# An $\tilde{O}(mn)$ Gomory-Hu Tree Construction Algorithm for Unweighted Graphs

Anand Bhalgat* Ramesh Hariharan† Telikepalli Kavitha‡ Debmalya Panigrahi§

## ABSTRACT

We present a fast algorithm for computing a Gomory-Hu tree or cut tree for an unweighted undirected graph $G = (V, E)$. The expected running time of our algorithm is $\tilde{O}(mc)$ where $|E| = m$ and $c$ is the maximum $u$-$v$ edge connectivity, where $u, v \in V$. When the input graph is also simple (i.e., it has no parallel edges), then the $u$-$v$ edge connectivity for each pair of vertices $u$ and $v$ is at most $n-1$; so the expected running time of our algorithm for simple unweighted graphs is $\tilde{O}(mn)$.

All the algorithms currently known for constructing a Gomory-Hu tree [8, 9] use $n-1$ minimum $s$-$t$ cut (i.e., max flow) subroutines. This in conjunction with the current fastest $\tilde{O}(n^{20/9})$ max flow algorithm due to Karger and Levine[11] yields the current best running time of $\tilde{O}(n^{20/9}n)$ for Gomory-Hu tree construction on simple unweighted graphs with $m$ edges and $n$ vertices. Thus we present the first $\tilde{O}(mn)$ algorithm for constructing a Gomory-Hu tree for simple unweighted graphs. We do not use a max flow subroutine here; we present an efficient tree packing algorithm for computing Steiner edge connectivity and use this algorithm as our main subroutine. The advantage in using a tree packing algorithm for constructing a Gomory-Hu tree is that the work done in computing a minimum Steiner cut for a Steiner set $S \subseteq V$ can be *reused* for computing a minimum Steiner cut for certain Steiner sets $S' \subseteq S$.

**Categories and Subject Descriptors:** F.2.2[Theory of Computation]: Nonnumerical Algorithms and Problems

**General Terms:** Algorithms

*Indian Institute of Science, Bangalore. anand@csa.iisc.ernet.in

†Strand Life Sciences and House of Algorithms, Bangalore. ramesh@strandls.com; work partly done when at IISc.

‡Indian Institute of Science, Bangalore. kavitha@csa.iisc.ernet.in

§Bell Labs Research, Bangalore. pdebmalya@bell-labs.com; work partly done when at IISc.


*STOC'07,* June 11–13, 2007, San Diego, California, USA.


**Keywords:** Steiner edge connectivity, cut trees, Gomory-Hu trees, min cuts

## 1. INTRODUCTION

Let $G = (V, E)$ be an undirected graph with $|V| = n$ and $|E| = m$. The edge connectivity of two vertices $s, t \in V$, denoted by $\lambda(s, t)$, is defined as the size of the smallest cut that separates $s$ and $t$; such a cut is called a minimum $s$-$t$ cut. Gomory-Hu trees, also known as *cut trees*, represent the structure of minimum $s$-$t$ cuts for all pairs of vertices $s$ and $t$ of an undirected graph in a compact way.

Gomory and Hu [8] in a classical result in 1961 showed that the edge connectivities of all pairs of vertices in an undirected graph can be computed using $n-1$ (rather than the naïve $\binom{n}{2}$) max-flow computations. Their algorithm computes a weighted cut tree $\mathcal{T}$, known as the Gomory-Hu tree, on $V$, with the property that the edge connectivity between any two vertices $s$ and $t$ in the graph exactly equals the weight of the minimum weight edge on the unique $s$-$t$ path in $\mathcal{T}$. Further, the partition of the vertices produced by removing this edge from $\mathcal{T}$ is a minimum $s$-$t$ cut in the graph. Gomory-Hu trees have many applications in multi-terminal network flows. Examples were shown by Benczúr[2] that cut trees do not exist for directed graphs.

All known algorithms for building Gomory-Hu trees in undirected graphs use a minimum $s$-$t$ cut subroutine (which is the same as an $s$-$t$ max flow algorithm). Gomory and Hu showed how to compute the cut tree $\mathcal{T}$ using $n-1$ max flow computations and graph contractions. Gusfield proposed an algorithm that does not use graph contractions; all $n-1$ max flow computations are performed on the input graph. Goldberg and Tsioutsiouliklis [7] did an experimental study of the Gomory-Hu and Gusfield's algorithms for the cut tree problem and described efficient implementations of these algorithms.

Any max-flow based approach for constructing a Gomory-Hu tree would have a running time of $(n-1)\cdot$(time for computing a max flow). Till now faster algorithms for Gomory-Hu trees were by-products of faster algorithms for computing a max-flow [6]. The current fastest $\tilde{O}(n^{20/9})$ max flow algorithm due to Karger and Levine [11] yields the current best running time of $\tilde{O}(n^{20/9}n)$ for Gomory-Hu tree construction on simple unweighted graphs with $m$ edges and $n$ vertices.

In this paper we consider the problem of designing a faster algorithm for constructing a Gomory-Hu tree; thus our algorithm cannot use $n-1$ max flow subroutines in the input graph. We show the following theorem here.

THEOREM 1. *Let $G = (V, E)$ be a simple unweighted graph with $m$ edges and $n$ vertices. Then a Gomory-Hu tree for $G$ can be built in expected time $\tilde{O}(mn)$.*

Thus our algorithm is always faster by a factor of $\Omega(n^{2/9})$ (ignoring polylog $n$ factors) compared to the previous best algorithm. Our Gomory-Hu tree algorithm achieves a running time of $\tilde{O}(mn)$ by using an efficient construction of tree packing. Gabow [5] gave an efficient construction of spanning tree packing (actually arborescence packing but with relaxed directions, henceforth called *directionless trees*) that yielded an $\tilde{O}(mk)$ algorithm to determine the global edge connectivity, $k = \min_{u,v \in V} \lambda(u, v)$, of a directed graph with $m$ edges. This approach was generalized by Cole and Hariharan [3] for determining the Steiner edge connectivity of an undirected graph or an Eulerian directed graph. For any Steiner set $S \subseteq V$, their algorithm runs in time $\tilde{O}(mk^2)$ where $k = \min_{u,v \in S} \lambda(u, v)$. For undirected graphs, the running time of their algorithm improves to $\tilde{O}(m + nk^3)$. The algorithm in [3] was used by Hariharan et al. in [10] to design an algorithm with expected running time $\tilde{O}(m + nk^3)$ to compute a *partial* Gomory-Hu tree for representing the $\lambda(u, v)$ values for all pairs of vertices $u, v$ that satisfied $\lambda(u, v) \leq k$.

### *Our Techniques.*

Our algorithm extends the method used in [10] that computes all *small* $\lambda(u, v)$ values. The main idea in [10], to compute all $\lambda(u, v)$ values that are at most $k$, is to use the $\tilde{O}(m+nk^3)$ Steiner edge connectivity algorithm of Cole and Hariharan [3] as the main subroutine instead of a max-flow algorithm. Recall that this Steiner edge connectivity algorithm is based on an efficient construction of tree packing. The algorithm in [10] constructs these trees by choosing the root vertex of these trees uniformly at random from the set $S$ of vertices, whose $\min_{u,v \in S} \lambda(u, v)$ value is to be determined. It was shown that in expected time $\tilde{O}(m + nk^3)$, all the trees required to estimate the $\lambda(u, v)$ values which are at most $k$ can be built. However this method does not yield interesting results for large $k$, for instance when $k = \Theta(n)$, it would take $\tilde{\Theta}(n^4)$ time to compute all $u$-$v$ edge connectivities which are at most $k$; it is instead more efficient to construct a Gomory-Hu tree using $n - 1$ max flows.

We obtain our efficient algorithm for constructing a Gomory-Hu tree as follows:

- We present an improved tree packing algorithm for the Steiner edge connectivity problem[1]. For a Steiner set $S \subseteq V$, our algorithm determines the $\min_{u,v \in S} \lambda(u, v)$ value and a vertex partition corresponding to this cut in time $\tilde{O}(m \cdot (\min_{u,v \in S} \lambda(u, v)))$.

