithms with applications. In *7th SWAT*, pages 1–9, 2000.
41. J. van den Brand, L. Chen, R. Kyng, Y. P. Liu, R. Peng, M. P. Gutenberg, and A. S. Sushant Sachdeva. A deterministic almost-linear time algorithm for minimum-cost flow. In *64rd FOCS*, 2023. To appear.

42. J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.
43. A. Walker. New fast method for generating discrete random numbers with arbitrary frequency distributions. *Electronics Letters*, 10:127–128(1), April 1974.
44. N. E. Young. Randomized rounding without solving the linear program. In *6th SODA*, pages 170–178, 1995.

A A constant factor approximation of the minimum cut

Matula [32] gave an $O(m/\epsilon)$ time algorithm that finds a $(2+\epsilon)$ approximation of the minimum cut in an unweighted graph G. The algorithm proceeds in iterations, where each iteration takes $O(m)$ time, and either finds a $(2+\epsilon)$ approximate cut, or produces a subgraph G' that contains the minimum cut of G, but has only a constant fraction of the edges of G. Hence there are $O(\log n)$ iterations, and the total running time is $O(m)$ for any fixed ϵ.

Matula's algorithm can be easily extended to the weighted setting. Each iteration can be implemented in $O(m \log n)$ time (cf. [20]), and produces a subgraph G' with a constant factor of the total edge weight of G. Thus, the algorithm produces a $(2+\epsilon)$ approximation of the minimum cut in a weighted graph G in $O(m \log n \log W)$ time, where W is the sum of edge weights in G.

The running time can be decreased to $O(m \log^2 n)$ at the expense of a worse constant factor approximation as follows. Let G be a weighted graph. Let c denote the weight of the minimum cut in G. Compute a maximum spanning tree T of G, and let w^* be the minimum weight of an edge in T. It is easy to see [22] that $w^* \leq c \leq n^2 w^*$. Contract all edges e with $w(e) > n^2 w^*$. Clearly, this does not affect the minimum cut. If $w^* \leq n^3$, then all edge weights are now bounded by n^5, and we can run Matula's algorithm in $O(m \log^2 n)$ time, and obtain a $(2+\epsilon)$-approximate minimum cut. Otherwise, set $\tilde{w}(e) \leftarrow \lfloor w(e)/\frac{w^*}{n^3} \rfloor$, and delete all edges with $\tilde{w} = 0$. Call the resulting graph $\tilde{G}$. Observe that $\tilde{G}$ has integer edge weights bounded by n^5, so we can find a $(2+\epsilon)$-approximate minimum cut in $\tilde{G}$ in $O(m \log^2 n)$ time. Now scale up the edge weights in $\tilde{G}$ by $\frac{w^*}{n^3}$. The weight of each edge in $\tilde{G}$ is now off from its original weight in G by at most $\frac{w^*}{n^3} \leq \frac{c}{n^3}$. Thus, the weights of any cut in $\tilde{G}$ and in G differ by at most c/n. Hence, an $(2+\epsilon)$-approximate minimum cut in $\tilde{G}$ is an $O(1)$-approximate minimum cut in G.

B Proof of Lemma 1

We follow the classical alias method by Walker [43], see Kronmal and Peterson for $O(n)$ construction time [29]. This method, given $x_1, x_2, \ldots, x_n$, finds $a_1, a_2, \ldots, a_n \in \{1, 2, \ldots, n\}$ and $b_1, b_2, \ldots, b_n \in \{1, 2, \ldots, n\}$ and integers $q_1, q_2, \ldots, q_n$ such that choosing $i \in \{1, 2, \ldots, n\}$ with probability $p_i = x_i/X$ reduces to the following two-step process:

1. choose uniform $j \in \{1, 2, \ldots, n\}$,
2. choose uniform $k \in [Xn]$,
3. if $k < q_j$ then return a_j, otherwise return b_j.

First, we describe how to choose uniform $j \in [n]$ in $O(\log n)$ time or fail with probability at most $1/n^c$. Let $\ell = \lceil (c+1) \log n \rceil = O(\log n)$, and $y = \lfloor 2^\ell / n \rfloor$. We have the following inequalities:

$$2^\ell (1 - 1/n^c) \leq 2^\ell - n \leq n \cdot y \leq 2^\ell.$$

We choose uniform $j' \in [2^\ell]$ by choosing ℓ random bits in $O(\log n)$ time. If $j' \geq n \cdot y$ then we fail, which happens with probability at most $1/n^c$. Otherwise, we return $\lfloor j'/y \rfloor$.

Next, instead of first choosing uniform $k \in [Xn]$ and then checking if $k < q_j$, we combine the two steps. Let $\ell' = \lceil (c+1) \log n \rceil = O(\log n)$, and choose integer y' such that $Xn/2^{\ell'} \leq y'$ and $Xn/(2^{\ell'} - 1) > y'$, which is possible as long as $Xn \geq 2^{2\ell'}$. To ensure this, for $X = n^{O(c)}$ we simply choose uniform $k \in [Xn]$ or fail with probability at most $1/n^c$ as described in the previous paragraph in $O(\log n)$ time. Otherwise, we have the following inequalities:

$$(2^{\ell'} - 1) y' < Xn \leq 2^{\ell'} y'.$$

We choose ℓ' random bits in $O(\log n)$ time to form a uniform $x \in [2^{\ell'}]$. If $x = 2^{\ell'} - 1$ or $x = \lfloor q_j / y' \rfloor$ then we fail, which happens with probability at most $2/n^c$. Otherwise, if $x < \lfloor q_j / y' \rfloor$ then we return a_j, and otherwise we return b_j.

The above procedure is equivalent to first choosing i with probability p_i/X and then choosing whether to fail with probability at most $3/n^c$ or return i.