Building chain and cactus representations of all minimum cuts from Hao-Orlin in the same asymptotic run time *

L. Fleischer†

May 18, 1998

Abstract

A cactus tree is a simple data structure that represents all minimum cuts of a weighted graph in linear space. We describe the first algorithm that can build a cactus tree from the asymptotically fastest deterministic algorithm that finds all minimum cuts in a weighted graph — the Hao-Orlin minimum cut algorithm. This improves the time to construct the cactus in graphs with n vertices and m edges from $O(n^3)$ to $O(nm \log n^2/m)$.

1 Introduction

A minimum cut of a graph is a non-empty, proper subset of the vertices such that the sum of the weights of the edges with only one endpoint in the set is minimized. An undirected graph on n vertices can contain up to $\binom{n}{2}$ minimum cuts [5, 8]. For many applications, it is useful to know many, or all minimum cuts of a graph, for instance, in separation algorithms for cutting plane approaches to solving integer programs [2, 6, 10], and in solving network augmentation and reliability problems [4, 3, 21]. Many other applications of minimum cuts are described in [1, 24].

In 1976, Dinits, Karzanov, and Lomonosov [8] published a description of a very simple data structure called a cactus that represents all minimum cuts of a weighted, undirected graph in linear space. This is notable considering the number of possible minimum cuts in a graph, and the space needed to store one minimum cut.

Any algorithm that is capable of computing all minimum cuts of a graph implicitly produces a data structure that represents all the cuts. The cactus tree is special because it is simple. The size of a

*Columbia University Computational Optimization Research Center Technical Report TR-98-01. A preliminary version of this paper will appear in *Integer Programming and Combinatorial Optimization, 6th International IPCO Conference*, Houston, 1998.

†email: lisa@ieor.columbia.edu. Department of Industrial Engineering and Operations Research, Columbia University. Supported in part by ONR through an NDSEG fellowship, by AASERT through grant N00014-95-1-0985, by an American Association of University Women Educational Foundation Selected Professions Fellowship, by the NSF PYI award of Éva Tardos, and by NSF through grant DMS 9505155.

cactus tree is linear in the number of vertices in the original graph, and any cut can be retrieved in time linearly proportional to the size of the cut. In addition, the cactus displays explicitly all nesting and intersection relations among minimum cuts. This, in addition to the compactness of a cactus, is unique among representations of minimum cuts.

Karzanov and Timofeev [20] outlined the first algorithm to build a cactus for an unweighted graph. Although their outline lacks some important details, it does provide a framework for constructing correct algorithms [25]. In addition, it can be extended to weighted graphs.

The Karzanov-Timofeev outline breaks neatly into two parts: generating a sequence of all minimum cuts, and constructing the cactus from this sequence. This two-phase approach applies to both unweighted and weighted graphs. It is known that the second phase can be performed in $O(n^2)$ time for both weighted and unweighted graphs [25, 20, 22]. The bottleneck for the weighted case is the first phase. Thus the efficiency of an algorithm to build a cactus tree depends on the efficiency of the algorithm used to generate an appropriate sequence of all minimum cuts of an undirected graph.

The earliest algorithm for finding all minimum cuts in a graph uses maximum flows to compute minimum (s,t)-cuts for all pairs of vertices (s,t). Gomory and Hu [16] show how to do this with only n (s,t)-flow computations. The fastest known maximum flow algorithm, designed by Goldberg and Rao [14, 13], runs in $O(\min\{m^{1/2}, n^{2/3}\} m \log(n^2/m) \log U)$ time where U is the maximum capacity of an edge. Another fast, deterministic maximum flow algorithm, designed by Goldberg and Tarjan [15], runs in $O(nm \log(n^2/m))$ time. Hao and Orlin [17] show how a minimum cut can be computed in the same asymptotic time as one run of the Goldberg-Tarjan algorithm. Using ideas of Picard and Queyranne [23], the Hao-Orlin algorithm can be easily modified to produce all minimum cuts in the same asymptotic time. Karger and Stein [19] describe a randomized algorithm that finds all minimum cuts in $O(n^2 \log^3 n)$ time. Recently, Karger has devised a new, randomized algorithm that finds all minimum cuts in $O(n^2 \log n)$ time. While this algorithm is the asymptotically fastest algorithm for finding all minimum cuts, it is not known how to build a cactus efficiently using this algorithm. The algorithm builds a data structure that represents all minimum cuts in $O(k + n \log n)$ space, where k is the number of minimum cuts.

The Karzanov-Timofeev framework requires a sequence of all minimum cuts found by generating all (s,t) minimum cuts for a given s and a *specific* sequence of t's. The Hao-Orlin algorithm also generates minimum cuts by finding all (s,t) minimum cuts for a given s and a sequence of t's. However, the order of the t's cannot be predetermined in the Hao-Orlin algorithm, and it may not be an appropriate order for the Karzanov-Timofeev framework.

All minimum cuts, produced by any algorithm, can be sequenced appropriately for constructing a cactus tree in $O(n^3)$ time. This is not hard to do considering there are at most $\binom{n}{2}$ minimum cuts, and each can be stored in $O(n)$ space.

The main result of this paper is an algorithm that constructs an appropriate sequence of minimum cuts within the same time as the asymptotically fastest, deterministic algorithm that finds all minimum cuts in weighted graphs, improving the deterministic time to construct the cactus in graphs with n vertices and m edges from $O(n^3)$ to $O(nm\log(n^2/m))$.

This paper describes an algorithm that modifies the output of the Hao-Orlin algorithm, rearranging the minimum cuts into an order suitable for a cactus algorithm based on Karzanov-Timofeev framework. The algorithm presented here runs in $O(nm + n^2 \log n)$ time—at least as fast as the Hao-Orlin algorithm. Since an algorithm based on the Karzanov-Timofeev outline requires $O(n^2)$ time, plus the time to find and sort all minimum cuts, this leads to the fastest known deterministic algorithm to construct a cactus of a weighted graph.

As noted earlier, the paper in which Karzanov and Timofeev [20] describe how to build a cactus is lacking some important details. Some of these are provided by Naor and Vazirani [22] in a paper that describes a parallel cactus algorithm. Both of these algorithms are not correct since they construct cacti that may not contain all minimum cuts of the original graph. De Vitis [25] provides a complete and correct description of an algorithm to construct a cactus, based on the ideas in [20] and [22]. Karger and Stein [19] give a randomized algorithm for constructing the chain representation in $O(n^2 \log^3 n)$ time. Benczúr [3] outlines another approach to build a cactus without using the chain representation. However, it is not correct since it constructs cacti that may not contain all minimum cuts of the original graph. Gabow [11, 12] describes a linear-sized representation of minimum cuts of an unweighted graph and gives an $O(m + \lambda^2 n \log(n/\lambda))$ time algorithm to construct this representation, where λ is the number of edges in the minimum cut.

2 Preliminaries

2.1 Definitions and Notation

We assume the reader is familiar with standard graph terminology as found in [7]. A *graph* $G = (V, E)$ is defined by a set of vertices V, with $|V| = n$ and a set of edges $E \subseteq V\text{x}V$, with $|E| = m$. A *weighted graph* also has a weight function on the edges, $w : E \to \Re$. For the purposes of this paper, we assume w is non-negative. A *cut* in a graph is a non-empty, proper subset of the vertices. The weight of a cut C is the sum of weights of edges with exactly one endpoint in the set. The weight of edges with one endpoint in each of two disjoint vertex sets S and T is denoted as $w(S, T)$. A *minimum cut* is a cut C with $w(C, \overline{C}) \leq w(C', \overline{C'})$ for all cuts C'.