- It follows from the analysis in [10] that the above subroutine for computing Steiner edge connectivity yields a Gomory-Hu tree algorithm with expected running time $\tilde{O}(m + nc^2)$, where $c = \max_{u,v \in V} \lambda(u, v)$. We give an improved analysis here and show that the expected running time of our algorithm in unweighted graphs is $\tilde{O}(mc)$. [Note that $m$ can be much smaller than $nc$.]

- This implies an $\tilde{O}(mn)$ algorithm for constructing a Gomory-Hu tree when the input graph is simple and unweighted, since $\max_{u,v \in V} \lambda(u, v) \leq n - 1$ in such graphs.

It follows from [10] that other corollaries of our efficient Steiner edge connectivity algorithm are (i) an algorithm with expected time $\tilde{O}(m+nk^2)$ for building a partial Gomory-Hu tree[2] corresponding to parameter $k$; (ii) an algorithm with expected time $\tilde{O}(m+nk^2)$ algorithm that splits a given $T \subseteq V$ of even cardinality into two odd cardinality components such that the size $k$ of the resulting cut is minimized.

## 1.1 Preliminaries

We review the Gomory-Hu tree construction algorithm from [8] now. The algorithm to construct a Gomory-Hu tree uses submodularity of cuts stated below as Fact 1.

FACT 1 (SUBMODULARITY OF CUTS). *If $A$ and $B$ are two subsets of vertices in $G$ and $\delta(X)$ represents the size of the cut $(X, V \setminus X)$, then $\delta(A)+\delta(B) \geq \delta(A \cap B)+\delta(A \cup B)$.*

Fact 1 leads to the following theorem, which is used in the Gomory-Hu tree construction algorithm.

THEOREM 2. *If $(S, V \setminus S)$ is a minimum $s$-$t$ cut in $G$ and $u, v$ are any pair of vertices in $S$, then there exists a minimum $u$-$v$ cut $(S^*, V \setminus S^*)$ such that $S^* \subset S$.*

The Gomory-Hu tree construction algorithm [8] initializes the cut tree $\mathcal{T}$ to a single node that contains the entire vertex set. At any step of the algorithm, it picks a node $S$ of $\mathcal{T}$ containing more than one vertex and chooses any two vertices $s$ and $t$ in $S$. The entire subtree subtended at each neighbor of $S$ is contracted into a single node and a max flow computation is performed from $s$ to $t$ in the new graph. Theorem 2 ensures that the minimum $s$-$t$ cut thus obtained (we call it $C$) is also a minimum $s$-$t$ cut in the original graph. Now, in $\mathcal{T}$, the node $S$ is split into $S_1$ and $S_2$ according to $C$ and the two nodes thus formed are joined by an edge of weight equal to the size of $C$. Further, all the neighboring subtrees of $S$ become neighboring subtrees of $S_1$ or $S_2$ depending upon which side of $C$ they lie on. The algorithm terminates when all the nodes of $\mathcal{T}$ become singleton sets. Thus $\mathcal{T}$ is a weighted tree whose nodes are the vertices of $V$. It can be shown that $\mathcal{T}$ captures all-pairs min-cuts.

### 1.1.1 An alternative implementation

In the above algorithm, an alternative to computing a minimum $s$-$t$ cut for some pair $s, t \in S$ is to compute a *minimum Steiner cut* for the Steiner set $S$ in the new graph. Given a subset $S \subseteq V$, called the Steiner set, the *Steiner edge connectivity* is the size of the smallest cut $C$ that splits $S$ into two non-empty components; such a cut is called a minimum Steiner cut. Thus a minimum Steiner cut is also a minimum $u$-$v$ cut for all those pairs $u, v \in S$ that lie on different sides of the cut $C$. The starting point of our Gomory-Hu tree construction algorithm is Algorithm 1.1.

[1] we would like to point out that *any* fast Steiner edge connectivity algorithm does not suffice here; our main algorithm needs an efficient subroutine for the problem: given a graph $G'$ and a vertex $r$ in $G'$, compute all min $r$-$x$ cuts for vertices $x$ in $G'$.

[2] this is basically a *contracted* Gomory-Hu tree where all edges with weight more than $k$ are contracted

**Algorithm 1.1** A simple algorithm for constructing a Gomory-Hu tree for the graph $G = (V, E)$

– Initialize the tree $\mathcal{T}$ to a single node, which is the entire vertex set $V$.
– Initialize the queue $Q$ to the queue containing only one element, which is the set $V$.
**while** the queue $Q$ is not empty **do**
– delete the first element of $Q$; call this element $S$.
– call the *minimum Steiner cut algorithm* with the set $S$ as the Steiner set in the new graph obtained by contracting the entire subtree rooted at each neighbor in $\mathcal{T}$ of $S$ into a single node.
– let $S_1$ and $S_2$ be the two components that $S$ is split into by the above cut and let $c$ be the size of this cut; update $\mathcal{T}$ by splitting the node $S$ into nodes $S_1$ and $S_2$ and introduce an edge with weight $c$ between $S_1$ and $S_2$.
– set the neighbors of $S_1$ and $S_2$ in the tree $\mathcal{T}$ appropriately.
– insert the node $S_1$ (similarly, $S_2$) in the queue $Q$ if $S_1$ (resp., $S_2$) contains more than one vertex.
**end while**

It is easy to see that Algorithm 1.1 is just an alternative implementation of the algorithm by Gomory and Hu. The running time of Algorithm 1.1 is $O(n\cdot$(time to compute a min Steiner cut)). We will present in the next section a tree packing algorithm with running time $\tilde{O}(mc)$ to compute a minimum Steiner cut in an undirected graph $G$, where $c$ is the value of this cut. This, however, would only result in an algorithm with running time $\tilde{O}(n \cdot (mc))$ for constructing a Gomory-Hu tree.

As a first attempt to get a faster algorithm, let us consider the following special case. Suppose each minimum Steiner cut that Algorithm 1.1 computes, splits the the Steiner set $S$ into two subsets $S_1, S_2$ of cardinality $|S|/2$ each (let us assume that $n$ is a power of 2). Then the depth of the computation tree[3] is $\log n$. We can show the following claim.

Claim 1. *If the computation tree has depth* $\log n$*, then the running time of Algorithm 1.1 is* $\tilde{O}(mc)$*, where* $c = \max_{u,v\in V} \lambda(u, v)$.

We will show how to extend Claim 1 in Section 3. We will first present our efficient tree packing algorithm for computing a minimum Steiner cut in Section 2.

## 2. THE IMPROVED MINIMUM STEINER CUT ALGORITHM

In this section we first give an overview of Gabow's algorithm [5] for directionless tree packing to find the global minimum cut and then review the algorithm by Cole and Hariharan [3] (henceforth called the CH algorithm) to compute a minimum Steiner cut for a given Steiner set $S \subseteq V$. Then we present our improved minimum Steiner cut algorithm.

[3]Each node of the computation tree corresponds to a minimum Steiner cut subroutine - the two children of the node corresponding to the subroutine for Steiner set $S$ are the subroutines for Steiner sets $S_1$ and $S_2$ that $S$ splits into, by the minimum Steiner cut for $S$ that Algorithm 1.1 computes.

First, the given undirected graph is made directed by orienting the edges in both directions. We will use the following terms here.

(i) an $r$-cut is a directed cut where $r$ is on the source side of the cut.

(ii) a *directionless r-spanning tree* is a spanning tree of directed edges rooted at a specified root vertex $r$ such that $r$ has an in-degree of 0 in the tree.

(iii) $con(v)$ denotes the edge connectivity of $v$ from the root vertex $r$.

Gabow's algorithm is based on following result due to Edmonds.

