

A Near-Linear Time Algorithm for Constructing a Cactus Representation of Minimum Cuts

David R. Karger*

Debmalya Panigrahi*

Abstract

We present an $\tilde{O}(m)$ (near-linear) time Monte Carlo algorithm for constructing the *cactus* data structure, a useful representation of all the global minimum edge cuts of an undirected graph. Our algorithm represents a fundamental improvement over the best previous (quadratic time) algorithms: because there can be quadratically many min-cuts, our algorithm must avoid looking at all min-cuts during the construction, but nonetheless builds a data structure representing them all. Our result closes the gap between the (near-linear) time required to find a single min-cut and that for (implicitly) finding all the min-cuts.

1 Introduction

In this paper, we give an $\tilde{O}(m)$ -time Monte Carlo algorithm for constructing the *cactus representation* of all (global) minimum cuts¹ in an undirected graph with n vertices and m edges of arbitrary capacity. This improves on the previous best $\tilde{O}(n^2)$ time algorithm of Karger and Stein [5].

The cactus is an elegant data structure introduced by Dinitz et. al. [1] that represents all (possibly $\Theta(n^2)$) minimum cuts of an undirected graph using an $O(n)$ -edge undirected graph. The representing graph is a tree of cycles—a collection of cycles connected to each other by non-cycle edges that form a tree. Each vertex in the original graph is mapped to a node of the cactus² (though the mapping can be non-injective and non-surjective). In the cactus, removing any tree edge, or any pair of edges from the same cycle, divides the cactus nodes in two. Each such partition induces a corresponding partition of the original graph vertices; these are precisely the minimum cuts of the original graph. Thanks to this correspondence, a cactus makes it easy to enumerate all minimum cuts, to find a minimum cut separating any two vertices if one exists, and to compute other useful characteristics of the min-cuts of the original graph.

*Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139. email: {karger, debmalya}@mit.edu. Research supported by NSF contract CCF-0635286.

¹Throughout this paper, “min-cut” will refer to a global minimum edge cut.

²To avoid ambiguity, we will use the term *vertex* for the original graph and *node* for the cactus.

Karzanov and Timofeev gave a sequential algorithm for constructing the cactus [6] that was parallelized by Naor and Vazirani [8]. This algorithm used an explicit listing of all the minimum cuts of a graph. An undirected graph potentially has $\Theta(n^2)$ min-cuts, each of which can be described explicitly in $\Theta(n)$ space; thus, the explicit listing uses $\Theta(n^3)$ space. Any cactus construction using such a listing will of course require $\Theta(n^3)$ time just to read its input. This was unimportant when min-cut algorithms were slow, as the time to find the min-cuts dominated the time to construct the cactus. However, as faster min-cut algorithms were developed [3, 7, 5, 4], faster cactus construction algorithms became imaginable. Karger and Stein [5] gave an $\tilde{O}(n^2)$ -time cactus construction based on the *chain representation* of minimum cuts. Using a fixed root r , the chain representation represents all min-cuts separating r from a given vertex x in $O(n)$ space. This is possible because those cuts form a nested sequence of $O(n)$ sets $S_1 \subseteq S_2 \subseteq \dots$, so can be represented via the differences $S_{i+1} - S_i$ which in total have size n^2 . The Karger-Stein algorithm is thus linear in the (maximum) size of the representation it uses.

Subsequently, Karger [4] gave a near-linear time algorithm for finding a minimum cut in a graph; however, no algorithm with similar time bounds was given for creating the cactus; the quadratic-space intermediate representation used by the previous algorithms suggested this would be difficult.

To achieve near-linear time, we need to find a better way to examine all the minimum cuts of the graph that need to be incorporated in the cactus. Indeed, it is not only the size of these cuts’ representation that is at issue: since there can be as many as $\Theta(n^2)$ min-cuts even in a graph with n edges (consider the cycle), a near-linear time cactus algorithm does not even have time to *encounter* all the minimum cuts, let alone build a large representation of them. This is the challenge we surmount in the paper: to go directly from the size- m graph to the size- n cactus, without ever “unpacking” the set of minimum cuts that we need to examine for the construction.

Our algorithm is Monte Carlo, as are all algorithms

for constructing the cactus—or even finding a minimum cut—whose time bounds beat $\tilde{O}(mn)$. It works with high probability, but offers no way to check whether it has produced the correct answer.

1.1 Our Approach. Although there may be $\Theta(n^2)$ min-cuts, we show that the cactus can be constructed by finding only $O(m+n)$ *minimal* min-cuts. (The concept of minimal min-cuts was introduced by Gabow in [2].) For intuition, suppose the cactus is a tree, i.e. that the cactus has no cycles, and that the nodes of the cactus are in 1-1 correspondence with the vertices of the graph. To build the cactus, we need only identify the tree edges in the cactus. To do so, pick some vertex r of the graph, and imagine we root the cactus at this vertex. Then the minimum cuts correspond to subtrees of this tree, and the subtree rooted at v is precisely the *minimal* (in number of vertices) min-cut separating v from the root r . Enumerating the contents of all such subtrees could generate $\Theta(n^2)$ work. So instead, we aim to identify the *parent* of each cactus node. If w is the parent of v , then the subtree rooted at w , besides being a minimal min-cut (for w) is the *second smallest* min-cut separating v from r . Thus, to construct the cactus, we label each vertex w with its minimal min-cut, and we find the parent of v by finding the vertex labeled by the second smallest min-cut for v among these minimal min-cuts.

In general, multiple graph vertices can map to the same cactus node. All these vertices have the same minimal min-cut and second smallest min-cut. These vertices are therefore indistinguishable and are treated as a single vertex in the cactus construction.

