Minimum Cuts in Near-Linear Time

David R. Karger*

March 24, 1999

Abstract

We significantly improve known time bounds for solving the minimum cut problem on undirected graphs. We use a "semi-duality" between minimum cuts and maximum spanning tree packings combined with our previously developed random sampling techniques. We give a randomized (Monte Carlo) algorithm that finds a minimum cut in an m-edge, n-vertex graph with high probability in $O(m \log^3 n)$ time. We also give a simpler randomized algorithm that finds *all* minimum cuts with high probability in $O(n^2 \log n)$ time. This variant has an optimal $\mathcal{RNC}$ parallelization. Both variants improve on the previous best time bound of $O(n^2 \log^3 n)$. Other applications of the tree-packing approach are new, nearly tight bounds on the number of *near-minimum* cuts a graph may have and a new data structure for representing them in a space-efficient manner.

1 Introduction

The minimum cut problem has been studied for many years as a fundamental graph optimization problem with numerous applications. Initially, the problem was considered a harder variant of the s-t minimum cut problem and was solved in $\tilde{O}(mn^2)$ time (where $\tilde{O}(f)$ denotes $O(f \operatorname{polylog} f)$) on m-edge, n-vertex graphs [FF62, GH61]. Improvements then followed: first to $\tilde{O}(mn)$ time [HO94, NI92a], showing that minimum cuts were as easy to find as maximum flows; then to $\tilde{O}(n^2)$ time [KS96], showing they were significantly easier.

In this paper we give an algorithm with an $O(m \log^3 n)$ running time. Initially attacked as a harder variant of the maximum flow problem, the minimum cut problem turns out to be solvable in near-linear time. Side effects of our analysis include new combinatorial theorems on the structure, enumeration and representation of minimum cuts. We also give a relatively simple algorithm that runs in linear time on a large class of graphs and in $O(n^2 \log n)$ time on all graphs. This algorithm still dominates all previous ones and can also be parallelized optimally (that is, with time-processor product no worse than the sequential algorithm's) to yield the best-known $\mathcal{RNC}$ algorithm for the problem.

Our algorithm is based on a "semi-duality" between minimum cuts and undirected spanning tree packings. It is therefore fundamentally different from past approaches based on flows or on edge contraction. An algorithm developed by Gabow [Gab95] uses *directed tree* packing to solve the problem, but ours appears to be the first to use *undirected tree* packings. It thus introduces a different approach to finding minimum cuts in undirected graphs.

1.1 The Problem

This paper studies the *minimum cut* problem. Given a graph with n vertices and m (possibly weighted) edges, we wish to partition the vertices into two non-empty sets so as to minimize the number (or total weight) of edges crossing between them. More formally, a *cut* (A, B) of a graph G is a partition of the vertices of G into two nonempty sets A and B. An edge (v, w) *crosses* cut (A, B) if one of v and w is in A and the other in B. The *value* of a cut is the number of edges that cross the cut or, in a weighted

*MIT Laboratory for Computer Science, Cambridge, MA 02138.
email: `karger@lcs.mit.edu`
URL: `http://theory.lcs.mit.edu/~karger`.
Research supported in part by ARPA contract N00014-95-1-1246 and NSF contract CCR-9624239.

graph, the sum of the weights of the edges that cross the cut. The minimum cut problem is to find a cut of minimum value. We use c to denote the value of this minimum cut. Throughout this paper, the graph is assumed to be connected; otherwise the problem is trivial. We also require that the edge weights be non-negative, since optimizing with negative weights is $\mathcal{NP}$-complete by a transformation from the maximum-cut problem [GJ79, page 210]. We distinguish the minimum cut problem from the *s-t minimum cut problem* in which we require that two specified vertices s and t be on opposite sides of the cut; in the minimum cut problem there is no such restriction.

1.2 Applications

The minimum cut problem has several applications. Picard and Queyranne [PQ82] survey applications including graph partitioning problems, the study of project networks, and partitioning items in a database. In information retrieval, minimum cuts have been used to identify clusters of topically related documents in hypertext systems [Bot93]. The problem of determining the connectivity of a network arises frequently in issues of network design and network reliability (one of our previous papers [Karb] exploits an extremely tight connection between minimum cuts and network reliability). Minimum cut computations are used to find the *subtour elimination constraints* that are needed in the implementation of cutting plane algorithms for solving the traveling salesman problem [DFJ54, LLKS85]. Padberg and Rinaldi [PR90] and Applegate [App92] have reported that solving min-cut problems was the computational bottleneck in their state-of-the-art cutting-plane based TSP algorithm, as well as other cutting-plane based algorithms for combinatorial problems whose solutions induce connected graphs.

1.3 Past Work

The minimum cut problem was originally approached as a variant of the s-t minimum cut problem and solved using flow techniques [FF62]. The most obvious method is to compute s-t min-cuts for all pairs of vertices s and t; this requires $\binom{n}{2}$ flow computations. Gomory and Hu [GH61] showed that the problem could be solved by n flow computations; using present flow algorithms [GT88] this gives a time bound of $\tilde{O}(mn^2)$. After a lengthy period of little progress, several new algorithms appeared. Hao and Orlin [HO94] showed how the n flow computations could be performed in the time for one, improving the running time to $\tilde{O}(mn)$. Gabow [Gab95] gave an alternative "augmenting-tree" scheme that required $\tilde{O}(mc)$ time on an unweighted graph with minimum cut c.

Recently [Kar94a, Kara], we introduced sampling as an attack on minimum cuts. We showed that sampling could be used to find approximate minimum cuts in $\tilde{O}(m)$ time and to find them exactly in $\tilde{O}(m\sqrt{c})$ time (combining sampling with Gabow's algorithm). The same sampling technique is crucial to this paper as well.

The above approaches all use maximum flow techniques that treat undirected graphs as directed graphs (Gabow's algorithm packs directed trees rather than directed paths, but still addresses directed graphs). Recently a flowless undirected-graph approach based on *edge contraction* was discovered. If we can identify an edge that does not cross the minimum cut, then we can merge its endpoints into a single new vertex without affecting the minimum cut. Performing $n-2$ valid contractions will reduce the graph to two vertices, at which point there is only one nontrivial cut that must therefore correspond to the minimum cut. Nagamochi and Ibaraki [NI92b, NI92a] devised a *sparse certificate* computation that found a contractible edge in $O(m)$ time; this led to an $O(mn)$-time minimum cut algorithm. Karger and Stein [KS96] showed that random edge contraction worked well, leading to an algorithm running in $O(n^2 \log^3 n)$ time.

1.4 Our Results

In this paper we present a different approach to minimum cuts that yields faster algorithms than were previously known. We use a "semi-duality" between minimum cuts and maximum packings of undirected spanning trees—arguably a more natural dual than the directed flows or directed spanning trees used previously [GH61, HO94, Gab95]. Our approach does not rely fundamentally on flows *or* edge contractions. For a weighted, undirected graph we give:

- A randomized algorithm that finds a minimum cut with constant probability in $O(m\log^2 n)$ time and with high probability in $O(m\log^3 n)$ time. This significantly improves the previous best $O(n^2\log^3 n)$ bound of [KS96]. A technical refinement improves the time bound by an additional $\log\log n$ factor.
- A variant that finds a minimum cut with constant probability in $O(n^2)$ time and finds *all* (possibly $\Omega(n^2)$) minimum cuts with high probability in $O(n^2\log n)$ time. While this only gives a speedup of $\Theta(\log^2 n)$ over the algorithm of [KS96], the new algorithm is quite simple, relying only on computing minimum spanning trees, finding least common ancestors and evaluating expression-trees. It runs in $O(m)$ time on a natural class of graphs.
- A parallel version of our simpler algorithm that runs in $O(\log^3 n)$ time using $n^2/\log^2 n$ processors.
- New bounds of $\Theta(n^{\lfloor 2\alpha\rfloor})$ on the number of cuts with value at most α times the minimum that a graph can have. For non-half-integral α this improves on a previous best general bound of $O(n^{2\alpha})$ [Kar93]. This bound is the first to match the "step function" behavior of the known lower bounds with respect to α.
- A new data structure that represents all minimum and near-minimum cuts. Besides being smaller than previous representations [KS96], this representation highlights certain structural features of a graph's near-minimum cuts.

1.5 Our Methods

We now summarize the approach of the paper. At its core is the following definition:

Definition 1.1. Let T be a spanning tree of G. We say that a cut in G *k-respects* T if it cuts at most k edges of T. We also say that T *k-constrains* the cut in G.

Nash-Williams [NW61] proved that any graph with minimum cut c contains a set of $c/2$ edge-disjoint spanning trees. These $c/2$ trees must divide up the c minimum cut edges. It follows that any such *tree packing* contains a tree that 2-constrains the minimum cut. Using this idea, we show that for any graph G, we can find (by packing trees) a small set of spanning trees such that every minimum cut 2-respects some of the spanning trees. This lets us reduce to the following problem: find the minimum cut in G that 2-respects a given spanning tree. While this might appear a harder problem than the original, it turns out that the added constraints make the problem easier. Our scheme can be thought of as a variant of the "branch and bound" techniques used for hard optimization problems: we consider several families of added constraints, one of which is satisfied by the optimum solution. We can find the optimum by solving each constrained optimization problem.

To introduce our approach, we show in Section 2 that a maximum packing of spanning trees always contains many trees that 2-constrain any minimum cut. An immediate application of this observation in Section 3 yields the tightest known bounds on the number of near-minimum cuts in a graph.

Next we turn to the problem of actually finding the minimum cut. It follows from the previous paragraph that we can find the minimum cut by examining each tree of a maximum packing and finding the smallest cut that 2-respects each tree. Although finding a maximum tree packing is hard, we show in Section 4 that Gabow's minimum cut algorithm [Gab95] or the Plotkin-Shmoys-Tardos fractional packing algorithm [PST91] can be used to find a packing that is sufficient for our purposes. Unfortunately, in a graph with minimum cut c, such a packing contains roughly c trees, so checking all of them is prohibitively slow. Indeed, even finding the packing (with either algorithm) takes time proportional to c. To eliminate this factor, we use random sampling techniques that we developed previously [Kar94a, Kar94b, Kara] to reduce the packing time to $O(m+n\log^3 n)$ and the number of trees in the packing to $O(\log n)$.

Once we have the packing, it remains to examine each tree in it. We begin in Section 5 by showing how to find the smallest cut that *1-respects* a given tree in $O(m)$ time. Our algorithm involves a simple dynamic program that determines the value of the cut "induced" by removing each edge of the given tree. Besides introducing our techniques, this algorithm can be used to find the minimum cut in $O(m+n\log^3 n)$ time on a natural family of "fat" graphs containing many spanning trees.

In some cases we will not be able to find a tree that 1-constrains the minimum cut, so we move on to finding the minimum cut 2-respecting a given tree. We present two solutions. In Section 6 we extend the simple dynamic program of Section 5 to compute the value of the cut induced by removing each *pair* of tree edges. The algorithm runs in $\Theta(n^2)$ time—optimal for an algorithm that computes the values for all $\binom{n}{2}$ pairs. By running on $O(\log n)$ trees it identifies *all* minimum cuts with high probability in $O(n^2 \log n)$ time. In Section 6.2, we also show how it can be parallelized optimally.