Theorem 3 (Edmond's relaxed theorem [4]). *The minimum value of an r-cut* $c_{min}$ *is exactly equal to the maximum number of edge-disjoint directionless r-spanning trees such that each vertex* $v \neq r$ *has in-degree* $c_{min}$ *over all the trees.*

The directionless $r$-spanning trees are constructed one at a time. Given the first $i-1$ trees $T_1, \ldots, T_{i-1}$, the $i$th tree $T_i$ is constructed in several rounds. $T_i$ is built from a forest (let us name this forest $T_i$ as well) initially comprising $n$ singleton vertices. Each distinct tree in this forest is called a component. Each round in the construction process runs in $O(m+n)$ time and reduces the number of components in the $i$th forest $T_i$ by at least half, leading to a time complexity of $O((m+n)\log n)$ per tree, and $O(c_{min}(m+n)\log n)$ overall.

Any particular round begins with several connected components, each of which has exactly one *deficient* vertex, i.e., a vertex whose total in-degree in $T_1, \ldots, T_i$ is $i-1$ (all other vertices have in-degree $i$ in $T_1, \ldots, T_i$). Each of these connected components gets processed in this round. Consider one such component $C$ with a deficient vertex $v$. Gabow's algorithm now computes the minimum set $M$ containing edges satisfying at least one of the following properties:
(1) $e \notin T_1, \ldots, T_i$ and is directed into $v$.
(2) $e \in T_j$ for $1 \leq j \leq i$ and is in the fundamental cycle formed by adding some edge $f \in M$ to $T_j$.
(3) $e \notin T_1, \ldots, T_i$ and is directed into a vertex into which some other edge in $M$ is also directed.

Computing $M$ needs a closure-like algorithm, and Gabow shows how to perform this in time proportional to the number of edges and vertices involved in $M$. Gabow shows that this closure like algorithm has one of two possible outcomes: (i) there exists an edge $e \in M$ which connects a vertex in $T_i$ outside $C$ to a vertex inside $C$, or (ii) no such edge exists in $M$. In Case (i), there exists a sequence of swap adjustments to $T_1, \ldots, T_i$ culminating in the addition of $e$ to $T_i$, which ensures that $v$ is no longer deficient; further, no new deficient vertices are created in the process and the number of connected components is reduced by 1. In Case (ii), Gabow shows that the set $S$ of vertices into which edges of $M$ are directed occur contiguously in $T_1, \ldots, T_i$, and $(V \setminus S, S)$ forms an $r$-cut of size $i-1$.

### *The Cole-Hariharan Algorithm.*

The CH algorithm [3] for minimum Steiner cut is based on a relaxed extended tree packing theorem from [1][4].

[4]Actually a slightly stricter version of the theorem appears in the cited reference.

THEOREM 4. *Given an Eulerian directed graph $G$ and any vertex $r$, there exist $k$ edge disjoint directionless trees $T_1, T_2, \ldots, T_k$ rooted at $r$ such that each vertex $v \neq r$ in $G$ appears exactly $con(v)$ times over all the trees and has in-degree exactly $con(v)$ over these trees, where $con(v)$ is the edge connectivity of $r$ to $v$ and $k = \max_{v \neq r}\{con(v)\}$.*

As in Gabow's algorithm, the CH algorithm builds directionless trees one at a time. Any Steiner vertex is chosen as the root $r$. Given the first $i-1$ trees with the property that each vertex $v$ occurs in exactly $\min\{con(v), i-1\}$ trees and has in-degree exactly equal to its number of occurrences in these trees, the $i$th tree is constructed in several rounds. Each round in the algorithm, decreases the number of components in the last tree by at least a constant factor. This is done by finding a series of adjustments to the trees so that all the properties of the trees (number of occurrences of a vertex and its in-degree) remain valid but components in the last tree merge to reduce their total number by at least a constant factor. However, there are additional complications here. Gabow's algorithm can stop the moment it finds a small $r$-cut (of size $i-1$, say). However the CH algorithm must continue in this situation; before doing so, it shrinks this entire $r$-cut into a *black supervertex*, adjusts edges in the trees incident on this black supervertex so the degree of each occurrence of the black supervertex is at most 2, and then proceeds further. This leads to a hierarchy of white vertices and black supervertices as the algorithm proceeds.

### *White vertices and Black supervertices.*

Let us suppose that we have constructed $i-1$ trees and are in the process of constructing the $i$th tree. Now all the vertices in the graph, except the root, are arranged in several layers. At the topmost layer, a vertex is either a singleton *white vertex* or is a part of a collection of vertices called a *black supervertex*. A black supervertex $b$ is a subset of vertices not containing the root $r$ such that its in-degree in the graph is at most $j \leq i-1$ and it has been discovered as a small cut during the construction of the $(j+1)$th tree. All the vertices of $b$ appear contiguously in all the trees constructed so far and further, from this stage onwards, vertices in $b$ are contiguous in all the trees they appear in. Each black supervertex $b$ appears exactly $con(b)$ many times over all the trees, possibly occurring multiple times in a single tree.

All vertices that do not belong to any black supervertex are white vertices. They can either be vertices with edge connectivity of at least $i$ from $r$, or vertices whose edge connectivity is $i-1$ from $r$ but no cut of size $i-1$ separating it from the root $r$ has been discovered yet. All white vertices which are in the second category will eventually be placed inside a black supervertex (which will be an edge connectivity cut separating it from $r$) by the end of the construction of the $i$th tree. White vertices occur $i$ times in the trees, exactly once per tree, and have an in-degree of either $i$ or $i-1$ depending on whether it is a deficient vertex or not.

Black supervertices internally have a hierarchical structure. So a black supervertex with in-degree $j \leq i-1$ is composed of a nonempty set of singleton vertices which have a connectivity of $j$ from $r$ and other black supervertices which have an edge connectivity $\leq j$. We overload our nomenclature and call singleton vertices which have an edge connectivity of $j$ from $r$ as white vertices at this particular level of nesting. The black supervertices, on the other hand, have a further hierarchy of white vertices and black vertices internally. The important properties here are:

- if white vertex $v$ appears immediately inside a black supervertex $b$, then $v$ appears once immediately inside each occurrence of $b$; the total in-degree of $v$ is the edge connectivity of $b$.
- if a black supervertex $b'$ with edge connectivity $j'$ appears immediately inside a black supervertex $b$, then $b'$ appears a total of $j'$ times immediately inside the various occurrences of $b$.
- when a set of vertices (and supervertices) is demarcated as a new black supervertex, all these vertices are contiguous in all the trees. Thus, a black supervertex always remains contiguous is all the trees.
- the following invariant is maintained: *a black supervertex has a degree of at most 2 in each occurrence.* (As we shall see, this property is actually maintained only for a special set of black supervertices known as the maximal black supervertices in the CH algorithm.)

The last property enables the algorithm to view the trees as trees on white vertices with paths (not necessarily properly oriented) of black supervertices connecting the white vertices.

### *The CH Algorithm for minimum Steiner cut.*

Each round in constructing the tree $T_i$ can be split into the following steps.

1. In each component of $T_i$ where there exist vertices with edge connectivity $i-1$ from $r$, find a *white maximal* set of vertices such that it represents an $r$-cut of size $i-1$ . Call such a set of vertices a *maximal black supervertex.*
2. If a maximal black supervertex contains a Steiner vertex, then terminate the algorithm and output $i-1$ as the Steiner edge connectivity; otherwise, balance the degrees of all occurrences of maximal black supervertices such that no occurrence has degree greater than 2.
3. Run a closure process on each component to find a *releasable or joining path*[5] joining this component to some other component.
4. Perform a sequence of transformations to the trees such that the releasable paths from the last step are released. These paths are added to the tree $T_i$ to reduce the number of components by at least a constant factor.

The CH algorithm first determines black supervertices that are white maximal in their components and runs the closure with such a vertex because of the following reason: since $b$ is *white maximal*, there is no other black supervertex $b'$ in $b$'s component such that $b \subset b'$ and there is some white vertex $w \in b' \setminus b$. It is clear that if such a black supervertex $b$ can be identified, then the closure process of the CH

[5]A path begins and ends at white vertices and goes via only black supervertices; a path is *releasable* if the trees can be modified such that the path is freed without violating the properties of the trees.

algorithm cannot get stuck in an $r$-cut of size $i-1$ and will eventually find a joining path. This is because the closure process will start with a seed path containing a white vertex in $b$'s component that is outside $b$ and there is no cut of size $i-1$ containing this white vertex along with $b$ which is limited to $b$'s component alone.