An (S,T)*-cut* is a cut that contains S and is disjoint from T. A (S,T) *minimum cut* is a minimum cut that separates S and T. Note that if the value of the minimum (S,T)-cut is greater than the value of the minimum cut, there are no (S,T) minimum cuts.

2.2 The Structure of Minimum Cuts of a Graph

A graph can have at most $\binom{n}{2}$ minimum cuts. This is achieved by a simple cycle on n vertices: there are $\binom{n}{2}$ choices of pairs of edges broken by a minimum cut. This is also an upper bound as shown in [5, 8] or, with a simpler proof, in [18].

Let λ be the value of a minimum cut in graph $G = (V, E)$.

Lemma 2.1 *Let $A \neq B$ be disjoint minimum cuts such that $T = A \cup B \neq V$ is also a minimum cut. Then $w(A, \overline{T}) = w(B, \overline{T}) = w(A, B) = \lambda/2$.*

Proof: Summing up the values of the edges leaving sets A and B, we get $2w(A,B) + w(T, \overline{T}) = w(A, \overline{A}) + w(B, \overline{B}) = 2\lambda$. Since T is a minimum cut, $2w(A,B) = \lambda$ and the last equality is shown. The proof is completed by noting the symmetrical relations between A, B, and $\overline{T}$. ∎

Lemma 2.2 *If S_1 and S_2 are minimum cuts such that none of $A = S_1 \cap S_2$, $B = S_1 \backslash S_2$, $C = S_2 \backslash S_1$, or $D = \overline{S_1 \cup S_2}$ is empty, then*

1. *A, B, C, and D are minimum cuts,*
2. *$w(A, D) = w(B, C) = 0$,*
3. *$w(A, B) = w(B, D) = w(D, C) = w(C, A) = \lambda/2$.*

Proof: Since S_1 and S_2 are minimum cuts, we have $2\lambda = w(S_1, \overline{S_1}) + w(S_2, \overline{S_2}) = w(A, \overline{A}) + w(D, \overline{D}) + 2w(B, C) = w(C, \overline{C}) + w(B, \overline{B}) + 2w(A, D)$. Since none of A, B, C, or D is empty, this sequence of equations implies that all of A, B, C, and D are minimum cuts and $w(B, C) = w(A, D) = 0$. The last item is a consequence of item 1 and Lemma 2.1. ∎

Two cuts $(S_1, \overline{S_1})$ and $(S_2, \overline{S_2})$ that meet the conditions of the above lemma are called *crossing* cuts.

A fundamental lemma further explaining the structure of minimum cuts in a graph is the Circular Partition Lemma. This lemma is proven by Bixby [5] and Dinits, et al. [8], with alternate proofs in [3, 25].

Definition 2.3 *A* circular partition *is a partition of V into $k \geq 3$ disjoint subsets $V_1, V_2, \ldots, V_k$, such that*

- *$w(V_i, V_j) = \lambda/2$ when $i - j = 1 \mod k$ and equals zero otherwise.*
- *For $1 \leq a < b \leq k$, $A = \cup_{i=a}^{b-1} V_i$ is a minimum cut, and if B is a minimum cut such that B or $\overline{B}$ is not of this form, then B or $\overline{B}$ is contained in some V_i.*

Lemma 2.4 (Bixby [5]; Dinits, Karzanov, Lomonosov [8]) *If S_1 and S_2 are crossing cuts in G, then G has a circular partition $\{V_1, V_2, \ldots, V_k\}$ such that each of $S_1 \cap S_2$, $S_1 \backslash S_2$, $S_2 \backslash S_1$, and $\overline{S_1 \cup S_2}$ equal $\cup_{i=a}^{b-1} V_i$ for appropriate choices of a and b.*

The proof uses the fact that the union and intersection of crossing cuts are tight, as established by Lemma 2.2, to argue that any minimum cut not represented in a circular partition must be contained in one of the sets of the partition.

It may be that G has more than one circular partition. Let $P := \{V_1, \ldots, V_k\}$ and $Q := \{U_1, \ldots, U_l\}$ be two distinct circular partitions of V. These partitions are *compatible* if there is a unique i and j such that $U_r \subset V_i$ for all $r \neq j$ and $V_s \subset U_j$ for all $s \neq i$.

Lemma 2.5 *Any two circular partitions of a graph G are compatible.*

Proof: Assume P and Q are two circular partitions of V as defined above. By definition, for every $1 \leq i \leq k$, there is a $1 \leq j \leq l$ such that either V_i or $\overline{V_i}$ is contained in U_j. Thus, if there is an i such that V_i and V_{i+1} are contained in two different sets of the partition A, then each $\overline{V}_p$, $1 \leq p \leq k$, $p \neq i, i+1$ is not contained in any U_j. Hence, each V_p, $1 \leq p \leq k$ is contained in some U_q, $1 \leq q \leq l$. By definition of circular partition, each V_p must be contained in a unique U_q, and hence $P = Q$. If $V_p \subset U_j$, for all $1 \leq p \leq k$, then $U_j = V$ and $\overline{U_j} = \emptyset$, contradicting the fact that Q is a circular partition. Hence there is at least one i and j such that $\overline{V_i}$ is contained in U_j. This implies that all $V_r \subset U_j$, $r \neq i$, and that $\overline{U_j} \subset V_i$, which further implies that all $U_s \subset V_i$, $s \neq j$. ■

A set of sets is called *laminar* if any pair of sets are either disjoint or one set is contained in the other. Laminar sets that do not include the empty set can be represented by a tree where the root node corresponds to the entire underlying set, and the leaves correspond to the sets that contain no other sets. The parent of a set A is the smallest set properly containing A. Since the total number of nodes in a tree with n leaves is less than $2n$, the size of the largest set of laminar sets of n objects is at most $2n - 1$.

Corollary 2.6 *There are at most $n-2$ distinct circular partitions of a graph on n vertices.*

Proof: Represent each circular partition by one of the sets in the partition with the property that this set contains at least two vertices. This is possible if there is more than one circular partition of G. By Lemma 2.5, these sets do not cross. There can be at most $2n-1$ laminar sets of n objects, and thus at most $n-1$ laminar sets containing more than one object. In addition, the entire set of objects cannot represent any circular partition, so there are at most $n-2$ distinct circular partitions of a graph on n vertices. ∎

2.3 The Cactus Representation

If G has no circular partitions, then G has no crossing cuts. In this case, the minimum cuts of G are *laminar* sets of vertices: any two sets are either mutually disjoint or one set is contained in the other. Since a laminar system on a ground set of n objects can contain at most $2n-2$ proper subsets, if G has no crossing cuts, it has at most $2n-2$ minimum cuts.

We can represent these minimum cuts of G with a tree: Consider the smallest side of every cut and represent the cut with a single node. Connect two nodes corresponding to cuts A and B if $A \subset B$ and there is no minimum cut C such that $A \subset C \subset B$. Finally, connect the nodes of all largest cuts to a single extra node. A vertex of G is mapped to the node of this tree that corresponds to the smallest cut containing this vertex. If there is no such node, this vertex is mapped to the extra node. (With this mapping there may be many nodes of the tree that do not have vertices mapped to them, since the tree may contain $2n-1$ nodes, while there are n vertices.) There is a 1-1 correspondence between the minimum cuts of G and the minimum cuts of this tree: the vertices mapped to the nodes in a cut of the tree form the corresponding cut in G.