The second algorithm, presented in Section 7, implicitly considers all $\binom{n}{2}$ pairs of tree edges without taking $\Omega(n^2)$ time to do so. We show how a *dynamic tree* data structure [ST83] can be used to evaluate a particular tree in $O(m \log^2 n)$ time. Thus evaluating the necessary $O(\log n)$ trees takes $O(m \log^3 n)$ time and gives the claimed bound.

We finish by giving some extensions of the algorithm. In Section 8 we give a fast and simple algorithm for a large class of "fat" graphs. In Section 9 we describe some refinements that improve the running time of our more complicated algorithm by some small factors.

The initial tree-packing step is the only one using randomization; all subsequent operations in the algorithm are deterministic. The fact that random sampling is used means that our algorithms, like those of [KS96], are *Monte Carlo* algorithms. Although the probability of success is high, there is no fast way to certify the correctness of the final answer (indeed, the fastest known method is to use one of the deterministic minimum cut algorithms).

2 Cuts and Tree Packings

In this section, we give some background on tree packings and their connection to cut problems. For the moment, let us restrict attention to unweighted graphs. An unweighted *tree packing* is a set of edge-disjoint spanning trees in a graph. The *value* of the packing is the number of trees in it. Clearly, every spanning tree in an unweighted packing must use at least one edge of every cut. Thus, at most c trees can be packed in a graph with minimum cut c. Nash-Williams [NW61] gave a related lower bound:

Theorem 2.1 ([NW61]). *Any undirected graph with minimum cut c contains a tree packing of value at least $c/2$.*

Nash-Williams actually proved a tighter result: if c_r is the value of the minimum r-way cut in the graph, then the value of the maximum tree packing is exactly $\lfloor \min_r c_r/(r-1) \rfloor$. But observe that the minimum r-way cut must have $c_r \geq rc/2$, since each of the r components of the cut must have c edges leaving it. Thus

$$\min \frac{c_r}{r-1} \geq \min \frac{rc}{2(r-1)} \geq c/2$$

Theorem 2.1 is not universally tight—for example, a single tree has minimum cut 1 and contains one spanning tree. However, it is existentially tight since, for example, a cycle has minimum cut 2 but has maximum tree packing value 1. We will find later that the tightness of a graph with respect to tree packings is a factor that determines the hardness of finding its minimum cuts using our algorithm. The factor-of-2 gap is particular to undirected graphs: Edmonds [Edm72] proved that a *directed* graph with *directed* minimum cut c has a packing of exactly c edge-disjoint directed spanning trees (Gabow exploits this in his minimum cut algorithm [Gab95]).

Since any graph has a packing with $c/2$ trees, it follows that a maximum packing will contain at least $c/2$ trees. Consider any minimum cut. Since the edges of the minimum cut must be partitioned among the trees of the maximum packing, the average number of min-cut edges per tree is at most $c/(c/2) = 2$. It follows that at least one of the trees in such a packing has at most 2 minimum-cut edges. In other words, at least one tree 2-constrains the minimum cut.

We will strengthen this argument in Lemma 2.3. First let us see why it is useful. If we are given a tree, and are able to determine which of its edges cross the minimum cut, then in fact we will have found the minimum cut. For consider a tree in which all min-cut edges have been noted. Start at any vertex and traverse the tree path to any other vertex. Each time we cross a min-cut edge, we know that we have switched sides of the minimum cut. It follows that two vertices are on the same side of the minimum cut if and only if the number of minimum cut edges on the path between them is even. More concretely, suppose a tree has only one minimum cut edge. Then removing this edge separates the tree into two subtrees, each of which is one side of the minimum cut. If a tree contains two minimum cut edges, then removing them separates the tree into three pieces. The "central" piece that is adjacent to both other pieces forms one side of the minimum cut, while the two remaining pieces form the other side.

The above paragraph demonstrates that any subset of the tree edges defines a unique two-way cut of the graph G. It is also clear that every cut defines a unique set of tree edges—namely, those crossing the cut—and that this correspondence between cuts and sets of tree edges is a bijection.

2.1 Weighted Packings

We now extend the above discussion to argue that *many* of the trees in a maximum packing must 2-respect that minimum cut. This is useful since it means that a randomly selected tree is likely to 2-constrain the minimum cut. We also generalize the tree packing definition to allow for weighted trees packed into weighted graphs.

Definition 2.2. A (weighted) *tree packing* is a set of spanning trees, each with an assigned weight, such that the total weight of trees containing a given edge is no greater than the weight of that edge. The *value* of the packing is the total weight of the trees in it.

If all edge and tree weights are integers, we can treat a weight-w edge as a set of w parallel edges, and a weight-w tree as a set of w identical trees. This recovers the unweighted notion of a tree packing as a set of edge-disjoint trees. The value of the packing becomes the number of (unweighted equivalent) trees. The definition also allows for fractional weights. This can be ignored, though, since we can always multiply all edge and tree weights by the least common denominator of the fractions to return to the integral version. It follows that Nash-Williams' Theorem 2.1 applies unchanged to weighted tree packings. We now give our main packing lemma.

Lemma 2.3. *Given any weighted tree packing of value βc and any cut of value αc, at least a $\frac{1}{2}(3 - \alpha/\beta)$ fraction of the trees (by weight) 2-constrain the cut.*

Corollary 2.4. *In any maximum packing, half the trees (by weight) 2-constrain the minimum cut.*

Proof. Suppose that tree T has weight w_T (so $\sum_T w_T = \beta c$) and that we choose a tree T at random with probability proportional to its weight. Define the random variable x_T to be one less than the number of edges of T crossing the α-minimum cut. Note that x_T is always a nonnegative integer. Since we have a packing, we know that no α-minimum cut edges are shared between the trees. It follows that

$$\begin{aligned} \sum w_T(x_T + 1) &\leq \alpha c \\ \sum w_T x_T &\leq \alpha c - \sum w_T \\ &\leq \alpha c - \beta c \end{aligned}$$

and thus

$$\begin{aligned} E[x_T] &= \frac{1}{\sum w_T} \sum w_T x_T \\ &\leq (\alpha/\beta - 1) \end{aligned}$$

It follows from Markov's inequality that $x_T < 2$ with probability at least $1 - 1/2(\alpha/\beta - 1) = \frac{1}{2}(3 - \alpha/\beta)$. Since x_T is an integer, $x_T < 2$ implies that $x_T \leq 1$, meaning that the α-minimum cut 2-respects T.

The corollary follows by taking $\alpha = 1$ and observing that $\beta \geq 1/2$ by Theorem 2.1. ☐

3 Combinatorics

As a first application of the concept of respecting constraint trees, we tighten the bounds on the number of small cuts in a graph. This section can be skipped without impacting the remainder of the paper. Our discussion is limited to *two-way* cuts—that is, sets set of edges crossing a two-way vertex partition. In contrast, other combinatorial results [KS96] also apply to multiway cuts.

Definition 3.1. A cut in G is *α-minimum* if its value is at most α times the minimum cut value.

Previously [Kar93, Kar94a, Kara], we proved that the number of α-minimum cuts in a graph is $O(n^{2\alpha})$. Others have since tightened this bound for small α: Benczur [Ben95] bounded the number of 6/5-minimum cuts by $O(n^2)$, Nagamochi, Nishimura and Ibaraki [NNI94] gave a bound of $O(n^2)$ for the number of 4/3-minimum cuts, and subsequently Henzinger and Williamson [HW96] showed that the number of cuts *strictly less* than 3/2 times the minimum is $O(n^2)$. In an unweighted n-vertex cycle, any even-size set of up to $\lfloor 2\alpha \rfloor$ edges is the edge set of an α-minimum cut. Therefore, the number of α-minimum cuts can be as large as

$$\binom{n}{2} + \binom{n}{4} + \cdots + \binom{n}{\lfloor 2\alpha \rfloor} = \Theta(n^{\lfloor 2\alpha \rfloor}). \tag{1}$$

We give an upper bound that is very close to this lower bound, matching its "step function" behavior as a function of α.

Our bound uses tree packings. It simplifies the proof to restrict attention to unweighted tree packings—collections of edge-disjoint spanning trees—in unweighted graphs. The discussion of the previous section showed that this loses no generality. We begin with a weaker but easily proved lemma:

Lemma 3.2. *For any constant α, there are $O(n^{\lfloor 2\alpha \rfloor})$ α-minimum cuts.*

Proof. Let $k = \lfloor 2\alpha \rfloor$. Number the α-minimum cuts from 1 to ℓ. Our goal is to bound ℓ by $O(n^k)$.

Recall Nash-Williams' Theorem 2.1 [NW61] which says that any graph with minimum cut c contains at least $c/2$ edge-disjoint spanning trees. Suppose that we pick one of these trees at random. Let $y_i = 1$ if we pick a tree that k-constrains the i^{th} α-minimum cut and 0 otherwise. Since this cut contains at most αc edges and each edge is in at most one of our $c/2$ spanning trees, the *expected* number of cut edges in our chosen spanning tree is at most 2α. It follows from Markov's inequality [MR95] and the fact that $1 + \lfloor 2\alpha \rfloor > 2\alpha$ that the chosen tree has at most $k = \lfloor 2\alpha \rfloor$ cut edges with constant probability. That is, $y_i = 1$ with constant probability, meaning $E[y_i] = \Omega(1)$. Thus

$$E[\sum_{i=1}^{\ell} y_i] = \Omega(\sum_{i=1}^{\ell} 1) = \Omega(\ell).$$

On the other hand, as discussed in the previous section, each cut that k-respects a given tree is in 1-1 correspondence with the set of at most k edges of the tree that it cuts. There are $\binom{n-1}{k} = O(n^k)$ such sets of edges, so no tree can k-constrain more than this many cuts. Thus

$$E[\sum y_i] \leq \max \sum y_i = O(n^k).$$

Combined with the previous bound, this means $\ell = O(n^k)$ as claimed. □

We now give refine the above lemma with a more complicated proof.

Theorem 3.3. *The number of α-minimum cuts is at most*

$$\frac{1}{\lfloor 2\alpha \rfloor + 1 - 2\alpha} \binom{n}{\lfloor 2\alpha \rfloor} (1 + O(1/n)).$$

Remark. For α a half integer, this quantity is within $1 + o(1)$ of the lower bound in Equation 1. The bound remains tight to within a constant factor for all values of α except those infinitesimally less than an integer (e.g., $\alpha = 3 - 1/n$). For such values, we get a better bound by considering cuts of value at most $\lceil 2\alpha \rceil /2$; the theorem then gives a bound of $\binom{n}{\lceil 2\alpha \rceil}(1 + o(1))$.

