Efficient Algorithms for Steiner Edge Connectivity Computation and Gomory-Hu Tree Construction for Unweighted Graphs*

Anand Bhalgat† Richard Cole‡ Ramesh Hariharan§ Telikepalli Kavitha¶
Debmalya Panigrahi ‖

Abstract

We first consider the Steiner edge connectivity problem on an unweighted undirected or Eulerian directed graph with n vertices and m edges. This problem involves finding the edge connectivity of a specified subset S of vertices, i.e. the cardinality of the minimum cut in the graph that separates the vertices in S into two parts. We give a deterministic algorithm for this problem that runs in $\tilde{O}(m + nc^2)$ time, where c is the Steiner edge connectivity of S. Our algorithm extends an algorithm due to Gabow[Gab95] that finds the minimum cut in a graph by constructing an edge-disjoint spanning tree packing.

We apply this Steiner edge connectivity algorithm to solve our second problem, that of constructing Gomory-Hu trees for undirected unweighted graphs. A Gomory-Hu tree is a succinct data structure for storing pairwise edge connectivity for (or equivalently, maximum flow between) all pairs of vertices in an undirected graph. All previous algorithms for computing a Gomory-Hu tree [GH61, Gus90] use $n - 1$ maximum flow computations. The fastest Gomory-Hu tree algorithm on unweighted graphs with m edges and n vertices has an expected running time of $\tilde{O}(mn + n^2F)$, where F is the maximum pairwise connectivity between any pair of vertices in the graph. This algorithm uses an $\tilde{O}(m + nf)$-time Las Vegas algorithm for computing maximum flow due to Karger and Levine [KL02], where f is the maximum flow. We improve the time complexity of constructing a Gomory-Hu tree to $\tilde{O}(mF)$, which is $\tilde{O}(mn)$ for simple graphs. The novelty of our approach is in replacing maximum flow computations by Steiner edge connectivity computations and showing that with high probability the entire time taken for the Gomory-Hu tree construction by this method is $\tilde{O}(mF)$.

*This paper combines results from [CH03], [HKP07] and [BHKP07]. This work was supported in part by NSF grants CCF 0515127 and IDM 0414763.

†University of Pennsylvania, Philadelphia, PA. bhalgat@cis.upenn.edu. Work partly done when at the Indian Institute of Science, Bangalore.

‡New York University, New York, NY. cole@cs.nyu.edu.

§Strand Life Sciences and House of Algorithms, Bangalore. ramesh@strandls.com. Work partly done when at the Indian Institute of Science, Bangalore and while visiting New York University, New York, NY.

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

‖Massachusetts Institute of Technology, Cambridge, MA. debmalya@mit.edu. Work partly done when at Bell Labs Research, Bangalore and the Indian Institute of Science, Bangalore.

1 Introduction

This paper addresses two related problems, *Steiner edge connectivity* computation and *Gomory-Hu tree* construction, for an unweighted graph $G = (V, E)$ on n vertices and m edges. We consider only Eulerian directed graphs and undirected graphs.

1.1 Steiner Edge Connectivity

Given an arbitrary subset $S \subseteq V$ of *terminal* vertices, we seek to find the smallest cut in the graph such that not all terminal vertices appear on the same side of the cut. We call such a cut a *Steiner min-cut* and its value defines the *Steiner Edge Connectivity* of S. This generalizes the two more well studied notions of a *global min-cut*, which requires only finding the smallest cut so that both sides are non-empty, and an *s-t min-cut*, which requires finding the smallest cut so that s and t are on opposite sides.

The Steiner edge connectivity of arbitrary sets of terminal vertices was previously studied by Dinitz and Vainshtein [DV94], motivated by the fact that these various notions of connectivity above could have substantially different values, e.g., in a complete graph K_n with each edge replaced by a path, the global min-cut is 2 but the Steiner edge connectivity of the original vertices is still $n - 1$.

Note that by Menger's theorem, the value of the Steiner min-cut c equals $min_{u,v \in S} c(u, v)$, where $c(u, v)$ is the number of edge-disjoint paths from u to v, (or from v to u; both are equivalent by Eulerianness). In fact, this can be simplified further to $min_{v \in S} c(r, v)$, where r is any arbitrary vertex in S; we will refer to r as the root. This readily suggest a max-flow approach to the problem, i.e., find the max-flow from r to every other vertex in S and take minimum of these flows. The fastest max-flow algorithm is due to Karger and Levine [KL02] and runs in expected $\tilde{O}(m + nf)$ time, where f is the max-flow. This gives a Steiner edge connectivity algorithm with expected time complexity $\tilde{O}((m + nF)|S|)$ for undirected graphs and Eulerian directed graphs, where $F = max_{v \in S} c(r, v)$.

Our Approach. We give a faster algorithm for this problem — our algorithm is deterministic and runs in $\tilde{O}(nc^2 + m)$ time for Eulerian directed graphs and undirected graphs, with inverse polynomial failure probability. Here, c denotes the Steiner edge connectivity of S. Note in particular that our algorithm is independent of $|S|$.

We will focus on Eulerian directed graphs in the description below. Indeed Eulerian directed graphs and undirected graphs are interconvertible as described in Theorem 6.

We do not use the max-flow approach. Instead we use the r-cut approach, defined as follows. Our task of finding the Steiner min-cut boils down to finding a minimum cut so that r is on one side and some vertex from S is on the other side. More specifically, we insist that r is on the source side, and the value of a cut is the number of edges going from the source side to the sink side. We call such cuts *r-cuts*.

We now appeal to some classical theorems by Edmonds listed below [Edm69, Edm72]. To state these theorems, we need the following definitions. An *r-arborescence* is a directed spanning tree rooted at a specified root vertex r with all edges directed away from r. A *directionless r-spanning tree* is also a spanning tree rooted at r like an arborescence but has the weaker constraint that only edges incident on r must be directed away from the root; all other edges can be arbitrarily directed.

Theorem 1 (Edmonds' Theorem [Edm69])**.** *The maximum number of edge disjoint r-arborescences in a directed graph equals the minimum cardinality of an r-cut.*

Theorem 2 (Edmonds' Relaxed Theorem [Edm72])**.** *The maximum number of edge disjoint r-arborescences in a directed graph equals the maximum number of edge disjoint directionless r-spanning trees in the same*

graph with the property that each vertex $v \neq r$ has total in-degree c over all these r-spanning trees, where c is the total number of trees constructed. (From the definition of directionless r-spanning trees, r has a total in-degree of 0 in the trees.)

Gabow [Gab95] used the above theorems to obtain an algorithm for determining global connectivity in $\tilde{O}(nc^2 + m)$ time, where c is the global min-cut value. This algorithm attempts to build as many edge-disjoint directionless r-spanning trees as possible; when it can build no more, the count of trees built gives the global min-cut value. Equivalently, one could build r-arborescences as well; however that turns out to be more expensive [BHKP08], and therefore directionless trees are preferable. Our algorithm uses the same paradigm and starts where Gabow's algorithm ends, i.e., we show how we can continue the tree building process to continue building directionless trees (though of a slightly different variety, in particular these trees are not spanning trees) beyond the global min-cut count. Note however that while Gabow's algorithm above works for arbitrary directed graphs as well in $\tilde{O}(mc)$ time, our algorithm works only on Eulerian directed graphs.

We are aided by the following theorems in our quest for building more directionless trees. Let $con(v) = c(r, v)$ denote the maximum number of edge disjoint paths from the root vertex r to vertex v. The following theorem appears in [BJFJ95], although their setting is slightly more general (namely, the graphs need not be Eulerian, but the number of edge-disjoint paths from the root to every vertex whose in-degree is smaller than its out-degree is at least the number of trees one is seeking). The proof is based on an edge-splitting lemma due to Lovász and does not immediately lead to an efficient algorithm.

Theorem 3 (The Tree Packing Theorem)**.** *Given an Eulerian directed graph G, there exists a collection of edge-disjoint trees rooted at r such that each vertex $v \neq r$ in G appears in exactly $con(v)$ trees and all edges in the trees are directed away from the root. (These trees need no longer contain all vertices in G.)*

The above theorem clearly implies its relaxed version stated below.

Theorem 4 (The Relaxed Tree Packing Theorem)**.** *Given an Eulerian directed graph G, there exists a collection of directionless edge-disjoint trees rooted at r, such that each vertex $v \neq r$ in G appears exactly $con(v)$ times over all trees (possibly occurring multiple times in a tree) and has in-degree exactly $con(v)$ over these trees. Edges in these trees are allowed to have arbitrary directions, except for those incident on r, which must be directed away from r.*

Our main contribution is a fast constructive proof of a coarser version of the above relaxed theorem. In this coarser version, we contract c trees, where c is the Steiner min-cut value. A vertex v with $con(v) \geq c$ appears exactly once in each of these trees and with total in-degree $con(v)$ over all trees; we call such vertices *white*. In contrast, vertices with $con(v) < c$ are partitioned into vertex sets which are then contracted into what are called *black vertices*. Each black vertex b represents a set of vertices $\mathcal{B}$ in the original graph; such a vertex b appears $con(\mathcal{B})$ times and with total in-degree $con(\mathcal{B})$ over all trees, where $con(\mathcal{B}) = max_{v \in \mathcal{B}} con(v)$.

Note that the partition into black vertices is discovered on the fly as tree construction progresses. We will discover cuts of size k while trying to construct the $k + 1$th tree, and each such cut (i.e., the side not containing r) will be shrunk into a new black vertex. We will now reduce the number of occurrences of this black from $k + 1$ to k. At this point, we bring in a key idea that is needed for tree building to proceed; we will rearrange the trees so each black has degree at most 2, in the process allowing for possibly multiple occurrences of a black in the same tree. We call this *degree balancing*. The whole tree construction procedure terminates when a black containing a terminal vertex from S is discovered. At that point, we will show that the set of vertices represented by this black is indeed a Steiner min-cut.

1.2 Gomory-Hu Tree Construction

A *Gomory-Hu tree* (also known as a *cut tree*) is an $O(n)$-space data structure which represents the pairwise edge connectivity of all pairs of vertices in an undirected graph. More precisely, it is a weighted tree $\mathcal{T}$ on V, with the property that the pairwise edge connectivity between any two vertices s and t in the graph equals the minimum weight of an 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, i.e. a cut of cardinality equal to the s-t edge connectivity. An undirected graph has at least one Gomory-Hu tree, but it might not be unique; on the other hand, examples by Benczúr[Ben95] show that Gomory-Hu trees need not exist for directed graphs. Gomory-Hu trees have many applications in multi-terminal network flows.

All the previous algorithms for constructing Gomory-Hu trees in undirected graphs use max-flow subroutine. Gomory and Hu [GH61] gave the first algorithm for constructing Gomory-Hu trees using $n-1$ max-flow computations and graph contractions. Gusfield [Gus90] proposed an algorithm that does not use graph contractions; all $n-1$ max flow computations are performed on the input graph. Goldberg and Tsioutsiouliklis [GT01] did an experimental study of these two algorithms and described efficient implementations for them. The fastest Gomory-Hu tree algorithm on simple unweighted graphs with m edges and n vertices has an expected running time of $\tilde{O}(mn + n^2F)$, where F is the maximum pairwise edge connectivity of a pair of vertices in the graph, using the $\tilde{O}(m + nF)$ Las Vegas algorithm for max-flow due to Karger and Levine [KL02].