Cole and Hariharan showed that all the above steps can be performed in $\tilde{O}(m)$ time for each round, except the first step, which takes $\tilde{O}(m+i^2n)$ time. We use Steps 2-4 of their algorithm and give an implementation of Step 1 with a running time of $\tilde{O}(m)$ to achieve an overall complexity of $\tilde{O}(mk)$ for constructing the $k$ trees, where $k = \min_{u,v \in S} \lambda(u,v)$. Let us first review Steps 2-4 of the CH algorithm.

### *Step 2: Balancing the degree of black supervertices.*

This step is also called the *global degree balancing step*. For each maximal black supervertex $b$ discovered in this round, we need to make the degree of each occurrence of $b$ in the trees $T_1, \ldots, T_i$ to be at most 2.

The idea here is the following: let the black supervertex $b$ occur with degree $d > 2$ in tree $T_j$. Since $b$ has a total degree of $2(i-1)$ in the graph, there must exist at least $d-2$ leaf occurrences of $b$ in the trees $T_1, \ldots, T_{i-1}$. Let $b'$ be one such leaf occurrence. The path from $b'$ to its nearest white vertex $w$ is traversed and cut at $w$; then the path $b'$-$w$ is attached to the instance of $w$ in $T_j$ and finally for a neighbor $x$ of $b$, the edge $x$-$b$ is replaced by $x$-$b'$, where $x$ is chosen so that $w$ is not in the subtree rooted at $x$ (treating $b$ as a root). This step can be performed efficiently for all the maximal black supervertices discovered in this round, by traversing each tree $T_1, \ldots, T_{i-1}$ in a bottom-up manner. This finishes the degree balancing step in $T_1, ..., T_{i-1}$.

Balancing the degree in $T_i$ is done separately, since we use this step to generate a *seed path* which will be used to start the closure process in Step 3. We go through all the components of $T_i$ that contain an $(i-1)$-sized cut (or maximal black supervertex) discovered in the previous step, in a sequential order and process them. This process could change a component and its vertices could merge with other components in $T_i$. So when we process a component $C$, we refer to the *dynamic* $C$ which includes all the new vertices that have merged with $C$ due to processing the earlier components in our order. We describe below how to process a component $C$ with a maximal black supervertex $b$.

A special case is when the black supervertex $b$ is an entire component $C$. Then we do not need to do anything since there is no white vertex left in $C$ and so none of the vertices in $C$ shall occur in the final version of the tree $T_i$; we just discard $C$ in this case. So let us assume that the degree of $b$ is $d \geq 1$. Since the total degree of $b$ in $T_1, \ldots, T_i$ is $2(i-1)$, it is easy to see that there are $d$ leaf occurrences of $b$ in the trees $T_1, \ldots, T_{i-1}$.

- if $d = 1$, then there is a leaf occurrence of $b$ in some tree $T_j$, $1 \leq j \leq i-1$. Form the path $x$-$b$-$y$ by detaching the $x$-$b$ path from $C$ (where $x$ is the nearest white vertex to the occurrence of $b$ in $T_i$) and detaching the $y$-$b$ path from $T_j$ (where $y$ is the nearest white vertex to the occurrence of $b$ in $T_j$) and joining $x$-$b$ with $b$-$y$. This $x$-$b$-$y$ path is used as a seed path for Step 3.

- for a general $d \geq 2$ we do the following: let $T_{\ell_1}, \ldots, T_{\ell_d}$ be the $d$ trees in $T_1, \ldots, T_{i-1}$ where $b$ occurs as a leaf. Detach the $d$ paths $b$-$x_\ell$ from these $d$ leaf occurrences of $b$, where $x_\ell$ is the nearest white vertex to $b$ in the tree $T_\ell$. Attach these $b$-$x_\ell$ paths to $b$ in $C$, creating an occurrence of $b$ with degree $2d$. There are two cases here:

  – None of the vertices $x_\ell$ belongs to $C$. In this case, the component $C$ is fragmented into subtrees rooted at the $d$ children of $b$, and each of these subtrees uses the edge linking it to $b$ to pair itself to one of the vertices $x_\ell$, which belongs to another component. Thus we create $d$ occurrences of $b$ with degree 2 and the component $C$ is no longer present in $T_i$. So there is no seed path generated for the component $C$ here and the number of components in $T_i$ decreases by 1.

  – The other case is that there is at least one $x_\ell$ which belongs to $C$. Then we can pair $2d-2$ of the $2d$ edges incident to $b$ so that this pairing creates $d-1$ new occurrences of $b$ with degree 2 and leaves two edges $b$-$x_\ell$ and say, $b$-$y$ unused, and the number of connected components in $T_i$ does not decrease (though some of the subtrees rooted at children of $b$ in $C$ might now be linked through $b$ to other components). We use the two paths $b$-$x_\ell$ and $b$-$y$ (as was done in the previous case of $d = 1$) to form the seed path $x_\ell$-$b$-$y$ for Step 3.

### *Step 3: The Closure Process.*

The closure process aims to find an augmenting path for each component in a round. Each closure process starts with the seed path generated in Step 2. This seed path is added to the closure set $\mathcal{C}$. The trees are scanned in a round-robin fashion and each new path $p$ added to $\mathcal{C}$ from tree $T_j$ is applied to the next tree $T_{j+1}$. The paths that form a fundamental cycle in $T_{j+1}$ due to the addition of $p$ are now added to the closure set $\mathcal{C}$.

The main difference in the closure process here and the closure process in Gabow's algorithm is the following: *in addition to the white-to-white paths, one also adds some black-to-white paths.* In particular, if $w_1$-$w_2$ is a white-to-white path to be added to $\mathcal{C}$ where $w_1$ is closer to the closure set, then we also add the paths from all the intermediate black vertices to $w_1$. Clearly, all paths in $\mathcal{C}$ have at least one white vertex as a terminal vertex. The other terminal vertex could be white or black. (If black, it is a supervertex). Additionally, there can be one or more black vertices on the path, though no white vertex can be present internal to the path.

The set of paths added to $\mathcal{C}$ from tree $T_j$ is applied to tree $T_{j+1}$ in an order such that one of the terminal white vertices (if there is only one terminal vertex, then that vertex itself) is always in the closure. In case the terminal black supervertex of some path is not present in tree $T_{j+1}$, then we reserve the path for applying to the next tree which contains the black supervertex. If a tree contains multiple copies of a black supervertex which is also a terminal of a path, then the path produces multiple fundamental cycles - all paths on these cycles are included in $\mathcal{C}$. Also if a path directed into a white vertex $v$ is included in $\mathcal{C}$, then all the unused paths directed into $v$ are also included in $\mathcal{C}$. If the other terminal (other than $v$) of such a path is a black supervertex $b$, then we detach a leaf occurrence of $b$ along with the path to its nearest white vertex $w$ (say from tree $T_j$) and consider the path $w$-$b$-$v$ formed by attaching the path $w$-$b$ to the path $b$-$v$ as a single unused white-to-white path. Now, all unused white-to-white paths directed into $v$ (and all their

corresponding black-to-white sub-paths moving inside out from the closure) are also included in $\mathcal{C}$. This entire process is known as *cyclic scanning*. The cyclic scanning procedure is terminated when a path which connects two components enters $\mathcal{C}$.

### *Step 4: Modifying the trees and releasing edges.*

In this step, we find a sequence of adjustments to be made to the trees $T_1, \ldots, T_i$ (without affecting their properties) so that the joining edges, determined in the last step, are released. These edges are added to the last forest $T_i$ to reduce its number of components. This procedure is performed simultaneously for all the components in $T_i$ and the number of components in $T_i$ reduces by at least a constant factor.