Proof. As before, we assume G is unweighted and pack $c/2$ edge-disjoint trees in the graph. Consider a bipartite graph B, with one side consisting of the trees in the Nash-Williams packing, and the other of the α-minimum cuts. To avoid confusion, we refer to the vertices of this bipartite graph as *nodes* and to its edges as *arcs*. Draw an arc from a tree-node T to a cut-node C if the tree k-constrains the cut in G. We will now assign weights to these arcs so that:

1. every tree node in B has weighted arc degree *at most* some D;

2. every cut node in B has weighted arc degree *at least* some d

Since there are $c/2$ trees in the packing, we see from Property 1 that the total weight of arcs in B is at most $cD/2$. It follows from Property 2 that the number of cuts in B is at most $cD/2d$. We show that this quantity can be made small by an appropriate choice of arc weights.

We now define the weight assignment. Let $k = \lfloor 2\alpha \rfloor$. For a given arc (T, C), if tree T r-constrains cut C for some $r \leq k$ but does not $(r+1)$-constrain C, then we give the arc weight $1 + k - r$. If more than k edges of T cross the cut C, we give the arc weight 0. Since a tree can exactly r-constrain at most $\binom{n}{r}$ distinct cuts, if follows that the tree-side degree

$$D \leq \sum_{1 \leq r \leq k} \binom{n}{r}(1 + k - r) = \binom{n}{k}(1 + O(1/n)).$$

Now consider the cut-side degree d. Take any cut C. For tree T, let r_T denote the number of edges of T that cross C. Then the weighted degree of C can be written as

$$\sum_{r_T \leq k} (1 + k - r_T) \geq \sum_T (1 + k - r_T)$$

(since all additional terms on the right hand side are negative)

$$= (1 + k)(c/2) - \sum_T r_T$$

$$\geq (1 + k)(c/2) - \alpha c$$

(since each of the αc edges of the cut contribute to r_T for at most one T)

$$= (1 + k - 2\alpha)(c/2).$$

This is a lower bound on the weight d incident on any cut-side node in B. Since we have shown above that $D \leq \binom{n}{k}(1 + O(1/n))$, it follows that our bound on the number of cut nodes, $cD/2d$, is no more than the bound claimed in the statement of the theorem.

Note that our weight assignment analysis works for any value of k, but that taking $k = \lfloor 2\alpha \rfloor$ gives the best ratio. An argument based on linear programming duality can be used to show that this proof cannot be improved: no other assignment of arc weights can give a better bound. We defer this argument to the appendix. $\square$

3.1 Weighted Graphs

Although our proof discussed only unweighted graphs, it clearly extends to weighted graphs as well. One way to see this is to redo the proof with a weighted tree packing, but this becomes notationally messy. A simpler way to extend the argument to graphs with integer edge weights is to replace a weight-w edge with w parallel unweighted edges, creating an unweighted graph with the same near-minimum cuts. This argument can be extended to graphs with rational edge weights as well: simply multiply all edge weights by their least common denominator, creating an integer-weighted graph with the same near-minimum cuts. Finally, real valued edge weights can be handled by considering them as the limits of rational-valued edge weight sequences.

3.2 Discussion

Note that a cycle has exactly $\binom{n}{\lfloor 2\alpha \rfloor} + \cdots + \binom{n}{2}$ minimum cuts. Thus for α a half integer, our upper bound is tight to within $1 + o(1)$. The gap widens when α is between two half-integers, reaching $\Theta(n)$ when α is infinitesimally less than a half-integer. No family of graphs has been exhibited which has more α-minimum cuts than the cycle, so one is tempted to conjecture that the bound for the cycle is also the upper bound. The only counterexample of which we are aware is the 4-clique [NNI94], which has 10 4/3-minimum cuts (each singleton or pair of vertices) as compared to the 6 of a 4-cycle. But perhaps this is a unique exception.

Another interesting open question regards the number of cuts *strictly less* than α times the minimum. For α a half integer, our above theorem only gives a bound of $O(n^{\lceil 2\alpha \rceil})$; a bound of $O(n^{\lfloor 2\alpha \rfloor})$, as exhibited by the cycle, seems more plausible. Such a bound has been proven for $\alpha = 3/2$ [HW96].

4 Algorithms for Finding Good Trees

We now apply the ideas of the previous combinatorial sections algorithmically. We prove that in any graph, we can find quickly a small set of trees such that the minimum cut 2-respects some of them. Therefore, we can find the minimum cut by enumerating only the cuts that 2-respect these few trees.

4.1 Packing Algorithms

Finding the trees is in a sense trivial from the preceding discussion, which showed that a maximum tree packing had the right property. Unfortunately, maximum tree packings in undirected graphs are hard to find. Gabow and Westermann [GW92] gave an algorithm for unweighted graphs that runs in $\tilde{O}(\min(mn, m^2/\sqrt{n}))$ time. Barahona [Bar95] gives an algorithm with an $\tilde{O}(mn)$ running time for weighted graphs. Both of these running times are dramatically larger than the one we want to achieve, so we will use two other approaches that find non-maximum tree packings.

One approach is due to Gabow. Although it does not yield a maximum packing, it yields a packing of value $c/2$—sufficient for our purposes. Gabow's algorithm is actually an algorithm for directed graphs: in such a graph, it finds c disjoint directed spanning trees (guaranteed by Edmonds' theorem [Edm72]) in $O(mc\log(n^2/m))$ time. We can use this algorithm by turning each of our undirected graph's edges into two edges, one in each direction. This gives us a directed graph with directed minimum cut c. Gabow's algorithm finds a packing of c directed-edge disjoint trees in this graph. If we now ignore edge directions, we get a set of c trees such that each of our undirected edges is in at most 2 trees. It follows that if we give each of our trees a weight of $1/2$, we have a packing of value $c/2$ to which our previous arguments (Lemma 2.3) apply.

An alternative approach is due to Plotkin, Shmoys, and Tardos [PST91]. They give an algorithm that packs spanning trees by repeatedly adding one weighted tree to the packing. In each iteration, which tree to add and the weight it is given are determined by a minimum-cost spanning tree computation, using costs determined by the current packing. After $\tilde{O}(c/\epsilon^2)$ iterations (which take time $\tilde{O}(mc/\epsilon^2)$), the packing has value at least $(1-\epsilon)$ times the maximum, and thus at least $(1-\epsilon)c/2$. Although it is more complicated that Gabow's algorithm in that it works with fractional values, this method has the attraction of working only with undirected spanning trees, emphasizing that our algorithm is able to avoid any reliance on directed-graph subroutines. It will also be useful when we develop parallel algorithms.

4.2 Sampling

Unfortunately, both of the above algorithms have running times dependent on c, so using them directly is unacceptable. So we use a random sampling step to reduce the effective minimum cut in the graph to $O(\log n)$, which means that both schemes can run in $\tilde{O}(m)$ time.

Theorem 4.1. *Given any weighted undirected graph G, in $O(m + n\log^3 n)$ time we can construct a set of $O(\log n)$ spanning trees such that the minimum cut 2-respects $1/3$ of them with high probability.*

Proof. In a previous paper [Kara], we showed how to construct, in linear time for any ϵ, a *skeleton* graph H on the same vertices with the following properties:

- H has $m' = O(n\epsilon^{-2}\log n)$ edges,
- the minimum cut of H is $c' = O(\epsilon^{-2}\log n)$,
- the minimum cut in G corresponds (under the same vertex partition) to a $(1+\epsilon)$-times minimum cut of H.

We already know that any tree packing of value $c'/2$ in the skeleton H will have many trees that 2-constrain the minimum cut in H. Intuitively, for small ϵ, since the minimum cut in G is a near-minimum cut in H, the tree packing will also contain many trees that 2-constrain this near-minimum cut. We now formalize this intuition.

Set $\epsilon = 1/6$ in the skeleton construction. Since H has minimum cut c', Gabow's algorithm can be used to find a packing in H of weight $c'/2$ in $O(m'c'\log n) = O(n\log^3 n)$ time. The original minimum cut of G has at most $(1+\epsilon)c'$ edges in H, so by Lemma 2.3, a $\frac{1}{2}(1-2\epsilon) = 1/3$-fraction of the trees 2-constrain this cut in H. But this $(1+\epsilon)$-minimum cut of H has the same vertex partition as the minimum cut of G, implying that the same trees 2-constrain the minimum cut of G. □

Remark. The randomized construction of the skeleton in this proof is the only step of our minimum cut algorithm that involves randomization. A deterministic replacement of Theorem 4.1 would yield a deterministic minimum cut algorithm.

An alternative construction based on the Plotkin-Shmoys-Tardos algorithm can also be applied to the sample. We can use that algorithm to find a tree packing of value $(1-\delta)c'/2$ in $O(n\log^3 n)$ time for any constant δ. Lemma 2.3 again shows that for $\delta = \epsilon = 1/6$, say, a $1/10^{th}$ fraction of the packed weight 2-constrains the $(1+\epsilon)$-minimum cut in H that corresponds to the minimum cut in G. Thus, if we choose a random tree (with probability proportional to its weight), there is a constant probability that it 2-constrains the minimum cut. Performing $O(\log n)$ such random selections picks at least one tree that 2-constrains the minimum cut with high probability. In fact, since the algorithm only packs $O((\log n)/\delta^2)$ distinct trees we can even afford to try them all. This gives another $O(m + n\log^3 n)$-time algorithm for finding a good set of $O(\log n)$ trees. It also has the following corollary that will be applied to our parallel minimum cut algorithm.

Corollary 4.2. *In $\mathcal{RNC}$ using $m+n\log n$ processors and $O(\log^3 n)$ time, we can find a set of $O(\log n)$ spanning trees such that with high probability, a constant fraction of them (by weight) 2-constrain the minimum cut.*

Proof. The skeleton construction can be performed as before (and is trivial to parallelize [Kara]). We have just argued that the algorithm of [PST91] can find an approximately maximum packing in the skeleton using $O(\log^2 n)$ minimum spanning tree computations. Minimum spanning trees can be found in parallel in $O(\log n)$ time [JM92] using $m' = O(n\log n)$ processors, and the other operation of [PST91] are trivial to parallelize with the same time and processor bounds. Thus the claimed bounds follow. □

The remainder of this paper is devoted to the following question: given a tree, find the minimum cut that 2-respects it. Applying the solution to the $O(\log n)$ trees described by Theorem 4.1 shows that we can find the minimum cut with high probability. In particular we have the following lemma:

Lemma 4.3. *Suppose the minimum cut that 2-respects a given tree can be found in $T(m,n)$ time. Then the minimum cut of a graph can be found with constant probability in $T(m,n) + O(m + n\log^3 n)$ time and with high probability in $O(T(m,n)\log n + m + n\log^3 n)$ time.*

Proof. We have seen above that a constant fraction (by weight) of the trees in our skeleton packing 2-constrain the minimum cut. So choosing a tree at random and analyzing it yields the minimum cut with constant probability. Trying all $O(\log n)$ trees identifies the minimum cut with high probability so long as the skeleton construction worked, which happens with high probability. □

We give two algorithms for analyzing a tree. The first algorithm (in Section 6) uses the fact that a tree contains only $\binom{n}{2}$ pairs of edges, one of which defines the minimum cut 2-respecting the tree. A very simple dynamic program computes the cut values defined by all $\binom{n}{2}$ pairs of edges in $O(n^2)$ time. The second algorithm (in Section 7) aims for a linear-time bound, and must therefore avoid enumerating all pairs of edges. We describe local optimality conditions showing that only certain pairs of edges can possibly be the pair defining the minimum cut, and give an algorithm that runs quickly by only trying these plausible pairs.