Our Contribution. In this paper, we design an algorithm for constructing a Gomory-Hu tree without using $n-1$ max-flow subroutines, but using our Steiner connectivity algorithm instead. Our algorithm has a time complexity of $\tilde{O}(mF)$; this improves upon the previous best time complexity of $\tilde{O}(mn + n^2F)$.

Theorem 5. *Let $G = (V, E)$ be an unweighted undirected graph with m edges and n vertices. A Gomory-Hu tree for G can be constructed in $\tilde{O}(mF)$ time, where $F = \max_{u,v \in V} c(u, v)$; the running time holds with inverse polynomial failure probability.*

To illustrate what role Steiner connectivity computation plays in constructing the Gomory-Hu tree, consider the following. Suppose we find a global minimum cut $(C, V - C)$. In this process we also find the pairwise minimum cut for all pairs of vertices which are separated by this cut. For Gomory-Hu tree construction, it remains to find the minimum cut between pairs of vertices which are either both in C or both in $V - C$. To do this, we define two subproblems. In the first, we designate all vertices in C as terminals, shrink all vertices in $V - C$ down to a single vertex and then solve an instance of the Steiner connectivity problem. This gives a partition $C_1, C_2, V - C$ of V and minimum cuts have been found for all vertex pairs separated by this partition. The second subproblem is analogous but has vertices in $V - C$ as terminals. Recursive processing yields minimum cuts for all pairs of vertices. Note that each recursive subproblem is an instance of the Steiner connectivity problem.

1.3 Other Applications

It follows from [CH03] that our Steiner connectivity algorithm also yields a fast implementation of the Williamson, Goemans, Mihail and Vazirani algorithm [WGMV95] for the Uniform Survivable Network Design Problem. This implementation runs in time $O((\max_v\{r_v\}^3 n \log n + \max_v\{r_v\}^2 n \log^2 n)$. Here, each vertex v of the given undirected graph has an associated non-negative integral label r_v, which is usually a small constant in practice, and the aim is to choose a collection of edges of minimum cost so that each pair of vertices v, w has $\min\{r_v, r_w\}$ edge disjoint paths.

We also have an algorithm with expected running time $\tilde{O}(m+nk^2)$ for constructing a partial Gomory-Hu tree with parameter k,[1] and, by [HKP07], an algorithm with expected running time $\tilde{O}(m + nk^2)$ that splits a given subset $T \subseteq V$ of even cardinality into two odd cardinality components, where k is the cardinality of the cut.

1.4 Roadmap

Section 2 describes preliminary results needed for our algorithm. In Section 3, we very briefly review Gabow's global connectivity algorithm. In Section 4, we give an outline of our Steiner connectivity algorithm. This algorithm has many different steps and procedures. Section 5 provides invariants and definitions of some key objects in the algorithm, defines input-output characteristics of each procedure, and shows that these procedure do indeed maintain the invariants. Detailed procedure descriptions appear in Sections 6, 7 and 8. The Gomory-Hu tree algorithm is presented in Section 9. Finally, we conclude and outline some possible directions of future work in Section 10.

2 Preliminaries

The following theorem shows that Eulerian directed graphs and undirected graphs are equivalent from the perspective of cut sizes.

Theorem 6. *An Eulerian directed graph G can be converted to an undirected graph G' such that each cut in G' is exactly twice the corresponding cut in G. Similarly, a undirected graph G can be converted to an Eulerian undirected graph G' such that each cut in G' is exactly the same as the corresponding cut in G.*

Proof. Ignoring directions in an Eulerian directed graph yields an undirected graph where each cut is exactly 2 times the value of the corresponding cut in the original directed graph. And converting each edge in an undirected graph into two edges, one in each direction, yields an Eulerian directed graph where each cut has exactly the same size as the corresponding cut in the original graph. □

Unless explicitly stated otherwise, we assume Eulerian directed graphs in the description below.

The following theorems help justify compression of small cuts, i.e., they help show that bigger cuts can be found correctly even after compressing smaller ones.

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 $\delta(A) + \delta(B) \geq \delta(A \cap B) + \delta(A \cup B)$.*

Theorem 7. *If $(S, V \setminus S)$ is a minimum s-t cut in G and for some pair of vertices u, v, $u, v \in S$, then there exists a minimum u-v cut $(S^*, V \setminus S^*)$ such that $S^* \subset S$.*

The following theorem helps prune the number of edges in the graph.

Theorem 8. *Given an undirected graph G with n vertices and m edges, an ordered collection of forests satisfying the following condition can be constructed in $O(n + m)$ time:*

- *Every edge is present in some forest.*

[1] A *partial Gomory-Hu tree* with parameter k is a *contracted* Gomory-Hu tree where all edges with weight more than k are contracted.

- *If a pair of vertices is not connected in a particular forest, then the pair is not connected in any subsequent forest. Consequently, at least i edges from each cut of cardinality $\geq i$ are present among the first i forests.*

Theorem 8 can be used to improve the running time of our algorithm as follows. Our algorithm builds directionless trees sequentially building one directionless tree at a time. To build the kth tree, we can restrict the set of edges to those in the first k Nagamochi-Ibaraki forests. By Theorem 8, the first k forests have $O(nk)$ edges and all cuts of size k or smaller are preserved. Also note that while the above theorem is stated for undirected graphs, an analogous fact holds for Eulerian directed graphs by Theorem 6.

3 Gabow's Algorithm for Global Connectivity

Recall from Section 1 that Gabow's algorithm [Gab95] is based on Theorem 2 which involves constructing a sequence of edge-disjoint directionless r-spanning trees. These directionless spanning trees are constructed one at a time. Given the first k spanning trees $T_1, \ldots, T_k$, the $k+1$th tree T_{k+1} is constructed in several rounds. T_{k+1} is built from a forest initially comprising n singleton vertices. Overloading our notation, we name this forest T_{k+1}. Each distinct tree in this forest is called a *component*.

Each round in the construction process runs in $O(n+m)$ time and reduces the number of connected components in the $k+1$th forest T_{k+1} by at least half, leading to an overall time of $O((n+m)\log n)$ per tree. By Theorem 8, the time to build T_{k+1} is $O(nk \log n + m)$. Gabow's algorithm needs to build $c+1$ trees, where c is the global min-cut (the algorithm aborts before building the last tree); the total time taken is thus $O(nc^2 \log n + m)$.

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 $T_1 \ldots T_{k+1}$ is k (all other vertices have in-degree $k+1$ in $T_1 \ldots T_{k+1}$). Each of these connected components gets processed in this round. Consider one such component $Comp$ with a deficient vertex v.

Closure Computation. Gabow's algorithm now computes the minimum set M containing edges satisfying at least one of the following properties:

- **Seed:** e is *unused*, i.e., $e \notin T_1 \ldots T_{k+1}$, and is directed into v.
- **Swap:** e is in one of $T_1 \ldots T_{k+1}$ and is in the fundamental cycle formed by some edge f in M with respect to that tree.
- **Incidence:** $e \notin T_1 \ldots T_{k+1}$ and is directed into a vertex into which some other edge in M is directed.

Clearly, computing M needs a *closure computation* algorithm, and Gabow shows how to perform this efficiently, i.e., in time proportional to the number of edges and vertices involved in M.

Transformation Sequence. Note that implicit in the closure rules itemized above is the intent to identify a transformation sequence, i.e., a sequence of rearrangements to the trees where edges are moved across trees and an edge connecting components is freed for adding to T_{k+1}. A transformation sequence for $Comp$ maintains one *free* edge with it at any point of time; it then uses this free edge to obtain another via specific operations. An unused edge directed into the deficient vertex v serves as the initial seed free edge. A swap adds a free edge to a tree and frees a tree edge from the resulting fundamental cycle. An incidence takes a free edge (either the seed edge or one obtained after a swap) and replaces that edge with another edge e directed into the same vertex; e now becomes a free edge, available for future swaps. It will be the case that

e will be derived from the pool of what are called *unused* edges, i.e., those which were outside $T_1, \ldots, T_{k+1}$ to begin with. Note that the above transformations preserve in-degrees of vertices in the trees plus free edge combined. These operations go on until the free edge at hand connects $Comp$ to another component in T_{k+1}.

Closure Outcomes. Gabow shows that the closure algorithm above has one of two possible outcomes.

- Either there exists a transformation sequence which connects $Comp$ to another component in T_{k+1}.
- Or the set C of vertices into which edges of M are directed occur *contiguously* in $T_1 \ldots T_{k+1}$, and further, $C, V - C$ actually forms an r-cut of size k.

In the former case, the algorithm has made progress towards reducing the number of connected components, and in the latter case, the algorithm terminates and claims that C is the global min-cut, of size k.

The Global Min-Cut. Of critical importance is the proof that C forms an r-cut of size k. This proof is based on the following facts.

- Each vertex in C other than v has in-degree $k+1$ in $T_1 \ldots T_{k+1}$ and vertex v has in-degree k in $T_1 \ldots T_{k+1}$.
- Vertices in C occur contiguously in each of $T_1 \ldots T_{k+1}$.
- Edges not in $T_1 \ldots T_{k+1}$ but directed into a vertex in C lie completely within C. Thus, edges directed into C from $V - C$ must all be in $T_1 \ldots T_{k+1}$.

An easy consequence of the first two properties is that the in-degree of C in $T_1 \ldots T_{k+1}$ is at most k. The third property ensures that the in-degree of C in the whole graph is also at most k. And an analogous counting argument shows that there are no cuts of size $k-1$, so C is indeed a min-cut.

4 Outline of the Steiner Connectivity Algorithm

In this section, we give an outline of our Steiner connectivity algorithm. As discussed earlier, this algorithm generalizes Gabow's global connectivity algorithm which was described in the previous section. Throughout this section and the subsequent detailed description of the algorithm, $G = (V, E)$ is an Eulerian directed graph with n vertices and m edges. Let S denote the set of terminal vertices whose Steiner connectivity we are interested in finding. We start with Gabow's algorithm. Say, the algorithm builds k trees with root vertex $r \in S$ and is now working on T_{k+1}. By Theorem 8, we assume that $m = O(nk)$ while building T_{k+1}.

As before, we run several rounds, at the end of which T_{k+1} is constructed. Consider a particular round. As earlier, for each component $Comp$ in T_{k+1}, the goal of this round is to modify the trees in such a way that $Comp$ gets connected to some other component. To this effect, we run Gabow's closure computation algorithm on $Comp$ with all unused edges directed into the unique deficient vertex in $Comp$ as *seed edges*. Suppose this closure computation gets stuck in a closure set C. This implies a cut $(V - C, C)$ of size k and Gabow's algorithm quits at this point returning this cut as the global min-cut. In contrast, our algorithm will need to continue further if $S \subseteq V - C$ (because the cut found above does not split S). It follows from Theorem 7 that there exists a minimum Steiner cut $(X, V - X)$ such that the entire set C is present on one side of the cut, i.e., $C \subset X$ or $C \subset V - X$. Thus for the purpose of finding a minimum Steiner cut, we can henceforth regard the entire set C as a single vertex.

