Parallel Minimum Cuts in $O(m \log^2 n)$ Work and Low Depth

DANIEL ANDERSON and GUY E. BLELLOCH, Carnegie Mellon University, Pittsburgh, PA, USA

We present a randomized $O(m \log^2 n)$ work, $O(\text{polylog}\, n)$ depth parallel algorithm for minimum cut. This algorithm matches the work bounds of a recent sequential algorithm by Gawrychowski, Mozes, and Weimann [ICALP'20], and improves on the previously best parallel algorithm by Geissmann and Gianinazzi [SPAA'18], which performs $O(m \log^4 n)$ work in $O(\text{polylog}\, n)$ depth.

Our algorithm makes use of three components that might be of independent interest. First, we design a parallel data structure that efficiently supports batched mixed queries and updates on trees. It generalizes and improves the work bounds of a previous data structure of Geissmann and Gianinazzi and is work efficient with respect to the best sequential algorithm. Second, we design a parallel algorithm for approximate minimum cut that improves on previous results by Karger and Motwani. We use this algorithm to give a work-efficient procedure to produce a tree packing, as in Karger's sequential algorithm for minimum cuts. Last, we design an efficient parallel algorithm for solving the minimum 2-respecting cut problem.

CCS Concepts: • **Theory of computation → Graph algorithms analysis**; **Parallel algorithms;**

Additional Key Words and Phrases: Minimum cut, parallel algorithms, graph algorithms, dynamic trees

ACM Reference format:
Daniel Anderson and Guy E. Blelloch. 2023. Parallel Minimum Cuts in $O(m \log^2 n)$ Work and Low Depth. *ACM Trans. Parallel Comput.* 10, 4, Article 18 (December 2023), 28 pages.
https://doi.org/10.1145/3565557

1 INTRODUCTION

Minimum cut is a classic problem in graph theory and algorithms. The problem is to find, given an undirected weighted graph $G = (V, E)$, a nonempty subset of vertices $S \subset V$ such that the total weight of the edges crossing from S to $V \setminus S$ is minimized. Early approaches to the problem were based on reductions to maximum s-t flows [16, 17]. Several algorithms followed that were based on edge contraction [21, 26, 31, 32]. Karger was the first to observe that tree packings [33] can be used to find minimum cuts [23]. In particular, for a graph with n vertices and m edges, Karger showed how to use random sampling and a tree packing algorithm of Gabow [10] to generate a set of $O(\log n)$ spanning trees such that, with high probability, the minimum cut crosses at most two edges of one of them. A cut that crosses at most k edges of a given tree is called a *k-respecting* cut. Karger then gives an $O(m \log^2 n)$-time algorithm for finding minimum 2-respecting cuts, yielding a randomized $O(m \log^3 n)$-time algorithm for minimum cut. Karger also gives a parallel algorithm for minimum 2-respecting cuts in $O(n^2)$ work and $O(\log^3 n)$ depth.

This research was supported by NSF Grants No. CCF-1901381, No. CCF-1910030, and No. CCF-1919223.
Authors' address: D. Anderson and G. E. Blelloch, 5000 Forbes Ave, Carnegie Mellon University, Pittsburgh, PA, 15213, USA; emails: {dlanders, guyb}@cs.cmu.edu.

2329-4949/2023/12-ART18 $15.00
https://doi.org/10.1145/3565557

Until very recently, these were the state-of-the-art sequential and parallel algorithms for the weighted minimum cut problem. A new wave of interest in the problem has recently pushed these frontiers. Geissmann and Gianinazzi [13] design a parallel algorithm for minimum 2-respecting cuts that performs $O(m \log^3 n)$ work in $O(\log^2 n)$ depth. Their algorithm is based on parallelizing Karger's algorithm by replacing a sequential data structure for the so-called *minimum path* problem, based on dynamic trees, with a data structure that can evaluate a *batch* of updates and queries in parallel. Their algorithm performs just a factor of $O(\log n)$ more work than Karger's sequential algorithm, but substantially improves on the work of Karger's parallel algorithm.

Soon after, a breakthrough from Gawrychowski, Mozes, and Weimann [11] gave a randomized $O(m \log^2 n)$ algorithm for minimum cut. Their algorithm achieves the $O(\log n)$ speedup by designing an $O(m \log n)$ algorithm for finding the minimum 2-respecting cuts, which was the bottleneck of Karger's algorithm. This is the first result to beat Karger's seminal algorithm in over 20 years.

An open question posed by Karger was whether a deterministic algorithm can achieve an $O\left(m^{1+o(1)}\right)$ runtime. This was recently resolved in the affirmative by Li [27] by derandomizing the construction of the spanning trees.

In our work, we combine ideas from Gawrychowski et al. and Geissmann and Gianinazzi with several new techniques to close the gap between the parallel and sequential algorithms. Our contribution can be summarized by:

Theorem 1.1. *The minimum cut of a weighted graph can be computed with high probability in $O(m \log^2 n)$ work and $O(\log^3 n)$ depth.*

We achieve this using a combination of results that may be of independent interest. First, we design a framework for evaluating mixed batches of updates and queries on trees work efficiently in low depth. This algorithm is based on parallel **Rake-Compress Trees (RC trees)** [1]. Roughly, we say that a set of update and query operations implemented on an RC tree is *simple* (defined formally in Section 3) if the updates maintain values at the leaves that are modified by an associative operation and combined at the internal nodes, and the queries read only the nodes on a root-to-leaf path and their children. Simple operation sets include updates and queries on path and subtree weights.

Theorem 1.2. *Given a bounded-degree RC tree of size n and a simple operation set, after $O(n)$ work and $O(\log n)$ depth preprocessing, batches of k operations from the operation-set, can be processed in $O(k \log(kn))$ work and $O(\log n \log k)$ depth. The total space required is $O(n + k_{max})$, where k_{max} is the maximum size of a batch.*

This result generalizes and improves on Geissmann and Gianinazzi [13] who give an algorithm for evaluating a batch of k path-weight updates and queries in $\Omega(k \log^2 n)$ work.

Next, we design a faster parallel algorithm for approximating minimum cuts, which is used as an ingredient in producing the tree packing used in Karger's approach (Section 4). To achieve this, we design a faster sampling scheme for producing graph skeletons, leveraging recent results on sampling binomial random variables, and a transformation that reduces the maximum edge weight of the graph to $O(m \log n)$ while approximately preserving cuts.

Last, we show how to solve the minimum 2-respecting cut problem efficiently in parallel, using a combination of our new mixed batch tree operations algorithm and the use of RC trees to efficiently perform a divide-and-conquer search over the edges of the 2-constraining trees (Section 5).

Theorem 1.3. *The minimum 2-respecting cut of a weighted graph with respect to a given spanning tree can be computed in $O(m \log n)$ work and $O(\log^3 n)$ depth with high probability.*

Application to the unweighted problem. The unweighted minimum cut problem, or edge connectivity problem, was recently improved by Ghafarri, Nowicki, and Thorup [15] who give an

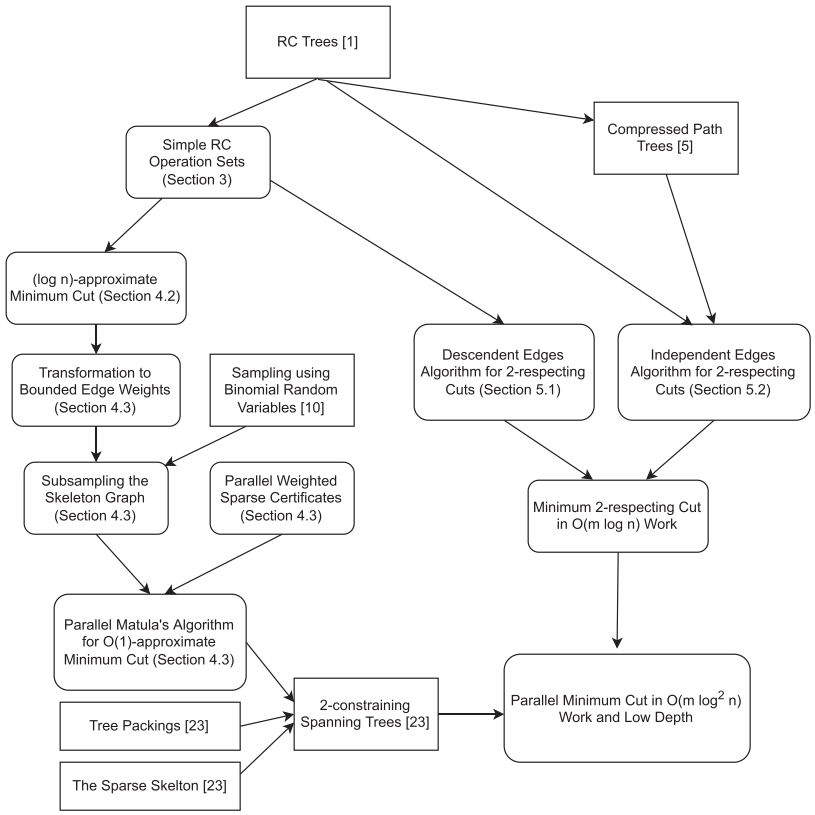


Fig. 1. An overview of the components of the algorithm and their dependencies. Square boxes are existing works, and rounded boxes are new results in this work.

$O(m \log n + n \log^4 n)$ work and $O(\text{polylog}\, n)$ depth randomized algorithm that uses Geissmann and Gianinazzi's algorithm as a subroutine. By plugging our improved algorithm into Ghafarri, Nowicki, and Thorup's algorithm, we obtain an algorithm that runs in $O(m \log n + n \log^2 n)$ work and $O(\text{polylog}\, n)$ depth **with high probability (w.h.p.).**

Overview of components. Our results are built upon numerous components that are novel, pre-existing, or combinations thereof. Figure 1 depicts the components of the algorithm as a flowchart. The core component is RC trees [1], from which we derive the framework of simple RC operation sets (Section 3) and batched-mixed operations on trees. This is used in multiple subsequent components of the algorithm. Table 1 shows the work bound for each of these components and compares them to existing work where relevant.

2 PRELIMINARIES

Model of computation. We analyze algorithms in the *work-depth* model using fork-join parallelism. A procedure can *fork* another procedure call to run in parallel and then wait for forked procedures to complete with a *join*. Work is defined as the total number of instructions performed by the algorithm and depth (also called span) is the length of the longest chain of sequentially dependent instructions [5]. The model can work-efficiently cross simulate the classic CRCW

Table 1. Summary of the Work Bounds of the Various Components of the Algorithm and Comments Comparing to Existing Work

Component	Work	Comments
Simple RC Operation Sets (Section 3)	$O(k \log(kn))$	A framework for evaluating queries on weighted trees
Minimum 2-respecting Cuts		
Descendant edges case (Section 5.1)	$O(m \log n)$	Improves on $O(m \log^3 n)$ in Reference [11]
Independent edges case (Section 5.2)	$O(m \log n)$	Improves on $O(m \log^3 n)$ in Reference [11]
Generating the 2-constraining spanning trees		
($\log n$)-approximate min cut (Section 4.2)	$O(m \log^2 n)$	Parallelizes ideas from Reference [21] using simple RC ops
Bounded edge weights (Section 4.3)	$O(m)$	Bounds weights by $O(m \log n)$ preserving an $O(1)$-min-cut
Subsampling the skeleton (Section 4.3)	$O(m \log^2 n)$	Samples $\log n$ skeleton graphs faster than $O(m \log^3 n)$ [23]
Parallel k-certificate (Section 4.3)	$O(km)$	Extends the algorithm of Reference [7] for weighted graphs
Parallel Matula's algorithm (Section 4.3)	$O(dm \log(W/m))$	Extends the algorithm of Reference [25] for weighted graphs

All components have $O(\text{polylog } n)$ depth.

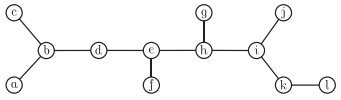
PRAM model [5], and the more recent Binary Forking model [6] with at most a logarithmic-factor difference in the depth.

Randomness. We say that a statement happens w.h.p. in n if for any constant c, the constants in the statement can be set such that the probability that the event fails to hold is $O(n^{-c})$. In line with Karger's work on random sampling [22], we assume that we can generate $O(1)$ random bits in $O(1)$ time. Since some of the subroutines we use require random $\Theta(\log n)$-bit words, these take $O(\log n)$ work to generate. The depth is unaffected, since we can always pre-generate the anticipated number of random words in parallel at the beginning of our algorithms.