In order to represent crossing minimum cuts, it is necessary to add cycles to this tree representing circular partitions of G. First note that for a circular partition $\{V_1, V_2, \ldots, V_k\}$, the minimum cuts represented by the partition, namely minimum cuts of the form $\cup_{i=a}^{b-1} V_i$ for any $1 \leq a < b \leq k$, can be described by the cuts of a simple cycle where each V_i corresponds to a node on the cycle. A cactus of a graph is a data structure that succinctly represents all minimum cuts of G by succinctly representing all circular partitions of G.

A *cactus* is a tree-like graph that may contain cycles as long as no two cycles share more than one vertex: every edge in the cactus lies on at most one cycle. Figure 1 contains an example of a cactus. A *cactus of a graph* G, denoted $\mathcal{H}(G)$, has in addition a mapping π that maps vertices of G to vertices of $\mathcal{H}(G)$, and a weight assigned to every edge. To distinguish the vertices of a graph from those of its cactus, we refer to the vertices of a cactus as *nodes*. If λ is the value of the minimum

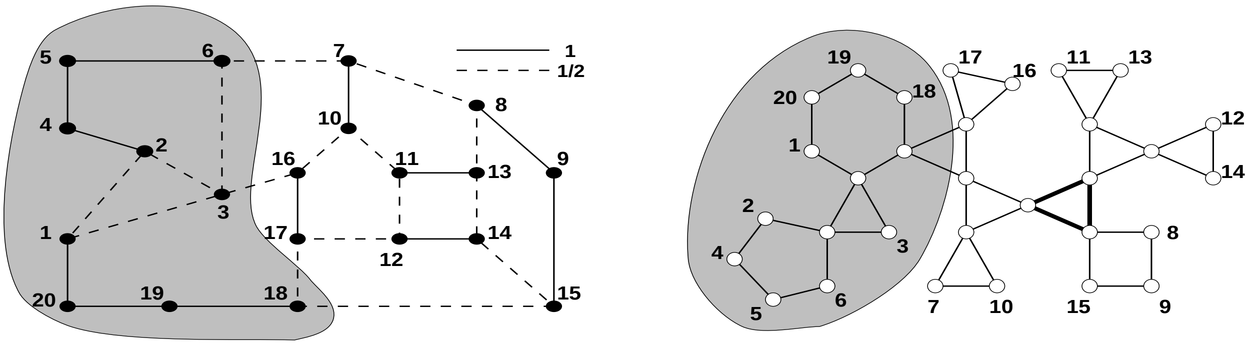

Figure 1: Cactus of a graph and an example of corresponding minimum cuts

cut in G, then each cycle edge of $\mathcal{H}(G)$ has weight $\lambda/2$, and each path edge has weight λ. The mapping π is such that every minimum cut M of $\mathcal{H}(G)$ corresponds to a minimum cut $\pi^{-1}(M)$ of G, and every minimum cut in G equals $\pi^{-1}(M)$ for some minimum cut M of $\mathcal{H}(G)$. If $\pi^{-1}(i) = \emptyset$, we say that i is an *empty* node.

Figure 1 contains a graph, its cactus, and an example of corresponding minimum cuts. In this example, there is a 1-1 correspondence between cycles of the cactus and circular partitions of G_x. For example, consider the bold-lined cycle in the cactus. This cycle corresponds to the following circular partition: $\{\{11-14\},\{8,9,15\},\{1-7,10,16-20\}\}$. In general, a cycle in the cactus corresponds to the circular partition defined by the sets of vertices mapped to the components of the cactus created when all edges of the cycle are deleted from the cactus. Theorem 2.9 establishes that for every graph G, there is always a cactus with a 1-1 correspondence between cycles of the cactus and circular partitions of G.

If x is in the subtour polytope, i.e. x is a solution to (1) - (3), then $\lambda = 2$ for G_x. Also, the degree constraints imply that each vertex of G_x is a minimum cut, so that each vertex of G_x is mapped to a unique node of the cactus.

The following theorem was originally proved by Dinits, Karzanov, and Lomonosov [8]. We present a different proof. Although the proof is constructive, the construction is not intended to be a guideline for constructing a cactus. The theorem, following corollary, and proofs also appear in [9], along with a detailed explanation of how to construct a cactus efficiently.

Theorem 2.7 (Dinits, Karzanov, Lomonosov [8]) *Every weighted graph has a cactus.*

Proof: Let $\Omega = \{C \mid C \text{ a minimum cut in } G\}$, and let $\Xi = \{\{V_1,\dots,V_k\} \mid \{V_1,\dots,V_k\} \text{ a circular partition of } G\}$. Given Ω and Ξ, we construct a cactus as follows. Let $\Psi = \{C \in \Omega \mid \forall \{V_1,\dots,V_k\} \in \Xi,\ \exists\, V_i \in \{V_1,\dots,V_k\} \text{ such that } C \subset V_i \text{ or } \overline{C} \subset V_i\}$. Note that Ψ does not contain crossing cuts, although there may be sets in Ψ that are not defined by any circular partition.

Step 1: Create a node ν_A for every element $A \in \Psi$. Connect in a cycle the nodes corresponding to sets in a circular partition. If $|\overline{A}| \leq |A|$, associate ν_A with $\overline{A}$ and rename ν_A to $\nu_{\overline{A}}$. If A is contained in Ψ, then either A or $\overline{A}$ may now be represented by two different nodes. For each cycle, note that at most one node ν_A in the cycle is renamed to $\nu_{\overline{A}}$. In addition, by compatibility of circular partitions, at most one cycle has no node renamed. Call the node of the cycle that is renamed the *parent* of the other nodes in the cycle. This is a parent of *type R*. If there is a cycle with no parent, this cycle is the *root* of the cactus.

Step 2: For any set A that is represented by two different nodes, *join* the (possibly empty) cycles containing the two nodes by creating a new node with label A, connecting this new node to any node adjacent to the original nodes labeled A, and deleting these original nodes labeled A along with all edges adjacent to them. If the deleted node was the parent of the nodes on one cycle, the new node is the new parent for these same nodes. Observe that, by the way we labeled the nodes, there cannot exist a sequence of cycles $C_1, C_2, \ldots, C_r$ for $r \geq 3$ such that C_i is joined to C_{i+1} for $1 \leq i \leq r$ and C_r is joined to C_1.

Step 3: Up to now, we have created a graph that consists of isolated nodes, isolated cycles, and trees of cycles. The next step connects these components into trees of cycles and edges. Connect node ν_A to node ν_B by an edge if (ν_A, ν_B) is not already an edge, $A \subseteq B$, and there is no other $C \in \Omega$ with $A \subset C \subset B$. In this case, ν_B is called the *parent* of ν_A. This is a parent of *type S*.

We now argue that no node has more than one parent (of any type): Suppose $B \neq B'$ are both parents of A of type S. By construction, $B \not\subset B'$ and $B' \not\subset B$. Also, since $B \cap B'$ is a tight set not equal to B and B' and $A \subset B \cap B'$, by construction it must be that $A = B \cap B'$. If $\overline{B \cup B'} = \emptyset$, then $\{A, \overline{B}, \overline{B'}\}$ is a circular partition, hence A is already connected to B and B' before Step 3, contradiction. Otherwise, B and B' are crossing cuts, contradicting the composition of Ψ. Now suppose $B^* \neq B$ are both parents of A where B^* is of type R and B is of type S. Since $A \subset B^* \cap B$ is a tight set and B is of type S, either $A = B^* \cap B$ or $B \subset B^*$. In the former case, the above arguments show a contradiction. In the latter case, since A and $\overline{B^*}$ are sets of the same circular partition, either $A = B$ (contradiction) or $B = A \cup A' = B^*$ where A' is another set of the same circular partition. Hence $\{A, A', \overline{B^*}\}$ is a circular partition, and thus A is already connected to B^*.