Things get even more complicated in the presence of cycles. We need to identify cycle edges of the cactus. We can still speak of rooting the cactus, and of nodes “below” others in the rooted cactus. Nodes on cycles, however, can have two “parents”, meaning there is no one second-smallest cut identifying a unique parent for such nodes. Instead, we show that if (v, w) is a cycle edge, then there is an edge e with one endpoint below v , and another below w . In this case, the smallest min-cut separating *both* endpoints of edge e from the root r is the one that detaches edge (v, w) from the rest of its cactus cycle. In other words, there is a minimal min-cut for some edge e that “certifies” that v and w are neighbors on a cactus cycle. Thus, enumerating the minimal min-cuts for *edges* in the graph lets us identify cycle edge of the cactus.

We similarly handle the last complication, that some node u of the cactus may be “empty”, with no graph vertex mapping to it. In this case, we show there is an edge e of the graph whose two endpoints have

this empty node as a least common ancestor—thus, the minimal min-cut separating (both endpoints of) e from r corresponds to the subtree rooted at the empty u . We go a step further— we show that if u is the parent of v in the cactus, then there is an edge f with exactly one endpoint in the subtree rooted at v such that the minimal min-cut separating (both endpoints of) f from r corresponds to the subtree rooted at u .

To find the minimal min-cuts, our algorithm builds on Karger’s $\tilde{O}(m)$ -time minimum cut algorithm [4]. Given a graph G , Karger’s algorithm uses random sampling to construct a set \mathcal{T} of $O(\log n)$ trees with the property that any minimum cut of G *2-respects* some tree of \mathcal{T} . That is, there will be some tree $T \in \mathcal{T}$ such that it has at most two edges crossing the given minimum cut. Removing this edge or pair of edges will divide the tree into two or three pieces that correspond to the vertex partition of the minimum cut (if two pieces, the partition is obvious; if three pieces, then the two non-adjacent pieces form one side of the minimum cut). The minimum cut algorithm finds a cut by inspecting all pairs of potentially removable tree edges—not explicitly, as that would take $\Omega(n^2)$ time, but by finding a “best match” second edge for each of the n edges of T . While this algorithm will find *some* min-cut, it will not find all.

We augment Karger’s algorithm to find all the minimal min-cuts. We know that each min-cut corresponds to a singleton or pair of edges from a tree $T \in \mathcal{T}$, so we seek the edges and pairs corresponding to minimal min-cuts. We piggyback on the part of Karger’s algorithm that hunts for minimum cuts, checking the values of cuts corresponding to certain singletons or pairs of edges. However, where Karger’s algorithm can stop once it finds *some* min-cut, we work more exhaustively to enumerate *all* minimal min-cuts. This puts us at risk of spending $\Omega(n^2)$ time encountering too-many *non-minimal* min-cuts; we must therefore prove and exploit structural theorems regarding these minimal min-cuts that let us terminate the exploration early so as to guarantee spending only $\tilde{O}(m)$ time.

Once we have this (implicitly represented) list of all minimal min-cuts, we construct the tree of minimal min-cuts of vertices using the second smallest min-cuts as described earlier. We now use the minimal min-cuts of edges to construct the cactus from this tree in $\tilde{O}(m)$ time.

Roadmap. We review Karger’s min-cut algorithm [4] and describe our modifications to the algorithm in section 2. In section 3, we describe our cactus construction algorithm using the modified version of the min-cut algorithm. We conclude and mention some open problems in section 4.

2 Modified mincut algorithm

In this section, we review Karger's $\tilde{O}(m)$ min-cut algorithm from [4] and then modify it to suit our purpose.

2.1 $\tilde{O}(m)$ time min-cut algorithm [4]. To describe this algorithm, we need to first define some terms that we will use throughout the paper.

DEFINITION 2.1. *A cut is said to k -respect a spanning tree of a graph if the spanning tree contains at most k edges of the cut. A cut is said to strictly k -respect a spanning tree of a graph if the spanning tree contains exactly k edges of the cut.*

Also, throughout the paper, if a property is said to hold *with high probability*, it means that the property does not hold with probability inversely polynomial in n , where the exponent of the polynomial can be boosted to an arbitrarily large constant without losing the property.

The following theorem provides the starting point of the min-cut algorithm.

THEOREM 2.1. (KARGER [4]) *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 each minimum cut 2-respects 1/3 of them with high probability.*

Throughout this discussion, we will consider these trees to be rooted at the same vertex. Since there are only $O(\log n)$ trees to consider, the problem of finding a minimum cut in the graph reduces to finding, given a spanning tree T , a min-cut that 2-respects T provided such a min-cut exists. This problem is further subdivided into finding any min-cut that 1-respects T and finding any min-cut that strictly 2-respects T . The first subproblem can be solved easily by a post-order traversal that leads to the following lemma.

LEMMA 2.1. (KARGER [4]) *The values of all cuts that 1-respect a given spanning tree can be determined in $O(m + n)$ time.*

The more involved case is that of finding a minimum cut that strictly 2-respects the spanning tree. We can however restrict the problem further.

DEFINITION 2.2. *A bough in a tree is a maximal path on the tree with a leaf at one end and having the property that all the other vertices have degree 2 in the tree.*

If we contract all the boughs into their immediate parent in the tree, then the number of leaves in the new tree is at most half of that in the original tree. This follows from the fact that each leaf in the new tree

consumes at least 2 leaves of the original tree. Thus, in $O(\log n)$ iterations, the entire tree will shrink into a single vertex. The problem then becomes one of finding the strictly 2-respecting min-cuts where one of the edges is on a bough in $\tilde{O}(m)$ time.

Now, we further sub-divide the problem.

DEFINITION 2.3. *The set of descendants of a vertex v in a spanning tree T is denoted by v^\downarrow . Similarly, the set of ancestors of v in T is denoted by v^\uparrow .*

If there is no scope of confusion, we will often drop the suffix and denote these sets by v^\downarrow and v^\uparrow respectively. Now observe that since the spanning tree is rooted, any set of edges in the tree can be uniquely represented by the set of lower end-points of the edges. Therefore, each strictly 2-respecting cut can be uniquely represented by two vertices in the tree. We sub-divide the problem based on the relative locations of these vertices.