5 Minimum Cuts that 1-Respect a Tree

To introduce our approach to analyzing a particular tree, we consider a simple special case: a tree that *1-constrains* the minimum cut. In such a tree there is one tree edge such that, if we remove it, the two resulting connected subtrees correspond to the two sides of the minimum cut. This section is devoted to a proof of the following:

Lemma 5.1. *The values of all cuts that* 1*-respect a given spanning tree can be determined in* $O(m+n)$ *time.*

Corollary 5.2. *The minimum cut that* 1*-respects a given spanning tree can be found in* $O(m+n)$ *time.*

We have already seen that being able to identify all cuts that 2-respect a tree will let us find the minimum cut. Later, we will exhibit a class of graphs for which finding the minimum 1-respecting cut will suffice for finding the minimum cut.

We begin with some definitions. Suppose that we root the tree at an arbitrary vertex.

Definition 5.3. $v^{\downarrow}$ is the set of vertices that are descendants of v in the rooted tree, including v.

Definition 5.4. $v^{\uparrow}$ is the set of vertices that are ancestors of v in the rooted tree, including v.

Note that $v^{\downarrow} \cap v^{\uparrow} = v$, where we allow v to denote the set $\{v\}$.

Definition 5.5. $\mathcal{C}(X, Y)$ is the total weight of edges with one endpoint in vertex set X and the other in vertex set Y. An edge with both endpoints in both sets is counted twice.

Thus, $\mathcal{C}(v, w)$ is the weight of the edge (v, w) if it exists, and 0 otherwise. For the most part we will focus on disjoint X and Y where double-counting edges is irrelevant.

Definition 5.6. $\mathcal{C}(S)$ is the value of the cut whose one side is vertex set S, *i.e.* $\mathcal{C}(S, \overline{S})$.

Once we root the tree, the cuts that 1-respect the tree have the form $\mathcal{C}(v^{\downarrow})$ for vertices v. Using this observation, we now prove Lemma 5.1.

5.1 1-respecting a path

As a first step, suppose the tree is in fact a path $v_1, \ldots, v_n$ rooted at v_1. We compute all values $\mathcal{C}(v_i^{\downarrow})$ in linear time using a dynamic program. First compute $\mathcal{C}(v_i, v_i^{\uparrow})$ and $\mathcal{C}(v_i, v_i^{\downarrow})$ for each v_i; this takes one $O(m)$-time traversal of the vertices' adjacency lists. We now claim that the following recurrence applies to the cut values:

$$\begin{aligned}
\mathcal{C}(v_n^{\downarrow}) &= \mathcal{C}(v_n, v_n^{\uparrow}) \\
\mathcal{C}(v_i^{\downarrow}) &= \mathcal{C}(v_{i+1}^{\downarrow}) + \mathcal{C}(v_i, v_i^{\uparrow}) - \mathcal{C}(v_i, v_i^{\downarrow})
\end{aligned}$$

This recurrence follows from the fact that when we move v_i from below the cut to above it, the edges from v_i to its descendants become cut edges while those from v_i to its ancestors stop being cut edges. It follows that all n cut values can be computed in $O(n)$ time by working up from v_n.

5.2 1-respecting a tree

We now extend our dynamic program from paths to general trees.

Definition 5.7. Given a function f on the vertices of a tree, the *treefix sum* of f, denoted $f^{\downarrow}$, is the function

$$f^{\downarrow}(v) = \sum_{w \in v^{\downarrow}} f(w).$$

Lemma 5.8. *Given the values of a function* f *at the tree nodes, all values of* $f^{\downarrow}$ *can be computed in* $O(n)$ *time.*

Proof. Perform a postorder traversal of the nodes. When we visit a node v, we already will have computed (by induction) the values at each of its children. Adding these values takes time proportional to the degree of v; adding in $f(v)$ gives us $f^{\downarrow}(v)$. Thus, the overall computation time is proportional to the sum of the node degrees, which is just the number of tree edges, namely $n-1$. □

We now compute the values $\mathcal{C}(v^{\downarrow})$ via treefix sums. Let $\delta(v)$ denote the (weighted) degree of vertex v. Let $\rho(v)$ denote the total weight of edges whose endpoints' least common ancestor is v.

Lemma 5.9. $\mathcal{C}(v^{\downarrow}) = \delta^{\downarrow}(v) - 2\rho^{\downarrow}(v)$.

Proof. The term $\delta^{\downarrow}(v)$ counts all the edges leaving descendants of v. This correctly counts each edge crossing the cut defined by $v^{\downarrow}$, but also double-counts all edges with both endpoints descended from v. But an edge has both endpoints descended from v if and only if its least common ancestor is in $v^{\downarrow}$. Thus the total weight of such edges is $\rho^{\downarrow}(v)$. □

Now note that the functions $\delta(v)$ and $\rho(v)$ can both be computed for all v in $O(m)$ time. Computing δ is trivial. Computing ρ is trivial if we know the least common ancestor of each edge, but these can be determined in $O(m)$ time [GT85, BV93, SV88]. From these quantities, according to Lemma 5.8, two $O(n)$-time treefix computations suffice to determine the minimum cut. This proves Lemma 5.1.

In Section 8, we will describe a class of graphs for which this "1-respects" test is sufficient to find the minimum cut.

6 Minimum Cuts that 2-respect a Tree in $O(n^2)$ Time

We now extend our dynamic program to find the minimum cut that 2-respects a given tree in $O(n^2)$ time. This will immediately yield an $O(n^2 \log n)$-time algorithm for finding all minimum cuts. The dynamic program can be parallelized, yielding an n^2-processor $\mathcal{RNC}$ algorithm for the problem. The trace of the dynamic program also provides a small-space data structure representing all the near-minimum cuts.

6.1 A sequential algorithm

This section is devoted to proving the following:

Lemma 6.1. *The values of all cuts that 2-respect a given tree can be found in $O(n^2)$ time.*

A sequential algorithm is an immediate corollary (using Lemma 4.3):

Theorem 6.2. *There is a Monte Carlo algorithm that finds a minimum cut with constant probability in $O(n^2)$ time and finds all minimum cuts with high probability in $O(n^2 \log n)$ time.*

We now prove the lemma. After rooting the given tree and applying the algorithm of Section 5 in case some tree 1-constrains the minimum cut, we can assume that (for the right tree) exactly 2 tree-edges cross the minimum cut. These two edges are the parent edges of two vertices v and w.

Definition 6.3. We say vertices v and w are *incomparable*, writing $v \perp w$, if $v \notin w^{\downarrow}$ and $w \notin v^{\downarrow}$. In other words, v and w incomparable if they are not on the same root-leaf path.

Suppose the minimum cut that 2-respects the tree cuts the tree-parent edges of vertices v and w. If $v \perp w$, then the parent edges of v and w define a cut with value $\mathcal{C}(v^{\downarrow} \cup w^{\downarrow})$. If (without loss of generality) $v \in w^{\downarrow}$, then their parent edges define a cut with value $\mathcal{C}(w^{\downarrow} - v^{\downarrow})$. We start with the first case. Assuming $v \perp w$,

$$\mathcal{C}(v^{\downarrow} \cup w^{\downarrow}) = \mathcal{C}(v^{\downarrow}) + \mathcal{C}(w^{\downarrow}) - 2\mathcal{C}(v^{\downarrow}, w^{\downarrow})$$

because $\mathcal{C}(v^{\downarrow})$ and $\mathcal{C}(w^{\downarrow})$ each include the edges of $\mathcal{C}(v^{\downarrow}, w^{\downarrow})$ that should not be counted in $\mathcal{C}(v^{\downarrow} \cup w^{\downarrow})$. Since the values $\mathcal{C}(v^{\downarrow})$ are already computed for all v, we need only compute $\mathcal{C}(v^{\downarrow}, w^{\downarrow})$ for every v and w. This can be done using treefix sums as in the previous section. First, let

$$f_v(w) = \mathcal{C}(v, w)$$

be the weight of the edge connecting v to w, or 0 if there is no such edge. It follows that

$$f_v^{\downarrow}(w) = \mathcal{C}(v, w^{\downarrow}).$$

Therefore, we can determine all n^2 values

$$g_w(v) = \mathcal{C}(v, w^{\downarrow}) = f_v^{\downarrow}(w)$$

in $O(n^2)$ time by performing n treefix computations, one for each f_v. At this point, since

$$g_w^{\downarrow}(v) = \mathcal{C}(v^{\downarrow}, w^{\downarrow}),$$

we can determine all the desired quantities with another n treefix sums, one for each g_w.

Now consider the second case of $v \in w^{\downarrow}$. In this case,

$$\mathcal{C}(w^{\downarrow} - v^{\downarrow}) \quad = \quad \mathcal{C}(w^{\downarrow}) - \mathcal{C}(v^{\downarrow}) + 2\mathcal{C}(v^{\downarrow}, w^{\downarrow} - v^{\downarrow}),$$

for subtracting $\mathcal{C}(v^{\downarrow})$ from $\mathcal{C}(w^{\downarrow})$ correctly subtracts the edges connecting $v^{\downarrow}$ to $w^{\uparrow}$, but also incorrectly subtracts all the edges running from $v^{\downarrow}$ to $w^{\downarrow} - v^{\downarrow}$. But we claim that

$$\mathcal{C}(v^{\downarrow}, w^{\downarrow} - v^{\downarrow}) = g_w^{\downarrow}(v) - 2\rho^{\downarrow}(v).$$

For $g_w^{\downarrow}(v)$ counts every edge with one endpoint in $v^{\downarrow}$ and the other in $w^{\downarrow}$, which includes everything we want to count but also incorrectly counts (twice) all edges with both endpoints in $v^{\downarrow}$. This is cancelled by the subtraction of $2\rho^{\downarrow}(v)$.

Thus, once we have computed each g_w, computing the cut value for each pair $v \in w^{\downarrow}$ in $O(n^2)$ time is trivial.

6.2 A parallel algorithm

We can parallelize the above treefix-based algorithms. Note that they involve two main steps: finding a proper packing of trees and finding the minimum cut constrained by each tree. Recall from Corollary 4.2 that the algorithm of [PST91] can be used to find a satisfactory packing in $O(\log^3 n)$ time using $m + n \log n$ processors. So we need only parallelize the above algorithms for a particular tree. But the only computations performed there are least common ancestor lookups, which can be performed optimally in parallel by, e.g., the algorithm of Schieber and Vishkin [SV88]; and treefix sum computations, which can also be performed optimally in parallel [KR90, Section 2.2.3].

Corollary 6.4. *Minimum cuts can be found in $\mathcal{RNC}$ with high probability in $O(\log^3 n)$ time using $n^2/\log^2 n$ processors.*

6.3 Data Structures

Previously, a linear-space representation of all minimum cuts was known [DKL76], but the best representation of approximately minimum cuts for general α required $\Theta(n^{2\alpha})$ space [KS96]. Benczur [Ben95] gave an $O(n^2)$-space representation for $\alpha < 6/5$. We can now do better.