At this point, we either have a tree rooted at either a single node or the cycle root, or a set of tree-like components. Each component has a node ν_A that has no parent. This node is the *root* of the component. All other sets represented by nodes in this component are contained in A. To create one connected component, introduce a new node, not associated with any set, and connect the root node of each component to this new node.

This final graph, H is cactus in form: it is a tree of cycles and paths. We need now to define the mapping of vertices of G to the nodes of this graph. A vertex $v \in G$ is mapped to the node of H that represents the smallest minimum cut A containing v. If no cut represented in H contains v, then v is mapped to the new node. Each cut-edge of H is given weight λ, and each cycle-edge is given weight $\lambda/2$. Now H is a cactus of G. If $|A| < |\overline{A}|$ is a minimum cut contained in Ψ, then this minimum cut corresponds to the minimum cut of H obtained by removing the edge (or pair of edges, if ν_A and the parent of ν_A are on the same cycle) adjacent to ν_A that connects ν_A to its parent. ■

Corollary 2.8 *Every weighted graph on n vertices has a cactus with no more than $2n-2$ vertices.*

Proof: After Steps 1 and 2, there may be at most one cut $(A, \overline{A})$ represented by two different nodes, in which case, $|A| = |\overline{A}|$ and the nodes are labeled ν_A and $\nu_{\overline{A}}$. In this case, ν_A and $\nu_{\overline{A}}$ are both connected to the new node in the above cactus. No other edge is incident to the new node: all other cuts $C \in \Psi$ do not cross A, so the smaller of C and $\overline{C}$ must be contained in either A or $\overline{A}$. This implies that the new node and its incident edges may be replaced by edge $(\nu_A, \nu_{\overline{A}})$ without losing representation of A, or any other minimum cut.

All other minimum cuts in Ψ are represented by exactly one node, corresponding to the smaller side of the cut. Among the nodes whose parent is the new node, arbitrarily select one, ν_A, and associate the new node with the set $\overline{A}$. Since none of the sets in Ψ cross, the remaining nodes in the cactus with these modifications, correspond to distinct laminar sets. Hence the total number of nodes in a cactus may be bounded by the maximum number of proper laminar sets of n objects, which is $2n-2$. ■

As defined here, and in [8], a graph does not necessarily have a unique cactus. For instance, a cycle on three nodes can also be represented by a cactus that is a star: an empty node of degree three, and three nodes of degree one each containing a distinct vertex. There are many rules that could make the definition unique [25, 22]. We will follow De Vitis [25] in the definition of a canonical cactus below.

Let i be a node on cycle Y of $\mathcal{H}(G)$. Let C_i^Y be the component containing i formed by removing the edges adjacent to i on Y, and let $V_i^Y = \pi^{-1}(C_i^Y)$. We call i *trivial* if $V_i^Y = \emptyset$. By removing V_i^Y and making the neighbors of i on Y neighbors in Y, we can assume $\mathcal{H}(G)$ has no trivial nodes. We also assume that $\mathcal{H}(G)$ has no empty, 3-way cut-nodes: an empty cut-node whose removal breaks $\mathcal{H}(G)$ into exactly three components can be replaced by a 3-cycle. Finally, we assume there are no empty nodes i in $\mathcal{H}(G)$ of degree three or less: either i is a 3-way cut-node, and handled as above, or it is a 2-way cut-node, and hence we can contract an incident cut-edge and still maintain a representation of all minimum cuts of G in $\mathcal{H}(G)$. A *canonical* cactus of a graph is a cactus with no trivial nodes, no empty 3-way cut-nodes, and no empty nodes with degree ≤ 3. For support graphs of solutions to (1) - (3), all cut nodes of the canonical cactus are empty, because all vertices are minimum cuts. The following theorem is not hard to prove.

Theorem 2.9 (De Vitis [25]) *Every weighted graph has a unique canonical cactus.*

Henceforth, the term cactus will be used to mean canonical cactus. In this canonical representation, every circular partition of G corresponds to a cycle of $\mathcal{H}(G)$ and every cycle of $\mathcal{H}(G)$ represents a circular partition of G. For cycle Y, the V_i^Y are the sets of the corresponding circular partition.

2.4 The Chain Representation of Minimum Cuts

Karzanov and Timofeev [20] make an important observation about the structure of minimum cuts, which they use to build the cactus: if two vertices s and t are adjacent in the graph G, then the minimum cuts that separate these vertices are nested sets. This motivates assigning vertices of G an *adjacency order* $\{v_1, \ldots, v_n\}$ so that v_{i+1} is adjacent to a vertex in $V_i := \{v_1, \ldots, v_i\}$. Let M_i be the set of minimum cuts that contain V_i but not v_{i+1}. We will refer to such minimum cuts as (V_i, v_{i+1}) *minimum cuts.* The following lemma summarizes the observation of [20].

Lemma 2.10 *If the vertices in G are adjacency ordered, then all cuts in M_i are non-crossing.*

Proof: Suppose not, and let $(X, \overline{X})$ and $(Y, \overline{Y})$ be two crossing cuts in M_i with $V_i \subseteq X \cap Y$. Then $v_{i+1} \in \overline{X} \cap \overline{Y}$ and Lemma 2.2 implies that $x(X \cap Y, \overline{X} \cap \overline{Y}) = 0$. But this contradicts the numbering of vertices which insures that $x(V_i, v_{i+1}) > 0$. Hence all cuts in M_i are non-crossing. ∎

This implies that the cuts in M_i form a nested chain and can be represented by a *path* P_i of sets that partition V: let $A_1 \subset A_2 \subset \cdots \subset A_l$ be the minimum cuts separating V_i and v_{i+1}. The first set X_1 of the path is A_1, and each additional set $X_r = A_r \backslash A_{r-1}$, with the final set X_{l+1} equal to $V \backslash A_l$. Each link of the path represents the minimum cut A_r that would result if the path were broken between sets X_r and X_{r+1}.

Note that, for any ordering of vertices, the M_i form a partition of the minimum cuts of G: v_i and v_{i+1} are the smallest, consecutively indexed pair of vertices that are separated by a cut, if and only if the cut is in M_i. The set of P_i for all $1 \leq i < n$ is called the *chain representation of minimum cuts.* For this reason, we will refer to these paths as *chains.*

2.5 From the Chain Representation to the Cactus

The algorithm outlined by Karzanov and Timofeev [20], refined by Naor and Vazirani [22], and corrected by De Vitis [25], builds the cactus representation from the chain representation of minimum cuts. For completeness, the algorithm is presented here. Let G_i represent the graph G with nodes in V_i contracted into one node. Let G_r be the smallest such graph that has a minimum cut of value λ (r is the largest index of such a graph). The algorithm starts with the cactus for G_r. $\mathcal{H}(G_r)$ is a pair of vertices incident to one edge. It then builds $\mathcal{H}(G_i)$, the cactus for G_i, from $\mathcal{H}(G_{i+1})$ using the following observation.