Black Vertices and White Vertices. We contract all vertices in C into a single vertex c (we will call such contracted vertices *black vertices* to distinguish them from other original uncontracted vertices which we call *white vertices*); all further computation happens on this new graph with fewer vertices. The trees $T_1, \ldots, T_{k+1}$ will need to be modified to reflect this contraction: this turns out to be straightforward because of the contiguity property, i.e., all vertices in C occur contiguously in each tree.

Once the trees have been modified to reflect this contraction, we run into our next challenge, namely, how to continue the algorithm? More specifically, the issue is that the vertex c has just k edges directed into it and therefore can never achieve in-degree $k+1$ in the trees. So we cannot complete building tree T_{k+1} while satisfying the invariant that each vertex has in-degree $k+1$, as required in Gabow's algorithm. Thus the in-degree invariant has to be relaxed for c. So c is not going to be a regular vertex like other vertices; however our algorithm cannot ignore c and the edges incident on c, since these edges could contribute to the connectivity between pairs of vertices in $V - C$. We need a new operation on all occurrences of c in $T_1, \ldots, T_{k+1}$ to be able to retain c in a special way so that we can continue to run the tree construction algorithm. Our crucial idea here is an operation called *degree-balancing* that we will perform on all occurrences of c. We describe this below.

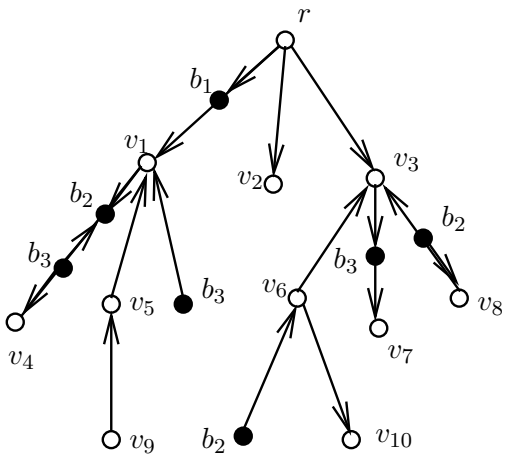

Figure 1: A degree-balanced tree.

Degree Balancing. Note that c has $k+1$ occurrences and in-degree k over all trees. Also note that this leaves no unused edge directed into c, i.e., all edges directed into c are used in $T_1, \ldots, T_{k+1}$. We rearrange edges within the trees $T_1, \ldots, T_{k+1}$ so that all occurrences of vertex c have degree at most 2 (see Fig. 1, where b_1, b_2 and b_3 are black vertices); the procedure which achieves this is called the *degree balancing* procedure (this procedure uses the Eulerian nature of the graph). This procedure could create multiple occurrences of a black vertex within the same tree, but the total number of occurrences and total in-degrees will still be preserved. Note that this is simply a rearrangement of edges; the graph itself doesn't change.

We always maintain that only white vertices can have degree 3 or more in the trees. The need for this degree at most 2 criterion for black vertices will become apparent in our modified closure computation procedure below. But note that this idea is akin to splitting-off c (pairing an incoming edge incident on c with an outgoing edge to create a new superedge so c virtually disappears from all future computations), but with a notable difference that we allow incoming edges to pair with each other and likewise for outgoing edges. This operation can be considered as *directionless* splitting-off of the vertex c, in the same spirit vis-a-vis splitting-off as Edmonds' Relaxed Theorem (Theorem 2) is in relation to Edmonds' Theorem (Theorem 1).

Superedges. After degree-balancing, edges in the trees can be organized into *superedges*, i.e., paths starting and ending at white vertices and carrying only black vertices internally. For instance, in Fig. 1, rb_1v_1 is a superedge, as are rv_2, rv_3, and $v_1b_2b_3v_4$. Paths that lead from a leaf black to their nearest white ancestors are called *partial superedges*, e.g., b_2v_6, b_3v_1. We use the term *superedge* to emphasize that these play essentially the same role in our algorithm that edges do in Gabow's global connectivity algorithm.

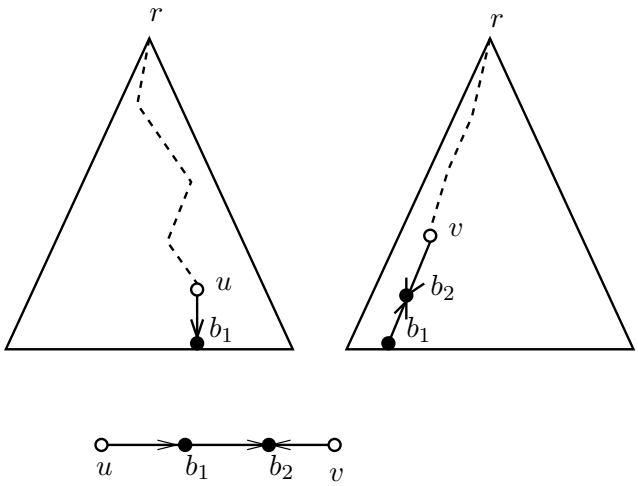

Figure 2: A seed superedge.

Seed Superedges. Our next challenge is to continue the task of connecting $Comp$ to another component in T_{k+1}. The sole deficient vertex in $Comp$ is now part of c and c has no unused edges directed into it to serve as seed edges to start another closure computation. We show that one of the following two cases occurs now:

- Either, $Comp$ comprises only the black vertex c, in which case we just discard $Comp$. This leaves k occurrences of c in the trees with each occurrence having degree 2 or less, and total in-degree over all occurrences equal to k, and we need to do nothing more for $Comp$.
- Or, $Comp$ comprises vertices other than c. Since c occurs $k+1$ times in the trees with each occurrence having degree at least 1 and at most 2, and since c has degree $2k$ in the current graph, there exist 2 leaf occurrences of c in the trees. Each leaf occurrence has an associated partial superedge leading to its nearest white ancestor; we remove these two partial superedges from their respective trees and combine them together as shown in Fig. 2 to yield a new superedge. We call this superedge a *seed superedge*; as we will see shortly, it plays the role of a seed edge in the next closure computation process. This leaves k occurrences of c in the trees and the seed superedge taken together, with each occurrence having degree 2 or less, and total in-degree over all occurrences equal to k.

In the former case, we are done with $Comp$ as it no longer exists. In the latter case, if one or both white endpoints of the seed superedge are outside $Comp$ then $Comp$ can be connected to other component(s) in T_{k+1} by re-pairing edges incident on the black vertex in question (see Section 6.3 for these details). The problem case is when both the white endpoints of the seed superedge belong to $Comp$. In this case, we use this seed superedge to start the next closure computation procedure in $Comp$, as described below.

Extended Closure Computation. Summarizing, our extended closure computation procedure has to deal with the following two cases on $Comp$. The first case is when $Comp$ contains a deficient white vertex

w (so w has in-degree k in the trees). In this case, unused edges directed into w serve as the seed edges for closure computation. The second case is when there is no deficient vertex in $Comp$ but there is a seed superedge with both white endpoints in $Comp$; this path starts the next closure computation. Fig. 3 captures this situation, illustrating that during a round each component of the current tree T_{k+1} (except the one that contains the root r) is either in deficient vertex case or the seed superedge case.

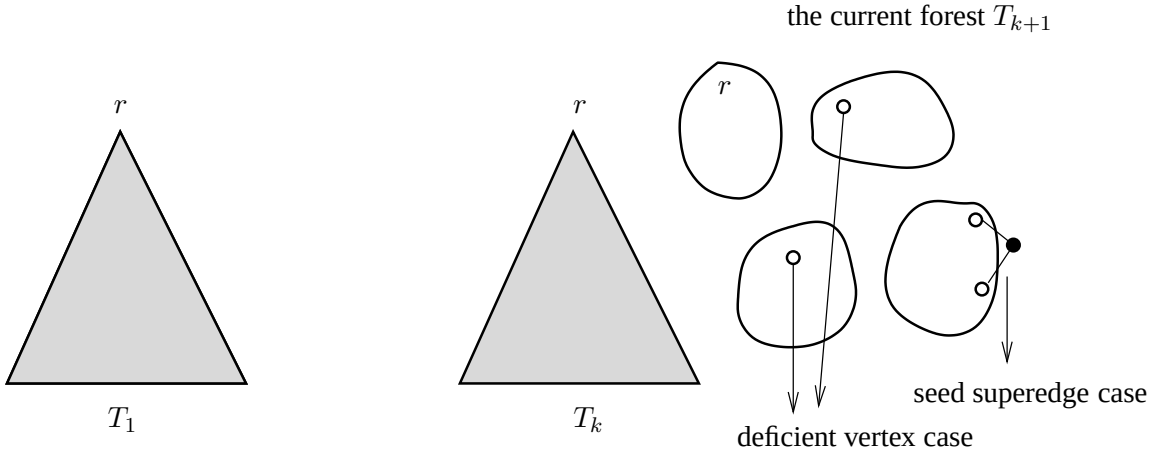

Figure 3: During a round, every component of T_{k+1}, except the one containing r, is in one of these 2 cases: seed superedge case/deficient vertex case.

Gabow's closure computation procedure runs into the following problem in this new setting with black vertices and white vertices. A black vertex has k or fewer occurrences and hence does not appear in every tree. This causes a fundamental problem in Gabow's round robin scheme. We solve this problem by using superedges instead of edges for most part; since superedges have white endpoints and since white vertices are present in every tree, we no longer run into the problem mentioned above.

However, working with white endpoints of superedges alone and ignoring the intervening black vertices leads to another problem. When Gabow's closure computation procedure ends by getting stuck in a closure set B, we can no longer claim that B constitutes a k-cut in the original graph. The reason why we cannot make this claim is because of lack of contiguity: white vertices in B are contiguous in the trees but once the intervening black vertices are brought into play as well, this contiguity breaks down. To actually claim that the closure set B is a k cut, B will have to be closed over the black vertices as well, i.e. B must be a set of white vertices and black vertices such that in each tree, the occurrences of the white vertices in B and the (possibly multiple or zero) occurrences of the black vertices in B must be contiguous. This property cannot be ensured by the swap and incidence rules alone. We will need an additional rule called the *mate* rule which essentially performs re-pairing on edges incident on black vertices. While the swap and incidence rules work on superedges and ignore black vertices, the mate rule considers black vertices with the intention of revising the pairing of edges that we performed during degree balancing. This revision results in the creation of new superedges from existing ones as shown in Fig. 4. With this, we will be able to show that any resulting closure set B is indeed a k-cut in the graph and therefore vertices in B can be contracted to a new black vertex for all future computation. On the other hand, if the above extended closure computation procedure identifies a connecting superedge, we will show that there exists a sequence of swap, incidence and mate operations which will release a connecting superedge enabling its addition to T_{k+1}, and consequent reduction in the number of components in T_{k+1}.