DEFINITION 2.4. *Consider any two vertices v and w in a spanning tree T . If $v \in w^\downarrow$ or $v \in w^\uparrow$, then we write $v \parallel w$ (v and w are said to be comparable); else, $v \perp w$ (v and w are said to be incomparable).*

Let us first show how to handle the case when $v \perp w$. We need to define some notation.

DEFINITION 2.5. $\mathcal{C}(X, Y)$ *is the sum of weights of edges with one end-point in vertex set X and the other in vertex set Y . (An edge with both endpoints in $X \cap Y$ is counted twice.) Overloading our notation, $\mathcal{C}(S) = \mathcal{C}(S, V - S)$.*

DEFINITION 2.6. *The v -precut at w (refer to Figure 1), 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 $v \perp w$ and ∞ otherwise.

DEFINITION 2.7. *The minimum v -precut, denoted \mathcal{C}_v , is the value $\min\{\mathcal{C}_v(w) \mid \exists(v', w') \in E, v' \in v^\downarrow, w' \in w^\downarrow\}$, and $\operatorname{argmin}\{\mathcal{C}_v(w) \mid \exists(v', w') \in E, v' \in v^\downarrow, w' \in w^\downarrow\}$ is called a minimum precut of v .*

The following lemma appears in [4].

LEMMA 2.2. *If it is determined by incomparable vertices, the minimum cut is $\min_v(\mathcal{C}(v^\downarrow) + \mathcal{C}_v)$.*

Calculating $\mathcal{C}(v^\downarrow)$ for each vertex v follows directly from Lemma 2.1- these are the respective cut values. So, we are left to calculate \mathcal{C}_v for each vertex v ; the minimum cut can then be found in additional $O(n)$ time.

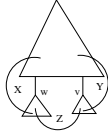


Figure 1: The v -precut at w , $\mathcal{C}_v(w) = X - Z = (X+Y) - (Y+Z) = \mathcal{C}(v^\downarrow \cup w^\downarrow) - \mathcal{C}(v^\downarrow) = (X+Z) - 2Z = \mathcal{C}(w^\downarrow) - 2\mathcal{C}(v^\downarrow, w^\downarrow)$.

Recall that we have already assumed that v is on a bough of the spanning tree. Let us restrict ourselves initially only to the case when v is a leaf. We maintain a value $val[w]$ at each vertex w and initialize it to $\mathcal{C}(w^\downarrow)$. Now, we require to subtract $2\mathcal{C}(v^\downarrow, w^\downarrow)$ from each $val[w]$. Further, once $val[w]$ has been computed for each w , we need to find a vertex which has the minimum value of $val[w]$. The dynamic tree data structure [9] helps solving both problems. Basically, it provides the following primitives:

- **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.

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

This procedure is now extended from a single leaf to an entire bough.

LEMMA 2.3. (KARGER [4]) *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 the first case, all the minimum precuts of u which are not ancestors of neighbors of v continue to be minimum precuts of v .*

This lemma leads to a simple bottom-up walk on the bough, where in each step, the value of \mathcal{C}_u computed inductively has to be compared with the values of $\mathcal{C}_v(w)$ in Lemma 2.3, which can be computed using an additional $O(d)$ dynamic tree operations, where d is the number of edges incident on vertex v in the graph.

We now describe the case of comparable v and w , i.e. $v \parallel w$. Without loss of generality, let us assume that $v \in w^\downarrow$ and v is on a bough. We need to compute $\mathcal{C}(w^\downarrow - v^\downarrow)$. It is shown in [4] that

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

For a given v , $\mathcal{C}(v^\downarrow, v^\downarrow)$ and $\mathcal{C}(v^\downarrow)$ are fixed and can be computed in $\tilde{O}(m)$ time for all the vertices by a minor extension of Lemma 2.1. Thus, it is sufficient to compute, for each vertex $w \in v^\uparrow$, the quantity $\mathcal{C}(w^\downarrow) + 2\mathcal{C}(v^\downarrow, w^\downarrow)$. $val[w]$ is initialized to $\mathcal{C}(w^\downarrow)$ for

each vertex using Lemma 2.1. Now, using a post-order traversal as earlier and the dynamic tree operations mentioned above, $2\mathcal{C}(v^\downarrow, w^\downarrow)$ is added to $val[w]$ for each $w \in v^\uparrow$. Once $val[w]$ has been computed for each w , we need to find a vertex which has the minimum value of $val[w]$. This can also be done using dynamic tree operations.

In summary, a strictly 2-respecting min-cut can be found in $O(D \log n)$ time for a bough which has a total of D edges incident on the vertices of the bough. After running the above procedure, $val[w]$ is reset to its original value by undoing all the operations (subtracting instead of adding in **AddPath**) in $O(D \log n)$ time. A different bough can now start running its procedure. Since an edge is incident on at most 2 boughs, the algorithm takes $\tilde{O}(m)$ time to process all the boughs. As discussed earlier, all the boughs are now folded up and a new phase begins.

3 Cactus construction algorithm

Unlike the min-cut algorithm presented above, we are not interested in finding only a single min-cut in the graph. To specify our goals, we need some more definitions.

DEFINITION 3.1. *Let r be the vertex that was selected to be the root of the $O(\log n)$ trees in the tree packing. Then, the size of a cut is the number of vertices not on the side of r in the cut.*

Note that the weight of edges in a cut is its *weight*; it is important to keep the distinction between size and weight of a cut in mind.