Corollary 2.11 *Let the vertices in G have an adjacency ordering. For each non-empty M_i, there is a path in the cactus of G_i that shares at most one edge with every cycle, and such that all cuts in M_i cut through this path.*

Proof: The path is the shortest path in G_i connecting the node containing v_{i+1} to the node containing V_i. Clearly all cuts in M_i cut through this path. If this path uses two edges of some cycle, this would imply that there are crossing cuts in M_i, contradicting Lemma 2.10. ■

By contracting the path described in the lemma into a single node, all cuts in M_i are removed from $\mathcal{H}(G_i)$, and no other minimum cuts are removed. Thus the resulting structure is $\mathcal{H}(G_{i+1})$.

Working in the opposite direction, in iteration i, the cactus algorithm replaces the node q in $\mathcal{H}(G_{i+1})$ that contains vertices $\{v_1, \ldots, v_i\}$ with the path P_i that represents the chain of cuts in M_i. Edges that were incident to q are then joined to an appropriate node in P_i. The mapping of vertices of G_i to vertices of the cactus $\mathcal{H}(G_i)$ are updated. Lastly, post-processing completes the construction of $\mathcal{H}(G_i)$, the canonical cactus of G_i. More precisely, the algorithm proceeds as follows.

(i) If $P_i = X_1, \ldots, X_k$ then replace q by k new nodes $q_1, \ldots, q_k$ with edges (q_j, q_{j+1}) for $1 \le j < k$.

(ii) For any tree or cycle edge (q, w) in $\mathcal{H}(G_{i+1})$, let $W \neq \emptyset$ be the set of vertices in w, or if w is an empty node, the vertices in any non-empty node w' reachable from w by some path of edges disjoint from a cycle containing (q, w). Find the subset X_j such that $W \subset X_j$ and connect w to q_j.

(iii) Let Q be the set of vertices mapped to q in $\mathcal{H}(G_{i+1})$. Assign to q_j the set $X_j \cap Q$. All other mappings remain unchanged.

(iv) Remove all empty nodes of degree ≤ 2 and all empty 2-way cut nodes by contracting an adjacent tree edge. Replace all empty 3-way cut-nodes with 3-cycles.

The correctness of this procedure is easy to prove by induction using the previous observations and the following two lemmas.

Lemma 2.12 *If (q, w) is a tree edge of $\mathcal{H}(G_{i+1})$ and $T \subset V$ is the set of vertices mapped to the subtree attached to q by (q, w), then $T \subseteq X_j$ for some j.*

Proof: Suppose T is split among several X_j. Let z be the smallest index such that $T' = X_z \cap T \neq \emptyset$. Since $V_i \subset Q$ and $Q \cap T = \emptyset$, $z > 1$. Thus $B = \cup_{i=1}^{z} X_i$ is a minimum cut in G_i that separates V_i and v_{i+1}, and T' and $T \backslash T'$. This implies that T and B are crossing cuts, and hence $T' = T \cap B$, $T \backslash T' = T \backslash B$, and $\overline{T}$ are cuts of a circular partition, and must lie on a cycle of $\mathcal{H}(G_{i+1})$. This contradicts the fact that T is attached to q by a tree edge. ■

Lemma 2.13 *Let (q, w) be a cycle edge of $\mathcal{H}(G_{i+1})$ that lies on cycle Y, and let $T_1, T_2, \ldots, T_r$ be the circular partition represented by Y with $V_i \subset Q = T_1$. Then either $\bigcup_{l=2}^{r} T_l \subset X_j$ for some j, or there are indices a and b such that $T_2 = X_a$, $T_3 = X_{a+1}$, $\ldots$, $T_r = X_b$.*

Proof: If a minimum cut B in M_i splits T_1, it cannot split any other T_l, since the B would cross the cut T_l, and hence the circular partition represented by Y should have been refined to replace T_l with $T_l \cap B$ and $T_l \backslash B$. Thus for all $l \geq 2$, $T_l \subseteq X_j$ for some j.

If none of the cuts in M_i split $\overline{T_1}$ then $\bigcup_{l=2}^{r} T_l \subset X_j$ for some j. Otherwise there is a cut that splits both T_1 and $\overline{T_1}$; these two cuts cross. This implies there is a circular partition that refines Y by splitting T_1 into T_1' and T_1''. Hence all cuts of the form $A = T_1' \cup_{j=2}^{l} T_j$, $l \leq r$ are in M_i, and any cut in M_i that splits $\overline{T_1}$ is of this form. ■

Clearly, operations (i)-(iii) of the above procedure can be performed in linear time, thus requiring $O(n^2)$ time to build the entire cactus from the chain representation of minimum cuts. Operation (iv) requires constant time per update. Each contraction removes a node from the cactus, implying no more than $O(n)$ contractions per iteration. Each cycle created is never destroyed, and since the total number of circular partitions of a graph is at most n, there can not be more than n of these operations over the course of the algorithm.

Theorem 2.14 *The canonical cactus of a graph can be constructed from the chain representation of all minimum cuts of the graph in $O(n^2)$ time.* ■

2.6 Finding All Minimum Cuts in a Weighted Graph

To find all minimum cuts, we make minor modifications to the minimum cut algorithm of Hao and Orlin [17]. The Hao-Orlin algorithm is based on Goldberg and Tarjan's preflow-push algorithm for finding a maximum flow [15]. It starts by designating a source vertex, u_1, assigning it a label of n, and labeling all other vertices 0. For the first iteration, it selects a sink vertex, u_2, and sends as much flow as possible from the source to the sink (using the weights as arc capacities), increasing the labels on some of the vertices in the process. The algorithm then repeats this procedure $n-2$ times, starting each time by contracting the current sink into the source, and designating a node with the lowest label as the new sink. The overall minimum cut is the minimum cut found in the iteration with the smallest maximum flow value, where the *value* of a flow is the amount of flow that enters the sink.

To find all minimum cuts with this algorithm, we use the following result. Define a *closure* of a set A of vertices in a directed graph to be the smallest set containing A that has no arcs leaving the set.

Lemma 2.15 (Picard and Queyranne [23]) *There is a 1-1 correspondence between the minimum (s,t)-cuts of a graph and the closed vertex sets in the residual graph of a maximum (s,t)-flow.*

Let U_i be the set of vertices in the source at iteration i of the Hao-Orlin routine, $1 \leq i \leq n-1$. If a complete maximum flow is computed at iteration i, Lemma 2.15 implies that the set of (U_i, u_{i+1}) minimum cuts equals the set of closures in the residual graph of this flow. This representation can be made more compact by contracting strongly connected components of the residual graph, creating a directed, acyclic graph (DAG). We refer to the graph on the same vertex set, but with the direction of all arcs reversed (so that they direct from U_i to u_{i+1}), as the *DAG representation of (U_i, u_{i+1}) minimum cuts.*

The Hao-Orlin routine does not necessarily compute an entire maximum flow at iteration i. Instead it computes a flow of value $\geq \lambda$ that obeys the capacity constraint for each edge, but leaves excess flow at some nodes contained in a *dormant* region that includes the source. Let R_i denote the vertices in the dormant region at the end of the i^{th} iteration of the Hao-Orlin algorithm. The following lemma follows easily from the results in [17].

Lemma 2.16 *R_i is contained in the source component of the DAG representation of (U_i, u_{i+1}) minimum cuts.*

It follows that at the end of iteration i of the Hao-Orlin algorithm, we can build a linear-sized DAG representation of all (U_i, u_{i+1}) minimum cuts.

The second and more serious problem with the Hao-Orlin algorithm is that the ordering of the vertices implied by the selection of sinks in the algorithm may not be an adjacency ordering as required for the construction phase of the Karzanov-Timofeev framework. This means that some of the DAGs may not correspond to directed paths. In the next section, we explain how to transform the (U_i, u_{i+1}) minimum cut DAGs into (V_i, v_{i+1}) minimum cut chains (directed paths) for some adjacency ordering $\{v_1, \dots, v_n\}$.