Note that the closure process does not provide an explicit sequence of what rule to apply in order to release a releasable edge. Instead, it provides a whole computation tree of all several possible sequences from which the desired sequence of rules can be extracted. There is one complication though. The computation tree is constructed on the static trees $T_1, \ldots, T_i$, i.e., no operations/changes on the trees are actually performed, rather they are simulated through appropriate data structures. But clearly the trees change as operations in the sequence are actually applied, and one could potentially get, for instance, a situation in which edge $e$ appears in the fundamental cycle of $f$ in the static trees but does not do so after previous operations have been applied. This problem is handled by showing that all changes made to edges before applying a particular operation are contained within certain contiguous sets $C_\ell$ of white vertices defined just prior to the operation and the portions outside are unaffected.

There are further problems that arise as a result of adding black-to-white paths to our closure. First, edges which have one endpoint outside and one endpoint inside the $C_\ell$'s could be rewired in the process so that their portions outside the $C_\ell$'s stay the same (by an edge portion, we mean the sequence of black internal vertices and one white endpoint) but the portion inside may change while remaining inside the $C_\ell$'s. Formally, we could work with *substitutes* of edges, which differ from the original edges only in the portion inside $C_\ell$'s and every operation goes through because the crux of the operation will be outside the $C_\ell$'s, all changes made so far will be inside, and the portions scanned to make the current change will enter $C_\ell$ immediately. Second, we need to determine which of $w_1$ or $w_2$ is closer to the closure set when a black to white path is applied with the black lying on the $w_1, w_2$ tree edge. This *nearness problem* can be solved easily on the static trees $T_1, \ldots, T_i$, but closure processes running on different components could conspire to make the actual dynamic answer different from the static one. However, it can be shown that such a swap will be the very last operation in its closure process, because it provides a joining edge and such a joining edge can be determined efficiently.

## 2.1 A faster algorithm for finding black supervertices

The CH algorithm uses a separate subroutine to initially demarcate all the maximal black supervertices. This subroutine does away with the possibility of the closure process getting stuck midway because of discovering an $r$-cut of size $i-1$. However the CH process is expensive and takes $\tilde{O}(m + i^2 n)$ time; we show a faster $\tilde{O}(m)$ procedure.

LEMMA 1. *If the closure process ends unsuccessfully (without finding the swap sequence) for a component during construction of $i$th tree, then the vertices spanned by the edges in the closure form an $i-1$ cut in the graph. This cut is a minimal $r$-$u$ cut for any white vertex $u$ present inside the closure where $r$ is the root of the tree.*

The new idea here is that we balance the degrees of all occurrences of $(i-1)$-sized cuts as and when we find them and the closure process treats it as a black supervertex and continues, till we discover that we have an $(i-1)$-sized cut that is white maximal in its component.

Our overall scheme is the following: as in the CH algorithm, we build directionless spanning trees one at a time, maintaining the in-degree requirement and the number of occurrences of each vertex in the trees. Given the first $i-1$ trees, we construct the $i$th tree $T_i$ in several rounds, where each round decreases the number of components in $T_i$ by at least a constant factor. Each round processes each component in $T_i$ by running the closure process starting with an unused path incident on the deficient vertex of the component. (The initialization is similar to Gabow's algorithm for finding the size of a minimum cut in a graph.) During this process, suppose we find a small cut (precisely, an $(i-1)$-sized cut which we call a black supervertex $b$).

- LOCAL DEGREE BALANCING: We degree balance all occurrences of $b$ in the trees $T_1, \ldots, T_i$. During the process of local degree balancing, we could realize that $b$ is actually a maximal black supervertex. Then we stop our local degree balancing step and go to the step "*Discovered a maximal black*". Otherwise (either $b$ is not maximal or it is maximal but we have not yet realized it), we go to the closure step.

- CLOSURE: The closure process is mostly similar to Step 3 of the CH algorithm and it builds the closure set of vertices, that is contiguous in all the trees $T_1, \ldots, T_i$, and contains $b$. If in this process, we find a vertex (either black or white) that is external to $b$'s component entering the closure set, then we realize that $b$ is a maximal black supervertex and stop the closure and go to the next step. Otherwise, we will discover an $(i-1)$-sized cut that contains $b$. Call this larger cut $b$ now and go back to the local degree balancing step.

- DISCOVERED A MAXIMAL BLACK: If we had discovered that $b$ is a maximal black supervertex while running the local degree balancing step, then we come to this step to run the *global degree balancing* step (Step 2 of the CH algorithm). If we had discovered that $b$ is a maximal black supervertex during the closure step, then all occurrences of $b$ in the trees $T_1, \ldots, T_i$ already have their degrees balanced, so we have a seed path (due to 2 leaf occurrences of $b$ in $T_1, \ldots, T_i$) and are ready to run Steps 3 and 4 of the CH algorithm.

- STEPS 3 AND 4 OF THE CH ALGORITHM: Now we are in the same stage as the end of Step 2 of CH algorithm - maximal black supervertices in components of $T_i$ that have $(i-1)$-sized cuts have been identified and we have balanced their degrees in $T_1, \ldots, T_i$. We run Steps 3 and 4 of the CH algorithm now and decrease the number of components in the last tree, by a constant factor.

*Remark.* Note that the difference between the local and global degree balancing steps is that the work done for local degree balancing a black supervertex $b$ is limited to $b$'s component itself, and therefore the the work done for different components does not overlap. However, this is not true when we try to balance the degrees of all occurrences of the maximal black supervertices, and hence we need a global procedure for degree balancing such vertices.

Now we elaborate on our steps.

### *Local Degree Balancing.*

We run this step when we encounter a black supervertex $b$ while performing the closure in a component $C$. If each occurrence of $b$ has a degree of at most 2 in the trees $T_1, \ldots, T_i$, then we do not need to do anything. However, if there is some occurrence of $b$ with degree $d > 2$ (say in tree $T_j$), there must be at least $d-2$ leaf occurrences of $b$ in the trees. Let $w_1, \ldots, w_{d-2}$ be the white vertices closest to these occurrences of $b$. We claim that any black supervertex $b' \supset b$, contained entirely in $b$'s component and containing at least one white vertex, must also contain each of $w_1, \ldots, w_{d-2}$. To prove this claim, we need Lemma 2.

LEMMA 2. *Consider any set of directionless spanning trees $T_1, \ldots, T_i$ ($T_i$ is a forest). Let these trees satisfy the degree restriction that each vertex $v$ in some black supervertex (except the root $r$) has exactly $con(v)$ occurrences and also has total in-degree of $con(v)$ in these trees, while all white vertices have exactly $i$ occurrences and have an in-degree of either $i$ or $i-1$. (A white vertex having an in-degree of $i-1$ is said to be deficient.) We claim that any set of vertices not containing $r$, containing at most one deficient white vertex and containing at least one white vertex, which forms an $r$-cut of size $i-1$ in the graph (call this set $X$) must be contiguous in all the trees.*

PROOF. The first observation is that there is at least one component of $X$ in each tree, since this set contains a white vertex. Thus $X$ has at least $i$ components. Now, if $X$ is not contiguous in at least one tree, then it has at least $i+1$ components. Let $W$ be the set of white vertices in $X$ and let $|W| = w$. Then, the total number of edges in the trees which are completely internal to $X$ (both terminal vertices in $X$) is at most $I(X) = (w \cdot i + \sum_{v \in X \setminus W} con(v)) - (i+1)$, since each $v \in W$ occurs exactly $i$ times while each $v \in X \setminus W$ occurs exactly $con(v)$ times. Due to the in-degree restrictions, the total number of edges directed into vertices in $X$ is at least $T(X) = (w-1) \cdot i + (i-1) + \sum_{v \in X \setminus W} con(v)$. Thus the number of external edges directed into vertices in $X$ is at least $T(X) - I(X) = i$. This contradicts the fact that $X$ is an $r$-cut of size $i-1$ in the graph. □

Now, we can see that any black supervertex $b' \supseteq b$, contained entirely in $b$'s component and containing at least one white vertex, must also contain $w_\ell$, where $w_\ell$ is the nearest white vertex to a leaf occurrence of $b$ in some tree $T_\ell$. This follows directly from the fact that if $w_\ell \notin b'$, then $b'$ is non-contiguous in $T_\ell$ and this immediately violates Lemma 2.