DEFINITION 3.2. *The minimal min-cut of a vertex v is the min-cut of least size which separates v from r . If v is not separated from r by any min-cut, then its minimal min-cut is undefined. Overloading the definition, the minimal min-cut of an edge (u, v) is the min-cut of least size that separates r from both u and v . As earlier, if no min-cut separates both u and v from r , then its minimal min-cut is undefined.*

In the following discussion, we will often refer to a cut by a subset of vertices—the side of the cut not containing the root vertex r .

DEFINITION 3.3. *Two cuts X and Y are said to be crossing if each of $X \cap Y, X - Y, Y - X$ and $X^C \cap Y^C$ is non-empty.*

LEMMA 3.1. (SEE EG, KARGER [4]) *If X and Y are crossing min-cuts, then $X \cap Y, X - Y, Y - X$ and $X \cup Y$ are min-cuts. Further,*

$$\mathcal{C}(X \cap Y, X^C \cap Y^C) = \mathcal{C}(X - Y, Y - X) = 0.$$

LEMMA 3.2. *The minimal min-cut of a vertex or edge is unique. Further, the minimal min-cut of a vertex v does not cross any other min-cut of the graph.*

Proof. If there are two minimal min-cuts of a vertex v (resp., edge (u, v)), then their intersection is a smaller min-cut containing v (resp., u and v), contradicting minimality. Similarly, if the minimal min-cut X of a vertex v crosses another min-cut Y , then either $X \cap Y$ or $X - Y$ is a smaller min-cut containing v , contradicting minimality.

We are now in a position to describe our plan. We first construct a list of $\tilde{O}(m)$ min-cuts containing the minimal min-cut of each vertex, if one exists. Then, we label each vertex with its corresponding minimal min-cut from the list and also, subsequently, find the minimal min-cuts of a *sufficient* set of edges. Finally, this set of minimal min-cuts for vertices and edges are used to construct a cactus representation of the graph.

3.1 Listing minimal min-cuts of vertices. We modify the min-cut algorithm described above to meet our objective. All our modifications are for the strictly 2-respecting scenario; the part of the algorithm that finds all 1-respecting min-cuts is exactly the same as above. Recall that the original algorithm has $O(\log n)$ phases, where the algorithm is run on progressively smaller trees in each phase. Now, consider a scenario where the minimal min-cut for a vertex u is defined by vertices v and w , where $v \perp w$ and $u \in v^\perp$. Our goal is to ensure that we identify this min-cut in the phase where we process v . However, due to the recursive process, v^\perp might now be a proper subset of the vertices compressed into a single node at this stage. In this case, we will fail to identify the minimal min-cut for u . So, we need to modify the structure of the algorithm slightly.

We also have $O(\log n)$ phases, but we maintain two trees in each phase. One tree is the shrunk tree S , identical to the earlier algorithm. The other tree T is the original spanning tree without any edge contraction.

Each vertex in the contracted tree S represents a set of vertices in the original tree T (refer to Figure 2). For each such set X , let $\ell(X)$ denote the *leader* of the set, which is the vertex of least depth in T . Conversely, each vertex v is the leader of a contracted vertex in S (i.e. set of vertices in T) in some phase; denote this set by $\ell^{-1}(v)$. Now, any such set X can have two possible structures in T :

- If X is a leaf in S , then it represents a subtree rooted at $\ell(X)$ in T .
- If X is a vertex with degree 2 in S , then it represents a subtree rooted at $\ell(X)$ in T , where one

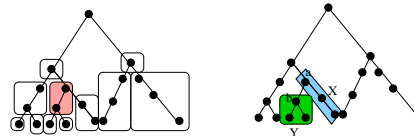


Figure 2: The tree on the left is a spanning tree T where the boughs in the different phases are marked. The shaded bough (on the left), for instance, is processed in phase 2 at which stage its two vertices correspond to sets X and Y in T (due to boughs being folded up). These sets are shown on the right; $a = \ell(X)$ and $b = \ell(Y)$.

of the children subtrees of $\ell(X)$ has been removed. This child subtree is rooted at $\ell(Y)$, where Y is the only child of X in S .

Our goal is to ensure that if the minimal min-cut of vertex u is defined by vertices v and w , where $v \perp w$ and $u \in v^\perp$, then this min-cut is identified when $\ell^{-1}(v)$ is processed.

We need another definition.

DEFINITION 3.4. *Consider any vertex v on a bough and let w be a minimum precut of v . If there exists no descendant x of w such that x is also a minimum precut of v , then w is said to be a minimal minprecut of v . If w is the only such vertex, it is said to be the unique minimal minprecut of v .*

LEMMA 3.3. *Let the minimal min-cut of vertex u 2-respect a tree T , where it is represented by vertices v and w , $v \perp w$ and $u \in v^\perp$. Then, w is the unique minimal minprecut of v .*

Proof. Clearly, w is a minimal minprecut of v , else there is a smaller min-cut separating u from root r . If there are multiple minimal minprecuts of v , then these min-cuts cross, violating Lemma 3.2.

To identify minimal minprecuts, we need to strengthen the **MinPath** primitive provided by the dynamic tree data structure. Recall that **MinPath**(v) for a variable val returns the minimum value of val among ancestors of v and a vertex which achieves this minimum value. If there are multiple ancestors of v achieving this minimum value, then the vertex returned by **MinPath** is ambiguous. However, we would like **MinPath** to return the closest ancestor of v achieving this minimum value. To achieve this, we run **AddPath**(v, ϵ) (for some $\epsilon > 0$) for each vertex v as a pre-processing step. Clearly, $val[v]$ now has a value $\epsilon|v^\perp|$. We choose a small enough ϵ so that this pre-processing step does not tamper with the ability of the algorithm to distinguish min-cuts from other cuts. Now, if we use our usual **MinPath** operations, we will always find the closest ancestor in case of a tie in the original graph.

We need to impose some additional structure on the minimal minprecuts of a vertex.