The Whole Algorithm. Now we are ready to bring together the above elements into a complete algorithm

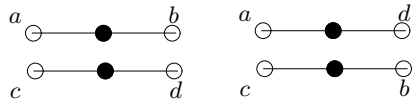

Figure 4: A Mate Operation; 2 superedges with a common black mate to yield two new superedges

(see Algorithm 1). We start with trees $T_1, \ldots, T_{k+1}$ with T_{k+1} having connected components each of which has a deficient vertex (except the component containing the root r). We now run several rounds, where each round reduces the number of connected components in T_{k+1} by a constant fraction.

The algorithm for a round performs extended closure computations and degree balancing steps on the various components in T_{k+1}. A particular component may have to go through several iterations of closure computations and degree-balancings; each such iteration will identify a new k-cut and create a new black vertex, degree-balance that vertex, and identify a new seed superedge for the whole component. When this sequence of iterations terminates for all components, we will be left with the task of performing a sequence of swap, incidence and mate transformations to actually release appropriate superedges and add them to T_{k+1} so the number of components in T_{k+1} reduces by a constant fraction. And if any of the k-cuts found above splits vertices in S then we are done.

Note that the above procedure will be performed on all components concurrently and therefore we need to ensure that the components do not interfere with each other. This will require that the degree-balancing steps and the steps for executing the transformation sequence be split into two parts. The first part will work purely within the component and can therefore be performed independently for each component. The second part will involve interference across components and will be performed by a global procedure which takes all components into account. For degree-balancing, we call these two parts *local* and *global* respectively.

Algorithm 1 Overall sequence of steps in a round.

for each component $Comp$ sequentially in arbitrary order **do**

1. Initialize $Seed(Comp)$ to the seed superedge for $Comp$, if one exists, and otherwise to the set of unused edges directed into the deficient vertex w in $Comp$.

repeat

2. Run *extended closure computation* on $Comp$ with $Seed(Comp)$ to obtain a new closure set B. Output B if B intersects with S, the set of terminal vertices whose Steiner connectivity is in question.

3. Compress B into a new black vertex and perform local degree-balancing on B resulting in a new seed superedge $Seed(Comp)$.

until either closure computation aborts before a new closure set is found or local degree balancing aborts because degree balancing requires going outside $Comp$

end for

4. Perform *global degree balancing* for all components $Comp$ for which local degree balancing aborted above, resulting in a new seed superedge $Seed(Comp)$ for each such component $Comp$.

5. For each component $Comp$, run *extended closure computation* with $Seed(Comp)$.

6. Use the running trace of Step 5 to identify transformation sequences for each component $Comp$, and execute these transformation sequences for a constant fraction of the components to reduce the number of components in T_{k+1} by a constant factor.

5 Algorithm Details

We now describe details, list out invariants and provide proofs of correctness and complexity. Section 5.1 lists out the invariants. Section 5.2 defines closure sets formally and shows that the cut output by Algorithm 1 is indeed a Steiner min-cut, provided the invariants hold. Section 5.3 defines transformations that we will perform in Step 6 of Algorithm 1 and shows that these transformations maintain the invariants. Section 5.4 then describes input-output characteristics of our key procedures, extended closure computation, degree balancing, and transformation sequence identification, and shows that each of these maintains the invariants as well. Section 6 describes details of degree balancing, Section 7 describes details of extended closure computation, and Section 8 describes details of obtaining transformation sequences.

5.1 Invariants and Some Useful Properties

We first define our key invariants.

(1) Each connected component $Comp$ of T_{k+1} (other than the one containing the root vertex r) contains at most one deficient white vertex whose total in-degree is k. If no such vertex exists, then $Comp$ has an associated seed superedge with both white endpoints inside $Comp$.

(2) A white vertex occurs exactly once in each of the trees $T_1, \ldots, T_{k+1}$.

(3) A white vertex (other than r) has total in-degree either k or $k+1$ in the trees and seed superedge taken together.

(4) Each black supervertex b has both total number of occurrences and in-degree equal to $C(b)$ (the edge connectivity of b in G) in the trees and the seed superedges taken together, where $C(b) \leq k$. Note that a black supervertex could have more than one occurrence per tree and does not contain the root r.

(5) Each occurrence of a black vertex in $T_1, \ldots, T_{k+1}$ has degree at most 2, and all degree 1 black occurrences are in T_1.

Superedges and Partial Superedges. Recall from Section 4 that edges in $T_1, \ldots, T_{k+1}$ are organized into superedges (white to white with only blacks internally) and partial superedges (white to leaf black).

Unused Edges. These are edges in the graph that are outside of $T_1, \ldots, T_{k+1}$ as well as the seed superedges. Note that all subsequent references to the term *unused edge* refer to edges which are *originally unused*, i.e., outside of $T_1, \ldots, T_{k+1}$ to begin with. As we perform tree transformations, unused edges could enter the new transformed trees; nevertheless we will still refer to these edges as unused edges. As we show below, an unused end must be directed into a white vertex; in addition, each unused edge with black endpoint b has an associated leaf occurrence of b in $T_1, \ldots, T_{k+1}$. Clearly, we can move the partial superedge associated with b across trees without violating Invariants 1-4 and the first part of Invariant 5; hence the second part of Invariant 5, i.e., without loss of generality, we assume that only tree T_1 has black leaves in it.

Lemma 1. *Assuming Invariants 1-5, every unused edge is directed into a white vertex.*

Proof. We need to show that every edge directed into a black vertex b is present in the trees. Invariant 4 says that every black vertex b has in-degree in the trees equal to its edge connectivity, say i (that is, the cut $(V - B, B)$ was discovered while building T_{i+1} and the set B got contracted to the black vertex b). So there are exactly i edges from vertices of $V - B$ directed into b. As per Invariant 4, all these i edges into b are present in the trees. Thus there is no unused edge directed into b. As this is true for any black vertex b, it follows that every unused edge is directed into a white vertex. □

Lemma 2. *Assuming Invariants 1-5, the number of leaf occurrences of any black b in $T_1, \ldots, T_{k+1}$ equals the number of unused edges with endpoint b; so there exists a one-to-one mapping from the set of leaf occurrences of b to the set of unused edges incident on b.*

Proof. Let i be the edge connectivity of a black vertex b. Then Lemma 1 tells us that the in-degree of b in the graph is equal to its in-degree in the trees, which is i (by Invariant 4). Since the graph is Eulerian, the total degree of b is $2i$, since its in-degree $=$ out-degree $= i$.

Also, the number of occurrences of b in the trees is i (by Invariant 4) and each occurrence is either a leaf occurrence or a degree 2 occurrence (by Invariant 5). Let there be ℓ leaf occurrences of b and $(i - \ell)$ degree 2 occurrences of b in the trees. Then the number of edges incident on b in the trees is $\ell + 2(i - \ell) = 2i - \ell$; thus are $2i - (2i - \ell) = \ell$ unused edges incident on b. Hence the number of leaf occurrences ℓ of b equals the number of unused edges with endpoint b. □

5.2 Closure Sets and Correctness of Steiner min-cut

The goal of this section is to define closure sets precisely. Assuming Invariants 1-5 hold, we also show that if Algorithm 1 terminates in Step 2 by finding a closure set B containing a white vertex from S, then B is indeed the desired Steiner min-cut.

Closure Set. Given a component $Comp$ in T_{k+1}, a closure set for $Comp$ is the minimal collection of black and white vertices B with the following properties:

- B contains both white endpoints and all black vertices in the seed superedge, if one exists, and the white deficient vertex in $Comp$, otherwise.
- B does not contain the root r.
- All of B's white vertices are in $Comp$.
- For all trees T_i, vertex occurrences in T_i of vertices in B are contiguous.
- All unused edges directed into vertices in B have both endpoints in B.

Definitions. In the description below, the term *current graph* denotes the graph in which some vertices have been compressed into black vertices, as opposed to the *original graph* where this is not the case. For any set of vertices B in the current graph, we define the number of *contiguous regions* for B as follows. Consider the graph $\mathcal{R}$ formed by the trees $T_1, \ldots, T_{k+1}$ plus the seed superedges for all components, and consider the subgraph induced by vertex occurrences corresponding to vertices in B. The number of connected components in this subgraph is the number of contiguous regions for B.

Lemma 3. *Consider any subset of vertices B not containing the root r. Let x denote the number of contiguous regions for B, y the number of deficient white vertices in B, and z the number of seed superedges which are completely contained within a contiguous region. Assuming Invariants 1-5, the in-degree of B in $\mathcal{R}$ is exactly $x - z - y$. Further, if B contains at least one white vertex then $x - z - y \geq k$.*

Proof. Let o denote the total number of occurrences in $\mathcal{R}$ of vertices in B. Each non-deficient white in B has in-degree $k+1$ and occurs $k+1$ times, each deficient white in B has in-degree k and occurs $k+1$ times, and each black has in-degree equal to its number of occurrences. So the total sum of vertex in-degrees in B is $o - y$. Edges completely within B contribute total in-degree to the tune of $x - z$ less than o (each

contiguous region contributes a deficiency of 1 but each seed superedge within a contiguous region offsets this by 1). Subtracting this internal in-degree from the sum of in-degrees of individual vertices in B, we get that the in-degree of B in R must be $(o - y) - (o - (x - z)) = x - z - y$.

If B has at least one white vertex then $x \geq k + z + y$, where the first term comes from trees $T_1, \ldots, T_k$ and the last from T_{k+1}. The lemma follows. □

Lemma 4. *Assuming Invariants 1-5, properties 1-4 of a closure set B imply that B has in-degree k in $\mathcal{R}$ and well as in trees $T1, \ldots, T_{k+1}$.*

Proof. Let B be a closure set in a component $Comp$. Property 1 and 4 imply that $x = k + 1$. Property 3 implies that either $z = 1, y = 0$ or $y = 1, z = 0$. Since property 2 holds, we can invoke Lemma 3; the first part of the lemma follows. The second part about trees follows from property 1. □

Lemma 5. *Assuming Invariants 1-5, a closure set B represents a cut of size k in the current graph. Further, B represents a min-cut in the current graph separating r from any white vertex in B.*

Proof. By Lemma 4, the in-degree of B in $\mathcal{R}$ is k. By property 5, the in-degree of B in the current graph is k as well. Further, since B contains at least one white vertex, the second claim in the lemma follows from by Lemma 3. □

Lemma 6. *Assuming Invariants 1-5, a closure set B represents a cut of size k in the original graph. Further, B represents a min-cut in the original graph separating r from any white vertex in B.*

Proof. Clearly a cut in the current graph induces a cut of the same size in the original graph (simply open up the black vertices). So the first part of the lemma follows from Lemma 5.

The second part of the lemma can be seen as follows. If the current graph comprises no black vertices then it is identical to the original and the lemma follows. So suppose there are black vertices in the current graph, and inductively assume that each of the corresponding closure sets B' represents a min-cut separating r from whites inside B' in the original graph. Then, by Theorem 7, it follows that for any white vertex v, there exists a min-cut (separating v from r) in the original graph which does not split any of the black vertices. Such a cut has a counterpart in the current graph as well of the same size. From Lemma 5, it follows that closure set B represents a min-cut separating r from whites inside B in the original graph as well. The lemma follows. □

Minimal Steiner min-cuts. We define a minimal Steiner min-cut $(B, V - B)$ in the original graph as a Steiner min-cut for which there exists a terminal vertex v such that $v \in B$, $r \in V - B$, and in addition, no proper subset of B containing v is a Steiner min-cut.