So if there is some $w_\ell \in \{w_1, \ldots, w_{d-2}\}$ missing in $b$'s component, then we know that $b$ is a maximal black supervertex in its component and so we do not perform any local degree balancing. On the other hand, if each $w_\ell$ is in $b$'s component, then we start walking up the tree $T_j$ from $w_\ell$ for each $\ell$ and simultaneously start walking up from $b$. During any of these walks, if we encounter a vertex outside the current component, then we stop that walk. We can have the following situations for any particular $w_\ell$.

1. The walk from $w_\ell$ hits some vertex in $b$. Then we have found the $b$-$w_\ell$ path in tree $T_j$. We associate $w_\ell$ with the last edge on the walk. (This is an edge incident on $b$ in $T_j$.)
2. The walk from $b$ hits $w_\ell$. Again, we have found the $b$-$w_\ell$ path in tree $T_j$ and we associate $w_\ell$ with the first edge on the walk. (This is again an edge incident on $b$ in $T_j$.)
3. Both walks hits a common vertex. Again, we have found the $b$-$w_\ell$ path in tree $T_j$ and we associate $w_\ell$ with the first edge on the walk from $b$. (This is again an edge incident on $b$ in $T_j$.)
4. Walks from $b$ and $w_\ell$ terminate on vertices in other components before any of the above conditions is met. Then $b$ is a maximal supervertex and we terminate this step immediately and go to the "*Discovered a maximal black*" step.

In these walks, to ensure that we spend time proportional to the size of the component in a round, we do not retrace any edge; all edges traced already are marked. In our walk up from $b$ and each of the $w_\ell$'s, we must have found paths between $b$ and $w_\ell$ in tree $T_j$; otherwise, we would not perform local degree balancing. Now, let $e$ be a path of black supervertices which is incident on $b$ but is not on this path between $b$ and some $w_\ell$. We can detach the subtree subtended at $e$ and connect it to $w_\ell$ through $b$ in $T_j$. For this purpose, we need to stitch the paths $b$-$w_\ell$ (from the leaf occurrence of $b$) and $e$ to form a single path of black vertices. Clearly, the degree of $b$ in the occurrence in $T_j$ will be decreased by 1. We perform this readjustment for the $d-2$ edges incident on $b$ in $T_j$; this can be done in time linear in the number of edges traversed.

### *Closure.*

The closure step performed here is mostly the same as the cyclic scanning process (Step 3 of CH algorithm) used to find maximal black supervertices. There is however one difference, which is due to the fact that unlike the closure step in CH algorithm, here the trees get modified due to local degree balancing. We present below how the traversals on the trees are done differently. Except for this, the other parts of the cyclic scanning process of the CH algorithm are implemented in exactly the same manner.

- When an edge $(x, y)$ is used in tree $T_j$, where $x$ is inside the closure set $\mathcal{C}$ and $y$ is outside it, the path between $y$ and $T_j \cap \mathcal{C}$ gets added to $\mathcal{C}$. The classical technique to obtain this path is to maintain depths of vertices in the trees and ensure that one always starts traversing up the tree from the deeper vertex. However, this is not possible here since the depths keep changing due to local degree balancing operations.
- Instead we do the following: if $r_c$ is the root of the closure set in $T_j$, then we traverse the paths both from $r_c$ and from $y$ to the root of the trees $r$, one edge at a time on each path. During these traversals, if we encounter on only one of these paths a vertex that does not belong to $b$'s component, then we still continue our

traversals on both paths one edge at a time, since we are assured that at least half the edges traversed (the edges on the path from $y$ to $T_j \cap \mathcal{C}$) get added to the closure.

- These traversals stop when either the traversal from the lower vertex (either $r_c$ or $y$) in $T_j$ hits the higher vertex or *both* the paths encounter vertices that are outside $b$'s component, in which case $b$ is maximal and we terminate the closure step and go to the "*Discovered a maximal black*" step.

Thus our closure process could terminate in one of two ways: (i) either we discover a larger $(i-1)$-sized cut $b' \supset b$ in $b$'s component or (ii) we discover that $b$ is a maximal black supervertex in its component.

– In Case (i), we go to the "*Local Degree Balancing*" step in order to degree balance all occurrences of $b'$ in $T_1, \ldots, T_i$.

– Case (ii) occurs when we realize the closure set of vertices that we are computing is forced to include an "external vertex" $v$, that is, one that does not belong to $b$'s component in $T_i$ (if we do not include $v$, then the contiguity property of $(i-1)$-sized cuts, refer Lemma 2, would get violated). Since we are constructing a *minimal* set of vertices in the closure that contains $b$ and is a candidate for the smallest $(i-1)$-sized cut that contains $b$, the fact that an external vertex is forced to belong to this set implies that $b$ is maximal in its component (refer Lemma 3 below). Hence in this case we go to the next step: "*Discovered a maximal black*".

LEMMA 3. *If an external vertex enters our closure set, then the current largest black supervertex $b$ is maximal.*

### *Discovered a maximal black.*

We come to this step either when we do local degree balancing for a black supervertex $b$ and discover that $b$ is a maximal black supervertex or we discover in the closure step that $b$ is a maximal black supervertex. In the former case, we need to do global degree balancing (Step 2 of the CH algorithm) followed by Steps 3 and 4 of the CH algorithm, whereas in the latter case, all occurrences of $b$ in $T_1, \ldots, T_i$ already have their degrees balanced due to the local degree balancing step on $b$. But before we go to Step 3 of the CH algorithm, we will first recompute the depths of all the vertices in $T_1, \ldots, T_i$, and then run Steps 3 and 4 of the CH algorithm. These steps ensure that the number of components in $T_i$ reduces by at least a constant factor.

### *Time Complexity of the Algorithm.*

We need to show that the modified algorithm has a time complexity of $\tilde{O}(m)$ in each round. We need to show this bound for the new steps in order to determine a maximal black supervertex; all the other steps are exactly the same as in the CH algorithm and have a time complexity of $\tilde{O}(m)$ per round according to their analysis [3]. The new process for finding maximal black supervertices involves: (1) a slightly modified cyclic scanning routine in our closure step and (2) local degree balancing of black supervertices.

At any stage, when we identify a black supervertex $b$, we contract the entire supervertex into a single node. This guarantees that we do not run the risk of traversing edges internal to the black supervertex in the future. Now, we make the following simple claims:

(i) all edges traversed during a closure step resulting in a black supervertex $b$ are internal to $b$; this ensures that the total time spent in the closure step in a round is $O(m)$,

(ii) all the edges traversed in order to ascertain whether a black supervertex $b$ is maximal are contained in the black supervertex of the next higher level of nesting, provided $b$ is not maximal,[6] and

(iii) if $b$ is maximal, then at least half the edges traversed both in the closure step and in ascertaining whether $b$ is maximal are contained in $b$'s component.

Thus, the total time complexity of our maximal black supervertex finding process is $\tilde{O}(m)$. Since $O(\log n)$ rounds are required to construct each tree, the total time complexity of finding a minimum Steiner cut of size $k$ is $\tilde{O}(mk)$. We have thus shown the following theorem.

THEOREM 5. *Given an undirected graph $G$ and a subset $S$ of its vertices, the edge connectivity $k$ of the vertices in $S$ can be determined in time $\tilde{O}(mk)$.*

## 3. A FAST CUT TREE ALGORITHM

In this section we present our fast algorithm for computing a Gomory-Hu tree. The important properties of our Steiner edge connectivity algorithm, which will be used by our Gomory-Hu tree construction algorithm, are the following:

- A collection of vertices $B$ becomes a *black supervertex* while constructing the $i$th tree if the edge connectivity of $r$ to $B$ is $i-1$ i.e., for all white vertices immediately inside $B$, edge connectivity from $r$ is $i-1$ and $B$ is the minimal Steiner cut of cardinality of $i-1$ for all the white vertices immediately inside $B$. From this stage onwards, vertices in $B$ are contiguous in all trees they appear in.