Theorem 6.5. *Given a graph with k α-minimum cuts, $\alpha < 3/2$, there is a data structure that represents them in $O(k + n \log n) = O(n^2)$ space that can be constructed in $O(n^2 \log n)$ time or in $\mathcal{RNC}$ using n^2 processors. With this data structure,*

1. *the value of a near-minimum cut can be looked up in $O(n)$ time, and*

2. *whether two vertices are on the same side of a given near-minimum cut can be determined in $O(\log^2 n)$ time.*

Proof. The output of our $\tilde{O}(n^2)$-time algorithm can serve as a data structure for representing the near-minimum cuts that 2-respect the trees it analyzes. We simply list the pairs of edges that together induce a near-minimum cut. Now consider any $\alpha < 3/2$. We know that with high probability every α-minimum cut 2-respects one of the $O(\log n)$ trees we inspect and will therefore be found.

Notice that given a cut (specified by a vertex partition) we can use this data structure to decide if it is α-minimum, and if so determine its value, in $O(n \log n)$ time: just check, for each of the $O(\log n)$ trees in the data structure, whether at most 2 tree edges cross the cut. If the cut is a small cut, there will be some tree for which this is true. We will have recorded the value of the cut with the edge pair (or singleton) that crosses it.

We can improve this query time to $O(n)$ if we use a perfect hash function to map each vertex set that defines a small cut to a size-$O(k)$ table that says which tree that particular small cut 2-respects. Given a query (vertex set) we can map that vertex set into the hash table and check the one tree indicated by the appropriate table entry.

Given the pair of tree edges that specify a cut, we can decide whether two vertices are on the side. We consider the tree, and ask whether the two given vertices remain connected in the tree as we remove one or the other of the cut edges. Such questions can be answered in $O(\log n)$ time using an *Euler Tour Tree* data structure [HK95, HdLT98]. □

In the sequential case, our data structure can actually be constructed deterministically if we are willing to take some extra time. Perform the initial tree packing in the original graph rather than the skeleton. By Lemma 2.3, we know that any one minimum cut 2-respects at least half the trees of this packing. Thus, if we pick a random tree, we expect it to 2-constrain half the minimum cuts. It follows that some tree 2-constrains half the minimum cuts. We can find such a tree *deterministically* via an exhaustive search. First, enumerate all the small cuts using, for example, the deterministic algorithm of [VY92]. Then, check each tree to find one that 2-constraints half the minimum cuts. In the case of an unweighted graph, we must check $c < m$ trees. In a weighted graph, we can start from a maximum packing with a polynomial number of distinct trees in it [Bar95] so that we can check them all in polynomial time. We now recursively find trees that 2-constrain the remaining minimum cuts that the first tree missed. Since we halve the number of remaining min-cuts each time, we only need $\log_2 n^{2\alpha}$ trees.

We can easily extend this approach to represent α-minimum cuts for any constant α. Theorem 4.1 generalizes to show that of the $O(\log n)$ trees we find there, a constant fraction will $\lfloor 2\alpha \rfloor$-respect any given α-minimum cut with high probability. We can therefore represent all α-minimum cuts by considering all sets of $\lfloor 2\alpha \rfloor$ edges in each tree. For any one tree, the values induced by all sets of $\lfloor 2\alpha \rfloor$ edges can be found in $O(n^{\lfloor 2\alpha \rfloor})$ time using treefix computations as above.

Lemma 6.6. *There is a data structure that represents all k of the α-minimum cuts in a graph using $O(n \log n + k\alpha) = O(\alpha n^{\lfloor 2\alpha \rfloor})$ space. Its randomized construction requires $O(n^{\lfloor 2\alpha \rfloor})$ time and space or $O(n^{\lfloor 2\alpha \rfloor})$ processors in $\mathcal{RNC}$. It can also be constructed deterministically in polynomial time.*

7 Near-Linear Time

We now reconsider the algorithm of Section 6 and show how its objective can be achieved in $O(m \log^2 n)$ time on any given tree. In this section we prove the following lemma.

Lemma 7.1. *The minimum cut that 2-respects a tree can be found in $O(m \log^2 n)$ time.*

Applying Lemma 4.3 yields our main theorem:

Theorem 7.2. *The minimum cut of a graph can be found with constant probability in $O(m \log^2 n + n \log^3 n)$ time or with high probability in $O(m \log^3 n)$ time.*

To prove the lemma, as in Section 6, we first consider finding the pair minimizing $\mathcal{C}(v^{\downarrow} \cup w^{\downarrow})$ for $v \perp w$; the almost identical case of minimizing $\mathcal{C}(w^{\downarrow} - v^{\downarrow})$ for $v \in w^{\downarrow}$ is deferred to the end of the section. Rather than explicitly computing $\mathcal{C}(v^{\downarrow} \cup w^{\downarrow})$ for all pairs v and w, which clearly takes $\Theta(n^2)$ time, we settle for finding for each v a vertex w that minimizes $\mathcal{C}(v^{\downarrow} \cup w^{\downarrow})$. This will clearly let us identify the minimum cut.

7.1 Precuts

Recall that for any vertices $w \perp v$,

$$\mathcal{C}(v^{\downarrow} \cup w^{\downarrow}) = \mathcal{C}(v^{\downarrow}) + \mathcal{C}(w^{\downarrow}) - 2\mathcal{C}(v^{\downarrow}, w^{\downarrow}).$$

We now factor out the contribution of $\mathcal{C}(v^{\downarrow})$ to this quantity:

Definition 7.3. The *v-precut at w,* denoted $\mathcal{C}_v(w)$, is the value

$$\mathcal{C}_v(w) = \mathcal{C}(v^{\downarrow} \cup w^{\downarrow}) - \mathcal{C}(v^{\downarrow}) = \mathcal{C}(w^{\downarrow}) - 2\mathcal{C}(v^{\downarrow}, w^{\downarrow})$$

if $w \perp v$ and ∞ otherwise.

Definition 7.4. The *minimum v-precut*, denoted $\mathcal{C}_v$, is the value

$$\min\{\mathcal{C}_v(w) \mid \exists (v', w') \in E,\ v' \in v^{\downarrow},\ w' \in w^{\downarrow}\}$$

The point of this somewhat odd definition is to notice that for a given v, only certain vertices w are candidates for forming the minimum cut with v, as is seen in the following lemma.

Lemma 7.5. *If it is determined by incomparable vertices, the minimum cut is* $\min_v(\mathcal{C}(v^{\downarrow}) + \mathcal{C}_v)$.

Proof. Let the minimum cut be induced by v and a vertex $w \perp v$. It suffices to show that w is in the set over which we minimize to define $\mathcal{C}_v$ in Definition 7.4. But this follows from the fact that each side of the minimum cut induces a connected subgraph of G, for this implies that there must be an edge (v', w') from $v^{\downarrow}$ to $w^{\downarrow}$. To see that each side of the minimum cut must be connected, note that if one side of the minimum cut has two disconnected components then we can reduce the cut value by moving one of these components to the other side of the cut. □

Suppose that we have already used the linear-time procedure of Section 5 to compute the values $\mathcal{C}(v^{\downarrow})$ for every v. It follows from Lemma 7.5 that it suffices to compute $\mathcal{C}_v$ for every vertex v: the minimum cut can then be found in $O(n)$ time. We now show how these minimum precuts can be computed in $O(m \log^2 n)$ time on any tree. We first show that we can find $\mathcal{C}_v$ for a leaf v in time proportional to its degree. We then extend this approach to handle a *bough*—a path rising from a leaf until it reaches a vertex with more than one child. Finally, we handle general trees by repeatedly pruning boughs.

We begin by giving each vertex w a variable `val`$[w]$. While working on vertex v, this variable will be used to accumulate $\mathcal{C}_v(w)$. Initially, we set `val`$[w] = \mathcal{C}(w^{\downarrow})$.

7.2 A Leaf

First consider a particular leaf v. To compute $\mathcal{C}_v$, consider the following procedure:

1. For each vertex w, subtract $2\mathcal{C}(v, w^{\downarrow})$ from `val`$[w]$ (so that `val`$[w] = \mathcal{C}_v(w)$).

2. $\mathcal{C}_v$ is the minimum of `val`$[w]$ over all $w \perp v$ that are ancestors of a neighbor of v.

The correctness of this procedure follows from Lemma 7.5, the initialization of `val`$[w]$, and the definition of $\mathcal{C}_v$.

To implement this procedure efficiently, we use the *dynamic tree* data structure of Sleator and Tarjan [ST83]. Given a tree, this structure supports (among others) the following operations on a node v:

`AddPath`(v, x): add x to `val`$[u]$ for every $u \in v^{\uparrow}$.

`MinPath`(v): return $\min_{u \in v^{\uparrow}}$ `val`$[u]$ as well as the u achieving this minimum.

Each such dynamic tree operation takes $O(\log n)$ amortized time on an n-vertex tree.

We use dynamic trees to compute $\mathcal{C}_v$. Recall that $\mathcal{C}(v, u)$ is the weight of the edge connecting vertices v and u, or 0 if no such edge exists. Apply procedure `LocalUpdate` from Figure 7.2.

Algorithm `LocalUpdate`(v)

1. Call `AddPath`(v, ∞).
2. For each edge (v, u), call `AddPath`$(u, -2\mathcal{C}(v, u))$.
3. For each edge (v, u), call `MinPath`(u)
4. Return the minimum result of Step 3.

Figure 1: procedure `LocalUpdate`

Lemma 7.6. *After Step 2 in a call to* `LocalUpdate`(v), *every* w *has* `val`$[w] = \mathcal{C}_v(w)$.

Proof. Step 1 assigns an infinite `val`$[w]$ to every ancestor w of v, as required in the definition of $\mathcal{C}_v(w)$. Next, $2\mathcal{C}(v, u)$ is subtracted in Step 2 from every `val`$[w]$ such that $u \in w^{\downarrow}$. Therefore, the total amount subtracted from `val`$[w]$ is $2\mathcal{C}(v, w^{\downarrow})$, as required in Definition 7.3. It follows that after Step 2, `val`$[w] = \mathcal{C}_v(w)$. □

Lemma 7.7. `LocalUpdate`(v) *called on a leaf* v *returns* $\mathcal{C}_v$.

Proof. Step 3 minimizes over $w \perp v$ that are ancestors of neighbors of v. According to Definition 7.4 this will identify $\mathcal{C}_v$. □

It follows that for a leaf v with d incident edges, we can find $\mathcal{C}_v$ via $O(d)$ dynamic tree operations that require $O(d \log n)$ time.

7.3 A Bough

We generalize the above approach. Let a *bough* be a maximal path starting at a leaf and traveling upwards until it reaches a vertex with more than one child (this vertex is not in the bough).

Lemma 7.8. *Let* v *be a vertex with a unique child* u. *Then either* $\mathcal{C}_v = \mathcal{C}_u$, *or else* $\mathcal{C}_v = \mathcal{C}_v(w)$ *for some ancestor* w *of a neighbor of vertex* v.