DEFINITION 3.5. Consider a vertex v and let its minimal minprecuts be w_1, w_2, \dots, w_k , where each $w_i \perp v$. Let ℓ_i be the lca of w_i and v in tree T . Clearly, each ℓ_i lies on the path connecting v to root r ; let ℓ_o be the shallowest vertex among the ℓ_i s. Correspondingly, let w_o be a minimal minprecut of v such that lca of w_o and v is ℓ_o ($w_o = w_i$ for some i). We call w_o an outermost minimal minprecut of v . If w_o is unique, it is called the unique outermost minimal minprecut of v .

The following lemma is an extension of Lemma 2.3.

LEMMA 3.4. Let v be a vertex in T , $X = \ell^{-1}(v)$ and w be the unique outermost minimal minprecut of v . If X is a leaf in S , then there exists at least one edge between X and w^\downarrow . On the other hand, if X is a non-leaf in S , let Y be the only child of X in S and $u = \ell(Y)$. Then, either there exists at least one edge between X and w^\downarrow , or w is the unique outermost minimal minprecut of u .

Proof. If X is a leaf in S , then by definition of a minimum precut, X and w^\downarrow must be connected. Otherwise, let there be no edge between X and w^\downarrow . We need to prove that w is the unique outermost minimal minprecut of u . First, note that by definition of minimum precut, u^\downarrow and w^\downarrow must be connected since $v^\downarrow = X \cup u^\downarrow$. Further, by Lemma 2.3, $C_u(w) = C_v(w) = C_v = C_u$ and if any descendant of w is a minimal minprecut of u , then it is also a minimal minprecut of v , contradicting the minimality of w . Thus, w is a minimal minprecut of u . If z is a minimal minprecut of u such that the lca of z and u is either an ancestor of or the same as the lca of w and u , then $z \perp v$ and is a minimal minprecut of v . Thus, w is the unique outermost minimal minprecut of u .

If w and v represent a minimal min-cut for a vertex $u \in v^\downarrow$, where $w \perp v$, then w is the unique outermost minimal minprecut of v . Thus, our goal is to identify the unique outermost minimal minprecut of a vertex, if it exists. To process a leaf X in a bough in S , we run **AddPath** followed by **MinPath** queries for the other endpoint of each edge with one endpoint in X . For processing a vertex X of degree 2 in S , assume inductively that we have already found the unique outermost minimal minprecut w corresponding to its child Y in S , provided such a vertex w exists. We now run **AddPath** followed by **MinPath** queries for the other endpoint of each edge with one endpoint in X . Also, we check if w is a minprecut of $v = \ell^{-1}(X)$ by inspecting the value of $val[w]$. The set of minimal minprecuts identified contains the unique outermost minimal minprecut of v , if

it exists. We now run lca queries to determine if v has a unique outermost minimal minprecut among the minimal minprecuts identified. The total time consumed by this procedure is $O(d \log n)$, where d edges are incident on vertices in X .

We now describe the algorithm used to identify all strictly 2-respecting minimal min-cuts which are represented by comparable vertices. When we process vertex v , we would like to find all such min-cuts represented by v and w , where $v \in w^\downarrow$. Recall that Karger's min-cut algorithm allows us to compute $val[w]$ for each w such that inspecting $val[w]$ for each vertex w reveals all the min-cuts we want to identify. However, whereas in the min-cut algorithm, only one **MinPath** query needs to be run at vertex v , a single query would only reveal the deepest w which forms such a min-cut with v , but would not reveal additional vertices satisfying the property further up the spanning tree. To overcome this challenge, we maintain a list at each vertex w , denoted by $desc[w]$, which contains its descendants v with which it has been found to form a min-cut that is potentially minimal for some vertex. This list is initially empty for each vertex and populated by the following procedure (which is run, for each vertex being processed on a bough in S , after the **AddPath** calls): *Run a MinPath query at v . Let w be returned by this query. If v, w do not form a min-cut, then stop; else, if $desc[w]$ is non-empty, then add v to $desc[w]$ and stop; otherwise, add v to $desc[w]$ and recurse at w (ie, run a **MinPath** query at w and so on).* The correctness of the procedure is established by the following lemma.

LEMMA 3.5. Let $u \in v^\downarrow$ or $u \perp v$ in T . If both u and v represent min-cuts with some vertex $w \in v^\uparrow \cap u^\uparrow$, then any min-cut represented by v and any z such that $z \in w^\uparrow$ is not a minimal min-cut for any vertex.

Proof. This follows directly from the observation that any min-cut represented by v and z must necessarily cross the min-cut represented by w and u . If such a min-cut is minimal for a vertex, then Lemma 3.2 is violated.

The **AddPath** queries clearly take $\tilde{O}(m)$ time in the above procedure. All the **MinPath** queries associated with the processing of a vertex v , except the last query, result in populating the previously empty $desc$ list of some vertex; thus, there are at most n such queries. The last query can be charged to the vertex being processed; thus there are at most n such queries as well. Overall, $O(n)$ **MinPath** queries are made. Thus, the above procedure has a time complexity of $\tilde{O}(m)$.

3.2 Labeling minimal min-cuts of vertices. We will now label each vertex with the smallest min-cut containing it among those that 2-respect a fixed spanning

tree T . As discussed earlier, the vertices representing a min-cut can be used to classify the min-cuts into 3 categories: 1-respecting min-cuts (category 1) and strictly 2-respecting min-cuts where the vertices are incomparable (category 2) or comparable (category 3). We label each vertex with the smallest min-cut containing it in each category. Finally, for each vertex, we find the minimum among its $O(\log n)$ labels corresponding to the 3 categories of edges in the $O(\log n)$ trees.

Category 1 (1-respecting). Lemma 2.1 states that in $O(n)$ time, we can find the weights of all 1-respecting cuts. Assuming that we know the weight of a min-cut using the algorithm in [4], it immediately follows that we can identify all the 1-respecting min-cuts in $O(n)$ time. To label each vertex with the minimal min-cut containing it, we contract all the edges in T that do not represent min-cuts. Then, the smallest min-cut containing vertex v is the edge connecting the contracted vertex containing v to its parent. Thus, we simply label all vertices in a contracted set by the root of the set. This takes $O(n)$ time.