- The algorithm terminates when a black supervertex containing a Steiner vertex is produced. Let us denote this supervertex by $B_S$. It is easy to see that $(B_S, V \setminus B_S)$ forms a minimum Steiner cut for Steiner set $S$ and $B_S$ is the side of this cut not containing the root $r$. Further, all the vertices of $B_S$ appear contiguously in all trees at the termination of the algorithm, and if one continues to construct more trees, these vertices shall always remain contiguous. This means that in further stages of the algorithm, $B_S$ can be thought of as a single vertex.

- Another property is that $B_S$ is a minimal Steiner min-cut, which means that no proper subset of $B_S$ can be a Steiner min-cut.

These properties lead to the idea of *reusing* the work done by our minimum Steiner cut algorithm for a Steiner set $S$ for one of the sets $S_1$ or $S_2$ that $S$ gets split into. Each node of our computation tree $\Lambda$ will consist of a call to the minimum Steiner cut algorithm with a particular Steiner set. The subproblem corresponding to the root of $\Lambda$ will be

[6]This follows from the fact that each of the $w_i$'s (nearest white vertices to leaf occurrences of $b$) along with $b$ are part of the black supervertex at the next higher level of nesting.

the computation of a minimum Steiner cut $(A, V \setminus A)$ for the Steiner set $V$. Our minimum Steiner cut algorithm will compute this cut by picking some vertex $r$ in $V$ as the root and building $k$ trees rooted at $r$, where $k$ is the size of this cut; let the vertex $r \in A$. The subproblem associated with one of the two children of the root of $\Lambda$ will be the minimum Steiner cut computation for the Steiner set $A$ in the graph $G_1$ that is obtained by contracting all the vertices in $V \setminus A$ into a single node $s_0$.

The crucial idea now (which was also used in [10]) is the following: *in order to compute this cut $(X, Y)$ where the vertices in $A \cup \{s_0\}$ are partitioned into $X$ and $Y$, we do not have to start from scratch but continue from where we stopped after building the $k$ trees which determined the cut $(A, V \setminus A)$.* In these $k$ trees, all the vertices in $V \setminus A$ appeared contiguously - so contracting them and regarding them as one node comes for free. So the first $k$ trees which were built, can be regarded as a part of the process of computing the minimum Steiner cut for the Steiner set $A$ in the graph $G_1$. More generally, at any stage of our algorithm, when we compute the minimum Steiner cut for a Steiner set $S$, this set $S$ corresponds to a node in the partial Gomory-Hu tree and the subtrees rooted at each neighbor of this node in the partial Gomory-Hu tree are contracted to single nodes. We compute the minimum Steiner cut for the Steiner set $S$ in this graph. And most importantly, this minimum Steiner cut need not be computed from scratch and the trees used at its parent in the computation tree can be reused if the root node of those trees belongs to $S$.

In order to reuse trees, Algorithm 3.1 uses a queue $Q$ which stores along with each Steiner set for which the minimum Steiner cut algorithm has to be called, the number of trees already constructed and the trees themselves. This will allow us to reuse trees constructed earlier at a node in the computation tree for one of its child subproblems. For any subproblem extracted from the queue, if no tree has already been constructed for this subproblem, then the algorithm chooses a root uniformly at random from the Steiner set. The algorithm then proceeds to compute trees using our minimum Steiner cut algorithm and stops when a black supervertex containing some Steiner vertex is found. This indicates a minimum Steiner cut and the Gomory-Hu tree $\mathcal{T}$ is modified accordingly and two new subproblems are spawned on $S_1$ and $S_2$, with trees constructed at this stage preserved for reuse in the subproblem for $S_2$ since $r \in S_2$ (because $S_1 \subseteq B_j$ and $r \notin B_j$).

**Algorithm 3.1** Our algorithm for constructing a Gomory-Hu tree for the graph $G = (V, E)$

- Initialize the tree $\mathcal{T}$ to a single node containing the entire vertex set $V$.
- Initialize the queue $Q$ to the queue containing the single element $(V, 0, \emptyset)$.

{*The format of a tuple in the queue is (Steiner set, number of trees already constructed, these trees).*}

**while** the queue $Q$ is not empty **do**
- $flag :=$ TRUE
- delete the first element $(S, i, \{T_1, \ldots, T_i\})$ from $Q$.
- if $i = 0$ then pick a vertex in $S$ uniformly at random as the root $r$.

**while** $flag =$ TRUE **do**
- construct tree $T_{i+1}$ rooted at $r$.
- find black supervertices.

**if** $\exists$ some black supervertex $B_j$ such that $B_j \cap S \neq \emptyset$ **then**
- $S_1 := B_j \cap S$;
- $S_2 := S \setminus S_1$;
- Split node $S$ into $S_1$ and $S_2$ and connect them by an edge of weight $i$ in $\mathcal{T}$.
- Set the neighbors of $S_1$ and $S_2$ in $\mathcal{T}$ appropriately.
- if $|S_1| > 1$ then insert $(S_1, 0, \emptyset)$ in the queue $Q$.
- if $|S_2| > 1$ then insert $(S_2, i, \{T_1, \ldots, T_i\})$ in the queue $Q$.

{*The trees used for $S$ are being reused for $S_2$ since the root $r$ of these trees is in $S_2$.*}
- $flag :=$ FALSE

**else**
- $i := i + 1$

**end if**

**end while**

**end while**

### *Analysis.*

Let us view Algorithm 3.1 in terms of its computation tree. Each node of the computation tree involves a minimum Steiner cut computation in some contracted graph $G'$. The cost of this minimum Steiner cut computation is $\tilde{O}(m'c')$, where $m'$ is the number of edges in $G'$ and $c'$ is the value of the minimum Steiner cut. Since $c' \leq c = \max_{u,v \in V} \lambda(u, v)$, this cost is also $\tilde{O}(m'c)$. By bounding the running time of this subproblem by $\tilde{O}(m'c)$, the child subproblem where the trees constructed for the parent can be reused comes completely for free, since the number of edges in the child subproblem is at most $m'$ and the size of the cut is at most $c = \max_{u,v \in V} \lambda(u, v)$. When we invoke a minimum Steiner cut computation in $G'$, we charge a cost of $\tilde{O}(c)$ to each edge of the graph $G'$ so that we accumulate a cost of $\tilde{O}(m'c)$ to pay for this minimum Steiner cut computation.

Note that the edges crossing the Steiner cut of a subproblem associated with a node $x$ in the computation tree are repeated in the subproblems associated with *both* the children of $x$, while an edge not crossing the cut is a part of only one of these two child subproblems. If an edge is repeated, we say that the original edge is a part of the left child subproblem and a new edge has been added to the right child subproblem in the computation tree. Lemma 4 shows that at most $2m$ new edges are spawned during the course of the algorithm, and therefore there are at most $3m$ edges overall in play.

LEMMA 4. *The total number of new edges created during the course of the algorithm due to repetition of edges in siblings is at most $2m$.*

PROOF. All the new edges correspond to edges in a Steiner cut at a particular stage of the algorithm. So the total number of new edges spawned is $c_1 + c_2 + \cdots + c_{n-1}$, where $c_1, \ldots, c_{n-1}$ are the values of the $n - 1$ minimum Steiner cuts. These values $c_1, \ldots, c_{n-1}$ are the weights of the $n - 1$ edges of the Gomory-Hu tree $\mathcal{T}$. Now we will show that the sum of weights of all edges of $\mathcal{T}$ is at most $2m$. Root the Gomory-Hu tree $\mathcal{T} = (V, \mathcal{E})$ at an arbitrary vertex and define the function $l : \mathcal{E} \rightarrow V$ such that $l(e)$ is the *deeper* of the two Steiner vertices of $e$ in $\mathcal{T}$. It is easy to see that $T$ is an one-to-one mapping. For any edge $e \in \mathcal{E}$, we

have $w(e) \leq \deg(l(e))$ ($deg()$ refers to degree in the graph not in the Gomory-Hu tree). Summing over all the edges in $\mathcal{E}$ and noting that the function $l$ is one-to-one, we have $\sum_{e\in\mathcal{E}} w(e) \leq \sum_{v\in V} \deg(v) = 2m$. □