In other words, given the value $\mathcal{C}_u$, we need only "check" ancestors of neighbors of v to determine $\mathcal{C}_v$.

Proof. We know that $\mathcal{C}_v = \mathcal{C}_v(w)$ for a certain w. Suppose w is not an ancestor of a neighbor of v. Then there is no edge from v to $w^{\downarrow}$, meaning $\mathcal{C}(v^{\downarrow}, w^{\downarrow}) = \mathcal{C}(u^{\downarrow}, w^{\downarrow})$. It follows that $\mathcal{C}_u \leq \mathcal{C}_u(w) = \mathcal{C}_v(w) = \mathcal{C}_v$. But $u \in v^{\downarrow}$ implies that for any x, $\mathcal{C}(u^{\downarrow}, x^{\downarrow}) \leq \mathcal{C}(v^{\downarrow}, x^{\downarrow})$, so $\mathcal{C}_u \geq \mathcal{C}_v$. Therefore $\mathcal{C}_u = \mathcal{C}_v$. □

We use the above lemma in an algorithm for processing an entire bough. Given a bough with d edges incident in total on its vertices, we show how to process all vertices in the bough in $O(d \log n)$ time. We use the recursive procedure `MinPrecut` from Figure 7.3. This procedure is initially called on the topmost vertex in the bough. Although we have formulated the procedure recursively to ease the proof exposition, it may be more natural to unroll the recursion and think of the algorithm as executing up from a leaf.

We claim that `MinPrecut` computes the desired quantity. To prove this, we prove a stronger statement by induction:

Lemma 7.9. *A call to* `MinPrecut`(v)

1. *sets* `val`$[w] = \mathcal{C}_v(w)$ *for each* w, *and*
2. *returns* $\mathcal{C}_v$.

```
Algorithm MinPrecut(v)

if v is the leaf of the bough
    call LocalUpdate(v) (also computes and returns C_v)
else
    Let u be the child of v
    c_1 ← MinPrecut(u) (updates some val[·] entries)
    c_2 ← LocalUpdate(v)
    return min(c_1, c_2)
```

Figure 2: procedure `MinPrecut`

Proof. By induction. We first prove Claim 1. The base case is for v a leaf, and follows from our analysis of `LocalUpdate` for a leaf in Lemmas 7.6 and 7.7. Now suppose v has child u. After calling `MinPrecut`(u), we know by induction that `val`$[w] = \mathcal{C}_u(w)$. Thus, after we execute `LocalUpdate`(v), which (as shown in Lemma 7.6) decreases each `val`$[w]$ by $\mathcal{C}(v, w^{\downarrow})$, entry `val`$[w]$ is updated to be

$$\begin{aligned} \mathcal{C}_u(w) - 2\mathcal{C}(v, w^{\downarrow}) &= (\mathcal{C}(w^{\downarrow}) - 2\mathcal{C}(u^{\downarrow}, w^{\downarrow})) - 2\mathcal{C}(v, w^{\downarrow}) \\ &= \mathcal{C}(w^{\downarrow}) - 2\mathcal{C}(v \cup u^{\downarrow}, w^{\downarrow}) \\ &= \mathcal{C}(w^{\downarrow}) - 2\mathcal{C}(v^{\downarrow}, w^{\downarrow}) \\ &= \mathcal{C}_v(w) \end{aligned}$$

We now prove Claim 2. Suppose v has child u. According to Lemma 7.8, there are two possibilities for $\mathcal{C}_v$. One possibility is that $\mathcal{C}_v = \mathcal{C}_u$. But by induction, the recursive call to `MinPrecut`(u) sets $c_1 = \mathcal{C}_u$. On the other hand, if $\mathcal{C}_v \neq \mathcal{C}_u$, then by Lemma 7.8, $\mathcal{C}_v = \mathcal{C}_v(w)$ for an ancestor of a neighbor of v. But then c_2 is set to $\mathcal{C}_v$ when we call `LocalUpdate`(v). In either case, the correct value is $\min(c_1, c_2)$. □

Based on this lemma, the correctness of the algorithm for finding all minimum precut values in a bough is immediate.

7.4 A Tree

We now use the bough algorithm repeatedly to analyze a particular tree. We still assume for now that the cut is defined by $v \perp w$; we consider the other case in the next subsection. We have already argued that we can perform the computation for a single bough using a number of dynamic tree operations proportional to the number of edges incident on the bough. Doing so will change the values `val`$[w]$. However, we can return them to their original values by repeating the execution of `MinPrecut` on the bough, replacing each addition operation by a subtraction.

It follows that we can process a collection of boughs in sequence, since each computation on a bough starts with the `val`$[w]$ variables back at their initial values $\mathcal{C}(w^{\downarrow})$. Now note that every edge of G is incident on at most 2 boughs. It follows that the total size (number of incident edges) of all boughs of our tree is $O(m)$. Therefore, the time to process all boughs of the tree with the algorithm of the previous section is $O(m \log n)$.

Once we have processed all boughs, we know that we have found the minimum cut if even one of the two tree-edges it cuts is in a bough. If not, we can contract each bough of the tree (that is, merge all vertices in a bough into the vertex from which that bough descends) without affecting the minimum cut. This can be done in $O(m)$ time (using a bucket sort on the edge endpoints). It leaves us with a new tree that 2-constrains the minimum cut of the contracted graph. Furthermore, every leaf of the new tree had at least two descendants in the original tree (else it would have been in a bough). This means that the new tree has at most half as many leaves as the old. It follows that after $O(\log n)$ iterations of the procedure for boughs followed by contractions of boughs, we will have reduced the number of tree-leaves to 0, meaning that we will have contracted the entire graph. Clearly at some time before this, we found the minimum cut.

7.5 Comparable v and w

It remains to explain the procedure for finding $\mathcal{C}_v(w)$ when (without loss of generality) $v \in w^{\downarrow}$. We proceed much as before. In Section 6, we showed that

$$\mathcal{C}(w^{\downarrow} - v^{\downarrow}) = \mathcal{C}(w^{\downarrow}) - \mathcal{C}(v^{\downarrow}) + 2(g_w^{\downarrow}(v) - 2\rho^{\downarrow}(v)).$$

It is therefore sufficient to compute, for each v, after factoring out $\mathcal{C}(v^{\downarrow}) + 4\rho^{\downarrow}(v)$, the quantity

$$\mathcal{C}_v(w)' = \min_{w \in v^{\uparrow}} \mathcal{C}(w^{\downarrow}) + 2g_w^{\downarrow}(v).$$

We again do so using the dynamic tree data structure. We initialize $\mathtt{val}[w] = \mathcal{C}(w^{\downarrow})$. We then use dynamic tree operations to add $2g_w^{\downarrow}(v)$ to $\mathtt{val}[w]$. Suppose first that v is a leaf. Then for each neighbor u of v, we call $\mathtt{AddPath}(u, 2\mathcal{C}(v,u))$. Having done so, we can extract our answer as $\mathtt{MinPath}(v)$. This follows because $g_w^{\downarrow}(v) = \sum_{x \in v^{\downarrow}} \mathcal{C}(x, w^{\downarrow})$. So to add $2g_w^{\downarrow}(v)$ to w, we want to add twice the capacity of every edge from v to each descendant $u \in w^{\downarrow}$.

Also as before, we can process an entire bough by working up from the leaf. For each vertex v' in the bough, we execute $\mathtt{AddPath}(u', \mathcal{C}(v', u'))$ operations for each neighbor u' of v' and then compute $\mathtt{MinPath}(v')$. By the time we call $\mathtt{MinPath}(v')$, we will have $\mathtt{val}[w]$ properly updated for each ancestor w of v'. After processing each of the boughs, we contract all of them and recurse on the resulting tree. Note that unlike the case of $w \perp v$, here we perform only one $\mathtt{MinPath}(v')$ operation for each vertex v' in the bough, rather than performing one for each of its neighbors.

8 A Simple Class of Graphs

In this section, we describe a class of graphs that are easier to solve. On these graphs we can find the minimum cut using only our simple algorithm for cuts that 1-respect a tree. Since we can recognize these graphs quickly, in practice we might often be able to avoid running the more complicated algorithm for the 2-respecting case.

The Nash-Williams lower bound of $c/2$ trees in a graph with minimum cut c is often pessimistic. The number of trees in the maximum packing may be as large as c (the exact number is determined by the value of the "sparsest cut" of the graph as discussed in Section 2 (cf. [NW61]). Call G a *δ-fat* graph if its maximum packing contains more than $(1+\delta)c/2$ trees.

Theorem 8.1. *For any constant $\delta > 0$, the minimum cut in a δ-fat graph can be found with constant probability in $O(m + n\log^3 n)$ time and with high probability in $O(m\log n + n\log^3 n)$ time.*

Proof. Consider a maximum packing in a δ-fat graph. It has $(1+\delta)c/2$ trees (by weight) sharing c minimum cut edges. Thus the average number of minimum cut edges per tree is at most $2/(1+\delta) < 2$. Thus, by Markov's inequality, there is a constant probability that the number of min-cut edges in a randomly chosen tree is less than 2, meaning that it must be exactly 1.

As before, we cannot actually take the time to find a maximum packing. However, we can use the same skeleton construction as Theorem 4.1. Consider the skeleton H with minimum cut $c' = O(\log n)$. We have shown [Kar98] that if a graph is δ-fat then, with high probability, any skeleton we construct from it (with $c' = O(\log n)$) is $(\delta - \epsilon)$-fat, where ϵ is a constant that can be made arbitrarily small by proper choice of c'. Since the skeleton is $(\delta-\epsilon)$-fat, the maximum packing has $(1+\delta-\epsilon)c'/2$ trees. We can therefore find a packing of $(1+\delta/2-\epsilon)c'/2$ trees in $O(n\log^3 n)$ time using the Plotkin-Shmoys-Tardos algorithm (note that Gabow's algorithm cannot be applied here as it always gives a packing of weight exactly $c'/2$ regardless of the graph's fatness). These trees share the at most $(1+\epsilon)c'$ edges of the cut in H corresponding to the minimum cut of G. Thus, so long as $\epsilon \ll \delta$ ($\epsilon = \delta/5$ suffices), the average number of cut edges per tree is a constant less than 2. Thus, as in Lemma 2.3, a constant fraction of the trees (by weight) will 1-respect the minimum cut of G.

Given such a tree, we can use the algorithm of Section 5 to identify the minimum cut in linear time. The constant-probability argument follows by selecting a random tree and applying our linear-time algorithm to find the minimum cut that 1-respects it. The high probability bound follows from selecting $O(\log n)$ random trees and running our algorithm on each. On sparse graphs, the bottleneck is now the packing algorithm of [PST91]. □

The above algorithm can be applied even if we do not know initially that our graph is δ-fat, for the heuristic is actually "self certifying." Once we have used the tree to find the minimum cut in the graph, we can compare the size of our tree-packing to the value c'' of the corresponding cut in the skeleton (which we know is near-minimum with high probability). If the packing has $(1+\delta)c''/2$ trees, then we know the skeleton is δ-fat. We have shown [Kar98] that with high probability the skeleton is δ-fat if and only if the original graph is. Unfortunately, the certification is correct with high probability but not with certainty (there is a small chance the skeleton is δ-fat but the graph is not). Thus, the checking heuristic does not guarantee that we have found the minimum cut. Rather, we have the following:

Lemma 8.2. *If a graph is δ-fat, then with high probability the δ-fat heuristic certifies itself and returns the minimum cut. If a graph is not δ-fat, then with high probability the δ-fat heuristic does not certify itself.*

Thus, so long as we are satisfied with a Monte Carlo algorithm, we can halt if the algorithm certifies itself.