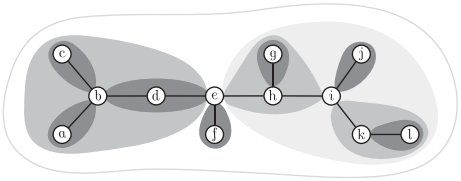
Our algorithms are Monte Carlo, i.e., correct w.h.p. but run in a deterministic amount of time. We can use Las Vegas algorithms, which are fast w.h.p. but always correct, as subroutines, because any Las Vegas algorithm can be converted into a Monte Carlo algorithm by halting and returning an arbitrary answer after the desired time.

Tree contraction. Parallel tree contraction is a technique developed to efficiently apply various operations over trees in logarithmic parallel depth [30], and was also later applied to dynamic trees [2]. Tree contration consists of a set of rake and compress operations. The *rake* operation removes a leaf vertex and merges it with its parent. The *compress* operation removes a vertex of degree two and replaces its two incident edges with a single edge joining its neighbors. Miller and Reif [30] observed that rakes and compresses can be applied in parallel as long as they are applied to an independent set of vertices. They describe a random-mate technique that ensures that any tree contracts to a single vertex in $O(\log n)$ rounds w.h.p., and using a total of $O(n)$ work in expectation. Gazit, Miller, and Teng [12] give a deterministic version with the same bounds, and Blelloch et al. [6] give a version that works in the binary-forking model. Miller and Reif's algorithm applies to bounded-degree trees, but arbitrary-degree trees can typically be handled by converting them into bounded-degree trees. For a rooted tree, the root is never removed, and is the final surviving vertex.

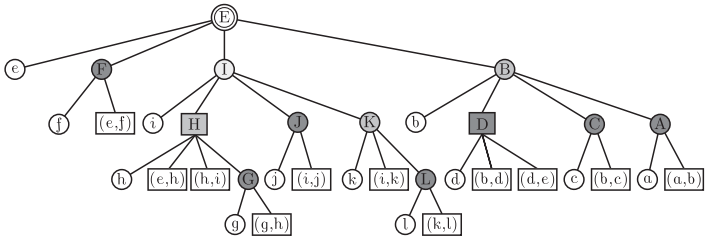
Rake-compress trees. The RC tree [1, 2] of a tree T encodes a recursive clustering of T corresponding to the result of tree contraction, where each cluster corresponds to a rake or compress (see Figure 2). A cluster is defined to be a connected subset of vertices and edges of the original tree. Importantly, a cluster can contain an edge without containing its endpoints. The *boundary vertices* of a cluster C are the vertices $v \notin C$ such that an edge $e \in C$ has v as one of its endpoints. All of the clusters in an RC tree have at most two boundary vertices. A cluster with no boundary vertices is called a *nullary cluster* (generated at the top-level *root* cluster), a cluster with one boundary is a



(a) A tree



(b) A recursive clustering of the tree produced by tree contraction. Clusters produced in earlier rounds are depicted in a darker color.



(c) The corresponding RC tree. Unary clusters (from rakes) are shown as filled circles, binary clusters as rectangles, and the finalize (nullary) cluster at the root with two concentric circles. The leaf clusters are labeled in lowercase, and the composite clusters are labeled with the uppercase of their representative. The shade of a cluster corresponds to its height in the clustering. Lower heights (i.e., contracted earlier) are darker.

Fig. 2. A tree, a clustering, and the corresponding RC tree [1].

unary cluster (generated by the rake operation) and a cluster with two boundaries is *binary cluster* (generated by the compress operation). The *cluster path* of a binary cluster is the path in T between its boundary vertices. Nodes in an RC tree correspond to clusters, such that a node is the disjoint union of its children.

The leaf clusters of the RC tree are the vertices and edges of the original tree, which are nullary and binary clusters, respectively. Note that all non-leaf clusters have exactly one vertex (leaf) cluster as a child. This vertex is that cluster's *representative* vertex. The recursive clustering is then defined by the following simple rule: Each rake or compress operation corresponds to a cluster, such that the operation that deletes vertex v from the tree defines a cluster with representative vertex v whose non-leaf subclusters are all of the clusters that have v as a boundary vertex. Clusters therefore have the useful property that the constituent clusters of a parent cluster C share a single boundary vertex in common—the representative of C, and their remaining boundary vertices become the boundary vertices of C.

In this article, we will be considering rooted trees. In this case the root of the tree is also the representative of the top level nullary cluster of the RC-tree, e.g., vertex e in Figure 2. Non-leaf binary clusters have a binary subcluster whose cluster path is above the representative vertex in the input tree, which we will refer to as the *top cluster*, and a binary subcluster whose cluster path is below the representative vertex, which we call the *bottom cluster*. We will also refer to the binary subcluster of a unary cluster as the top cluster as its cluster path is also above the representative vertex. In our pseudocode, we will use the following notation. For a cluster x: $x.v$ is the representative vertex, $x.t$ is the top subcluster, $x.b$ is the bottom subcluster, $x.U$ is a list of unary subclusters, and $x.p$ is the parent cluster.

Compressed path trees. For a weighted (unrooted) tree T and a set of *marked* vertices $V \subset V(T)$, the compressed path tree is a weighted tree T_c on some subset of the vertices of T including V with the following property: for every pair of vertices $(u, v) \in V \times V$, the weight of the lightest edge on the path from u to v is the same in T and T_c. The compressed path three T_c is defined as the smallest such tree. Alternatively, the compressed path tree is the tree T with all unmarked vertices of degree less than three spliced out, where each spliced-out path is replaced by an edge whose weight is the lightest of the weights on the path it replaced. It is not hard to show that T_c has size less than $2|V|$. Compressed path trees are described in Reference [4], where it is shown that given an RC tree for the tree T and a set of k marked vertices, the compressed path tree can be produced in $O(k \log(1 + n/k))$ work and $O(\log^2 n)$ depth w.h.p. Gawrychowski et al. [11] define a similar notion which they call "topologically induced trees," but their algorithm is sequential and requires $O(k \log n)$ work (time).

Karger's minimum cut algorithm. Karger's algorithm for minimum cuts [23] is based on the notion of *k-respecting cuts*. Karger's algorithm is the following two-step process.

(1) Find $O(\log n)$ spanning trees of G such that w.h.p., the minimum cut 2-respects at least one of them
(2) Find, for each of the aforementioned spanning trees, the minimum 2-respecting cut in G

Karger solves the first step using a combination of random sampling and *tree packing*. Given a weighted graph G, a tree packing of G is a set of weighted spanning trees of G such that for each edge in G, its total weight across all of the spanning trees is no more than its weight in G. The underlying tree packing algorithms used by Karger have running time proportional to the size of the minimum cut, so random sampling is first used to produce a sparsified graph, or *skeleton*, where the minimum cut has size $\Theta(\log n)$ w.h.p. The sampling process is carefully crafted such that the resulting tree packing still has the desired property w.h.p.

Given the skeleton graph, Karger gives two algorithms for producing tree packings such that sampling $\Theta(\log n)$ trees from them guarantees that, w.h.p., the minimum cut 2-respects one of them. The first approach uses a tree packing algorithm of Gabow [10]. The second is based on the packing algorithm of Plotkin et al. [34], and is much more amenable to parallelism. It works by performing $O(\log^2 n)$ minimum spanning tree computations. In total, Step 1 of the algorithm takes $O(m + n \log^3 n)$ time.

For the second step, Karger develops an algorithm to find, given a graph G and a spanning tree T, the minimum cut of G that 2-respects T. The algorithm works by arbitrarily rooting the tree, and considering two cases: when the two cut edges are on the same root-to-leaf path, and when they are not. Both cases use a similar technique; they consider each edge e in the tree and try to find the best matching e' to minimize the weight of the cut induced by the edges $\{e, e'\}$. This is achieved by using a dynamic tree data structure to maintain, for each candidate e', the value that the cut would have if e' were selected as the second cutting edge, while iterating over the possibilities of e and updating the dynamic tree. Karger shows that this step can be implemented sequentially in $O(m \log^2 n)$ time, which results in a total runtime of $O(m \log^3 n)$ when applied to the $O(\log n)$ spanning trees.

3 BATCHED MIXED OPERATIONS ON TREES

The batched mixed operation problem is to take an off-line sequence of mixed operations on a data structure, usually a mix of queries and updates, and process them as a batch. The primary reason for batch processing is to allow for parallelism on what would otherwise be a sequential execution of the operations. We use the term *operation-set* to refer to the set of operations that can be applied among the mixed operations. We are interested in operations on trees, and our results

apply to operation-sets that can be implemented on an RC tree in a particular way, defined as follows.

Definition 3.1. An implementation of an operation-set on trees is a *simple RC implementation* if it uses an RC representation of the trees and satisfies the following conditions.

(1) The implementation maintains a value at every RC cluster that can be calculated in constant time from the values of the children of the cluster,
(2) every query operation is implemented by traversing from a leaf to the root examining values at the visited clusters and their children taking contant time per value examined, and using constant space, and
(3) every update operation involves updating the value of a leaf using an associative constant-time operation, and then reevaluating the values on each cluster on the path from the leaf to the root.

Note that every operation has an *associated leaf* (either an edge or vertex). Also note that setting (i.e., overwriting) a value is an associative operation (just return the second of the arguments). For simple RC implementations, all operations take time (work) proportional to the depth of the RC tree, since they only follow a path to the root taking constant time at each cluster. Although the simple RC restriction may seem contrived, most operations on trees studied in previous work [2, 3, 37] can be implemented in this form, including most path and subtree operations. This is because of a useful property of RC trees, that all paths and subtrees in the source tree can be decomposed into clusters that are children of a single path in the RC tree, and typically operations need just update or collect a contribution from each such cluster.

Example. As an example, consider the following two operations on a rooted tree (the first an update, and the second a query):

- addWeight(v, w): adds weight w to a vertex v
- subtreeSum(v): returns the sum of the weights of all of the vertices in the subtree rooted at v

ALGORITHM 1: The subtreeSum query.

```
1: procedure subtreeSum(v : vertex)
2:     w ← 0
3:     x ← v; p ← x.p
4:     while p is a binary cluster do
5:         if (x = p.t) or (x = p.v) then
6:             w ← w + p.b.w + p.v.w + Σ_{u∈p.U} u.w
7:         x ← p; p ← x.p
8:     return w + p.v.w + Σ_{u∈p.U} u.w
```

These operations can use a simple RC implementation by keeping as the value of each cluster the sum of values of all its children. This satisfies the first condition, since the sums take constant time. Single-edge clusters in the RC tree start with the initial weight of the edge, while single-vertex clusters start with zero weight. An addWeight(v, w) adds weight w to the vertex v (which is a leaf in the RC tree) and updates the sums up to the root cluster. This satisfies the third condition, since addition is associative and takes constant time. The query can be implemented as in Algorithm 1, where $x.w$ is the weight stored on the cluster x. It starts at the leaf for v and goes up the RC tree keeping track of the total weight underneath v. Note that x will never be a unary cluster, so if not

the representative or top subcluster of p, it is the bottom subcluster with nothing below it in this cluster. Observe that SUBTREESUM only examines values on a path from the start vertex to the root and the children along that path. Each step takes constant time and requires constant space, satisfying the second condition. The operation-set therefore has a simple RC implementation.

3.1 Batched Mixed Operations Algorithm

We are interested in evaluating batches of operations from an operation-set on trees with a simple RC implementation. In particular, we prove Theorem 1.2. Our algorithm is similar to that of Geissmann and Gianinazzi [13], except we work with a cluster-based decomposition of trees based on RC trees instead of their path-based decomposition based on heavy-light decomposition.