LEMMA 5. *The expected total cost due to an edge in the algorithm is $\tilde{O}(c)$.*

PROOF. Consider an intermediate step in the algorithm where a subproblem $C$ containing $s$ Steiner vertices is split by the minimum Steiner cut into two subproblems $C_1$ and $C_2$, containing $s_1$ and $s_2$ Steiner vertices respectively, where $s_1 + s_2 = s$. For any edge $e$ which is present in either $C_1$ or $C_2$, let $X_e$ be a random variable denoting the cost due to the edge $e$ in $C_1$ or $C_2$. An edge $e$ in $C_1$ pays a cost of $c \log n$ if the root $r_c$ of the spanning trees constructed for $C$ is in $C_2$, else (the root $r_c$ is present in $C_1$) $e$ pays nothing since $C_1$ would inherit the trees built for $C$. Since the root of the spanning trees is chosen uniformly at random at any stage of the algorithm, the probability that $r_c \in C_2$ is $s_2/s$ and the probability that $r_c \in C_1$ is $s_1/s$. Thus the expected cost of $e$ in $C_1$ is $(s_2/s)\tilde{O}(c)$. Similarly, if $e$ is an edge in $C_2$, then its expected cost is $(s_1/s)\tilde{O}(c)$. Thus

$$E[X_e] = \begin{cases} (s_2/s)\tilde{O}(c) & \text{if } e \in C_1 \\ (s_1/s)\tilde{O}(c) & \text{if } e \in C_2 \end{cases}$$

However, we need to consider a subtle point here. The equation above is valid if our algorithm always picks the same minimum Steiner cut to split $C$ on, irrespective of the choice of root at $C$. However, note that our algorithm does not return a single Steiner min-cut, but all the minimal Steiner min-cuts where the side not containing the root is minimal. So our algorithm needs to decide which Steiner min-cut to split $C$ on. We are always interested in retaining as many Steiner vertices as possible in the part of the Steiner cut which contains the root since we shall get the corresponding subproblem for free in the next iteration. So the algorithm always splits $C$ along that particular cut (among the minimal Steiner min-cuts) which retains the maximum number of Steiner vertices on the side of the root.

Using linearity of expectation, the expected total cost of an edge is the sum of its expected costs at each computation it is part of. Let the edge $e$ trace a path along the computation tree corresponding to the Steiner sets $S_1, S_2, \ldots, S_k$, where $S_k \subset S_{k-1} \subset \ldots \subset S_1 \subseteq V$ and $k \leq n$. We need to consider two situations. If the edge was present in the original graph, then $S_1 = V$ and the expected cost due to the edge at the different subproblems it is part of is at most $(1+(s_1-s_2)/s_1+(s_2-s_3)/s_2+\ldots+(s_{k-1}-s_k)/s_{k-1})\tilde{O}(c)$; on the other hand, if the edge was created by a subproblem due to repetition of the Steiner cut edges in both its children, then $S_1 \subset V$ and the expected cost due to this edge at the different sub-problems it is part of is at most $(s_0 - s_1)/s_0 + (s_1 - s_2)/s_1 + \ldots + (s_{k-1} - s_k)/s_{k-1})c \log n$, where $s_0$ is the cardinality of the Steiner set in the parent of $S_1$ in the computation tree. If $n_0, n_1, n_2, \ldots$ are positive integers, where $n_0 > n_1 > n_2 > \ldots$, then

$$\frac{n_0 - n_1}{n_0} + \frac{n_1 - n_2}{n_1} + \frac{n_2 - n_3}{n_2} + \ldots \frac{n_{k-1} - n_k}{n_{k-1}} \leq$$
$$\frac{1}{n_0} + \frac{1}{n_0 - 1} + \ldots + \frac{1}{n_1 + 1} + \frac{1}{n_1} + \frac{1}{n_1 - 1} + \ldots + \frac{1}{n_k + 1}$$

where we are upper bounding the term $(n_i - n_{i+1})/n_i$ on the left hand side by the sum $1/n_i + 1/(n_i - 1) + \ldots + 1/(n_{i+1} + 1)$. The right hand side is at most $H_{n_0}$, the $n_0$th Harmonic number, which is $\ln(n_0) + \Theta(1)$. Thus it follows from the above inequality that the total cost due to an edge is at most $(\ln n + \Theta(1))\tilde{O}(c)$, which is $\tilde{O}(c)$. □

Since we split the cost of the algorithm among its edges and since there are at most $3m$ edges by Lemma 4, we have shown Theorem 6.

THEOREM 6. *A Gomory-Hu tree in an unweighted graph $G = (V, E)$ with $m$ edges and $n$ vertices can be computed in expected time $\tilde{O}(mc)$, where $c = max_{u,v\in V}\lambda(u, v)$.*

We can actually show a stronger result that the running time of our algorithm is $\tilde{O}(mc)$ with high probablity. The computation tree $\Lambda$ essentially consists of various subproblems, where any contiguous left-going path is a single subproblem which takes time $\tilde{O}(m_i c)$, where $m_i$ is the number of edges in this subproblem. Let us split $\Lambda$ into *layers*. *Layer*$(j)$ consists of all those nodes/subproblems whose path from the root of $\Lambda$ has exactly $j$ right turns. The time taken for *all* the subproblems in any single layer is $\tilde{O}(\sum_i m_i c)$, where $\sum_i m_i$ is the total number of edges involved in all the subproblems associated with this layer. It follows from Lemma 4 that $\sum_i m_i$ in any one layer is at most $3m$, thus it takes $\tilde{O}(mc)$ time per layer. We can show that with probability $1 - 1/n$, the number of layers is $O(\log n)$.

### *Conclusion and Future work.*

We presented the first Gomory-Hu tree algorithm which runs in expected time $\tilde{O}(mn)$. Derandomization, extension to the capacitated case, and improvement in speed are interesting open problems.

**Acknowledgments.** We would like to thank the reviewers for pointing out the correct references to us.

## 4. REFERENCES

[1] J. Bang-Jensen, A. Frank, and B. Jackson, *Preserving and increasing local edge connectivity in mixed graphs*, SIAM J. Discrete Mathematics 8(2), pp. 155-178, 1995.

[2] András A. Benczúr, *Counterexamples for Directed and Node Capacitated Cut-Trees*, SIAM J. Computing 24 (3), pp. 505-510, 1995.

[3] R. Cole and R. Hariharan, *A Fast Algorithm for Computing Steiner Edge Connectivity*, Proc. of the 35th Annual ACM Symposium on Theory of Computing, San Diego, pp. 167-176, 2003.

[4] J. Edmonds, *Submodular functions, matroids, and certain polyhedra* Calgary International Conference on Combinatorial Structures and their Application, Gordon and Breach, New York, 1969, pp. 69–87.

[5] Harold N. Gabow, *A matroid approach to finding edge connectivity and packing arborescences*, J. Comput. System Sci. 50, pp. 259-273, 1995.

[6] A. V. Goldberg and S. Rao, *Beyond the Flow Decomposition Barrier*, JACM 45(5), pp. 783-797, 1998.

[7] A. V. Goldberg and K. Tsioutsiouliklis, *Cut Tree Algorithms: An Experimental Study*, J. Algorithms 38(1), pp. 51-83, 2001.

[8] R. E. Gomory and T. C. Hu, *Multi-terminal network flows*, J. Soc. Indust. Appl. Math. 9(4), pp. 551-570, 1961.

[9] D. Gusfield, *Very Simple Methods for All Pairs Network Flow Analysis*, SIAM J. Computing 19(1), pp. 143-155, 1990.

[10] R. Hariharan, T. Kavitha, and D. Panigrahi, *Efficient Algorithms for Computing All Low s-t Edge Connectivities and Related Problems*, Proc. of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 127-136, 2007.

[11] D. Karger and M. Levine, *Finding Maximum Flows in Undirected graphs seems easier than Maximum Matching*, Proc. of the 30th Annual ACM Symposium on Theory of Computing, pp. 69-78, 1997.