Lemma 7. *Let k denote the value of the Steiner min-cut. A cut $(B, V - B)$ is a minimal Steiner min-cut in the original graph if and only if B appears as a closure set while constructing T_{k+1} and B contains a terminal white vertex.*

Proof. First, suppose B is a minimal Steiner min-cut and let v be the associated terminal vertex. Then v must become part of a closure set B' when constructing some tree in Algorithm 1. By Lemma 6, that tree must be T_{k+1}, otherwise the min-cut separating v from r in the original graph will not be k. Applying 1 and the minimality of B, we can conclude that $B = B'$. That shows one side of the lemma.

Second, suppose B appears as a closure set while constructing T_{k+1} and it contains a terminal white vertex v. By Lemma 6, B is a Steiner min-cut in the original graph. It remains to show minimality. Consider

any subset B' of B containing v. By the definition of a closure set, B is the minimal set which satisfies properties 1-5. So B' violates at least one of these 5 properties (in particular, one of properties 1, 4 and 5). We show that B' must have cut size at least $k+1$.

If B' violates only property 5, then Lemma 4 says that B' would have in-degree k in the trees plus at least one additional in-degree outside among the unused edges. So suppose B' violates either property 1 or property 4. Recall the parameters x, y, z from Lemma 3. Note that $x \geq k+1$ because $v \in B$. If B' violates property 1, then $y = z = 0$ and $x - y - z \geq k+1$. And if it violates property 4, then $x > k+1$ and $y + z$ can contribute at most 1, so $x - y - z \geq k+1$. So B' must have cut size at least $k+1$ in $\mathcal{R}$, and therefore in the current graph, and therefore in the original graph as well. □

5.3 Transformation Sequences

We are given a collection of trees $T_1, \ldots, T_{k+1}$ satisfying Invariants 1-5 and a collection of components $Comp_1, \ldots, Comp_h$ in T_{k+1}. Each component either has an associated seed superedge or a collection of seed edges; in the latter case, all these seed edges are directed into the same vertex. We now define a *transformation sequence* for this collection of components as follows. These will be the sequences applied in Step 6 of Algorithm 1.

We start with $e_1, \ldots, e_h$, each denoting either the seed superedge, if one exists, or exactly one of the seed edges for their respective components, otherwise. A transformation sequence is defined as any sequence comprising only the operations listed below and satisfying the following additional properties.

- Each of these operations results in some tree modifications while updating one or two of the e_i's.
- Each e_i is either a superedge or it is an edge directed into a white endpoint in $Comp$.
- At least one white endpoint of e_i is in $Comp$.
- The e_i's are *free*, i.e., outside of the current trees obtained from the original ones via already executed transformations (contrast these from unused edges, which are free initially but need not stay free as these transformations happen).

The eventual goal of defining a transformation sequence is to reduce the number of connected components in T_{k+1} by a constant fraction while maintaining Invariants 1-5.

Transformation Operations. The transformations listed below include swap and incidence operations (as in Gabow's algorithm (see Section 3) along with mate or re-pairing operations illustrated in Fig. 4. In fact, the operations listed below could be combinations of these operations, i.e., a mate followed by a swap etc. In addition, there are operations which combine a partial superedge in a tree with an unused edge (see Lemma 2) to create a new superedge.

1. Swap: Swap-in a superedge e_i into some tree and remove a whole superedge f or a single edge f directed into a white vertex from its fundamental cycle; the swapped-out superedge/edge is the new e_i.

2. Mate-Swap: Mate superedge e_i with a superedge f in some tree as shown in Fig 5. f must not be in the fundamental cycle of e_i and must share a black vertex with e_i. Mating results in two new superedges, both of which are added to the tree in question. From the resulting fundamental cycle, swap out either a whole superedge or one edge directed into a white vertex from a whole superedge. This resulting edge/superedge replaces e_i.

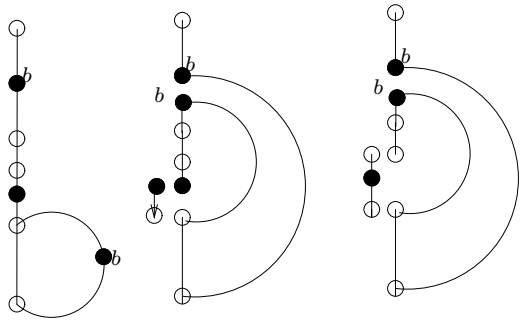

Figure 5: Operation 2: The first figure is before the operation is performed, and the second shows two cases after the operation is performed, namely, only an edge is freed or a whole superedge is freed.

3. Incidence: Replace the given edge e_i directed into white vertex, say w, with another edge outside of $T_1 \ldots T_k$ and $e_1, \ldots, e_k$, directed into w.

4. Mate: Mate two superedges e_i, e_j as shown in Fig.4; they must have a black vertex in common.

5. Join: Consider an edge e_i with black endpoint b and suppose one of the trees has a leaf occurrence of b (such an occurrence is indeed mandated by Lemma 2). Pull out the partial superedge associated with b and append it to edge e_i to obtain a new superedge e_i.

6. Join-Swap Consider an edge e_i with black endpoint b and suppose one of the trees has a leaf occurrence of b (such an occurrence is indeed mandated by Lemma 2). Swap in e_i into this tree such that its endpoint b is identified with the above leaf occurrence; from this fundamental cycle swap out either a whole superedge or one edge directed into a white vertex.

7. Connect Consider a superedge e_i which has one white endpoint in $Comp_j$, $j \neq i$ (recall one endpoint must always be in $Comp_i$). Then add e_i to T_{k+1} and combine the two components to yield a new set of components. e_j becomes the associated superedge/edge for the new component.

Lemma 8. *The set of modified trees $T_1 \ldots T_{k+1}$ and e_i's resulting from a transformation sequence satisfy Invariants 1-5 (assuming Invariants 1-5 hold to begin with and assuming operation 7 does not apply for any edge e_i at the end of the sequence).*

Proof. Invariant 2 and the first part of Invariant 5 are easily seen to hold. The last part of Invariant 5 is violated though, but it is easy to sweep all leaf black occurrences into T_1 in $O(nk)$ time after all transformations are done.

We show Invariants 1, 3 and 4 below. We need the following terminology first. Let $\mathcal{E}$ denote the initial pool of superedges/edges e_i at the beginning of the sequence, and let $\mathcal{S}$ denote the subset that comprises superedges. Let $\mathcal{T}$ denote the initial pool of trees. Let $\mathcal{E}'$, $\mathcal{S}'$ and and $\mathcal{T}'$ denote their final counterparts at the end of the sequence. Note $\mathcal{S}$ is exactly the set of seed superedges initially, and $\mathcal{S}'$ is the counterpart at the end of the sequence.

An exploration of the above operations shows that in-degrees in $\mathcal{T} \cup \mathcal{E}$ are identical to in-degrees in $\mathcal{T}' \cup \mathcal{E}'$, and the number of black vertex occurrences in $\mathcal{T} \cup \mathcal{S}$ is identical to that in $\mathcal{T}' \cup \mathcal{S}'$.

Now, Consider Invariant 4. Note that in-degrees of blacks in $\mathcal{T} \cup \mathcal{E}$ equals that in $\mathcal{T} \cup \mathcal{S}$, and likewise in-degrees of blacks in $\mathcal{T}' \cup \mathcal{E}'$ equals that in $\mathcal{T}' \cup \mathcal{S}'$ (because edges in $\mathcal{E} - \mathcal{S}$ are directed into whites and

likewise for $\mathcal{E}' - \mathcal{S}'$). Thus both in-degrees and black vertex occurrences in $\mathcal{T} \cup \mathcal{S}$ are identical to those in $\mathcal{T}' \cup \mathcal{S}'$. Invariant 4 follows.

Finally, consider Invariants 1 and 3. The in-degree of each white in $\mathcal{T} \cup \mathcal{E}$ is $k+1$. Therefore, the same holds for $\mathcal{T}' \cup \mathcal{E}'$. Now consider $\mathcal{T}' \cup \mathcal{S}'$. Each component with associated edge in $\mathcal{E}' - \mathcal{S}'$ has exactly one vertex with in-degree k in $\mathcal{T}' \cup \mathcal{S}'$; all other whites continue to have in-degree $k+1$ in $\mathcal{T}' \cup \mathcal{S}'$. Superedges in $\mathcal{S}'$ have both endpoints within their respective components (because operation 7 does not apply any more). Invariants 1 and 3 follow. □

Component-Specific Transformation Sequences. A transformation sequence specific to component $Comp_i$ is one in which only e_i changes and all other e_j's stay the same. The eventual transformation sequence of interest will comprise individual component-specific sequences for each component (as in Section 8.1.2) followed by a portion where multiple e_i's could change simultaneously (as in Section 8.3.2).

5.4 Key Procedures

Next we define input-output characteristics of our 3 key procedures. Details of each procedure appear in subsequent sections. We need the following definitions first.

Definitions. Given a set of black and white vertices X, let $||X||$ denote the number of edges with both endpoints in X. Let $w(Comp)$ denote the number of white vertices in $Comp$.

Extended Closure Computation (Steps 2 and 5). Extended closure computation will process each component $Comp$ independently in arbitrary order, it results in the following outcomes.

- Either it identifies a new closure set B in $Comp$.
- Or, in the event that new closure sets are not identified in any of the components, it guarantees the existence of a transformation sequence (see Section 5.3) that reduces the number of components in T_{k+1} by a constant fraction.

In the former case, the time taken by this procedure is $O(||B|| + k)$ for component $Comp$. In the latter case, the time taken by this procedure is $O(nk)$ over all components. Note that closure computation makes no changes to the trees so the invariants stay unaffected. Details of extended closure computation appear in Section 7.

Local Degree Balancing (Step 3). Given a component $Comp$ and a new black vertex b_{Comp} in $Comp$, this procedure does one of two things.

- Either it determines that there are no more closure sets in $Comp$. The total time taken in this case is $O(w(Comp) * k)$.
- Or,
  - First, it rearranges edges amongst the trees so every occurrence of b_{Comp} has degree at most 2 and leaf occurrences stay in T_1. Note that every unused edge stays unused in the process.
  - Second, it reduces the number of occurrences of b_{Comp} by 1 and identifies and releases a seed superedge for $Comp$ by joining together two partial superedges associated with leaf occurrences of b_{Comp}. This seed superedge is guaranteed to have both white endpoints in $Comp$.

– All changes made to T_{k+1} are within $Comp$, i.e., the connected components in T_{k+1} stay the same.

If another closure set B' is discovered subsequently in $Comp$ (while running closure computation initiated by the seed superedge obtained after degree balancing b_{Comp}), then the time taken by the local degree balancing step for B is $O(w(B') * k)$, where $w(B')$ is the number of white vertices in B'; otherwise it is $O(w(Comp) * k)$.

Details of local degree balancing appear in Section 6.2.