PROOF SKETCH OF THEOREM 1.2. The preprocessing just builds an RC tree on the source tree, and sets the values for each cluster based on the initial values on the leaves. This can be implemented with the Miller-Reif algorithm [30], in the binary forking model [6], or deterministically [12]. All take linear work and logarithmic depth (w.h.p. for the randomized versions). Our algorithm for each batch is then implemented as follows:

(1) Timestamp the operations by their order in the sequence.
(2) Collect all operations by their associated leaf, and sort within each leaf by timestamp. This can be implemented with a single sort on the leaf identifier and timestamp.
(3) For each leaf use a prefix sum on the update values to calculate the value of the leaf after each operation, starting from the initial value on the leaf.
(4) Initialize each query using the value it received from the prefix sum. We now have a list of operations on each leaf sorted by timestamp. For each update we have its value, and for each query we also have its partial evaluation based on the value. We prepend the initial value to the list, and call this the *operation list*. An operation list is *non-trivial* if it has more than just the initial value.
(5) For each level of the RC tree starting one above the deepest, and in parallel for every cluster on the level for which at least one child has a non-trivial operation list:
   (a) Merge the operation lists from each child into a single list sorted by timestamp.
   (b) Calculate for each element in the merged operations list, the latest value of each child at or before the timestamp. This can be implemented by prefix sums.
   (c) For each list element, calculate the value at that timestamp from the child values collected in the previous step.
   (d) For queries, use the values and/or child values to update the query.

This algorithm needs to have children with non-trivial operation lists identify parents that need to be processed. This can be implemented by keeping a list of all the clusters at a level with non-trivial operation lists left-to-right in level order. When moving up a level, clusters that share the same parent can be combined. An illustration of the merging process is depicted in Figure 3 using the operations from Algorithm 1.

We first consider why the algorithm is correct. We assume by structural induction (over subtrees) that the operation lists contain the correct values for each timestamped operation in the list. This is true at the leaves, since we apply a prefix sum across the associative operation to calculate the value at each update. For internal clusters, assuming the child clusters have correct operation lists (values for each timestamp valid until the next timestamp, and partial result of queries), we properly determine the operation lists for the cluster. In particular for all timestamps that appear in children we promote them to the parent, and for each we calculate the value based on the current value, by timestamp, for each child.

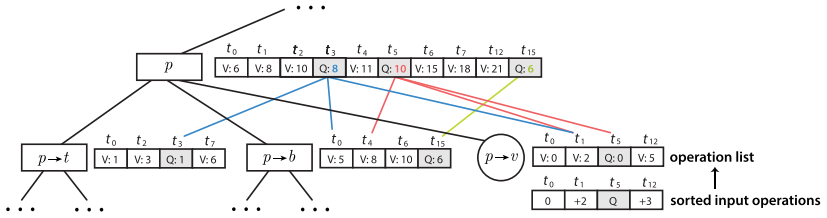


Fig. 3. Merging the operation lists for a binary cluster consisting of AddWeight and subtreeSum operations. Values in the operation sequence, denoted $V : v$, are computed by aggregating the latest values of the children at the given timestamp. For example, at t_6 in p, the algorithm adds 3 from $p.t$ at t_2, 10 from $p.b$ at t_6, and 2 from $p.v$ at t_1. Queries, denoted $Q : q$, are updated at each level by using the latest values of the children. For example, to update the query at t_3, it takes the current value of 1 from $p.t$ at t_3, then adds the weight of 5 from $p.b$ at t_0, and the weight of 2 from $p.v$ at t_1, as per Algorithm 1, since the conditional on Line 5 is true. Similarly, to update the query at t_5, the conditional is also true, and the most recent timestamps in $p.b$ and $p.v$ are t_4 and t_1, so it accumulates those values.

We now consider the costs. The cost of the batch before processing the levels is dominated by the sort that takes $O(k \log k)$ work and $O(\log k)$ depth. The cost at each level is then dominated by the merging and prefix sums that take $O(k)$ work and $O(\log k)$ depth accumulated across all clusters that have a child with a non-trivial operation list. If the RC tree has depth $O(\log n)$, then across all levels the cost is bounded by $O(k \log n)$ work and $O(\log n \log k)$ depth. The total work and depth is therefore as stated. The space for each batch of size k is bounded by the size of the RC tree, which is $O(n)$, and the total space of the operation lists at any two adjacent levels, which is $O(k)$. □

3.2 Path Updates and Path/Subtree Queries

We now consider implementing mixed operations consisting of updating paths, and querying both paths and subtrees. We will use these in Sections 3.3 and 5. In particular we wish to maintain, given a weighted rooted tree $T = (V, E)$, a data structure that supports the following operations.

- AddPath(u, v, w): For $u, v \in V$ adds w to the weight of all edges on the u to v path.
- QuerySubtree(v): Returns the lightest weight of an edge in the subtree rooted at $v \in V$,
- QueryPath(u, v): For $u, v \in V$, returns the lightest weight of an edge on the u to v path.
- QueryEdge(e): Returns $w(e)$

To implement these, we first implement the simpler operations AddPath'(v, w), which adds weight w to the path from v to the root; and QueryPath'(u, v), which requires that v be the representative vertex of an ancestor of u in the RC tree. The more general forms can be implemented in terms of these with a constant number of calls given the **lowest common ancestor (LCA)** in the original tree for AddPath and in the RC tree for QueryPath.

Lemma 3.2. *The AddPath', QuerySubtree, QueryPath', and QueryEdge operations on bounded degree trees can be supported with a simple RC implementation.*

Proof sketch. Our simple RC implementation for combining values and AddPath' is given in Algorithm 2. The queries are given in Algorithm 3. The value of each vertex (leaf) in the cluster is the total weight added to that vertex by AddPath'. The value for each unary cluster consists of: m, the minimum weight edge in the cluster; and w, the total weight of AddPaths' originating in the cluster. For each binary cluster, we separate the minimum weights on and off the cluster path. In particular, the value of each binary cluster consists of: m, the minimum weight edge not

ALGORITHM 2: A simple RC implementation of AddPath'.

```
1: using VertexV = int
2: using UnaryV = struct { m : edge, w : int }
3: using BinaryV = struct { m : edge, l : edge, w : int }
4: procedure f_UNARY(w_v : VertexV, (m_t, l_t, w_t) : BinaryV, U : UnaryV list)
5:     w' ← w_v + Σ_{u∈U} u.w
6:     m_u ← min_{u∈U} u.m
7:     return { min(m_t, l_t + w', m_u), w_t + w' }
8: procedure f_BINARY(w_v : VertexV, (m_t, l_t, w_t) : BinaryV, (m_b, l_b, w_b) : BinaryV, U : UnaryV list)
9:     w' ← w_v + w_b + Σ_{u∈U} u.w
10:    m_u ← min_{u∈U} u.m
11:    return { min(m_t, m_b, m_u), min(l_t + w', l_b), w_t + w' }
12: procedure AddPath'(v : vertex, w : int)
13:    v.value ← v.value + w
14:    Reevaluate the f(·) on path to root.
```

on the cluster path; l, the minimum edge on the cluster path due to all AddPath' originating in the cluster; and w, the total weight of AddPaths' originating in the cluster. The f_{binary} and f_{unary} calculate the values for unary and binary clusters from the values of their children. We initialize each vertex with zero, and each edge e with $(m = 0, l = w(e), w = 0)$.

It is a simple RC implementation, since (1) the $f(\cdot)$ can be computed in constant time, (2) the queries just traverse from a leaf on a path to the root (possibly ending early) only examining child values, taking constant time per level and constant space, and (3) the update just sets a leaf using an associative addition, and reevaluates the values to the root.

We argue the implementation is correct. First, we argue by structural induction on the RC tree that the values as described in the previous paragraph are maintained correctly by f_{binary} and f_{unary}. In particular, assuming the children are correct, we show the parent is correct. The values are correct for leaves, since we increment the value on vertices with AddPath', and initialize the edges appropriately. To calculate the minimum edge weight of a unary cluster f_{unary} takes the minimum of three quantities: the minimum off-path edge of the child binary cluster, the overall minimum edge of any of the child unary clusters, and, importantly, the minimum edge on the cluster path of the child binary cluster plus the AddPath' weight contributed by the unary clusters and the representative vertex (i.e., $\min(m_t, l_t + w', m_u)$). This is correct, since all paths from those clusters to the root go through the cluster path, so it needs to be adjusted. The off-path edges and child unary clusters do not need to be adjusted, since no path from the representative vertex goes through them. The minimum weight is therefore correct. The total AddPath' weight is correct, since it just adds the contributions.

For binary clusters, we need to separately consider the minimum off- and on-path edges. For the off-path edges the parts that are off the cluster path are the off-path edges from the two binary children, plus all edges from the unary children (i.e., $\min(m_t, m_b, m_u)$). For the on-path edges both the top and bottom binary clusters contribute their on-path edges. The on-path edges from the bottom binary cluster do not need to be adjusted, because no vertices in the cluster are below them. The on-path edges from the top binary cluster need to be adjusted by the AddPath' weights from all vertices in the bottom cluster, all vertices in unary child clusters, and the representative vertex, since they are all below the path (this sum is given by w'); see Figure 4. The minimum of the resulted adjusted top edge and bottom edge is then returned, which is indeed the minimum edge on the path accounting for AddPaths' on vertices in the cluster.

ALGORITHM 3: A simple RC implementation of QueryEdge, QueryPath', and QuerySubtree.

```
1:  procedure QuerySubtree(v : vertex)  // Returns the lightest weight of an edge in the subtree rooted at v
2:    m ← ∞; l ← ∞                      // m: min edge not on the cluster path, l: min edge on the cluster path so far
3:    x ← v; p ← x.p
4:    while p is a binary cluster do    // Accumulate weights until we reach a unary cluster
5:      if (x = p.t) or (x = p.v) then  // If x is in the top half of the cluster, then all
6:        w' ← p.b.w + p.v.w + Σ_{u∈p.U} u.w   //  AddPaths originating below it will add to
7:        l ← min(l + w', p.b.l)        // the weight of all edges on the cluster path
8:        m ← min(m, p.b.m, min_{u∈p.U} u.m)
9:      x ← p; p ← x.p
10:   w' ← p.v.w + Σ_{u∈p.U} u.w
11:   return min(l + w', m, min_{u∈p.U} u.m)   // The lightest weight edge is either on the cluster path or not
12: procedure QueryEdge(e : edge)       // Returns the weight of the edge e
13:   w ← w(e)
14:   x ← e; p ← x.p
15:   while p is a binary cluster do
16:     if x = p.t then                 // if e is on the cluster path of the top half of the
17:       w ← w + p.b.w + p.v.w + Σ_{u∈p.U} u.w   // current cluster, then all AddPaths originating
18:     x ← p; p ← x.p                  // below the top cluster will add to the weight of e
19:   return w + p.v.w + Σ_{u∈p.U} u.w
20: procedure QueryPath'(u : vertex, v : vertex)   // Returns the lightest edge on the path from u to v
21:   m ← ∞; t ← ∞; b ← ∞               // such that v is the representative of an ancestor of u in the RC tree
22:   x ← u; p ← x.p
23:   while not p.v = v do
24:     w' ← p.v.w + Σ_{u∈p.U} u.w
25:     if p is a unary cluster then    // When p is a unary cluster, and u originated in the
26:       if x = p.t then m ← min(t + w', m)   // top subcluster, the weight of all AddPaths below
27:       else m ← min(p.t.l + w', m)   // is added to the edges between u and the boundary,
28:       t ← ∞; b ← ∞                  // otherwise it is added to all edges on the top cluster path
29:     else
30:       w' ← w' + p.b.w               // The weight of all AddPaths below is added to the top cluster path.
31:       if x = p.t then t ← t + w'; b ← min(b + w', p.b.l) // If u originated in the top subcluster, then this
32:       else if x = p.b then t ← min(p.t.l + w', t)   // affects both t and b. If u originated in the
33:       else t ← p.t.l + w'; b ← p.b.l   // bottom subcluster, then it affects only t. Otherwise, u
34:     x ← p; p ← x.p                  // originated in a unary subcluster so x is not in the cluster path.
35:   if x = p.t then l ← b
36:   else if x = p.b then l ← t        // u either connects to v in the direction of the top boundary of p (a
37:   else return m                     //  weight of t), the bottom boundary of p (a weight of b) or neither (m)
38:   while p is a binary cluster do
39:     w' ← p.v.w + p.b.w + Σ_{u∈p.U} u.w   // There might still be more AddPath operations below, so we
40:     if (x = p.t) then l ← l + w'    // continue up the RC tree to accumulate any that remain.
41:   return min(m, l)
```