Category 2 (strictly 2-respecting, incomparable). The smallest min-cut containing a vertex is the smallest among the min-cuts represented by its ancestors in the bough containing it in S . We trace the path along the bough downward maintaining the smallest encountered min-cut C . The label given to a vertex v is the min-cut stored as C when v is encountered. Clearly, this takes $O(1)$ time for each vertex along the walk, and therefore $O(n)$ time overall.

Category 3 (strictly 2-respecting, comparable). We perform a post-order tree traversal using a mergeable minheap to hold all the minimal min-cuts which contain the current vertex u . These min-cuts are exactly the min-cuts whose lower vertex has been encountered but the upper vertex has not been encountered yet. Labeling u with the smallest min-cut in the heap takes $O(1)$ time. Now, all the cuts whose upper vertex is u are removed from the heap and the heap is passed on to the parent of u , say v . All the heaps passed up from its children are now merged at v in amortized $\tilde{O}(1)$ time. This takes $\tilde{O}(n)$ time overall.

3.3 Minimal min-cuts of edges. As described earlier, we also need to find the minimal min-cuts of edges. We need the following definition.

DEFINITION 3.6. A certificate of a min-cut X is an edge e such that X is the minimal min-cut of e , i.e. X is the min-cut of least size separating both endpoints of e from the root vertex r .

The following lemma states a property that ensures the existence of a large weight of certificates for a min-

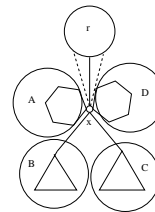


Figure 3: Maximal min-cuts contained in a min-cut represented by an empty node- A, B, C and D are the maximal min-cuts contained in the min-cut represented by the empty node x .

cut.

LEMMA 3.6. Consider a min-cut X . Let $Y \subset X$ be a min-cut satisfying the following properties:

- Y is a maximal min-cut contained in X , i.e. there does not exist a min-cut Z such that $Y \subset Z \subset X$.
- Y does not cross any other min-cut.

Then, the total weight of certificates of X with one endpoint in Y is at least $c/2$, where c is the weight of a min-cut.

Proof. The properties satisfied by Y ensure that any edge between Y and $X - Y$ is a certificate of X . Now, if the total weight of edges between Y and $X - Y$ is less than $c/2$, then $X - Y$ is a cut of weight less than c .

We now show that the above lemma ensures that the min-cuts of interest to us have many certificates. Recall from the introduction that we are interested in min-cuts which are represented either by empty cactus nodes or by consecutive nodes on a cactus cycle.

Let us consider the first category of min-cuts, i.e. those represented by empty cactus nodes. As discussed earlier, we are interested not only in finding such a min-cut X but also in identifying its children in the cactus, i.e. identify all the cycles and subtrees below this empty node in the cactus. Each such cycle/subtree Y is a maximal min-cut contained in X , i.e. there is no min-cut Z with $Y \subset Z \subset X$. For each such maximal min-cut Y , we are interested in finding a certificate of X with exactly one endpoint in Y . The following lemma, combined with Lemma 3.6 shows that the total weight of such certificates is large for each such Y .

LEMMA 3.7. For any min-cut represented by an empty node in the cactus, each maximal min-cut contained in it satisfies the properties of Y in Lemma 3.6.

Proof. The maximal min-cuts contained in a min-cut represented by an empty cactus node are the subtrees and cycles below the node (see Figure 3). Any min-cut

represented by a subtree or a cycle in the cactus does not cross any other min-cut.

Now, we consider the second category of min-cuts, those represented by a pair of contiguous cycle nodes in the cactus. The cactus representation itself shows that the total weight of edges between the min-cuts represented by the cycle nodes is $c/2$.

We can now find the minimal min-cuts of edges using the algorithm for constructing minimal min-cuts of vertices. We construct a set of $\Theta(\log n)$ graphs from the input graph G , where each edge (of weight, say w) in G is contracted with probability $\min(w/2c, 1)$ independently in each new graph.

LEMMA 3.8. *Let G' be any new graph produced by random contraction of edges of G . If X and Y satisfy the condition in Lemma 3.6, then with $\Theta(1)$ probability, there exists a certificate of X with exactly one endpoint in Y which is contracted in G' , and no edge in the cut $(X, V - X)$ is contracted.*

Proof. Let the certificates of X with one endpoint in Y have weight w_1, w_2, \dots, w_k , where $\sum_{i=1}^k w_i \geq c/2$ by Lemma 3.6. If there is an edge of weight $\geq 2c$ in this set, then it is necessarily contracted. Thus, let us assume that $w_i < 2c, \forall i$. Then, the probability that none of the certificates is contracted is given by

$$\prod_{i=1}^k \left(1 - \frac{w_i}{2c}\right) \leq \prod_{i=1}^k \left(1 - \frac{1}{2c}\right)^{w_i} \leq \left(1 - \frac{1}{2c}\right)^{c/2} \leq e^{-1/4}.$$

On the other hand, let the edges in the cut $(X, V - X)$ have weight W_1, W_2, \dots, W_l , where $\sum_{i=1}^l W_i = c$. Then, the probability that none of these edges is contracted is given by

$$\prod_{i=1}^l \left(1 - \frac{W_i}{2c}\right) \geq \left(1 - \frac{\sum_{i=1}^l W_i}{2c}\right) = 1/2.$$

The following corollary follows immediately.