3 The Algorithm

In this section, we discuss how to transform the DAG representation of all minimum cuts into the chains that can be used to construct a cactus representation of all minimum cuts. To do this, we use an intermediate structure called an (S, T)-cactus.

Definition 3.1 *An (S, T)-*cactus *is a cactus representation of all minimum cuts separating vertex sets S and T.*

Note that an (S, T)-cactus is not necessarily a structure representing minimum (S, T)-cuts, but overall minimum cuts that separate S and T. If the minimum (S, T)-cut has value greater than the minimum cut, then the (S, T)-cactus is a single nodes containing all vertices.

Lemma 3.2 *An (S, T)-cactus is a path of edges and cycles.*

Proof: In the cactus representation of all minimum cuts in a graph, consider the smallest minimum cut of the cactus containing all nodes in S and contract the side of the cut containing S into one node. Repeat the same operation for set T. Note that the resulting graph still has the structure of a cactus: by Lemma 2.2, there is a smallest minimum cut containing S, and this must correspond to a (connected) minimum cut in the cactus. Also note that any minimum cut dividing S and T has not been contracted. Now consider a simple path from S to T in this shrunk cactus. Every minimum cut separating S and T breaks through this path. The cactus representing all (S, T) minimum cuts is the graph that includes this path and all cycles of the larger cactus that share at least one edge with this path. Thus the (S, T)-cactus is a path of edges and these cycles. ∎

Each cycle in an (S, T)-cactus has a unique node that is closest to S (the *source-side node*) and a unique node that is closest to T (the *sink-side node*). The other nodes on the cycle form two paths between these two end nodes. We will call these *cycle-paths.* Thus each cycle in the (S, T)-cactus can be described by a source-side node, a sink-side node, and two cycle-paths of nodes connecting them. The *length* of a cycle-path refers to the number of nodes on the cycle-path. A zero length cycle-path consists of just one arc joining the source-side node to the sink-side node.

Without loss of generality, we assume that no cycle of an (S, T)-cacti has a zero length cycle-path. Any cycle of an (S, T)-cactus that contains a zero length cycle-path can be transformed into a path by deleting the arc in the zero length cycle-path. This operation does not delete any (S, T)-cut represented by the (S, T)-cactus.

Note that Lemma 2.10 implies that a (V_i, v_{i+1})-cactus for adjacency ordering $\{v_1, v_2, \ldots, v_n\}$ is a path without cycles, the nested chain of minimum cuts required for the Karzanov-Timofeev framework.. A first step in building these chains is to construct the $n-1$ (S, T)-cacti of the $n-1$ DAG's output by the Hao-Orlin algorithm. From these (S, T)-cacti, we can then construct chains needed for the cactus algorithm.

3.1 Building an (s, t)-cactus

Benczur [3] sketches how to build an (S, T)-cactus from the DAG representation of all minimum cuts separating S and T. The algorithm uses a topological sort of the nodes of the DAG, with some post-processing that can also be performed in $O(m+n)$ time. For completeness, we describe the algorithm here.

For two different cycles of the (S, T)-cactus, all the nodes on the cycle closer to the source must precede all the nodes on the further cycle in any topological ordering. Thus a topological sort sequences the nodes of the DAG so that nodes on a cycle in the (S, T)-cactus appear consecutively. It remains to identify the start and end of a cycle, and to place the nodes of a cycle on the proper chain.

For a node i on cycle Y of a cactus $\mathcal{H}(G)$, define C_i^Y to be the component containing i formed by removing the edges adjacent to i on Y, and define $V_i^Y = \pi^{-1}(C_i^Y)$. If $V_i^Y = \emptyset$ for some Y, we can remove C_i^Y and make the neighbors of i on Y neighbors in Y. Thus we assume $\mathcal{H}(G)$ has no cycle nodes i with $V_i^Y = \emptyset$.

Lemma 3.3 *Let Y be a cycle on three or more nodes of cactus $\mathcal{H}(G)$ then*

1. *The sum of the costs on edges between any two vertex sets V_i^Y and V_j^Y of adjacent nodes i and j on cycle Y equals $\lambda/2$.*
2. *There are no edges between vertex sets V_i^Y and V_h^Y of nonadjacent nodes i and h on Y.*

Proof: If i and j are adjacent on cycle Y of cactus tree K, then $V_i^Y \cup V_j^Y$ is a minimum cut, since $C_i^Y \cup C_j^Y$ is a minimum cut of K that separates $\{i, j\}$ from the other nodes on Y. We can then apply Lemma 2.1 to V_i^Y and V_j^Y to prove (1). To show (2), we note that the total value of the edges leaving V_i^Y is λ. If j_1 and j_2 are the two nodes adjacent to i on Y, then $V_{j_1}^Y \cap V_{j_2}^Y = \emptyset$.

Combining these observations with the first claim, we conclude that edges leaving V_i^Y go to either $V_{j_1}^Y$ or $V_{j_2}^Y$, thus completing the proof. ■

Since we can construct an (S,T)-cactus from the cactus of the complete graph by contracting nodes, this lemma also applies to (S,T)-cacti: on a cycle of an (S,T)-cactus, only adjacent nodes share edges. With this observation, we can now identify when a cycle starts as we scan the topological order of the DAG output by the Hao-Orlin algorithm. In topological order and one at a time, we remove nodes along with their outgoing arcs from the DAG. The first time that that there are two nodes left in the graph with no incoming arcs, we know that the node we have just removed is the source-side node of a new cycle. Before the next cycle starts, the number of nodes with no incoming arcs left in the DAG will return to just one, so that we will be able to identify the source-side node of the next cycle. By removing nodes and their incoming arcs in reverse topological order, we can similarly identify the sink-side nodes of each cycle. It remains to place the remaining cycle nodes on their respective cycle paths. This can be done as we walk through the topological sort. Two nodes that are adjacent in the topological source in between a source-side node and a sink-side node that share no edges are on opposite cycle-paths, and otherwise are on the same cycle path, in the order they appear in the topological sort.

Lemma 3.4 (Benczúr [3]) *A DAG representing strongly connected components of the residual graph of a maximum (s,t)-flow can be transformed into an (s,t)-cactus in $O(m+n)$ time.* ■

3.2 Constructing the chains

In this section we describe how to build chains H_i that can be used to construct a cactus representation of all minimum cuts from the (U_{j-1},u_j)-cacti, $1<j\le n$. Let D_j be the (U_{j-1},u_j)-cactus for $1<j\le n$. We start by fixing an adjacency ordering $\{v_1,\dots,v_n\}$ with $v_1=u_1$ and let $\sigma: V\to V$ be the permutation $\sigma(i)=j$ if $v_i=u_j$.

The algorithm we present below builds each chain H_i, $1<i\le n$, one at a time by reading cuts off of the D_j. The basic idea is to use the following fact established below: If any two cuts from D_j are such that one is completely contained within the other, then the index of the chain to which the first belongs is no more than the index of the chain to which the second belongs. Thus, all the minimum cuts belonging in H_i that are contained in one D_j are consecutive; and, as we walk through the cuts from U_{j-1} to u_j in D_j, the index of the chain to which a cut belongs is non-decreasing. The validity of these statements is established in the next two subsections.