QuerySubtree(v) accumulates the appropriate minimum weights within a subtree as it goes up the RC tree. It starts at the node for which v is its representative vertex. As with the calculation of values it needs to separate the on-path and off-path minimum weight. Whenever coming as the upper binary cluster to the parent, QuerySubtree needs to add all the contributing AddPath' weights from vertices below it in the parent cluster (the representative vertex, the lower binary

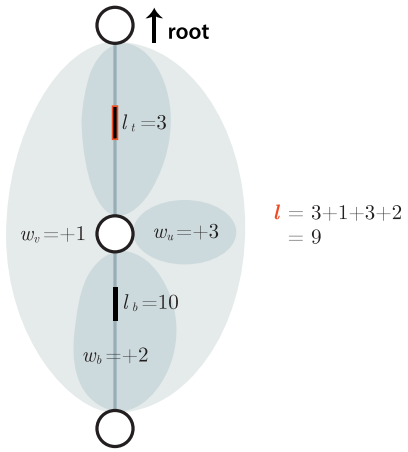


Fig. 4. When a binary cluster joins its children, all AddPaths' that originated in the vertex, bottom, or unary subclusters will affect all of the edges in the top cluster path. Here, $w' = w_v + w_b + w_u = 6$ weight is added to edges on the top cluster path due to AddPath operations from below. The minimum weight edge on the cluster path is therefore $\min(l_t + w', l_b) = \min(3 + 6, 10) = 9$, which is the edge from the top cluster path, highlighted in red.

cluster, and the unary clusters, see Figure 4) to the current minimum on-path weight. A minimum is then taken with the lower on-path minimum edge to calculate the new minimum on-path edge weight (Line 7). The off-path minimum is the minimum of the current off-path minimum, the minimum off-path edge of the bottom cluster and the minimums of the unary clusters (Line 8). Once we reach a unary cluster, we are done, since for a unary cluster all subtrees of vertices within the cluster are fully contained within the cluster. The final line therefore just determines the overal minimum for the subtree rooted at v by considering the on-path edges adjusted by AddPath' contributions, the off-path edges, and all edges in child unary clusters.

QueryEdge(e) simply adds the total weight of all AddPath' operations that occurred beneath e to the weight of e. Specifically, at each iteration of the loop, w contains the $w(e)$ plus the total weight of all AddPath' operations originating at any vertex below e that is contained in the current cluster x. As the query moves up the RC tree, if the parent cluster is a binary cluster and x is its top subcluster, then the vertices not yet accounted for are those in the bottom subcluster, the representative vertex, and the unary subclusters. If x is the bottom subcluster of its binary parent, or one of its unary subclusters, then no vertices in p but not x are below e. When the while loop terminates, p is a unary cluster and x is its binary subcluster. At this point, the representative of p, and all unary subclusters of p are below e, and hence their weight is added to the total. Since p is a unary cluster, there exists no additional vertices below e in the tree, and hence the final weight contains the contributions of all AddPath' operations originating below e.

Last, QueryPath' works by maintaining three values, m, t, b. To make defining them easier, consider, at each iteration of the main loop (Lines 23–34) in which the current cluster x is a binary cluster, the vertex c, which is the closest vertex to u on the cluster path of x (if u is on the cluster path of x, say $c = u$). Then, we can define m as the minimum weight edge on the path from u to c (which will be ∞ if u is on the cluster path of x), t as the minimum weight edge above c on the cluster path of x, and b as the minimum weight edge below c on the cluster path of x. If x is a unary cluster, then t and b are ∞ (undefined), and m is simply the minimum weight edge on

the path from u to the boundary of x. Observe that it is important for the algorithm to maintain both t and b, because it does not know in advance whether v is above or below the current cluster path. It remains to argue that the implementation correctly maintains these values, and that the postprocessing is correct.

Each time the algorithm moves up to the next highest cluster, it first computes w', the total weight of all AddPath' operations originating below the representative vertex. If the cluster is a unary cluster, and u originated from the top (binary) subcluster, then the path from u to the boundary of p consists of the previous path from u to c (the lightest edge on which is m), and the path from c to the boundary of p (the lightest edge on which is t). Since w' weight has been added to all edges on the path from c to the boundary of p, the lightest such edge is now $t + w'$ and hence the lightest edge on the path from u to the boundary of p is $\min(t + w', m)$. If u did not originate in the top subcluster of p, then it came from one of the unary subclusters. In this case, the path from u to the boundary of p consists of the path from u to the boundary of x, and the cluster path of the top subcluster (which begins at the boundary of x and ends at the boundary of p), and hence the lightest edge is $\min(p.t.l + w', m)$. Since the current cluster is a unary cluster, t and b are undefined (Line 28).

If the next cluster is a binary cluster, then we reason as follows. If u originated in the top subcluster, then the path from c to the top boundary remains the same, but w' weight is added to every edge (including t). The cluster path below c now consists of the edges previously below c to the bottom boundary of x, and additionally those on the cluster path of the bottom subcluster (the edges from the bottom boundary of x to the bottom boundary of p). The edges below c on the cluster path of the top subcluster (including b) have had their weight increased by w', and hence the lightest edge on the path from c to the bottom boundary of p is now $\min(b + w', p.b.l)$. Similarly, if u originated in the bottom subcluster, then the path from c to the bottom boundary hasn't changed, so b is unchanged, and no weight is added to the edge t. However, since the path from c to the top boundary of p now includes the cluster path of the top subcluster, the lightest edge from c to the top boundary is now $\min(p.t.l + w', t)$. Otherwise, u must have originated from a unary subcluster of p, and hence the cluster path of p contains no edges from x, so t is simply the lightest edge in the top subcluster, and b is the lightest edge in the bottom subcluster.

Once the main loop terminates (Lines 23–34), by the loop condition, it must be because the current cluster x has v as a boundary. If u originated in the top subcluster of the latest p, then v must be the bottom boundary of p, and hence the path from u to v consists of the path from u to c and the path from c to v, which goes toward the bottom boundary of p and hence contains b. Conversely, if u originated in the bottom subcluster of p, then the path from u to v goes toward the top boundary of p and hence contains t. If u originated in a unary subcluster, then the path from u to v just joins u to the boundary of x, hence the lightest edge is m. If not, then the lightest edge is either m, or b or t, respectively. The weight of b or t might still be affected by AddPath' operations from below, so the total weight of such operations is accumulated by continuing up the RC tree and added to determine the final weight. □

Corollary 3.3. *Given a bounded-degree tree of size n, any sequence of k AddPath, QuerySubtree, QueryPath, and QueryEdge operations can be evaluated in $O(n + k \log(nk))$ work, $O(\log n \log k)$ depth and $O(n + k)$ space.*

Proof. The LCAs required to convert AddPath to AddPath' and QueryPath to QueryPath' can be computed in $O(n + m)$ work, $O(\log n)$ depth, and $O(n)$ space [36]. The rest follows from Theorem 1.2 and Lemma 3.2. □

3.3 Improving Previous Results

Using our batched mixed operations on trees algorithm, we can improve previous results on finding 2-respecting cuts. In particular, we can shave off a log factor in the work of Geissmann and Gianinazzi's parallel algorithm [13], and we can parallelise Lovett and Sandlund's sequential algorithm [28].

Geissmann and Gianinazzi find 2-respecting cuts by first finding an $O(m)$ sequence of mixed AddPath and QueryPath operations for each of $O(\log n)$ trees. They show how to find each sequence in $O(m \log n)$ work and $O(\log n)$ depth. On each set they then use their own data structure to evaluate the sequence in $O(m \log^2 n)$ work and $O(\log^2 n)$ depth, for a total of $O(m \log^3 n)$ work and $O(\log^2 n)$ depth across the sets. Replacing their data structure with the result of Corollary 3.3 improves their results to $O(m \log^2 n)$ work.

Lovett and Sandlund significantly simplify Karger's algorithm by first finding a heavy-light decomposition—i.e., a vertex disjoint set of paths in a tree such that every path in the tree is covered by at most $O(\log n)$ of them. It then reduces finding the 2-respecting cuts to a sequence of AddPath and QueryPath operations on the decomposed paths induced by each non-tree edge, for a total of $O(m \log n)$ operations. Using Geissmann and Gianinazzi's $O(n \log n)$ work $O(\log^2 n)$ algorithm for finding a heavy-light decomposition [13, Lemma 7], and the result of Corollary 3.3 again gives an $O(m \log^2 n)$ work, $O(\log^2 n)$ depth algorithm.

4 PRODUCING THE TREE PACKING

We follow the general approach used by Karger to produce a set of $O(\log n)$ spanning trees such that w.h.p., the minimum cut 2 respects at least one of them. We have to make several improvements to achieve our desired work and depth bounds. At a high level, Karger's algorithm works as follows.

(1) Compute an $O(1)$-approximate minimum cut c
(2) Sample edges from the unweighted multigraph corresponding to the weighted graph G, where an edge with weight w is represented as w parallel edges, with probability $\Theta(\log n/c)$
(3) Use the tree packing algorithm of Plotkin [34] to generate a packing of $O(\log n)$ trees

In this section, we describe the tools required to parallelise this algorithm.

4.1 A Parallel Version

Step 2 is trivial to parallelize, as the sampling can be done independently in parallel. The sampling procedure produces an unweighted multigraph with $O(m \log n)$ edges, and takes $O(m \log^2 n)$ work and $O(\log n)$ depth.

In Step 3, Plotkin's algorithm consists of $O(\log^2 n)$ **minimum spanning tree (MST)** computations on a weighting of the sampled graph, which has $O(m \log n)$ edges. Naively this would require $O(m \log^3 n)$ work, but we can use a trick of Gawrychowski et al. [11]. Since the sampled graph is a multigraph sampled from m edges, each invocation of the MST algorithm only cares about the current lightest of each parallel edge, which can be maintained in $O(1)$ time, since the weights of the selected edges change by a constant each iteration. Using Cole, Klein, and Tarjan's linear-work MST algorithm [8] results in a total of $O(m \log^2 n)$ work in $O(\log^3 n)$ depth w.h.p.

The only nontrivial part of parallelizing the tree production is actually Step 1, computing an $O(1)$-approximate minimum cut. In the sequential setting, Matula's algorithm [29] can be used, which runs in linear time on unweighted graphs, and on weighted graphs in $O(m \log^2 n)$ time. Karger and Motwani [25] give a parallel version of Matula's algorithm, but it takes $O(m^2/n)$ work. Ghaffari and Kuhn [14] present a distributed version of Matula's algorithm in the CONGEST model

that runs in $\tilde{O}((D + \sqrt{n})/\epsilon^5)$ rounds. We show how to compute an approximate minimum cut in $O(m \log^2 n)$ work and $O(\log^3 n)$ depth, which allows us to prove the following.

Theorem 4.1. *Given a weighted graph, in $O(m \log^2 n)$ work and $O(\log^3 n)$ depth, a set of $O(\log n)$ spanning trees can be produced such that the minimum cut 2-respects at least one of them w.h.p.*

We achieve our bounds by improving Karger's algorithms and speeding up several of the components. We use the following combination of ideas, new and old.

(1) We extend a k-approximation algorithm of Karger [21] to work in parallel, allowing us to produce a $\log n$-approximate minimum cut in low work and depth.
(2) We use a faster sampling technique for producing Karger's skeletons for weighted graphs. This is done by transforming the graph into a graph that maintains an approximate minimum cut but has edge weights each bounded by $O(m \log n)$, and then using binomial random variables to sample all of the multiedges of a particular edge at the same time, instead of separately. Subsampling is then used to sample the same graph with decreasing probabilities.
(3) We show that the parallel sparse k-certificate algorithm of Cheriyan, Kao, and Thurimella [7] for unweighted graphs can be modified to run on weighted graphs.
(4) We show that Karger and Motwani's parallelization of Matula's algorithm can be generalized to weighted graphs.
(5) We use the $\log n$-approximate minimum cut to allow the algorithm to make just $O(\log \log n)$ guesses of the minimum cut such that at least one of them is an $O(1)$ approximation.

4.2 Parallel $\log n$-approximate Minimum Cut

To compute an $O(1)$-approximate minimum cut, our first step is actually to compute a $\log n$-approximate minimum cut. We parallelize an algorithm of Karger for computing k-approximate minimum cuts that is efficient when $k = \Omega(\log n)$ [21].