COROLLARY 3.1. *For each min-cut X that is represented by either an empty node or a pair of adjacent cycle nodes in the cactus, and maximal min-cut Y contained in X , at least one certificate of X with exactly one endpoint in Y is contracted and no edge in $(X, V - X)$ is contracted, in at least one of the $\Theta(\log n)$ contracted graphs, with high probability.*

Note that a vertex in a contracted graph represents a set of vertices and edges on them in the original graph. A union-find data structure can keep track of the composition of a vertex in each contracted graph (and to construct the contracted graphs) in $\tilde{O}(n)$ time.

We also keep track of the size of a vertex, i.e. number of original vertices contracted into the vertex. This serves to compute the sizes of the min-cuts in the above algorithm. Thus, we can afford to find the minimal min-cut for each vertex only, for each of the contracted graphs and the original graph. Then, we use the information maintained above to label each (original) vertex and (original) edge with the smallest min-cut containing it in each of the contracted graphs. Note that while each vertex will necessarily be given its correct label, some edges might have wrong labels (which correspond to larger min-cuts containing them) or have no label at all. We show later that the set of edges correctly labeled is sufficient.

3.4 Cactus construction from minimal min-cuts.

We initially form a partition of the vertices into sets containing vertices with the same minimal min-cut. Any pair of vertices in such a set is not separated by any min-cut in the graph; hence each of these sets can be contracted into a single vertex. In the remaining discussion, a vertex will denote such a contracted vertex.

We state the following lemma; the proof of existence follows directly from the properties of the cactus data structure, while the algorithm is omitted due to lack of space.

LEMMA 3.9. *Among the vertex-minimal min-cuts, each vertex has a unique second smallest min-cut containing it. Further, each vertex can be labeled by its second-smallest min-cut in $\tilde{O}(m)$ time overall.*

We now form a tree T by connecting vertex v with vertex u if the minimal min-cut of u is the second smallest min-cut of v . This tree can be constructed in $O(n)$ additional time. The following lemma is a direct consequence of the construction.

LEMMA 3.10. *The minimal min-cut containing any vertex v is the sub-tree of T subtended at v .*

We discard all edges that are not between incomparable vertices in T in $O(m)$ time using lca queries in tree T .

LEMMA 3.11. *The minimal min-cut of an edge that either has both endpoints in a single vertex in T , or whose endpoints are in comparable vertices in T , is also the minimal min-cut of some vertex.*

Proof. Let an edge have both endpoints in the same vertex in tree T . Since its endpoints are not separated by any min-cut in the graph, any min-cut containing either of its endpoints contains both its endpoints. Hence, the minimal min-cut containing both endpoints is also minimal for each endpoint individually. Now, let

an edge have endpoints in comparable vertices (say u and v , where $u \in v^\perp$) in T . Then, the minimal min-cut of v also contains u , which implies that all min-cuts containing v also contain u . Thus, the minimal min-cut of the edge is also the minimal min-cut of v .

We now describe the construction of the cactus from tree T . In the following discussion, a contiguous part of the cactus representing min-cuts contained in a set of vertices S is referred to as a *cactus of S* . The following property follows immediately from the fact that a min-cut is contiguous in the cactus.

LEMMA 3.12. *Consider any vertex x and its children y_1, y_2, \dots, y_k in tree T . The vertices in the subtree of x in T (call this set X) are contiguous in the cactus. Further, the vertices in the subtree of each of y_i (call these sets Y_i) are contiguous in the cactus.*

For each node of this tree, our goal then becomes constructing the cactus representation for the vertices in its subtended subtree, assuming that the cactus representation of its children subtrees have been constructed recursively. Thus, we are interested in constructing the cactus representation of X with each Y_i contracted into a single vertex. This cactus will represent all the min-cuts that are contained in X but not in any Y_i . For this purpose, we need to identify the set of edges whose minimal min-cuts will be represented in the cactus for X . Let (u, v) be an edge. Let $x = lca(u, v)$ in tree T and y and z be children of x containing u and v respectively. Then, we define a function $f : E \rightarrow V \times V$ such that $f(u, v) = (y, z)$. The following lemma states that for the purpose of the construction, we can consider (u, v) to be an edge between y and z .

LEMMA 3.13. *Consider an edge e between vertices u and v , where $u \perp v$ in T . The minimal min-cut of e must be contained in x^\perp , where $x = lca(u, v)$, and must contain y^\perp and z^\perp , where y and z are children of x such that $u \in y^\perp$ and $v \in z^\perp$. Further, x, y, z can be identified for all edges in $\tilde{O}(m)$ time.*

Proof. The first proposition follows from the observation that $u, v \in x^\perp$ and x^\perp , being a minimal min-cut of a vertex, does not cross any min-cut according to Lemma 3.2. The second proposition follows from the non-crossing property of the min-cuts y^\perp and z^\perp . The value of x, y and z for each edge can be found using a post-order traversal maintaining a union-find data structure that keeps track of all edges whose one endpoint has been encountered but the other has not.

Thus, we can now concentrate on the following problem. We are given a set of vertices X comprising

subsets Y_1, Y_2, \dots, Y_k , where $\cup_{i=1}^k Y_i \subset X$ and $Y_i \cap Y_j = \emptyset$ for all $i \neq j$. For convenience, we assume that $Y_{k+1} = X - \cup_{i=1}^k Y_i$. We assume, for the purpose of this construction, that each Y_i is contracted into a single vertex. Further, we are given all edges between any pair of (contracted) vertices Y_i and Y_j and their corresponding labels. Our goal is to construct a cactus on the vertices Y_1, \dots, Y_{k+1} such that all the min-cuts contained in X but not in any of the Y_i s are represented by the cactus.

One further complication arises from the fact that while some edges are labeled with the minimal min-cuts containing them, other edges may have no label or incorrect labels. First, we remove all edges without a label since Corollary 3.1 ensures that each minimal min-cut has a (correctly) labeled certificate. Now, we need to distinguish between min-cuts having the correct label and those that have erroneous labels. The following property helps us make this distinction.