Global Degree Balancing (Step 4). Given a collection of components and given a new black vertex b_{Comp} in each such component $Comp$, this procedure does the following:

- First, it rearranges edges amongst the trees so every occurrence of b_{Comp} has degree at most 2 for every component $Comp$ and the number of occurrences of b_{Comp} reduces by 1. Note that every unused edge stays unused, and a simple $O(nk)$ time pass at the end moves all leaf black occurrences to T_1.
- In the above process, it reorganizes components in T_{k+1} but guarantees that the number of components in T_{k+1} only decreases in this step.
- Next, for each resulting component $Comp'$ in T_{k+1}, it releases a seed superedge for $Comp'$; this seed superedge is guaranteed to have both white endpoints in $Comp'$.
- Finally, for each resulting component $Comp'$ in T_{k+1}, it guarantees that there is no closure set inside $Comp'$.

The total time taken by this procedure is $O(nk)$. Details of global degree balancing appear in Section 6.3.

Lemma 9. *Local and Global degree balancing maintain Invariants 1-5 (assuming Invariants 1-5 hold to begin with).*

Proof. Local degree balancing in a component $Comp$ in T_{k+1} arranges edges incident on a black vertex b discovered in $Comp$ so that there are exactly k occurrences of b after this degree balancing step and each occurrence has degree at most 2. No component other than $Comp$ is affected by this step. Now we will show that Invariants 1-5 are maintained by this step for the component $Comp$.

1. Local degree balancing identifies and releases a seed superedge for $Comp$. Thus Invariant 1 is maintained.
2. The local degree balancing algorithm rearranges edges among the trees $T_1, \ldots, T_{k+1}$ by taking edges (w, b) incident on leaf occurrences of b and moving them to trees where b has high degree. No white vertex w is moved by this step, hence each white vertex occurs exactly once in each of the trees $T_1, \ldots, T_{k+1}$.
3. As local degree balancing only rearranges edges among the trees and creates a seed superedge, local degree balancing retains the total in-degree of each white vertex in the trees/seed superedge. Thus if each white vertex (other than r) has total in-degree either k or $k + 1$ prior to local degree balancing, then this is true after local degree balancing.

4. Invariant 4 (the total number of occurrences and the total in-degree of each black vertex in the trees/seed superedge) is retained for every black vertex other than b, as local degree balancing only rearranges edges among the trees/seed superedge. We now have to show that Invariant 4 is maintained for b also.

    - There are totally $k+1$ occurrences of b currently (one in each of the trees $T_1, \ldots, T_{k+1}$). If $Comp = b$, then this component is dropped from T_{k+1}, then there are exactly k occurrences of b in the trees henceforth. If $Comp$ has vertices other than b, then a seed superedge is created by joining two partial superedges edges. Thus two occurrences of b are merged into 1 occurrence, this results in exactly k occurrences in the trees/seed superedge.
    - The total in-degree of b in the trees is equal to k and after rearrangement of edges in the trees and creation of seed superedge, it follows that the total in-degree of b in the trees/seed superedge is equal to k after local degree balancing.

5. Local degree balancing ensures that each occurrence of b in the trees has degree at most 2, no other black vertex occurrence is affected by this step, and only T_1 has black leaf occurrences. Thus, the first part of Invariant 5 is maintained.

Global degree balancing works across different components and this step also rearranges edges incident upon each maximal black vertex whose degree needs to balanced in this step so that every occurrence of this vertex has degree at most 2. It follows from arguments similar to the ones given above for local degree balancing that Invariants 1-5 are maintained by this step. □

Identifying and Executing Transformation Sequences (Steps 5 and 6). Given a collection of components in T_{k+1} with the guarantee that each such component $Comp$ contains no closure sets, this step does the following:

- First, a constant fraction of the components are identified.
- Transformation sequences are determined for each of these chosen components in such a way that the sequences for the various components can be executed independently in an arbitrary order (so performing transformations for one component does not render transformations for another component invalid).
- Transformation sequences for the chosen components are then executed resulting in the number of components in T_{k+1} going down by a constant fraction.

Details of executing transformation sequences appear in Section 8. The time taken by this procedure will be $O(nk)$.

5.5 Correctness and Complexity

Lemma 10. *Invariants 1-5 hold at the beginning of Algorithm 1 and at the end of each step in Algorithm 1.*

Proof. Recall trees $T_1, \ldots, T_k$ are constructed one by one, and for each tree there are several invocations of Algorithm 1. At the beginning of the algorithm, when all we have is T_1 with each component in T_1 being singleton, the invariants clearly hold with $k = 0$. So after constructing $T_1, \ldots, T_k$ and while constructing T_{k+1}, let us assume the invariants hold at the beginning of a particular invocation of Algorithm 1, and show

that they continue to hold after each step. Lemmas 10 and 11 show that the steps local degree balancing, global degree balancing, and executing transformation sequences maintain Invariants 1-5. The extended closure computation step does not modify any trees. Thus Invariants 1-5 hold at the end of each step in the algorithm. □

Lemma 11. *The total time taken for one invocation of Algorithm 1 is $O(nk) = O(nk)$.*

Proof. First consider Step 2. The time taken in an iteration of Step 2 is $O(||B|| + k)$ if a closure set B is discovered; and the time taken over the last iterations of Step 2 over all components is $O(nk)$. It remains to account for all but the last iterations for each component. Note that $||B||$ added up over the various closure sets is $O(m)$ because no edge counts in two of the closure sets (because a closure set is compressed to a black vertex for future use). And the plus k adds up to $O(nk)$ because each closure set contains at least one white vertex.

Let us now estimate the time taken for local degree balancing. Consider components $Comp$ and let $\mathcal{B}_1, \ldots, \mathcal{B}_h$ denote the various closure sets obtained in successive iterations. The time complexity is $O(\sum_2^h w(B_i) * k) + O(w(Comp)*)k$. This sums to to $O(w(Comp) * k)$ for $Comp$ and $O(nk)$ overall.

Global degree balancing summed over all components takes $O(nk)$ time. Identifying and executing transformation sequences takes $O(n + m)$ time over all components. Thus the time taken for a round to construct T_{k+1} is $O(n + m)$. Note that m can be bounded by $O(nk)$ by using NI preprocessing step. Thus the time taken for a round is $O(nk)$. □

Theorem 9. *All minimal Steiner min-cuts for the specified terminal vertex set S with respect to the root can be determined in time $O(mk \log n) = O(nk^2 \log n + m)$, where k is the size of the Steiner min-cut.*

Proof. Each round reduces the number of components in T_{k+1} by a constant fraction. Thus $O(\log n)$ rounds are required to build tree T_{k+1} and as each round for T_{k+1} takes $O(nk)$ time (by Lemma 11), the total time to build T_{k+1} is $O(nk \log n)$. Hence the time taken to build trees $T_1, \ldots, T_{k+1}$ (during the construction of T_{k+1} we realise that the Steiner min-cut is k) is $O(nk^2 \log n)$. Correctness follows from Lemma 7. □

Theorem 10. *Given any subset X of vertices, the minimal min-cut separating v from the chosen root r can be found for all vertices $v \in X$ in time $O(mk \log n) = O(nk^2 \log n + m)$, where k is the value of the largest of these minimal min-cuts.*

Proof. By Lemma 7, constructing $k + 1$ trees suffices to find these min-cuts. The time taken to build trees $T_1, \ldots, T_{k+1}$ is $O(nk^2 \log n)$. □

6 Degree-Balancing Subroutines

Let B denote a closure set discovered for $Comp$. We contract B into a single vertex b and then we degree-balance b, i.e. we re-distribute superedges incident on b such that each occurrence of b in $T_1, \ldots, T_{k+1}$ has degree at most 2. Recall that once we degree-balance b, we also get a seed superedge containing b to resume the closure computation procedure on $Comp$.

6.1 Decision Procedure for Selecting Degree-Balancing Subroutine

First, we need the following decision procedure to determine whether to run local or global degree balancing. We consider each tree T_i in which the occurrence of b has degree greater than 2 and run the following procedure.

Consider an occurrence of b in tree T_i with degree $d+2, d>0$, and let $u_1, \ldots, u_{d+2}$ be the other white endpoints of the superedges incident on b in T_i. Since b currently has $k+1$ occurrences overall and total degree $2k$ in the trees $T_1, \ldots, T_{k+1}$, there exist d leaf occurrences of b which can be uniquely assigned to this occurrence of degree $d+2$ (see Lemma 2). We remove the d superedges incident on these d leaf occurrences from their respective trees and add them to T_i (note that b is a new black, so we call these superedges and not partial superedges when degree balancing is in progress). Let the resulting superedges be $e_1, \ldots, e_d$ and their white endpoints (i.e. the endpoints other than b) be $w_1, \ldots, w_d$. Each e_j causes a fundamental cycle between b and w_j in T_i. Our goal is to check whether there is any external white vertex on these d fundamental cycles. We do this as follows.

We traverse upwards from each of the w_i's and from b, progressing each traversal in a round-robin manner and marking any white vertices traversed. We traverse only white vertices and skip over black vertices internal to superedges. For each traversal, we say it is *successful* if it hits a previously marked white vertex or hits b, and we say it *fails* if it hits an external white vertex. A particular traversal halts when it encounters either success or failure. The whole procedure stops if either all but one traversal have succeeded or when two of the traversals have failed. In the former case, it is easily seen that all white vertices in the above fundamental cycles are within $Comp$. And in the latter case, there is at least one external white vertex in these fundamental cycles.

The above procedure is run on all relevant trees. If the latter case holds for any of these trees, then we run the global degree balancing procedure for b, and otherwise we run the local degree balancing procedure.

Lemma 12. *If one of the above fundamental cycles has an external white vertex, then if the next iteration of Step 2 for $Comp$ in Algorithm 1 were to be run, it would not find a closure set.*

Proof. We will show that the seed superedge yielded by degree balancing b will contain an occurrence of b. And clearly, it will have a white endpoint. It follows from property 4 in the definition of closure sets that any closure set for $Comp$ will then need to contain all the w_i's as well. Property 3 will then be violated. □

Lemma 13. *In the event that all white vertices in the above fundamental cycles are inside $Comp$, all of these white vertices are inside any future closure sets that Algorithm 1 discovers in $Comp$.*

Proof. Follows as in the proof of Lemma 12. □

If all white vertices in the above fundamental cycles are in $Comp$ then the time taken by the above procedure is proportional to the number of these white vertex occurrences. By Lemma 13, this is proportional to $w(B') * k$, if there is a subsequent closure set B' found for $Comp$, and $w(Comp) * k$ otherwise. And if there is an external white vertex in one of the fundamental cycles, then the time taken is proportional to the number of occurrences in the fundamental cycles of white vertices from $Comp$, which is $w(Comp) * k$, as required.