Mixed incremental connectivity and component weight queries. The following ingredient is useful in parallelizing Karger's k-approximate minimum cut algorithm. We show that that the following operations have a simple RC implementation, and hence can be efficiently implemented. Given a vertex-weighted, undirected graph with given initial vertex weights, we wish to support:

- SubtractWeight(v, w): Subtract weight w from vertex v
- JoinEdge(e): Mark the edge e as "joined"
- QueryWeight(v): Return the weight of the connected component containing the vertex v, where the components are induced by the joined edges

Lemma 4.2. *The SubtractWeight, JoinEdge, and QueryWeight operations can be supported with a simple RC implementation.*

Proof sketch. The values stored in the RC clusters are as follows. Vertices store their weight, and unary clusters store the weight of the component reachable via joined edges from the boundary vertex. A binary cluster is either *joined*, meaning that its boundary vertices are connected by joined edges, in which case it stores a single value—the weight of the component reachable via joined edges from the boundaries; otherwise, it is *split*, in which case it stores a pair—the weight of the component reachable via joined edges from the top boundary, and the weight of the component reachable via joined edges from the bottom boundary. We provide pseudocode for the update operations for Illustration in Algorithm 4.
The initial value of a vertex is its starting weight. The initial value of an edge is $(0, 0)$, indicating that it is split at the beginning. Note that f_{unary} and f_{binary} can be evaluated in constant time, and

ALGORITHM 4: A simple RC implementation of SubtractWeight and JoinEdge.

```
1:  procedure f_UNARY(v_v, t, U)
2:      if t = (t_v, b_v) then return t_v
3:      else return v_v + t + Σ_{u_v ∈ U} u_v
4:  procedure f_BINARY(v_v, t, b, U)
5:      if t = t_v and b = b_v then
6:          return t_v + b_v + v_v + Σ_{u_v ∈ U} u_v
7:      else if t = (t_{t_v}, t_{b_v}) and b = b_v then
8:          return (t_{t_v}, t_{b_v} + v_v + b_v + Σ_{u_v ∈ U} u_v)
9:      else if t = t_v and b = (b_{t_v}, b_{b_v}) then
10:         return (t_v + v_v + b_{t_v} + Σ_{u_v ∈ U} u_v)
11:     else if t = (t_{t_v}, t_{b_v}) and b = (b_{t_v}, b_{b_v}) then
12:         return (t_{t_v}, b_{b_v})
13: procedure SubtractWeight(v, w)
14:     v.value ← v.value - w
15:     Reevaluate the f(·) on path to root.
16: procedure JoinEdge(e)
17:     e.value ← 0
18:     Reevaluate the f(·) on path to root.
```

the structure of the updates involves setting the value at a leaf using an associative operation and re-evaluating the values of the ancestor clusters.

We can argue that the values are correctly maintained by structural induction. First consider unary clusters. If the top subcluster is split, then the representative vertex and unary subclusters are not reachable via joined edges, and hence the only reachable component is the component reachable inside the top subcluster from its top boundary, whose weight is t_v. If the top subcluster is joined, then the representative vertex is reachable, which is by definition the boundary vertex of the unary subclusters, and hence the reachable component is the union of the reachable components of all of the subclusters, whose weight is as given.

For binary clusters, there are four possible cases, depending on whether the top and bottom subclusters are joined or not. If both are joined, then the representative and hence the boundary of all subclusters is reachable from both boundaries, and hence the cluster is joined and the reachable component is the union of the reachable components of the subclusters. If either subcluster is split, then the reachable component at the corresponding boundary is just the reachable component of the subcluster, whose weight is as given. Last, if one of the subclusters is not split, then the corresponding boundary can reach the representative vertex, and hence the reachable components of the unary subclusters, whose weights are as given.

It remains to argue that we can implement QueryWeight with a simple RC implementation. Consider a vertex v whose component weight is desired and consider the parent cluster P of v, i.e., the cluster of which v is the representative. If P has no binary subclusters that are joined, then observe that P must contain the entire component of v induced by joined edges, since the only way for a component to exit a cluster is via a boundary that would have to be joined. Answering the query in this situation is therefore easy; the result is the sum of the weights of v, the unary subclusters of P, the bottom boundary weight of the top subcluster (if it exists), and the top bounary weight of the bottom subcluster (if it exists). Suppose instead that P contains a binary subcluster that is joined to some boundary vertex $u \neq v$. Since the subcluster is joined, u is in the same induced component as v, and hence QueryWeight(v) has the same answer as QueryWeight(u). By standard properties of RC trees, since u is a boundary of P, we also know that the leaf cluster

u is the child of some ancestor of P. Since the root cluster has no binary subclusters, this process of jumping to joined boundaries must eventually discover a vertex that falls into the easy case, and since such a vertex u is always the child of some ancestor is P, the algorithm only examines clusters that are on or are children of the root-to-v path in the RC tree, and hence the algorithm is a simple RC implementation. □

Invoking Theorem 1.2, we obtain the following useful corollary.

Corollary 4.3. *Given a vertex-weighted undirected graph, a batch of k SubtractWeight, JoinEdge, and QueryWeight operations can be evaluated in $O(k \log(kn))$ work and $O(\log n \log k)$ depth.*

Parallel k-approximate minimum cut. Karger describes an $O(mn^{2/k} \log n)$ time sequential algorithm for finding a cut in a weighted graph within a factor of k of the optimal cut [21]. It works by randomly selecting edges to contract with probability proportional to their weight until a single vertex remains, and keeping track of the component with smallest incident weight (not including internal edges) during the contraction.

His analysis shows that in a weighted graph with minimum cut c, with probability $n^{-2/k}$, the component with minimum incident weight encountered during a single trial of the contraction algorithm corresponds to a cut of weight at most kc, and therefore, running $O(n^{2/k} \log n)$ trials yields a cut of size at most kc w.h.p.

Although Karger's contraction algorithm is easy to parallelize using a parallel minimum spanning tree algorithm, keeping track of the incident component weights is trickier. To overcome this problem, we show that we can use our batch component weight algorithm to simulate the sequential contraction process efficiently. With this tool, we can determine the minimum incident weight of a component as follows:

(1) Compute an MST with respect to the weighted random edge ordering, where a heavier weight indicates that an edge contracts later
(2) For each edge $(u, v) \in G$, determine the heaviest edge in the MST on the unique (u, v) path
(3) Construct a vertex-weighted tree from the MST, where the weights are the total incident weight on each vertex in G. For each edge (u, v) in the MST in contraction order:
    - Determine the set of edges in G such that (u, v) is the heaviest edge on its MST path. For each such edge identified, SubtractWeight from each of its endpoints by the weight of the edge
    - Perform JoinEdge on the edge (u, v)
    - Perform QueryWeight on the vertex u

Observe that the weight of a component at the point in time when it is queried is precisely the total weight of incident edges (again, not including internal edges). Taking the minimum over the initial degrees and all query results therefore yields the desired answer.

Karger shows how to parallelize picking the (weighted) random permutation of the edges with $O(m \log^2 n)$ work. It can easily slightly modified to improve the bounds by a logarithmic factor as follows. The algorithm selects the edges by running a prefix sum over the edge weights. Assuming a total weight of W, it then picks m random integers up to W, and for each uses binary search on the result of the prefix sum to pick an edge. This process, however, might end up picking only the heaviest edges. Karger shows that by removing those edges the total weight W decreases by a constant factor, with high probability. To make this efficient, we must first preprocess the edge weights to make them polynomial in n. Gawrychowski et al. [11] describe a transformation that affects the value of the minimum cut by no more than a constant factor and bounds all edge weights

by $O(n^5)$. Therefore, repeating for $\log n$ rounds the algorithm will select all edges in the appropriate weighted random order. Each round takes $O(m \log n)$ work for a total of $O(m \log^2 n)$ work.

Replacing the binary search in Karger's algorithm with a sort of the random integers and merge into the the result of the prefix sum yields an $O(m \log n)$ work randomized algorithm. In particular, m random numbers uniformly distributed over a range can be sorted in $O(m)$ work and $O(\log n)$ depth by first determining for each number which of m evenly distributed buckets within the range it is in, then sorting by bucket using an integer sort [35] and finally sorting within buckets.

Step 1 therefore takes $O(m \log n)$ work and $O(\log^2 n)$ depth to compute the random edge permutation, and $O(m)$ work and $O(\log n)$ depth to run a parallel MST algorithm [24]. Step 2 takes $O(m \log n)$ work and $O(\log n)$ depth using RC trees [1, 2], and Step 3 takes $O(m \log n)$ work and $O(\log^2 n)$ depth by Corollary 4.3 and the fact that the algorithm performs a batch of $O(n)$ operations. By Karger's analysis, trying $O(n^{2/k} \log n)$ random contractions yields the following lemma, and setting $k = \log n$ gives our desired corollary.

Lemma 4.4. *For a weighted graph, a cut within a factor of k of the minimum cut can be found w.h.p. in $O(mn^{2/k} \log^2 n)$ work and $O(\log^2 n)$ depth.*

Corollary 4.5. *For a weighted graph, a cut within a factor of $\log n$ of the minimum cut can be found w.h.p. in $O(m \log^2 n)$ work and $O(\log^2 n)$ depth.*

4.3 Additional Tools and Lemmas

Transformation to bounded edge weights. For our algorithm to be efficient, we require that the input graph has small integer weights. Gawrychowski et al. [11] give a transformation that ensures all edge weights of a graph are bounded by $O(n^5)$ without affecting the minimum cut by more than a a constant factor. For our algorithm, $O(n^5)$ would be too big, so we design a different transformation that guarantees all edge weights are bounded by $O(m \log n)$ and only affects the weight of the minimum cut by a constant factor.

Lemma 4.6. *There exists a transformation that, given an integer-weighted graph G, produces an integer-weighted graph G' no larger than G, such that G' has edge weights bounded by $O(m \log n)$, and the minimum cut of G' corresponds to an $O(1)$-approximate minimum cut in G.*

Proof. Let G be the input graph and suppose that the true value of the minimum cut is c. First, we use Corollary 4.5 to obtain a $O(\log n)$-approximate minimum cut, whose value we denote by $\tilde{c}$ ($c \leq \tilde{c} \leq c \log n$). We can contract all edges of the graph with weight greater than $\tilde{c}$, since they cannot appear in the minimum cut. Let $s = \tilde{c}/(2m \log n)$. We delete (not contract) all edges with weight less than s. Since there are at most m edges in any cut, this at most affects the value of a cut by $sm = \tilde{c}/(2 \log n) \leq c/2$. Therefore, the minimum cut in this graph is still a constant factor approximation to the minimum cut in G.

Next, scale all remaining edge weights down by the factor s, rounding down. All edge weights are now integers in the range $[1, 2m \log n]$. This is the transformed graph G'. It remains to argue that the value of the minimum cut is a constant-factor approximation. First, note that the scaling process preserves the order of cut values, and hence the true minimum cut in G has the same value in G' as the minimum cut in G'. Consider any cut in G', and scale the weights of the edges back up by a factor s. This introduces a rounding error of at most s per edge. Since any cut has at most m edges, the total rounding error is at most $sm \leq c/2$. Therefore, the value of the minimum cut in G' is a constant factor approximation to the value of the minimum cut in G. □

Last, observe that this transformation can easily be performed in parallel by using a work-efficient connected components algorithm to perform the edge contractions, as is standard (see, e.g., Reference [26]).

Sampling binomial random variables. It will be helpful in the next step to be able to efficiently sample binomial random variables. We will use the following results due to Farach-Colton et al. [9].

Lemma 4.7 (Farach-Colton et al. [9], Theorem 1). *Given a positive integer n, one can sample a random variate from the binomial distribution $B(n, 1/2)$ in $O(1)$ time with probability $1 - 1/n^{\Omega(1)}$ and in expectation after $O(n^{1/2+\varepsilon})$-time preprocessing for any constant $\varepsilon > 0$, assuming that $O(\log n)$ bits can be operated on in $O(1)$ time. The preprocessing can be reused for any $n' = O(n)$.*

We can also use the following reduction to sample $B(n, p)$ for arbitrary $0 \leq p \leq 1$.