LEMMA 3.14. *If X_1, X_2, \dots, X_k are min-cuts such that $\cup_{i=1}^k X_i$ is also a min-cut, then among all edges with their two endpoints in different X_i s, the label corresponding to the smallest min-cut is a correct label.*

Proof. This follows from Lemma 3.8, coupled with the fact that each edge label is either correct (ie, gives the minimal min-cut of the edge) or gives a strictly larger min-cut containing both endpoints of the edge.

During the construction, we will *mark* any node for which the cactus has not been constructed yet. Initially, all the Y_i s are unmarked. Now, let (Y_i, Y_j) be the edge with the smallest label. We introduce a new node a as a parent of Y_i and Y_j and mark a . We move all edges with exactly one endpoint in $Y_i \cup Y_j$ to a and remove all edges between Y_i and Y_j . We then move on to the next smallest label among the surviving edges. The previous lemma ensures that the label we process at any stage is correct on account of being the minimum surviving label.

We now describe the construction for all the possible cases (refer to Figure 4). In general, at any stage of the construction, suppose the smallest label corresponds to an edge (a, b) . If both a and b are unmarked nodes (case (a)), then we have already constructed the cactus for a and b ; so we can assume that a and b are singleton vertices. In this case, we simply introduce a new node c which is the parent of a and b , and mark the new node. Now, suppose a is a marked node but b is not. Then, we introduce a new node c and make it the parent of b . However, the relationship between c and a is not clear at this stage. There are three possibilities: either c overlaps a , or it is the same cut as a or it is the

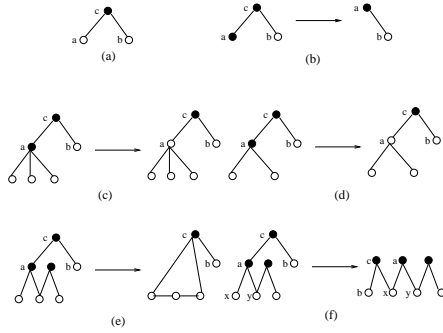


Figure 4: (Marked nodes are dark.) (a) a and b are both unmarked, (b-f) a is marked and b is unmarked; (b) $b \in a$, (c) a has more than 2 children, (d) a has no sibling, (e) a has siblings but both of a 's children are contained in c , and (f) a has siblings and $x \in c$ while $y \notin c$.

parent of a . To distinguish between these possibilities, we run a set of containment queries, each of which can be answered in $O(1)$ time using lca values in the spanning trees. First, we check if $b \in a$; if so, then $c = a$ (case (b)). In that case, we add b as a child of a and remove c ; a remains marked. Suppose the above check fails. Then, a and c are not the same min-cut. Now, if a has more than 2 children, then a has no sibling (case (c)); so, c must be the parent of a . We unmark a and mark c . On the other hand, if a has exactly 2 children x and y , then we check if $x \in c$ and $y \in c$. If both $x, y \in c$, then c is the parent of a (case (d)). In this case, if a has no sibling, we unmark a , mark c and add c as the parent of a . Otherwise, if a has siblings (case (e)), we mark c , remove a and its siblings, connect the children of the siblings of a (including the children of a) in a chain and close the chain using c to form a cycle; also, c is marked. The final possibility is that x is contained in c , but y is not (case (f)). In this case, c is marked and added as a sibling of a containing x and b . If both a and b are marked, then we need to run the above checks for both a and b .

Finally, we are left with no edges. This indicates that we have found all the min-cuts contained in X . At this stage, we can have two situations. If the node representing X in the cactus has more than one cycle/subtree below it, then X is the minimal min-cut for all the edges between these cycles/subtrees. In this case, there will be a single empty node at the highest level of the cactus formed. We replace this empty node with the node representing X . The other possibility is that the node representing X has exactly one cycle/subtree below it in the cactus. If it has exactly one subtree below it, then there is no edge with endpoints that are incomparable in T and have X as

their lca. If it has exactly one cycle below it, then there will be a cycle at the highest level of the cactus, where the last node added to the cycle is an empty node. In this case, we replace this empty node by the node representing X . This completes the construction of the cactus of X .

4 Conclusion and Future Work

We have presented an $\tilde{O}(m)$ algorithm for constructing the cactus representation of a graph. An important open question is whether there exists an $\tilde{O}(m)$ time computable certificate for the same problem. However, the apparently simpler question of whether there exists an $\tilde{O}(m)$ time computable certificate for the min-cut of an undirected graph is also open.

References

- [1] Efim A. Dinitz, Alexander 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, Moscow, 1976.
- [2] Harold N. Gabow. Applications of a poset representation to edge connectivity and graph rigidity. In *Proceedings of the 32nd Annual Symposium on the Foundations of Computer Science*, pages 812–821, 1991.
- [3] Jianxiu 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 3rd Annual ACM-SIAM Symposium on Discrete Algorithms.
- [4] David R. Karger. Minimum cuts in near-linear time. *Journal of the ACM*, 47(1):46–76, January 2000. A preliminary version appeared in Proceedings of the 28th ACM Symposium on Theory of Computing.
- [5] 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.
- [6] Alexander V. Karzanov and E. A. Timofeev. Efficient algorithm for finding all minimal edge cuts of a non-oriented graph. *Cybernetics*, 22:156–162, 1986.
- [7] Hiroshi Nagamochi and Toshihide Ibaraki. Computing edge connectivity in multigraphs and capacitated graphs. *SIAM Journal on Discrete Mathematics*, 5(1):54–66, February 1992.
- [8] Dalit Naor and Vijay V. Vazirani. Representing and enumerating edge connectivity cuts in \mathcal{RNC} . In F. Dehne, J. R. Sack, and N. Santoro, editors, *Proceedings of the 2nd Workshop on Algorithms and Data Structures*, volume 519 of *Lecture Notes in Computer Science*, pages 273–285. Springer-Verlag, August 1991.
- [9] 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.