6.2 The Local Degree Balancing Algorithm

Consider the occurrence of b in T_i with degree $d+2$ where we also added d superedges $e_1, \ldots, e_d$ incident upon b from other trees. We will match each of the edges $e_1, \ldots, e_d$ with one of the edges incident on the above occurrence of b using the algorithm described below. As there are $d+2+d = 2d+2$ edges incident upon b, our algorithm makes d pairs from these $2d+2$ edges so that by creating d new occurrences of b and making each new occurrence of b a degree 2 occurrence using the pairing, T_i remains connected. Note that the original occurrence of b also has degree 2 (since $2d+2-2d = 2$) now. Thus T_i will have $d+1$

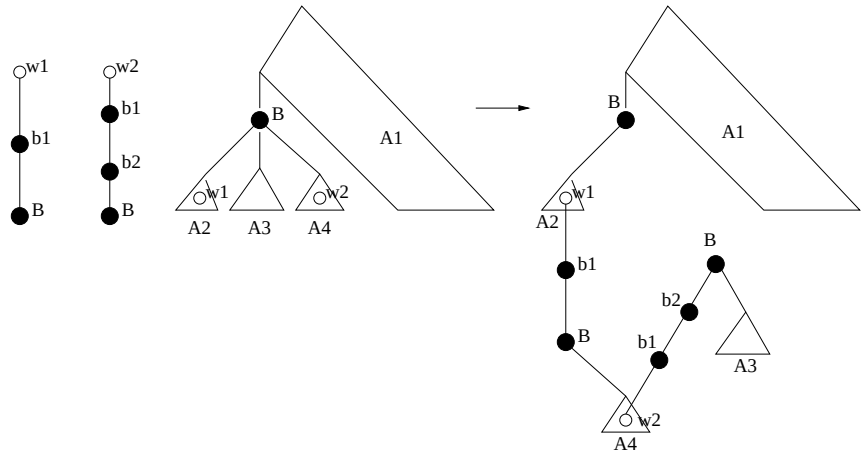

Figure 6: Local balancing black vertex B in a tree

occurrences of b now, each with degree 2. Note that the total number of occurrences of b remains $k+1$ in this process (d leaf occurrences are replaced by d new occurrences in T_i) (see Figure 6 for an example).

Define $CycEdg(e_j)$ to be that neighbor (a white vertex) u_ℓ of b in T_i such that the superedge (b, u_ℓ) is in the fundamental cycle of e_j. Our traversals described are also used to compute $CycEdg()$ for each of $e_1, \ldots, e_d$. For any traversal which terminated in b, the corresponding $CycEdg()$ is the last white vertex encountered before b. For any traversal that terminated in a vertex marked by b, the corresponding $CycEdg()$ is the first white vertex encountered after b in the traversal from b. Finally, for any traversal that terminated in a vertex marked $w \neq b$, the corresponding $CycEdg()$ is the same as $CycEdg(e_w)$, where e_w is the superedge between b and w. We now take e_1 and match it to any superedge other than $(b, CycEdg(e_1))$; for instance suppose we match it to the superedge (b, u_h). (Note that such a superedge always exists since at any stage, the occurrence of b with degree more than two has at least 2 neighbors more than the number of e_js left.) For all those e_j with $CycEdg(e_j) = u_h$, we now redefine $CycEdg(e_j)$ to be $CycEdg(e_1)$, and then repeat the above procedure until all the edges $e_1, \ldots, e_d$ are matched.

Lemma 14. *The above transformations keep T_i connected.*

Proof. We use induction on the sequence of transformations. Since T_i is initially connected, the base case follows. Suppose after a set of transformations, T_i is connected. Since we ensure that we do not pair the next superedge (say e_j) with $(b, CycEdg(e_j))$ where $CycEdg()$ is defined according to the *current* configuration of T_i, T_i stays connected. □

After the above procedure is completed for all occurrences of b, each occurrence of b has degree at most 2. We now take a pair of degree 1 occurrences, call them (u, b) and (v, b), to create the new superedge (u, v), which is the seed superedge to resume the closure computation for $Comp$. (Note that the number of occurrences of b in the trees and seed superedge taken together is now k.) All remaining degree 1 occurrences are moved to T_1. We know that u and v are white vertices in $Comp$ since prior to local degree balancing we traversed the edges incident on b and found no external white vertices.

Running time. Initializing the value of $CycEdge()$ for each of the d leaf superedges is done during the traversals and take the same time (i.e., $O(nk)$ overall) as the traversals. In order to update the function $CycEdge()$, we use the union-find data structure. Thus, the time taken here is $O(\alpha d)$ where α is the inverse

Ackermann function. Adding over all the trees, this is $O(\alpha k)$. Summing over the entire algorithm, this becomes $O(\alpha m)$. This follows from the fact that the sum of in-degrees of all the blacks over all the trees at any stage of our algorithm is at most $2m$. Lemma 15 proves this claim.

Lemma 15. *$\sum_b C(b) \leq 2m$, where the sum is over all the black vertices b that are discovered during the algorithm.*

Proof. Consider any black vertex b. This is a subset B of vertices that gets contracted to a single vertex. There is at least one white vertex $w \in B$ that was the deficient vertex in the component $Comp$ of B that triggered the closure computation in $Comp$ which eventually led to the formation of the cut $(V - B, B)$. Note that $(V - B, B)$ is a min r-w cut, thus the value of this cut is bounded from above by the degree of w. Thus $\sum_b C(b)$ is bounded from above by the sum of degrees of all vertices in G, which is $2m$. □

6.3 The Global Degree Balancing Algorithm

We now describe a single global procedure for degree balancing all those black vertices (these are necessarily maximal in their respective components) where we encountered an external white vertex during the tree traversals described earlier. This procedure is global in the sense that it handles all such maximal black vertices in various components of T_{k+1} together; the associated tree traversals will no longer be confined to the respective components. Consider any tree T_i and let $\mathcal{B} = \{b_1, \ldots, b_r\}$ be the set of black vertices in T_i in need of global degree balancing. Also, let $d_j > 2$ denote the degree of b_j in T_i, and let $d = \sum_{j=1}^{r}(d_j - 2)$.

Recall that for each b_j, our goal is to pair the superedges corresponding to the $d_j - 2$ leaf occurrences of b_j that we have added to T_i (we call these *leaf superedges*) with the superedges incident on b_j that were already present in T_i (we call these *tree superedges*). This will result resulting in $d_j - 1$ occurrences of b_j, each with degree 2. The condition for pairing above is that T_i should stay connected (or if $i = k + 1$, then the number of connected components should not reduce). This condition, which is the same as that for local degree balancing, requires that each leaf superedge e incident on b_j be paired with a tree superedge f incident on b_j such that *f is not in the fundamental cycle of e in T_i*. We show below how this can be achieved simultaneously for all the vertices $b_1, \ldots, b_r$ in $O(n)$ time, giving $O(nk)$ time per round over all the $k + 1$ trees. We need the following definition.

Descendant Subtrees. A *descendant subtree* of $b_j \in \mathcal{B}$ is a subtree rooted at any child v of b_j (where v is a white vertex). Thus, b_j has $d_j - 1$ descendant subtrees. Suppose b_j has a descendant subtree rooted at a white vertex v which neither contains any other vertex in $\mathcal{B}$, nor the endpoint of any leaf superedge added to T_i. (Such a descendant subtree is called a *friendly* descendant subtree.) Then, we can pair the superedge (b_j, v) with any of the leaf superedges incident on v (say (w, v)) while keeping tree T_i connected. As an additional advantage, the entire friendly descendant subtree along with the $w - b - v$ superedge can now be contracted into the vertex v for the remaining part of the procedure since this subtree will remain unchanged irrespective of the pairing of superedges (refer to Figure 7(a)). This property will prove critical in showing the efficiency of our procedure.

Our procedure, therefore, is the following. Initially all leaf superedges are unmatched; also, initially, all black vertices in $\mathcal{B}$ have degree greater than 2, and this degree reduces as pairing happens, eventually resulting in all occurrences having degree 2. We find a friendly descendant subtree, pair the corresponding tree superedge with any leaf superedge and contract the entire subtree along with the superedge formed by joining the tree and leaf superedge. We then repeat the procedure for a new friendly descendant subtree. We terminate when all the black vertices in $\mathcal{B}$ have degree 2 in each occurrence.

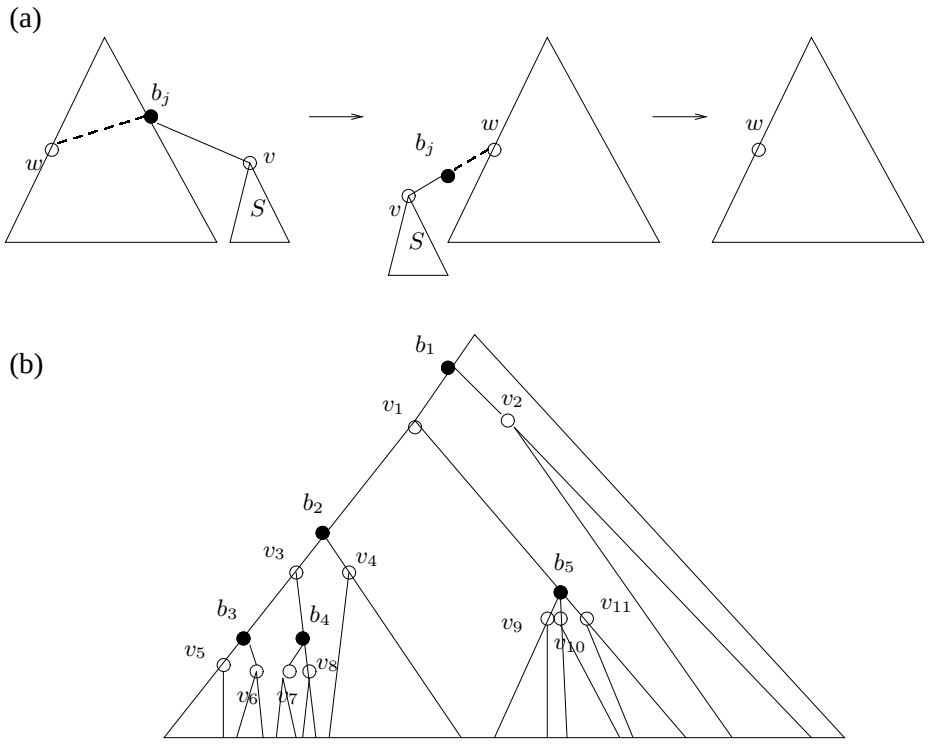

Figure 7: (a) The transformation in one iteration where S is a friendly descendant subtree. The first step performs the degree balancing and the second step contract the subtree S along with the superedge (w, v); (b) The mapping is as follows (S_i is the descendant subtree rooted at v_i): S_1 is mapped to either b_2 or b_5, S_3 is mapped to either b_3 or b_4, all other descendant subtrees have an empty map.

Lemma 16. *There always exists a friendly descendant subtree in tree T_i during the above procedure.*

Proof. We prove the property at the start of the procedure; since the proof is based on a counting argument and both the degree and the number of unmatched leaf edges decreases by 1 after each iteration, the proof continues to hold subsequently during the course of the algorithm. Let us define a mapping from each descendant subtree (call it S and let it be rooted at a white vertex v) to a black vertex b_j in $\mathcal{B}$ with the following property: *b_j is in the descendant subtree S, i.e. b_j is a descendant of v in T_i and no other black vertex in $\mathcal{B}$ is present on the path from v to b_j* (refer to Figure 7(b)). For descendant subtrees having multiple black vertices satisfying the above property, we select any such black vertex arbitrarily. On the other hand, for descendant subtrees not containing any black vertex in $\mathcal{B}$, the above mapping is empty. We are interested in counting the number of descendant subtrees whose map is empty for the above mapping.