Lemma 4.8 (Farach-Colton et al. [9], Theorem 2). *Given an algorithm that can draw a sample from $B(n', 1/2)$ in $O(f(n))$ time with probability $1 - 1/n^{\Omega(1)}$ and in expectation for any $n' \leq n$, then drawing a sample from $B(n', p)$ for any real p can be done in $O(f(n) \log n)$ time with probability $1 - 1/n^{\Omega(n)}$ and in expectation, assuming each bit of p can be obtained in $O(1)$ time.*

We note, importantly, that the model used by Farach-Colton et al. assumes that random $\Theta(\log n)$-size words can be generated in constant time. Since we only assume that we can generate random bits in constant time, we will have to account for this with an extra $O(\log n)$ factor in the work where appropriate. Note that this does not negatively affect the depth, since we can pre-generate as many random words as we anticipate needing, all in parallel at the beginning of our algorithm. Last, we also remark that although it might not be clear in their definition, the constants in the algorithm can be configured to control the constant in the $\Omega(1)$ term in the probability, and therefore their algorithms take $O(1)$ time and $O(\log n)$ time w.h.p.

To make use of these results, we need to show that the preprocessing of Lemma 4.9 can be parallelized. Thankfully, it is easy. The preprocessing phase consists of generating n^{ε} alias tables of size $O(\sqrt{n \log n})$. Hübschle-Schneider and Sanders [19] give a linear work, $O(\log n)$ depth parallel algorithm for building alias tables. Building all of them in parallel means we can perform the alias table preprocessing in $O(n^{1/2+\varepsilon})$ work and $O(\log n)$ depth. The last piece of preprocessing information that needs to be generated is a lookup table for decomposing any integer $n' = O(n)$ into a sum of a constant number of square numbers. This table construction is trivial to parallelize, and hence all preprocessing runs in $O(n^{1/2+\varepsilon})$ work and $O(\log n)$ depth.

Lemma 4.9. *Given a positive integer n, after $O(n^{1/2+\varepsilon})$ work and $O(\log n)$ depth preprocessing, one can sample random variables from $B(n, 1/2)$ in $O(\log n)$ work w.h.p., and from $B(n, p)$ in $O(\log^2 n)$ work w.h.p. The preprocessing can be reused for any $n' = O(n)$.*

Subsampling p-skeletons. Karger defines the p-skeleton $G(p)$ of an unweighted graph G as a copy of G where each edge appears with probability p. A p-skeleton therefore has $O(pm)$ edges in expectation. For a weighted graph, the p-skeleton is defined as the p-skeleton of the corresponding unweighted multigraph in which an edge of weight w is replaced by w parallel multiedges. The p-skeleton of a weighted graph therefore has $O(pW)$ edges in expectation, where W is the total weight in the graph. Karger gives an algorithm for generating a p-skeleton in $O(pW \log(m))$ work, which relies on performing $O(pW)$ independent random samples with probabilities proportional to the weight of each edge, each of which takes $O(\log(m))$ amortized time. In Karger's algorithm, given a guess of the minimum cut c, he computes p-skeletons for $p = \Theta(\log n/c)$. Since no edge of weight greater than c can be contained in the minimum cut, all such edges can be contracted, leaving us with $W \leq mc$, so the skeleton has $O(m \log n)$ edges and takes $O(m \log^2 n)$ work to compute. Since our algorithm does not know the minimum cut c yet, it uses guessing and doubling on p, and hence has to compute several p-skeletons, so $O(m \log^2 n)$ work is too slow. We overcome this problem using binomial random variables and subsampling.

Lemma 4.10. *Given a weighted graph G with edge weights bounded by $m^{2-\varepsilon}$, an initial sampling probability p and an integer k, there exists an algorithm that can produce the skeleton graphs $G(p), G(p/2), \ldots, G(p/2^k)$ in $O(m \log^2 n + km \log n)$ work w.h.p. and $O(k \log n)$ depth.*

Proof. Begin by using Lemma 4.9 and performing the required preprocessing for sampling binomial random variables from $B(m^{2-\varepsilon}, 1/2)$, which takes $O(m)$ work and $O(\log n)$ depth. To construct $G(p)$, for each edge e in the graph, sample a binomial random variable $x \sim B(w(e), p)$. The skeleton then contains the edge e with weight x (conceptually, x unweighted copies of the multiedge e). This results in the same distribution of graphs as if sampled using Karger's technique, and takes $O(m \log^2 n)$ work w.h.p. and $O(\log n)$ depth. For each additional skeleton $G(p')$ requested, subsample from the previous skeleton by drawing binomial random variables from $B(w_{G(2p')}(e), 1/2)$, which takes $O(m \log n)$ work w.h.p. and $O(\log n)$ depth. In total, to perform k rounds of sampling, this takes $O(m \log^2 n + km \log n)$ work w.h.p. and $O(k \log n)$ depth. □

Using subsampling here is important, since otherwise it would cost $O(km \log^2 n)$ work to sample all of the desired skeleton graphs. Additionally, note that Lemma 4.6 makes it easy to satisfy the requirement that all edge weights be bounded by $m^{2-\varepsilon}$.

Parallel weighted sparse certificates. A *sparse k-connectivity certificate* of an unweighted graph $G = (V, E)$ is a graph $G' = (V, E' \subset E)$ with at most $O(kn)$ edges, such that every cut in G of weight at most k has the same weight in G'. Cheriyan, Kao, and Thurimella [7] introduce a parallel graph search called *scan-first search*, which they show can be used to generate k-connectivity certificates of unweighted graphs. Here, we briefly note that the algorithm can easily be extended to handle weighted graphs. The scan-first search algorithm is implemented as follows.

ALGORITHM 5: Scan-first search [7]

```
1: procedure SFS(G = (V, E) : Graph, r : Vertex)
2:     Find a spanning tree T' rooted at r
3:     Find a preorder numbering to the vertices in T'
4:     For each vertex v ∈ T' with v ≠ r, let b(v) denote the least neighbor of v in preorder
5:     Let T be the tree formed by {v, b(v)} for all v ≠ r
```

Note that the scan-first search tree of a connected graph is always a tree. If the graph is disconnected, then the result is a scan-first search tree of each component. Using a linear work, low depth spanning tree algorithm, scan-first search can easily be implemented in $O(m)$ work and $O(\log n)$ depth. Cheriyan, Kao, and Thurimella show that if E_i are the edges in a scan-first search forest of the graph $G_{i-1} = (V, E \setminus (E_1 \cup \ldots E_{i-1}))$, then $E_1 \cup \ldots E_k$ is a sparse k-connectivity certificate. A sparse k-connectivity certificate can therefore be found in $O(km)$ work and $O(k \log n)$ depth by running scan-first search k times.

In the weighted setting, we treat an edge of weight w as w parallel unweighted multiedges. As always, this is only conceptual, the multigraph is never actually generated. To compute certificates in weighted graphs, we therefore use the following simple modification. After computing each scan-first search tree, instead of removing the edges present from G, simply lower their weight by one, and remove them only if their weight becomes zero. It is easy to see that this is equivalent to running the ordinary algorithm on the unweighted multigraph. We therefore have the following.

Lemma 4.11. *A sparse k-connectivity certificate for a weighted, undirected graph can be found in $O(km)$ work and $O(k \log n)$ depth.*

Parallelizing Matula's algorithm. Matula [29] gave a linear time sequential algorithm for $(2+\varepsilon)$-approximate edge connectivity (unweighted minimum cut). It is easy to extend to weighted

graphs so that it runs in $O(m \log n \log W)$ time, where W is the total weight of the graph. Using standard transformations to obtain polynomially bounded edge weights, this gives an $O(m \log^2 n)$ algorithm. Karger and Motwani [25] gave a parallel version of Matula's unweighted algorithm that runs in $O(m^2/n)$ work. Essentially, their version of Matula's algorithm does the following steps as indicated in Algorithm 6.

ALGORITHM 6: Approximate minimum cut

```
1: procedure MATULA(G = (V, E) : Graph)
2:     if |V| = 1 then return ∞
3:     local d ← minimum degree in G
4:     local k ← d/(2 + ε)
5:     local C ← Compute a sparse k-certificate of G
6:     local G′ ← Contract all non-certificate edges of E
7:     return min(d, MATULA(G′))
```

It can be shown that at each iteration, the size of the graph is reduced by a constant factor, and hence there are at most $O(\log n)$ iterations. Furthermore, the work performed at each step is geometrically decreasing, so the total work, using the sparse certificate algorithm of Cheriyan, Kao, and Thurimella [7] is $O(dm)$ and the depth is $O(d \log^2 n)$, where d is the minimum degree of G.

Here, we give a slight modification to this algorithm that makes it work on weighted graphs in $O(dm \log(W/m))$ work and $O(d \log n \log W)$ depth, where d is the minimum weighted degree of the graph. To extend the algorithm to weighted graphs, we can replace the sparse certificate routine with our modified version for weighted graphs, and replace the computation of d with the equivalent weighted degree. By interpreting an edge-weighted graph as a multigraph where each edge of weight w corresponds to w parallel multiedges, we can see that the algorithm is equivalent. To argue the cost bounds, note that like in the original algorithm where the size of the graph decreases by a constant factor each iteration, the total weight of the graph must decrease by a constant factor in each iteration. Because of this, it is no longer true that the work of each iteration is geometrically decreasing. Naively, this gives a work bound of $O(dm \log(W))$, but we can tighten this slightly as follows. Observe that after performing $\log(W/m)$ iterations, the total weight of the graph will have been reduced to $O(m)$, and hence, like in the sequential algorithm, the work must subsequently begin to decrease geometrically. Hence the total work can actually be bounded by $O(dm \log(W/m) + dm) = O(dm \log(W/m))$. We therefore have the following.

LEMMA 4.12. *Given a weighted graph with minimum weighted-degree d and total weight W, an $O(1)$-approximate minimum cut can be found in $O(dm \log(W/m))$ work and $O(d \log n \log W)$ depth.*

4.4 Parallel $O(1)$-approximate Minimum Cut

We have finally amassed the ingredients needed to produce a parallel $O(1)$-approximate minimum cut algorithm. Well, we need one more trick, unsurprisingly due to Karger. To produce the sampled skeleton graph, Karger's algorithm chooses the sampling probability inversely proportional to the weight of the minimum cut, which paradoxically is what we are trying to compute. This issue is solved by using guessing and doubling. The algorithm guesses the minimum cut and computes the resulting approximation. It can then use Karger's sampling theorem (Theorem 6.3.1 and Lemma 6.3.2 of Reference [20]) to verify whether the guess was too high.

Lemma 4.13 (Karger [20]). *Let G be a graph with minimum cut c and let $p = \Theta((\log n)/\varepsilon^2 c)$. Then w.h.p. the minimum cut in $G(p)$ has value in $(1 \pm \varepsilon)pc$.*

Lemma 4.14 (Karger [20]). *w.h.p., if $G(p)$ is constructed and has minimum cut $\hat{c} = \Theta((\log n)\varepsilon^2)$ for $\varepsilon \leq 1$, then the minimum cut c in G has value in $(1 \pm \varepsilon)\hat{c}/p$.*

If the true minimum cut is c, then the correct sampling probability for Karger's algorithm is $p = \Theta((\log n)\varepsilon^2 c)$, which produces a skeleton cut of size $\hat{c} = \Theta((\log n)/\varepsilon^2)$ w.h.p. If the algorithm makes a guess $C > 2c$ with corresponding probability $P = \Theta((\log n)/\varepsilon^2 C)$, then Lemma 4.14 says that the minimum cut in the skeleton graph is less than $\hat{c}$ w.h.p. The algorithm can therefore double the guess for P and try again, until the minimum cut in the skeleton is larger than $\hat{c}$, at which point we know that the P-skeleton approximates the minimum cut within a factor ε. To perform these steps efficiently, our algorithm does the following:

(1) Use Corollary 4.5 to compute a $\log n$-approximate minimum cut value C in $O(m \log^2 n)$ work and $O(\log^2 n)$ depth.
(2) Transform the graph using Lemma 4.6 to ensure that all weights are bounded by $O(m \log n)$ while retaining an $O(1)$-approximate minimum cut in $O(m \log^2 n)$ work and $O(\log^2 n)$ depth.
(3) Sample the skeleton graphs $G(\log^2 n/C), G(\log^2 n/(2C)), \ldots, G(\log n/C)$ using Lemma 4.10. This is $\log \log n \leq \log n$ skeletons, and hence this takes $O(m \log^2 n)$ work w.h.p. and $O(\log^2 n)$ depth.
(4) For each skeleton graph:
    - Compute a sparse $\Theta(\log n)$ certificate of the skeleton graph. This takes $O(m \log n)$ work and $O(\log^2 n)$ depth by Lemma 4.11.
    - Compute an $O(1)$-approximate minimum cut in the $\Theta(\log n)$ certificate using Matula's algorithm (Lemma 4.12). Since the certificate guarantees that the total weight is at most $O(n \log n)$ and hence that the minimum weighted degree is at most $O(\log n)$, this takes $O(m \log n \log \log n)$ work and $O(\log^2 n \log \log n)$ depth.