3.2.1 The Code

(s,t)-CactiToChains $(D_2, \ldots, D_n)$

0. For $i = 2$ to n,
0. $H_i = \emptyset$.
0.   For $j = 2$ to $n-1$,
1. $n_{ij} :=$ the node in D_j that contains v_i.
2.   If n_{ij} is not the source node,
3.   Identify the path of nodes from source to the node preceding n_{ij} in D_j.
4.   If $H_i = \emptyset$, let the source node of D_j be the source node of H_i.
5.   Else, add vertices in the source but not in H_i to the last node of H_i.
6.   Append the remaining path to H_i, skipping empty nodes;
7.   and, in D_j, contract this path into the source node.
8.   If n_{ij} is on a cycle in D_j,
9.   Append to H_i the the chain of the cycle that does not contain n_{ij}.
10.   Contract n_{ij} into the source node of D_j.
11.   Add a component to H_i that contains only u_j.
12.   If $H_i \neq \emptyset$, add the vertices not in H_i to the last node of H_i.

3.2.2 Correctness

The aim of this section is to prove that each H_i constructed by (s,t)-CactiToChains represents all cuts in M_i. Define a *phase* to be one pass through the outer while loop.

Lemma 3.5 *At the end of phase i, all vertices $v_k, k \leq i$ are contained in the source node of every remaining D_j.*

Proof: Proof is by induction. At start, all sources contain vertex $v_1 = u_1$. Assume vertices v_k, $k < i$ are contained in the sources of the remaining D_j at the start of phase i. For each j, if n_{ij} is not the source node, then n_{ij} is contracted into the source in Step 10. ■

Lemma 3.6 *In phase i, for all j, n_{ij} either lies on the path in D_j from the source to the first node of the cycle closest to the source, or is one of the two nodes on the cycle adjacent to this path.*

Proof: Figure 2 shows the possible locations of n_{ij} in D_j as specified in the lemma. We show that if there is a cycle between n_{ij} and the source S_j, then we can find crossing cuts in M_i, contradicting

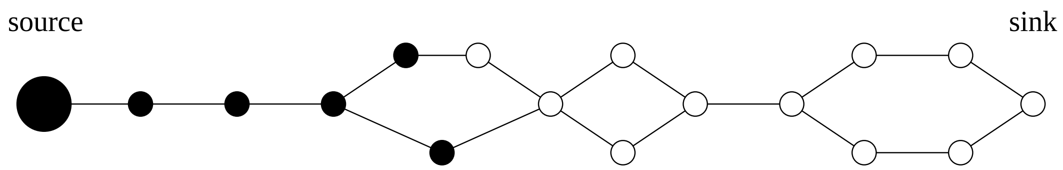

Figure 2: The (s,t)-cactus D_j, updated for iteration i. The black nodes represent the possible locations of n_{ij}.

our choice of the ordering of the v_i. If n_{ij} is not in one of the locations stipulated in the lemma, then some cycle contains at least one node that either precedes, or is unrelated to n_{ij}, on both of its cycle-paths. (Recall that we are assuming that every cycle-path has length greater than zero.) Thus there are two crossing minimum cuts that separate the source of D_j from v_i. Lemma 3.5 implies that V_{i-1} is contained in the source, and hence these are cuts in M_i, contradicting Lemma 2.2. ∎

Lemma 3.7 *The algorithm constructs chains that contain every vertex exactly once.*

Proof: By construction, each H_i is a chain that contains every vertex. Suppose, in iteration j, we are about to add a vertex u_r to the chain H_i. If u_r is in the source component of D_j, we explicitly check before we add u_r. Otherwise, $r \geq j$ and there is some cut that separates vertices in U_{j-1} and V_{i-1} from vertices u_r and u_j. All cuts added to H_i before this iteration come from a D_k with $k < j$, and thus contain a strict subset of vertices in U_{j-1}. By Lemma 2.10, the cuts separating V_{i-1} from v_i are nested, and so none of the previous cuts contain u_r in the source side. Thus we add u_r to H_i exactly once. ∎

Theorem 3.8 *Each chain H_i constructed by the algorithm contains all cuts in M_i.*

The proof of this theorem follows the next two lemmas, and establishes the correctness of the algorithm. H_i is *complete* when it contains all vertices. A node of H_i is *complete* when either a new node of H_i is started, or H_i is complete. For example, a node started at Step 11 is not complete until after the next execution of Step 5 or Step 12. A cut in D_j is obtained by either removing an arc on a path or a pair of arcs on two different cycle-paths of a cycle. The one or two nodes that are endpoints of these arcs on the source side are said to *represent* this cut.

Lemma 3.9 *Each complete node added to H_i represents the same cut that is represented by the node in the current D_j.*

Proof: Proof is by induction on j. The first node added to H_i is the source node of some D_j and clearly both these cuts are the same. Now assume that up until this last node, all previous nodes added to H_i represented the corresponding cut of the relevant D_k, $k \leq j$. Adding the current node from D_j, we add or have added in previous steps to H_i all the vertices that are in the source side of this cut. We must argue that there are no additional vertices in any node preceding this one in H_i. Recall that the cuts in M_i are nested. Thus all previous nodes added to H_i represent cuts strictly contained in the current cut. Hence the node added to H_i represents the same cut this node represents in the current D_j. ■

Lemma 3.10 *Each complete node added to H_i, $1 < i \leq n$, except the last, represents a distinct cut in M_i.*

Proof: Since each node added to H_i contains some vertex, and no vertex is repeated, each represents a distinct cut. When H_i is started, Lemma 3.5 implies that all vertices in V_{i-1} are contained in the source. Vertex v_i is only added in Step 11, when $j = \sigma(i)$. This starts the final node of H_i since $v_i = u_j$ is contained in the source nodes of all D_k, $k > j$. Hence all previous nodes added to H_i represent cuts that contain V_{i-1} but not v_i, and are thus cuts in M_i. ■

Proof of Theorem 3.8: For a fixed cut C, let v_i be the vertex of least index in $\overline{C}$. We show that this cut is represented in H_i. By the end of each phase k, $k < i$, the algorithm contracts all nodes on the path from the source to the node containing v_k. Since V_{i-1} is contained in the source side of C, this cut is not contracted in D_j at the start of phase i. If the cut C is represented in D_j by one node, it lies on the path of D_j from the expanded source to n_{ij}. Thus this node is added to H_i in either Step 5 or 6. Otherwise the cut is through the same cycle that contains n_{ij}. One node representing the cut is the node immediately preceding n_{ij}, currently source-side node of the cycle. The other node lies on the cycle-path not containing n_{ij}. This cut is added to H_i in Step 9, when this latter node is added. ■

3.2.3 Efficiency

Some care is needed to implement the above algorithm to complete within the same asymptotic time as the Hao-Orlin algorithm. Within the inner "for" loops, operations should not take more than $O(\log n)$ time. We show, that with minor preprocessing and record keeping, all operations will take no more than logarithmic time per operation, or can be charged to some set of minimum cuts.

Before starting the algorithm, we walk through each (s,t)-cactus from t to s, labeling every node with the smallest index of a vertex v_i contained in that node, and labeling every arc with the smallest index on the sink side of the arc. This takes linear time per (s,t)-cactus, for a total of $O(n^2)$ time.

We perform one more preprocessing step. For each vertex, we create an array of length $n-1$. In the array for vertex v_i, entry j indicates if v_i is in the source component of D_j. This takes $O(n^2)$ time to initialize; and, through the course of the algorithm, it will take an additional $O(n^2)$ time to maintain, since once a vertex is contracted into the source, it stays there.