Although fat graphs seem to form a natural class, it should be noted that they form a small class: a random graph (in the standard model of Bollobas [Bol85]) is highly unlikely to be fat. Its minimum cut is within $1+o(1)$ of the average degree k, and counting edges shows that a graph with average degree k can have at most $k/2$ spanning trees.

9 Small Refinements

In this section, we give two very small refinements of our algorithm that improve its running time to the unhappy bound of $O(m(\log^2 n)\log(n^2/m)/\log\log n)$. Such a messy bound suggests to us that further improvements should be possible.

9.1 Fewer trees

Our first improvement reduces the number of trees that we must analyze for 2-constrained cuts.

Lemma 9.1. *A minimum cut can be found with high probability in $O(m(\log^3 n)/\log\log n + n\log^6 n)$ time.*

Proof. We use the fact that trees can be analyzed more quickly for 1-respecting cuts than for 2-respecting cuts. In our initial sampling step (discussed in Section 4.2), set $\epsilon = 1/4\log n$. Using Gabow's algorithm to pack trees in this sample takes $O(n\log^7 n)$ time. A sampling-based speedup [Kara] improves the running time to $O(n\log^6 n)$.

In a second phase, choose $4\log^2 n$ trees at random from the packing and find the minimum cut that 1-respects each in $O(m\log^2 n)$ time. In a third phase, choose $(\log n)/\log\log n$ trees at random from the packing and find the minimum cut that 2-respects each of them in $O(m(\log^3 n)/\log\log n)$ time. Our speedup arises from checking $(\log n)/\log\log n$ trees for the "2-respects" case instead of $\log n$. We claim that this algorithm finds the minimum cut with probability $1-O(1/n)$.

To see this, consider the sampled graph. It has minimum cut c'. The minimum cut of G corresponds to a cut of value at most $(1+\epsilon)c'$ in the sample. The tree packing has value $\rho \geq c'/2$, of which, say, $\alpha\rho$ trees 1-respect the minimum cut and $\beta\rho$ trees 2-respect the minimum cut. Since any tree not counted by α or β must have at least 3 cut edges, we have

$$\begin{aligned}
\alpha\rho + 2\beta\rho + 3(1-\alpha-\beta)\rho &< (1+\epsilon)c' \leq 2(1+\epsilon)\rho \\
\alpha + 2\beta + 3(1-\alpha-\beta) &< 2(1+\epsilon) \\
3 - 2\alpha - \beta &< 2 + 2\epsilon \\
\beta &\geq 1 - 2\epsilon - 2\alpha
\end{aligned}$$

Suppose first that $\alpha > 1/4\log n$. Each time we choose a random tree, the probability is α that we will choose one that 1-respects the minimum cut. Thus if we choose $4\log^2 n$ random trees, the probability that none of them 1-respects the cut is at most

$$(1 - 1/4\log n)^{4\log^2 n} < 1/n.$$

Thus, with high probability, in the second phase we choose and analyze a tree that 1-respects the minimum cut, and therefore find the minimum cut.

Now suppose that $\alpha \le 1/4\log n$. It follows from the previous inequalities and the fact that $\epsilon = 1/4\log n$ that

$$\beta \ \ge \ 1 - 1/\log n$$

It follows that when we choose $(\log n)/\log\log n$ trees and test them for 2-respecting cuts in the third phase, our probability of failing to choose a tree that 2-respects the minimum cut is only

$$\begin{aligned}(1-\beta)^{(\log n)/\log\log n} &\le \left(\frac{1}{\log n}\right)^{(\log n)/\log\log n}\\ &= 1/n\end{aligned}$$

Thus, regardless which case holds, we find the minimum cut with probability at least $1 - 1/n$.

□

9.2 Solving one tree faster

Our second improvement reduces the time it takes to analyze a single tree.

Lemma 9.2. *The smallest cut that 2-respects a given tree can be found in $O(m\log n\log(n^2/m))$ time.*

Proof. We previously bounded the time to analyze a bough with d incident edges by $O(d\log n)$. We show that in an amortized sense, we can also bound this time by $O(n\log n)$. We thus break the time to process a tree with up to n leaves into two parts. The time to reduce the number of leaves (and boughs) to l is (as was shown in Section 7) $O(m(\log n)\log n/l)$. The time to process the l-leaf tree is (by the amortized bound) $O(nl\log n)$. Overall, the time to process the tree is

$$O((m\log n/l + nl)\log n).$$

Choosing $l = m/n$ yields the claimed time bound.

To prove the amortized time bound, suppose we process a bough with d incident edges. Afterward, we contract that bough to a single node. Assuming we merge parallel edges during the contraction, a single node can only have n incident edges. Thus, $d - n$ of the edges incident on the bough vanish. We charge the dynamic tree operation for each edge that vanishes to the vanishing edge, and charge the at-most n dynamic tree operation for non-vanishing edges to the algorithm. Over the entire course of the algorithm, each edge gets charged once (for a negligible total of $O(m)$ operations) while the algorithm gets charged $O(nl)$ times. □

10 Conclusion

This paper has presented two algorithms for the minimum cut problem. The first is quite simple and runs in $O(n^2\log n)$ time (and on fat graphs in $O(m\log n)$ time). The second algorithm is relatively complicated, but holds out the possibility that the minimum cut problem can be solved in linear time. Several probably unnecessary logarithmic factors remain in the running time, suggesting the following improvements:

- Give a dynamic path-minimization data structure taking constant amortized time per operation. This would reduce the running time by a $\log n$ factor. We are not using the full power of dynamic trees (in particular, the tree on which we operate is static, and the sequence of operations is known in advance), so this might be possible.

- Extend the approach for boughs directly to trees in order to avoid the $O(\log n)$ different phases needed for pruning boughs. This too would reduce the running time by a $\log n$ factor.

- Give a deterministic linear-time algorithm for finding a tree that 2-constrains the minimum cut. This would eliminate the $\log n$ factor required by our randomized approach. Note that randomization is used only to find the right tree; all remaining computation is deterministic. Thus any $o(mn)$-time algorithm for finding a good tree would yield the fastest known deterministic algorithm for minimum cuts.

Our algorithm is Monte Carlo. It would be nice to develop a deterministic, or at least a Las Vegas, algorithm based upon our ideas. One standard approach to getting a Las Vegas algorithm would be to develop a minimum cut "certifier" that would check the correctness of our Monte Carlo algorithm.

Another question is whether the near-linear time algorithm can be parallelized. This would require finding a substitute for or parallelization of the sequence of dynamic updates the algorithm performs.

Our tree-packing approach has also led to some progress on the enumeration of near-minimum cuts. The gap between upper and lower bounds is now extremely small and should be eliminated entirely. Except for the 4-clique and one graph on 7 vertices [NNI94], the cycle appears to have the most small cuts, so it is the upper bound which seems likely to be improved.

11 Acknowledgments

Thanks to Robert Tarjan for some helpful references and comments on dynamic merging. Thanks to Eric Lehman and Matt Levine for some careful reading and suggestions for presentation improvements.

A Optimality of the Cut Counting Proof

In Theorem 3.3, we gave an upper bound of

$$q_k = \frac{\sum_{r \le k}(k+1-r)\binom{n}{r}}{k+1-2\alpha}$$

on the number of α-minimum cuts a graph may have. The upper bound held for every (and thus the smallest) value of $k \ge \lfloor 2\alpha \rfloor$ (to ensure $q_k > 0$). The proof of this bound relied on a weighted bipartite graph on cuts and packed trees. It is not obvious that this proof has been "optimized." Perhaps a different assignment of edge weights would yield a better bound. In this section, we show that no better bound is possible. Our analysis was derived using linear-programming duality, but is presented in a simplified way that does not use duality.

Our upper-bound proof assigns a weight $w_r \ge 0$ to any edge connecting a cut C to a packed tree containing exactly r edges of C. The maximum weighted tree degree is therefore $\sum w_r \binom{n}{r}$. We then consider the smallest possible weighted degree that a cut C could have given these weights. If n_r trees contain exactly r edges of this minimum-degree cut, then its weighted degree is $\sum w_r n_r$. We deduce that the number of cuts is upper bounded by

$$R(\{w_r, n_r\}) = \frac{\frac{c}{2}\sum w_r \binom{n}{r}}{\sum w_r n_r}.$$

The argument assumes that n_r reflects the *smallest possible* cut degree, so our choice of weights w_r yields an upper bound of

$$\max_{n_r} R(\{w_r, n_r\})$$

on the number of cuts. Since we are interested in as tight a bound as possible, we would actually like to compute

$$\min_{w_r} \max_{n_r} R(\{w_r, n_r\}).$$

Of course, we need not consider all possible values n_r. In particular, since every tree has r edges of C for *some* r, we know that

$$\sum_r n_r = c/2. \tag{2}$$

Furthermore, since the trees must share the at most αc edges of C, we must have

$$\sum_r r \cdot n_r \leq \alpha c. \tag{3}$$

Finally, we must clearly have

$$n_r \geq 0 \tag{4}$$

for every r. Conversely, for any n_r satisfying these three equations, there clearly exist trees yielding these values n_r.

Using these restrictions on n_r, we chose a particular set of values w_r to show in Theorem 3.3 that

$$\min_{w_r} \max_{n_r} R(\{w_r, n_r\}) \leq \min_{k \geq \lfloor 2\alpha \rfloor} q_k.$$

We now prove that one cannot make a stronger argument. We exhibit a particular assignment of values n_r for which

$$\min_{w_r} R(\{w_r, n_r\}) = \min_{k \geq \lfloor 2\alpha \rfloor} q_k.$$

Thus, no weight assignment can prove a ratio better than $\min q_k$. Since we have given weights that prove a ratio of $\min q_k$, we have an optimal bound.

To show our analysis is optimal, fix k to minimize q_k and consider the following values for n_r:

$$\begin{aligned} n_r &= \frac{c}{2q_k}\binom{n}{r} \qquad (r = 1, \ldots, k) \\ n_{k+1} &= \frac{c}{2}\left(1 - \frac{1}{q_k}\sum_{r \leq k}\binom{n}{r}\right) \\ n_r &= 0 \qquad\qquad (r > k+1) \end{aligned}$$

We will shortly show that these n_r satisfy the three feasibility equations and thus correspond to some set of trees. We will also prove one more fact: for every r,

$$n_r \leq \frac{c}{2q_k}\binom{n}{r}. \tag{5}$$

This clearly holds with equality for $r < k+1$ and is trivial for $r > k+1$; we will show it holds for $r = k+1$ as well. Assuming it for now, observe that for any weights w_r, the ratio we get for our proof is

$$\begin{aligned} R(w_r, n_r) &= \frac{\frac{c}{2}\sum \binom{n}{r} w_r}{\sum n_r w_r} \\ &\geq \frac{\frac{c}{2}\sum_{r \leq k}\binom{n}{r}}{\sum_{r \leq k}\frac{c}{2q_k}\binom{n}{r}} \\ &= q_k. \end{aligned}$$

In other words, no weight assignment can yield a better bound than q_k in the proof of Theorem 3.3.