Since there are $O(\log \log n)$ skeleton graphs, the total work done by the final step is at most $O(m \log n (\log \log n)^2)$, which is at most $O(m \log^2 n)$, and the depth is $O(\log^3 n)$. The correctness of the algorithm follows from the sampling theorem (Lemma 4.14) and Karger's discussion [20]. Finally, we can conclude the following result.

Lemma 4.15. *Given a weighted, undirected graph, the weight of an $O(1)$-approximate minimum cut can be computed w.h.p. in $O(m \log^2 n)$ work and $O(\log^3 n)$ depth.*

5 FINDING MINIMUM 2-RESPECTING CUTS

We are given a connected, weighted, undirected graph $G = (V, E)$ and a spanning tree T. In this section, we will give an algorithm that finds the minimum 2-respecting cut of G with respect to T in $O(m \log n)$ work and $O(\log^3 n)$ depth.

Our algorithm, like those that came before it, finds the minimum 2-respecting cut by considering two cases. We assume that the tree T is rooted arbitrarily. In the first case, we assume that the two tree edges of the cut occur along the same root-to-leaf path, i.e., one is a descendant of the other. This is called the *descendant edges* case. In the second case, we assume that the two edges do not occur along the same root-to-leaf path. This is the *independent edges* case.

Since we are going to use RC trees, we require that G have bounded degree. Note that any arbitrary degree graph can easily be *ternarized* by replacing high-degree vertices with cycles of infinite weight edges, resulting in a graph of maximum degree three with the same minimum cut, and only a constant-factor larger size in terms of edges, which our bounds depend on.

5.1 Descendant Edges

We present our minimum 2-respecting cut algorithm for the descendant edges case. Let T be a spanning tree of a connected graph $G = (V, E)$ of degree at most three, and root T at an arbitrary vertex of degree at most two. The rooted tree is therefore a binary tree.

We use the following fact. For any tree edge $e \in T$, let F_e denote the set of edges $(u, v) \in E$ (tree and non-tree) such that the u to v path in T contains the edge e. Then the weight of the cut induced by a pair of edges $\{e, e'\}$ in T is given by

$$w(F_e \Delta F_{e'}) = w(F_e) + w(F_{e'}) - 2w(F_e \cap F_{e'}),$$

where Δ denotes the symmetric difference between the two sets. For each tree edge e, our algorithm seeks the tree edge e' that minimizes $w(F_e \Delta F_{e'})$, which is equivalent to minimizing

$$w(F_{e'}) - 2w(F_e \cap F_{e'}).$$

To do so, it traverses T from the root while maintaining weights on a tree data structure that satisfies the following invariant:

Invariant 1 (Current Subtree Invariant). *When visiting $e = (u, v)$, for every edge $e' \in$ Subtree(v), the weight of e' in the dynamic tree is $w(F_{e'}) - 2w(F_e \cap F_{e'})$.*

The initial weight of each edge e is therefore $w(F_e)$. Maintaining this invariant as the algorithm traverses the tree can then be achieved with the following observation. When the traversal descends from an edge $p = (w, u)$ to a neighboring child edge $e = (u, v)$, the following hold for all $e' \in$ Subtree(v):

(1) $(F_e \cap F_{e'}) \supseteq (F_p \cap F_{e'})$, since any path that goes through p and e' must pass through e.
(2) $(F_e \cap F_{e'}) \setminus (F_p \cap F_{e'})$ are the edges $(x, y) \in F_{e'}$ such that e is a *top edge* of the path $x - y$ in T (i.e., e is on the path from x to y in T, but the parent edge of e is not).

Therefore, to maintain the current subtree invariant, when the algorithm visits the edge e, it need only subtract twice the weight of all x–y paths that contain e as a top edge. This can be done efficiently by precomputing the sets of top edges. There are at most two top edges for each path x–y, and they can be found from the LCA of x and y in T. We need not consider tree edges, since they will never appear in $F_{e'}$. By maintaining the aforementioned invariant, the solution follows by taking the minimum value of $w(F_e)$ + QuerySubtree(v) for all edges $e = (u, v)$ during the traversal. As described, this algorithm is entirely sequential, but it can be parallelized using our batched mixed operations on trees algorithm (Corollary 3.3).

The operation sequence can be generated as follows. First, the weights $w(F_e)$ for each edge can be computed using the batched mixed operations algorithm (Corollary 3.3) where each edge (u, v) of weight w creates an AddPath(u, v, w) operation, followed by a QueryEdge(e) for every edge $e \in T$. This takes $O(m \log n)$ work and $O(\log^2 n)$ depth. The LCAs required to compute the sets of top edges can be computed using the parallel LCA algorithm of Schieber and Vishkin [36] in $O(m)$ work and $O(\log n)$ depth in total. By computing an Euler tour of the tree T (an ordered sequence of visited edges) beginning at the root, the order in which to perform the tree operations can be deduced in $O(n)$ work and $O(\log n)$ depth. Each edge in the Euler tour generates an AddPath operation for each of its top edges, followed by a QuerySubtree operation. Note that each edge is visited twice during the Euler tour. The second visit corresponds to negating the AddPath operations from the first visit. The solution is then the minimum result of all of the QuerySubtree operations. Since there are a constant number of top edges per path, and $O(m)$ paths in total, the operation sequence has length $O(m)$. Using Corollary 3.3, we arrive at the following.

Theorem 5.1. *Given a weighted, undirected graph G and a rooted spanning tree T, the minimum 2-respecting cut of G with respect to T such that one of the cut edges is a descendant of the other can be computed in in $O(m \log n)$ work and $O(\log^2 n)$ depth w.h.p.*

5.2 Independent Edges

The independent edge case is where the two cutting edges do not fall on the same root-to-leaf path. To solve the independent edges problem, we use the framework of Gawrychowski et al. [11], which is to decompose the problem into a set of subproblems, which they call *bipartite problems*. The key challenge in parallelizing the solution to the bipartite problem is dealing with the fact that the resulting trees might not be balanced. The algorithm of Gawrychowski et al. relies on performing a biased divide-and-conquer search guided by a heavy-light decomposition [18], and then propagating results up the trees bottom up. Since the trees may be unbalanced, this cannot be easily parallelized. Our solution is to use the recursive clustering of RC trees to guide a divide and conquer search in which we can maintain all of the needed information on the clusters.

Definition 5.2 (The Bipartite Problem). Given two weighted rooted trees T_1 and T_2 and a set of weighted edges that cross from one to the other, $L = \{(u, v) : u \in T_1, v \in T_2\}$, the bipartite problem is to select $e_1 \in T_1$ and $e_2 \in T_2$ with the goal of minimizing the sum of the weight of e_1 and e_2 plus the weights of all edges $(v_1, v_2) \in L$ such that v_1 is in the subtree rooted at the bottom endpoint of e_1 and v_2 is in the subtree rooted at the bottom endpoint of e_2. The size of a bipartite problem is the size of L plus the size of T_1 and T_2.

Gawrychowski et al. observe that if T_1 and T_2 are edge-disjoint subtrees of T, then, assigning weights of $w(F_e)$ to each tree edge and weights of $-2w(e)$ to each edge non-tree, the solution to the bipartite problem is the minimum 2-respecting cut such that $e_1 \in T_1$ and $e_2 \in T_2$. The independent edges problem is then solved by reducing it to several instances of the bipartite problem, and taking the minimum answer among all of them. We will show how to generate the bipartite problems efficiently, and how to solve them efficiently, both in parallel.

5.2.1 Generating the Bipartite Problems. The following parallel algorithm generates $O(n)$ instances of the bipartite problem with total size at most $O(m)$. For each edge e in T, the algorithm first assigns them a weight equal to $w(F_e)$. Now consider all non-tree edges, i.e., all edges $e \in E, e \notin T$, group them by the LCA of their endpoints in T, and assign them a weight of $-2w(e)$. This forms a partition of the $O(m)$ edges of G, each group identified by a vertex. Each vertex in T conversely has an associated (possibly empty) list of non-tree edges.

For each vertex v in T with a non-empty associated list of edges, create a compressed path tree of T with respect to the endpoints of the associated edges and v. Finally, for each such compressed path tree, root it at v (the common LCA of the edge endpoints). The bipartite problems are now generated as follows. For each vertex v with a non-empty list of non-tree edges, and the corresponding compressed path tree T_v, consider the children x, y of v in T_v. The bipartite problem consists of T_1, which contains the edge (v, x) and the subtree of T_v rooted at x, and likewise, T_2, which contains the edge (v, y) and the subtree of T_v rooted at y, and L, the associated list of non-tree edges. See Figure 5 for an illustration.

Lemma 5.3. *Given a tree and a set of non-tree edges, the corresponding bipartite problems can be generated in $O(m \log n)$ work and $O(\log^2 n)$ depth w.h.p.*

Proof. The edge weight values can be computed in the same way as before using our batched mixed operations on trees algorithm in $O(m \log n)$ work and $O(\log^2 n)$ depth. LCAs can be computed using the parallel LCA algorithm of Schieber and Vishkin [36] in $O(m)$ work and $O(\log n)$

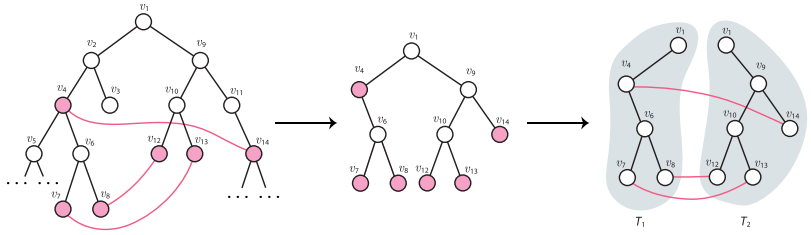


Fig. 5. The bipartite problems are generated by compressing the input tree with respect to the endpoints of the edges whose endpoints share an LCA, then splitting the tree into the left and right halves.

depth. Grouping the edges by LCA can be achieved using a parallel sorting algorithm in $O(m \log n)$ work and $O(\log n)$ depth. Together, these steps take $O(m \log n)$ work and $O(\log^2 n)$ depth. For each group, computing the compressed path tree takes $O(m_i \log(1 + n/m_i)) \leq O(m_i \log n)$ work and $O(\log^2 n)$ depth w.h.p., where m_i is the number of edges in the group. Performing all compressed path tree computations in parallel and observing that the edge lists of each vertex are a disjoint partition of the edges of G, this takes at most $O(m \log n)$ work and $O(\log^2 n)$ depth in total w.h.p. □

It remains only for us to show that the bipartite problems can be efficiently solved in parallel.

5.2.2 Solving the Bipartite Problems. Our solution is a recursive algorithm that utilizes the recursive cluster structure of RC trees. Recall that RC trees consist of unary and binary clusters (and the nullary cluster at the root, but this is not needed by our algorithm). Since the bipartite problems are constructed such that trees T_1 and T_2 always have a root with a single child, the root cluster of their RC trees consists of exactly one unary cluster.