Theorem 3.11 *Algorithm (s,t)-CactiToChains can be implemented to run in $O(n^2 \log n)$ time.*

Proof: In Step 1, we use the labeling of the arcs to walk through D_j starting from the source until we find n_{ij}. By Lemmas 3.5 and 3.6, all these cuts are (V_{i-1}, v_i)-cuts, and hence the arcs are labeled with v_i. The time to do this depends on the out degree of each node encountered in D_j (at most 2), and is charged to the minimum cuts we encounter. Note that if none of the arcs leaving the source is labeled with the current chain index i, then n_{ij} must be the source node. This takes care of Steps 1-3, and Step 8.

Let's put aside Steps 5 and 12 and concentrate on the remaining steps. In these, we are either adding nodes and vertices to H_i, or contracting nodes of D_j. This takes constant time per addition or contraction of vertex. Since each vertex is added at most once to H_i and contracted into the source of D_j at most once, over the course of the algorithm these steps take no more than $O(n^2)$ time.

Finally, we consider Steps 5 and 12. In these, we need to determine the difference of two sets that are increasing over the course of the algorithm. To do this efficiently, we wait until the end of a phase to add the vertices that would have been added at Step 5. At the end of each phase, we make a list of the vertices that are missing from H_i. This can be done in linear time per phase by maintaining an array in each phase that indicates if a vertex has been included in H_i. For each of the vertices not in H_i, we then need to determine to which node of H_i it should be added. The nodes that are possible candidates are the incomplete nodes—any node created in Step 11.

Vertex v is added to the node containing u_k in Step 5 if v is contained in the source of D_j, $j > k$ but is not contained in the source of D_k, for iterations k and j where n_{ik} and n_{ij} are not source nodes, and k is the highest index with these properties. Consider the D_j for which n_{ij} is not the source node. The sources of these D_j are nested for any fixed i, by the observation that the cuts of M_i are nested. That is, once a vertex is contained in one of them, it is contained in the rest. During each phase, we maintain a array-list of the consecutive j for which n_{ij} is not the source node. For each vertex, we can now binary search through these nested sources using the arrays described before the statement of the theorem to find the node of H_i that should contain v in $O(\log n)$ time. The time spent on this step over the course of the algorithm is then bounded by $O(n^2 \log n)$: logarithmic time per vertex per phase.

Note that we have added vertices to the sources of these D_j since we started the phase. All the vertices we added in Step 7 have also been added to H_i, so they are not among the vertices we need to place at the end (i.e. the vertices not in H_i). We do need to worry about the vertices added in Step 10, however. To avoid this problem, simply postpone the contractions described in Step 10 until after we have assigned all vertices to a node of H_i. ∎

Since $mn \log n^2/m$ is never smaller than $n^2 \log n$, we have the following corollaries.

Corollary 3.12 *The chain representation of all minimum cuts of a weighted graph can be constructed in $O(mn \log n^2/m)$ time.* ∎

Corollary 3.13 *The cactus representation of all minimum cuts of a weighted graph can be constructed in $O(mn \log n^2/m)$ time.* ∎

Acknowledgement

I would like to thank Éva Tardos for assistance with the proof of Theorem 3.11.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows.* Prentice Hall, 1993.

[2] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. Finding cuts in the TSP (a preliminary report). Technical Report 95-05, DIMACS, 1995.

[3] A. A. Benczúr. *Cut structures and randomized algorithms in edge-connectivity problems.* PhD thesis, Department of Mathematics, Massachusetts Institute of Technology, June 1997.

[4] A. A. Benczúr and D. R. Karger. Augmenting undirected edge connectivity in $\tilde{O}(n^2)$ time. In *Proceedings of the Ninth Annual ACM/SIAM Symposium on Discrete Algorithms*, pages 500–509, 1998.

[5] R. E. Bixby. The minimum number of edges and vertices in a graph with edge connectivity n and m n-bonds. *Networks*, 5:253–298, 1975.

[6] A. Caprara, M. Fischetti, and A. N. Letchford. On the separation of maximally violated mod-k cuts. Unpublished manuscript, 1997.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press / McGraw-Hill Book Company, 1990.

[8] E. A. Dinits, A. V. Karzanov, and M. V. Lomonosov. On the structure of a family of minimal weighted cuts in a graph. In A. A. Fridman, editor, *Studies in Discrete Optimization*, pages 290–306. Moscow Nauka, 1976. Original article in Russian. Article is not published in English. This manuscript is an English translation obtained from ITC-International Translations Centre, Schuttersveld 2, 2611 WE Delft, The Netherlands. (ITC 85-20220). A translation is also available from NTC-National Translations Center, Library of Congress, Cataloging Distribution Service, Washington DC 20541, USA (NTC 89-20265).

[9] L. Fleischer. Building the chain and cactus representations of all minimum cuts from Hao-Orlin in same asymptotic run time. In R. Bixby, E. A. Boyd, and R. Z. Rios Mercado, editors, *Integer Programming and Combinatorial Optimization*, Lecture Notes in Computer Science. Springer-Verlag, June 1998. Extended Abstract.

[10] L. Fleischer and E. Tardos. Separating maximally violated comb inequalities in planar graphs. In *Integer Programming and Combinatorial Optimization: 5th International IPCO Conference*, number 1084 in Lecture Notes in Computer Science, pages 475–489, 1996.

[11] H. N. Gabow. Applications of a poset representation to edge connectivity and graph rigidity. In *Proc. 32nd Annual Symp. on Found. of Comp. Sci.*, pages 812–821, 1991.

[12] H. N. Gabow. Applications of a poset representation to edge connectivity and graph rigidity. Technical Report CU-CS-545-91, Department of Computer Science, University of Colorado at Boulder, 1991.

[13] A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. *Journal of the ACM*. To appear.

[14] A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. In *38th IEEE Symposium on Foundations of Computer Science*, October 1997.

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

[16] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *J. Soc. Indust. Appl. Math*, 9(4):551–570, 1991.

[17] J. Hao and J. B. Orlin. A faster algorithm for finding the minimum cut in a graph. In *Proc. of 3rd ACM-SIAM Symp. on Discrete Algorithms*, pages 165–174, 1992.

[18] D. R. Karger. Random sampling in cut, flow, and network design problems. In *Proc. of 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 648–657, 1995.

[19] D. R. Karger and C. Stein. A new approach to the minimum cut problem. *Journal of the ACM*, 43(4):601–640, 1996.

[20] A. V. Karzanov and E. A. Timofeev. Efficient algorithms for finding all minimal edge cuts of a nonoriented graph. *Cybernetics*, 22:156–162, 1986. Translated from Kibernetika 2 (1986) pp. 8-12.

[21] D. Naor, D. Gusfield, and C. Martel. A fast algorithm for optimally increasing the edge connectivity. In *Proc. 28th Annual ACM Symposium on Theory of Computing*, pages 698–707, 1990.

[22] D. Naor and V. V. Vazirani. Representing and enumerating edge connectivity cuts in RNC. In *Proc. Second Workshop on Algorithms and Data Structures*, number 519 in Lecture Notes in Computer Science, pages 273–285. Springer-Verlag, 1991.

[23] J.-C. Picard and M. Queyranne. On the structure of all minimum cuts in a network and applications. *Mathematical Programming Study*, 13:8–16, 1980.

[24] J.-C. Picard and M. Queyranne. Selected applications of minimum cuts in networks. *INFOR*, 20(4):394–422, 1982.

[25] A. De Vitis. The cactus representation of all minimum cuts in a weighted graph. Technical Report 454, IASI-CNR, 1997.