We now need to justify the claims made earlier in Equations 2, 3, 4, and 5. Equation 2 is obvious from the definition of n_{k+1}. Next we prove Equation 3, showing that $\sum r \cdot n_r \leq \alpha c$:

$$\begin{aligned}
\sum r \cdot n_r &= \sum_{r \leq k} r \frac{c}{2q_k} \binom{n}{r} + (k+1)\frac{c}{2}\left(1 - \frac{1}{q_k}\sum_{r\leq k}\binom{n}{r}\right) \\
&= \frac{c}{2}\left(\sum_{r\leq k}\frac{r}{q_k}\binom{n}{r} + k + 1 - (k+1)\sum_{r\leq k}\frac{1}{q_k}\binom{n}{r}\right) \\
&= \frac{c}{2}\left(k + 1 - \sum \frac{k+1-r}{q_k}\binom{n}{r}\right) \\
&= \frac{c}{2}(k+1-(k+1-2\alpha)) \\
&= \alpha c
\end{aligned}$$

as claimed.

It remains only to prove Equations 4 and 5, which are non-obvious only for $r = k+1$. Plugging in the definition of n_{k+1}, we need to prove that

$$0 \leq \frac{c}{2}\left(1 - \frac{1}{q_k}\sum_{r\leq k}\binom{n}{r}\right) \leq \frac{c}{2q_k}\binom{n}{k+1}$$

or equivalently (multiplying by $2q_k/c$) that

$$\sum_{r\leq k}\binom{n}{k} \leq q_k \leq \sum_{r\leq k+1}\binom{n}{r}.$$

We begin with the upper bound. Since $q_k \leq q_{k+1}$ (by choice of k to minimize q_k), we deduce that

$$\begin{aligned}
q_k &= (k+2-2\alpha)q_k - (k+1-2\alpha)q_k \\
&\leq (k+2-2\alpha)q_{k+1} - (k+1-2\alpha)q_k \\
&= \sum_{r\leq k+1}(k+2-r)\binom{n}{r} - \sum_{r\leq k}(k+1-r)\binom{n}{r} \\
&= \sum_{r\leq k+1}\binom{n}{r}
\end{aligned}$$

as desired. Now consider the lower bound. If $k > \lfloor 2\alpha \rfloor$, then we must have $q_k \geq q_{k-1}$, so we can work much as before:

$$\begin{aligned}
q_k &= (k+1-2\alpha)q_k - (k-2\alpha)q_k \\
&\geq (k+1-2\alpha)q_{k+1} - (k+2\alpha)q_k \\
&= \sum_{r\leq k}(k+1-r)\binom{n}{r} - \sum_{r\leq k-1}(k-r)\binom{n}{r} \\
&= \sum_{r\leq k}\binom{n}{r}
\end{aligned}$$

If, on the other hand $k = \lfloor 2\alpha \rfloor$, then just recall that

$$q_k = \sum_{r\leq k}\frac{k+1-r}{k+1-2\alpha}\binom{n}{r}.$$

But note that for all $r \leq k$, the numerator $k+1-r \geq 1$ while the denominator $k+1-2\alpha \leq 1$. So the claim holds.

This completes the analysis of our proof, and shows that no other weight function assignment will improve on the one we have already given. Of course, this does not prove that there is no better bound on the number of cuts: a completely different proof could show one.

References

[ACM93] ACM-SIAM. *Proceedings of the 4^{th} Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1993.

[App92] David Applegate. AT&T Bell Labs, 1992. Personal Communication.

[Bar95] Francisco Barahona. Packing spanning trees. *Mathematics of Operation Research*, 20(1):104–115, February 1995.

[Ben95] András A. Benczúr. A representation of cuts within 6/5 times the edge connectivity with applications. In *Proceedings of the 36^{th} Annual Symposium on the Foundations of Computer Science*, pages 92–102. IEEE, IEEE Computer Society Press, October 1995.

[Bol85] Béla Bollobás. *Random Graphs.* Harcourt Brace Janovich, 1985.

[Bot93] Rodrigo A. Botafogo. Cluster analysis for hypertext systems. In *Proceedings of the 16^{th} Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 116–125, June 1993.

[BV93] Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, April 1993.

[DFJ54] G. B. Dantzig, D. R. Fulkerson, and S. M. Johnson. Solution of a large-scale traveling salesman problem. *Operations Research*, 2:393–410, 1954.

[DKL76] Efim A. Dinitz, A. V. Karzanov, and Micael V. Lomonosov. On the structure of a family of minimum weighted cuts in a graph. In A. A. Fridman, editor, *Studies in Discrete Optimization*, pages 290–306. Nauka Publishers, 1976.

[Edm72] Jack Edmonds. Edge-disjoint branchings. In R. Rustin, editor, *Combinatorial Algorithms*, pages 91–96. Algorithmics Press, 1972.

[FF62] Lester R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks.* Princeton University Press, Princeton, New Jersey, 1962.

[Gab95] Harold N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. *Journal of Computer and System Sciences*, 50(2):259–273, April 1995. A preliminary version appeared in Proceedings of the 23^{rd} ACM Symposium on Theory of Computing.

[GH61] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *Journal of the Society of Industrial and Applied Mathematics*, 9(4):551–570, December 1961.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, San Francisco, 1979.

[GT85] Harold N. Gabow and Robert E. Tarjan. A linear time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30:209–221, 1985.

[GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM*, 35:921–940, 1988.

[GW92] Harold N. Gabow and Herbert H. Westermann. Forests, frames, and games: Algorithms for matroid sums and applications. *Algorithmica*, 7(5):465–497, 1992.

[HdLT98] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge and biconnectivity. In *Proceedings of the* 29^{th} *ACM Symposium on Theory of Computing*, pages 79–89, New York, May 23–26 1998. ACM Press.

[HK95] Monika Rauch Henzinger and Valerie King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Proceedings of the* 27^{th} *ACM Symposium on Theory of Computing*, pages 519–527. ACM, ACM Press, May 1995.

[HO94] J. Hao and James B. Orlin. A faster algorithm for finding the minimum cut in a directed graph. *Journal of Algorithms*, 17(3):424–446, 1994. A preliminary version appeared in Proceedings of the 3^{rd} Annual ACM-SIAM Symposium on Discrete Algorithms.

[HW96] Monika Henzinger and David P. Williamson. On the number of small cuts in a graph. *Information Processing Letters*, 59:41–44, 1996.

[JM92] Donald B. Johnson and Panagiotis Metaxas. A parallel algorithm for computing minimum spanning trees. In *Proceedings of the* 4^{th} *Annual ACM-SIAM Symposium on Parallel Algorithms and Architectures*, pages 363–372. ACM, June 1992.

[Kara] David R. Karger. Random sampling in cut, flow, and network design problems. To appear. A preliminary version appeared in Proceedings of the 26^{th} ACM Symposium on Theory of Computing.

[Karb] David R. Karger. A randomized fully polynomial approximation scheme for the all terminal network reliability problem. To appear. A preliminary version appeared in Proceedings of the 27^{th} ACM Symposium on Theory of Computing.

[Kar93] David R. Karger. Global min-cuts in $\mathcal{RNC}$ and other ramifications of a simple mincut algorithm. In *Proceedings of the* 4^{th} *Annual ACM-SIAM Symposium on Discrete Algorithms* [ACM93], pages 21–30.

[Kar94a] David R. Karger. Random sampling in cut, flow, and network design problems. In *Proceedings of the* 26^{th} *ACM Symposium on Theory of Computing*, pages 648–657. ACM, ACM Press, May 1994. Mathematics of Operations Research, To appear.

[Kar94b] David R. Karger. *Random Sampling in Graph Optimization Problems.* PhD thesis, Stanford University, Stanford, CA 94305, 1994. Contact at `karger@lcs.mit.edu`. Available from `http://theory.lcs.mit.edu/~karger`.

[Kar98] David R. Karger. Random sampling and greedy sparsification in matroid optimization problems. *Mathematical Programmming B 82(1)*, 82(1–2):41–81, June 1998. A preliminary version appeared in Proceedings of the 34^{th} Annual Symposium on the Foundations of Computer Science.

[KR90] Richard M. Karp and Vijaya Ramachandran. Parallel algorithms for shared memory machines. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 869–932. MIT Press, Cambridge, MA, 1990.

[KS96] David R. Karger and Clifford Stein. A new approach to the minimum cut problem. *Journal of the ACM*, 43(4):601–640, July 1996. Preliminary portions appeared in SODA 1992 and STOC 1993.

[LLKS85] Eugene L. Lawler, J. K. Lenstra, A. H. G. Rinooy Kan, and David B. Shmoys, editors. *The Traveling Salesman Problem.* John Wiley & Sons, 1985.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms.* Cambridge University Press, New York, NY, 1995.

[NI92a] Hiroshi Nagamochi and Toshihide Ibaraki. Computing edge connectivity in multigraphs and capacitated graphs. *SIAM Journal on Discrete Mathematics*, 5(1):54–66, February 1992.

[NI92b] Hiroshi Nagamochi and Toshihide Ibaraki. Linear time algorithms for finding k-edge connected and k-node connected spanning subgraphs. *Algorithmica*, 7:583–596, 1992.

[NNI94] Hiroshi Nagamochi, Kazuhiro Nishimura, and Toshihide Ibaraki. Computing all small cuts in an undirected network. Technical Report 94007, Kyoto University, Kyoto, Japan, 1994.

[NW61] C. St. J. A. Nash-Williams. Edge disjoint spanning trees of finite graphs. *Journal of the London Mathematical Society*, 36:445–450, 1961.

[PQ82] J. C. Picard and M. Queyranne. Selected applications of minimum cuts in networks. *I.N.F.O.R: Canadian Journal of Operations Research and Information Processing*, 20:394–422, November 1982.

[PR90] M. Padberg and G. Rinaldi. An efficient algorithm for the minimum capacity cut problem. *Mathematical Programming*, 47:19–39, 1990.

[PST91] Serge Plotkin, David Shmoys, and Éva Tardos. Fast approximation algorithms for fractional packing and covering problems. In *Proceedings of the* 32^{nd} *Annual Symposium on the Foundations of Computer Science*, pages 495–504. IEEE, IEEE Computer Society Press, October 1991.

[ST83] Daniel D. Sleator and Robert E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, June 1983.

[SV88] Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing*, 17:1253–1262, December 1988.

[VY92] Vijay V. Vazirani and Mihalis Yannakakis. Suboptimal cuts: Their enumeration, weight, and number. In *Automata, Languages and Programming.* 19^{th} *International Colloquium Proceedings*, volume 623 of *Lecture Notes in Computer Science*, pages 366–377. Springer-Verlag, July 1992.