High-level idea. Recall that the goal is to select an edge $e_1 \in T_1$ and an edge $e_2 \in T_2$ that minimizes their costs plus the cost of all edges $(u, v) \in L$ such that u is a descendant of e_1 and v is a descendant of e_2. Our algorithm first constructs an RC tree of T_1, and weights the edges in T_1 and T_2 by their cost. At a high level, the algorithm then works as follows. Given a binary cluster c_1 of T_1, the algorithm maintains weights on T_2 such that for each edge $e_2 \in T_2$, its weight is the weight of e_2 in the original tree plus the sum of the weights of all edges $(u, v) \in L$ such that u is a descendant of the bottom boundary vertex of c_1, and v is a descendant of e_2. This implies that for a binary cluster of T_1 consisting of an isolated edge $e_1 \in T_1$, the weights of each $e_2 \in T_2$ are precisely such that $w(e_1) + w(e_2)$ is the value of selecting $\{e_1, e_2\}$ as the solution. This idea leads to a very natural recursive algorithm. We start with the topmost unary cluster of T_1 and proceed recursively down the clusters of T_1, maintaining T_2 with weights as described. When the algorithm recurses into the top binary child of a cluster, it must add the weights of all $(u, v) \in L$ that are descendants of that cluster to the corresponding paths in T_2. If recursing on the bottom binary subcluster of a binary cluster, then the weights on T_2 are unchanged. When recursing on a unary cluster, since it has no descendants, the algorithm uses the original weights of T_2. Once the recursion hits a binary cluster that consists of a single edge e_1, it can return the solution $w(e_1) + w(e_2)$, where e_2 is the lightest edge with respect to the current weights on T_2. Last, to perform this process efficiently, the algorithm *compresses*, using the compressed path tree algorithm [4], the tree T_2 every time it recurses, keeping only the vertices that are endpoints of the crossing edges that touch the current cluster of T_1.

Implementation. We provide pseudocode for our algorithm in Algorithm 7. Given a bipartite problem (T_1, T_2, L), we use the notation $L(C)$ to denote the edges of L limited to those that are incident on some vertex in the cluster C. Furthermore, we use $V_{T_2}(L(C))$ to denote the set of

vertices given by the endpoints of the edges in $L(C)$ that are in T_2. The pseudocode does not make the parallelism explicit, but all that is required is to run the recursive calls in parallel. The procedure takes as input a cluster C of T_1, a compressed version of T_2 with its original weights, and T_2', the compressed version of T_2 with updated weights. At the top level, it takes the cluster representing all of T_1 for the first argument, and the cluster for all of T_2 for the second and third argument. The Compress function compresses the given tree with respect to the given vertex set and its root, and returns the compressed tree still rooted at the same root. AddPaths(S) takes a set $S \subset L$ of edges and for each one, adds $w(u, v)$ to the root-to-v path, where $v \in T_2$, returning a new tree.

ALGORITHM 7: Parallel bipartite problem algorithm

```
1:  procedure Bipartite(C, T2, T2', L)
2:    if C = {e} then
3:      return w(e) + LightestEdge(T2')
4:    else
5:      T_cmp ← T2.Compress(V_T2(L(C.t)))
6:      T2'' ← T2'.AddPaths(L(C) \ L(C.t))
7:      T''_cmp ← T2''.Compress(V_T2(L(C.t)))
8:      ans ← Bipartite(C.t, T_cmp, T''_cmp, L(C.t))
9:      for each cluster C' in C.U do
10:       T_cmp ← T2.Compress(V_T2(L(C')))
11:       ans ← min(ans, Bipartite(C', T_cmp, T_cmp, L(C')))
12:     if C is a binary cluster then
13:       T_cmp ← T2.Compress(V_T2(L(C.b)))
14:       T'_cmp ← T2'.Compress(V_T2(L(C.b)))
15:       ans ← min(ans, Bipartite(T_cmp, T'_cmp, L(C.b)))
16:     return ans
```

Since this algorithm creates many copies of T_2, we must ensure that we can still identify and locate a desired vertex given its label. One simple way to achieve this is to build a static hashtable alongside each copy of T_2 that maps vertex labels to the instance of that vertex in that copy.

An ingredient that we need to achieve low depth is an efficient way to update the weights in T_2 when adding weights to a collection of paths. Although RC trees support batch-adding weights to paths, the standard algorithm does not meet our cost requirements. This is easy to achieve in linear work and $O(\log n)$ depth by propagating the total weight of all updates up the clusters, and then propagating back down the tree, the weight of all updates that are descendants of the current cluster. It remains to analyze the cost of the Bipartite procedure.

Theorem 5.4. *A bipartite problem of size m can be solved in $O(m \log m)$ work and $O(\log^3 m)$ depth w.h.p.*

Proof. First, since all recursive calls are made in parallel and the recursion is on the clusters of T_1, the number of levels of recursion is $O(\log m)$ w.h.p. We will show that the algorithm performs $O(m)$ work in total at each level, in $O(\log^2 m)$ depth w.h.p. Observe first that at each level of recursion, the edges L for each call are a disjoint partition of the non-tree edges, since each recursive call takes a disjoint subset. We will now argue that each call does work proportional to $|L|$. Since T_2 and T_2' are both compressed with respect to L, their size is proportional to $|L|$. AddPaths takes linear work in the size of T_2 and $O(\log m)$ depth, and hence takes $O(|L|)$ work and $O(\log m)$ depth. Compress(K) takes $O(|K| \log(1 + |T_2|/|K|)) \leq O(|K| + |T_2|)$ work and $O(\log^2 m)$ depth w.h.p. Since compression is with respect to some subset of L, all of the compress operations take $O(|L|)$ work

and $O(\log^2 m)$ depth w.h.p. In total, this is $O(|L|)$ work in $O(\log^2 m)$ depth w.h.p. at each level for each call. Since the Ls at each level are a disjoint partition of the non-tree edges, the total work per level is $O(m)$ w.h.p., and hence the desired bounds follow. □

Since there are $O(n)$ bipartite problems of total size $O(m)$, solving them all in parallel yields the following, which, when combined with Theorem 5.1, proves Theorem 1.3.

Theorem 5.5. *Given a weighted, undirected graph G and a rooted spanning tree T, the minimum 2-respecting cut of G with respect to T such that the cut edges are independent can be computed in $O(m \log n)$ work and $O(\log^3 n)$ depth w.h.p.*

Combining Theorem 4.1 with Theorem 1.3 on each of the $O(\log n)$ trees in parallel proves Theorem 1.1.

6 CONCLUSION

We present a randomized $O(m \log^2 n)$ work, $O(\log^3 n)$ depth parallel algorithm for minimum cut. It is the first parallel minimum cut algorithm to match the work bound of the best sequential algorithm, making it work-efficient. Finding a faster parallel algorithm for minimum cut would therefore entail finding a faster sequential algorithm. It remains an open problem to find a deterministic algorithm for minimum cut, even a sequential one, that runs in $O(m \operatorname{polylog} n)$ time.

ACKNOWLEDGMENTS

We thank the anonymous referees for their comments and suggestions, and Phil Gibbons, Sam Westrick, and Danny Sleator for their feedback on the manuscript. We thank Ticha Sethapakdi for helping with the figures.

REFERENCES

[1] Umut A. Acar, Daniel Anderson, Guy E. Blelloch, Laxman Dhulipala, and Sam Westrick. 2020. Parallel batch-dynamic trees via change propagation. In *Proceedings of the European Symposium on Algorithms (ESA'20)*.

[2] Umut A. Acar, Guy E. Blelloch, and Jorge L. Vittes. 2005. An experimental analysis of change propagation in dynamic trees. In *Proceedings of the SIAM Symposium on Algorithm Engineering and Experiments (ALENEX'05)*.

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

[4] Daniel Anderson, Guy E. Blelloch, and Kanat Tangwongsan. 2020. Work-efficient batch-incremental minimum spanning trees with applications to the sliding window model. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'20)*.

[5] Guy E. Blelloch. 1996. Programming parallel algorithms. *Commun. ACM* 39, 3 (Mar. 1996).

[6] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. 2020. Optimal parallel algorithms in the binary-forking model. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'20)*.

[7] Joseph Cheriyan, Ming-Yang Kao, and Ramakrishna Thurimella. 1993. Scan-first search and sparse certificates: An improved parallel algorithm for k-vertex connectivity. *SIAM J. Comput.* 22, 1 (1993), 157–174.

[8] Richard Cole, Philip N. Klein, and Robert E. Tarjan. 1996. Finding minimum spanning forests in logarithmic time and linear work using random sampling. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'96)*.

[9] Martín Farach-Colton and Meng-Tsung Tsai. 2015. Exact sublinear binomial sampling. *Algorithmica* 73, 4 (2015), 637–651.

[10] Harold N. Gabow. 1995. A matroid approach to finding edge connectivity and packing arborescences. *J. Comput. Syst. Sci.* 50, 2 (1995), 259–273.

[11] Paweł Gawrychowski, Shay Mozes, and Oren Weimann. 2020. Minimum cut in $O(m \log^2 n)$ time. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP'20)*.

[12] H. Gazit, Gary L. Miller, and ShangHua Teng. 1988. Optimal tree contraction in the EREW model. In *Concurrent Computations*. Plenum Press, 139–156.

[13] Barbara Geissmann and Lukas Gianinazzi. 2018. Parallel minimum cuts in near-linear work and low depth. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'18)*.

[14] Mohsen Ghaffari and Fabian Kuhn. 2013. Distributed minimum cut approximation. In *Proceedings of the International Symposium on Distributed Computing (DISC'13)*.
[15] Mohsen Ghaffari, Krzysztof Nowicki, and Mikkel Thorup. 2020. Faster algorithms for edge connectivity via random 2-out contractions. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA'20)*.
[16] Ralph E. Gomory and Tien Chung Hu. 1961. Multi-terminal network flows. *J. Soc. Indust. Appl. Math.* 9, 4 (1961), 551–570.
[17] J. X. Hao and James B. Orlin. 1994. A faster algorithm for finding the minimum cut in a directed graph. *J. Algor.* 17, 3 (1994), 424–446.
[18] Dov Harel and Robert Endre Tarjan. 1984. Fast algorithms for finding nearest common ancestors. *SIAM J. Computing* 13, 2 (1984), 338–355.
[19] Lorenz Hübschle-Schneider and Peter Sanders. 2019. Parallel weighted random sampling. Retrieved from https://arXiv:1903.00227.
[20] David Karger. 1995. *Random Sampling in Graph Optimization Problems*. Ph.D. Dissertation. Stanford University.
[21] David R. Karger. 1993. Global min-cuts in RNC, and other ramifications of a simple min-cut algorithm. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA'93)*.
[22] David R. Karger. 1999. Random sampling in cut, flow, and network design problems. *Math. Oper. Res.* 24, 2 (1999), 383–413.
[23] David R. Karger. 2000. Minimum cuts in near-linear time. *J. ACM* 47, 1 (2000), 46–76.
[24] David R. Karger, Philip N. Klein, and Robert E. Tarjan. 1995. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM* 42, 2 (1995), 321–328.
[25] David R. Karger and Rajeev Motwani. 1994. Derandomization through approximation: An NC algorithm for minimum cuts. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*.
[26] David R. Karger and Clifford Stein. 1996. A new approach to the minimum cut problem. *J. ACM* 43, 4 (1996), 601–640.
[27] Jason Li. 2021. Deterministic mincut in almost-linear time. In *Proceedings of the ACM Symposium on Theory of Computing (STOC'21)*.
[28] Antonio Molina Lovett and Bryce Sandlund. 2020. A simple algorithm for minimum cuts in near-linear time. In *Proceedings of the Scandinavian Symposium and Workshops on Algorithm Theory (SWAT'20)*.
[29] David W. Matula. 1993. A linear time 2+ ε approximation algorithm for edge connectivity. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA'93)*.
[30] Gary L. Miller and John H. Reif. 1989. Parallel tree contraction part 1: Fundamentals. In *Randomness and Computation*, Vol. 5. 47–72.
[31] Hiroshi Nagamochi and Toshihide Ibaraki. 1992. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM J. Discrete Math.* 5, 1 (1992), 54–66.
[32] Hiroshi Nagamochi and Toshihide Ibaraki. 1992. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica* 7, 1-6 (1992), 583–596.
[33] C. St. J. A. Nash-Williams. 1961. Edge-disjoint spanning trees of finite graphs. *J. London Math. Soc.* 1, 1 (1961), 445–450.
[34] Serge A. Plotkin, David B. Shmoys, and Éva Tardos. 1995. Fast approximation algorithms for fractional packing and covering problems. *Math. Oper. Res.* 20, 2 (1995), 257–301.
[35] Sanguthevar Rajasekaran and John H. Reif. 1989. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Computing* 18, 3 (1989).
[36] Baruch Schieber and Uzi Vishkin. 1988. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comput.* 17, 6 (1988), 1253–1262.
[37] Daniel D. Sleator and Robert Endre Tarjan. 1983. A data structure for dynamic trees. *J. Comput. Syst. Sci.* 26, 3 (1983), 362–391.

Received 20 December 2021; revised 2 September 2022; accepted 30 September 2022