The critical property of the mapping defined above is that not more than one descendant subtree can map to the same black vertex b_j in $\mathcal{B}$, since otherwise, one of the black vertices corresponding to the descendant subtrees is between b_j and the root of the other descendant subtree. Further, there is at least one black vertex b_l in $\mathcal{B}$ which is not the descendant of any other b_j in $\mathcal{B}$; therefore, b_l is not in any descendant subtree and no descendant subtree maps to b_l by the above mapping. Thus, at most $r-1$ descendant subtrees have a non-empty map by the above mapping. Now, the total number of descendant subtrees over all the black vertices in $\mathcal{B}$ is $\sum_{j=1}^{r} d_j - r$. Thus, at least $\sum_{j=1}^{r} d_j - 2r + 1$ descendant subtrees do not contain any black vertex in $\mathcal{B}$. Since there are $\sum_{j=1}^{r}(d_j - 2)$ unmatched leaf superedges, each with one white endpoint, at least 1 of the above descendant subtrees neither contains any black vertex in $\mathcal{B}$, nor has any white endpoint of an unmatched leaf superedge. □

Let this friendly descendant subtree $\mathcal{S}$ be rooted at a child v of a black vertex $b_j \in \mathcal{B}$; then the edge connecting b_j to v is paired with any of the leaf edges for b_j, causing this leaf edge to become matched, causing the degree of b_j to reduce by 1, and causing $\mathcal{S}$ to become compressed to a single vertex and no longer be considered a descendant subtree. The algorithm simply repeats this process until degrees for all vertices in $\mathcal{B}$ are down to 2.

We will now show that we can implement the above algorithm efficiently, namely we can find a friendly descendant subtree in each iteration using $O(n)$ time over all iterations. We perform a post-order traversal of tree T_i, where each walk up from the leaves stops when we have one of the following conditions:

- We reach the white endpoint v of an unmatched leaf superedge.
- We reach a child (white vertex, say w) of a black vertex (say b_j) in $\mathcal{B}$.

In the first case, the traversal waits at v until the leaf superedge incident on v is matched. In the second case, we are guaranteed that the subtree at w is a friendly descendant subtree. Thus, we match the (w, b_j) tree superedge with any leaf superedge incident on b_j. After the matching, one of the following two situations happens: either b_j still has a degree greater than 2, in which case the walk waits at b_j, or b_j now has a degree of 2 in which case the walk continues upward to the parent of b_j. Thus, at any stage, we have a set of walks waiting at white endpoints of leaf superedges or at black vertices in $\mathcal{B}$ with degree greater than 2. The above lemma however ensures that at any stage we always have at least one walk that does not get stuck, namely the walk in the friendly descendant subtree.

Running Time. Since each friendly descendant subtree is contracted after the superedge pairing, no superedge in the tree is traversed more than once. This ensures a running time of $O(n)$ for one tree and $O(nk)$ over $T_1, \ldots, T_{k+1}$.

We have already noted that any tree T_i stays connected after the above procedure. However, recall that T_{k+1} is actually a forest; we now show that the number of components in T_{k+1} remains unchanged during global degree balancing.

Lemma 17. *Let x be the number of components in T_{k+1} before global degree balancing. At any point of time during global degree balancing on T_{k+1}, the number of components in T_{k+1} is x.*

Proof. We prove this by induction. Assume that the claim holds before any particular change. Clearly, if a component splits into two fragments, then one of the two fragments is attached to an already existing component at the white endpoint of the leaf superedge. An entire component does not get incorporated in another component because the maximal black vertex in this component retains an occurrence with degree of 2 in the component. Thus the total number of components remain unchanged. □

After global degree balancing. Let b be a black vertex that has just undergone global degree balancing. Thus every occurrence of b has degree at most 2. Recall that there are currently $k+1$ occurrences of b in the trees $T_1, \ldots, T_{k+1}$ and the total degree of all occurrences of b is $2k$. We have the following two cases now.

- All the occurrences of b in $T_1, \ldots, T_k$ are degree 2 occurrences. Then b is an entire component in the forest T_{k+1}. In this case, we delete the component b from T_{k+1}. So b now has k occurrences in $T_1 \ldots T_{k+1}$.
- b has leaf occurrences in some of $T_1, \ldots, T_{k+1}$. Since b has $k+1$ occurrences in these trees and total degree $2k$, there are at least 2 leaf occurrences of b in $T_1, \ldots, T_{k+1}$.

  We detach two leaf superedges incident on b and pair them with each other; this superedge becomes the seed superedge for the component $Comp$ containing b. Let (x, y) be this seed superedge. We have the following possible situations.

  1. Both x and y belong to $Comp$: then Step 5 (closure computation) of Algorithm 1 needs to be run on $Comp$ initiated with (x, y) as the seed superedge.
  2. Both x and y belong to some other component $Comp'$: then we perform a mate between the seed superedge (x, y) and the superedge (u, v) that contains b in $Comp$. The resulting superedges from the mate ((x, u) and (y, v)) connect $Comp$ and $Comp'$ and no further processing is required for either $Comp$ or $Comp'$ in this round.
  3. One of x, y belongs to $Comp$ and the other belongs to another component $Comp'$: then the seed superedge is added to T_{k+1} to connect $Comp$ and $Comp'$; no further processing of either $Comp$ or $Comp'$ is required in this round.
  4. Neither x nor y belongs to $Comp$ but they belong to different components: then we mate the seed superedge with the edge containing b in $Comp$ and add the resulting edges to T_{k+1}. This splits $Comp$ into two parts and the two parts get attached to $Comp'$ and $Comp''$ (the components that x and y respectively belong to) respectively; no further processing of either $Comp'$ or $Comp''$ is required in this round.

Since the components in T_{k+1} might be re-organized by global degree balancing as shown above, we now need to show that we only need to run the closure computation procedure only once for each new component in T_{k+1}.

Lemma 18. *The closure computation procedure performed after global degree balancing does not lead to a cut of size k in any component.*

Proof. First, consider a component $Comp$ where a closure computation earlier found an external vertex (and not a new k-cut). If a new k-cut C is found by the closure computation now, then C clearly contains at least one white vertex which has been brought into $Comp$ from some other component by the global degree balancing procedure (otherwise, the earlier closure computation should also have found this k-cut). Since C is contiguous in T_{k+1}, C must also contain a black vertex b which was brought into $Comp$ by global degree balancing. Further, there must be another component of T_{k+1} which contains at least one occurrence of b, namely, the component where b was identified as a black vertex in the first place. Thus, some occurrence of b (specifically, an occurrence in $Comp$) is in C while some other occurrence (specifically, an occurrence in the original component of b) is not in C. This violates the definition of C.

Next, we consider a component $Comp$ which underwent global degree balancing for a black vertex b. For this component, if the closure computation finds another k-cut C, then C must contain b since the closure computation is initialized with a seed superedge containing b. By Lemma 12, b is a maximal black vertex in $Comp$ (considering the composition of $Comp$ before global balancing). Thus, for C to be a k-cut, it must contain white vertices which have entered $Comp$ from other components due to global degree balancing. Since C is contiguous in T_{k+1}, C must also contain a black vertex b' which was brought into $Comp$ by global degree balancing. There must be another component of T_{k+1} which contains at least one occurrence of b', namely, the component where b' was identified as a black vertex in the first place. Thus, some occurrence of b' (specifically, an occurrence in $Comp$) is in C while some other occurrence (specifically, an occurrence in the original component of b') is not in C. This again violates the definition of C. □

7 Extended Closure Computation Details

We are given a collection of trees $T_1, \ldots, T_{k+1}$ and components $Comp_1, \ldots, Comp_h$ in T_{k+1} with associated seed superedges/seed edges satisfying Invariants 1-5.

Behaviour. Extended closure computation will process each component independently in arbitrary order, and is described in Algorithm 2 for one such component $Comp$. It results in the following outcomes.

- Either it identifies a new closure set C in $Comp$.
- Or, in the event that new closure sets are not identified in any of the components, it guarantees the existence of a transformation sequence (see Section 5.3) that reduces the number of components in T_{k+1} by a constant fraction.

In the former case, the time taken by this procedure is $O(||C|| + k)$ for component $Comp$. In the latter case, the time taken by this procedure is $O(nk)$ over all components. Note that extended closure computation itself does not make any transformations, it merely guarantees the existence of a transformation sequence as above. Also recall that when called from Step 5 in Algorithm 1, the second case above will hold for all components (see Lemma 18). In that case, Step 6 in Algorithm 1 actually identifies and executes the associated transformation sequences; that procedure is described in subsequent sections.

Vertices and Vertex Occurrences. We use the term *vertex occurrence* for a white vertex v to denote a particular occurrence of v in the trees $T_1, \ldots, T_{k+1}$, and for a black vertex v to denote a particular occurrence of v in the trees $T_1, \ldots, T_{k+1}$ or the seed superedges.

Outline. Step 7 is the heart of Algorithm 2. The whole algorithm performs several invocations of Step 7, where each invocation involves traversing the path between some vertex occurrence v' (black or white) to C_i in some tree T_i, where C_i is a set of contiguous vertex occurrences in T_i. All vertex occurrences encountered on this path are then added to C_i. They are also added to C so they can used for initiating invocations of Step 7 in other trees. Note that C is a set of vertices while C_i's comprise not vertices but vertex occurrences. Another source of additions to C is Step 9 which looks at unused edges outside the trees. A white vertex w is said to be *incidence-ready* if there exists an edge directed into w in some tree T_i, $1 \leq i \leq k+1$, with both endpoints of the edge inside C_i. The order of additions to C_i, C in Step 7 and the order of processing vertices in Step 4 and in the for loop after step 4 are important and will be used in Lemma 23 for generating transformation sequences. Note that for all other components $Comp'$, we use explicit superscripts for their corresponding C_i, C, i.e., $C_i^{Comp'}, C^{Comp'}$.

Algorithm 2 Extended Closure Computation for Component $Comp$

0. Let u be any endpoint of the seed superedge, if its exists, and the common white endpoint of the seed edges, otherwise.
1. Initialize C to all black and white vertices (and not vertex occurrences) which occur in the seed superedge or all the seed edges, as the case may be.
2. Initialize C_i to vertex occurrence u for each tree $T_i, 1 \leq i \leq k+1$.
repeat
  for each tree $T_i, 1 \leq i \leq k+1$ in order **do**
    3. Let X denote the set of vertices in C which have vertex occurrences in T_i outside C_i.
    4. Order vertices in X so whites come first, blacks later, and each set is further sorted in increasing time order of entry into C.
    for each vertex $v \in X$ in order and every occurrence v' of v in T_i (in arbitrary order, except in T_1 where leaf occurrences, if any, are considered first) **do**
      5. If the path between C_i and v' in T_i contains a white vertex outside $Comp$, then terminate the algorithm.
      6. If v is black and either has an occurrence in the seed superedge for some other component $Comp'$, or the occurrence v' is present in $C_i^{Comp'}$ for a previously processed component $Comp'$, then terminate the algorithm.
      7. Add all white and black vertex occurrences between C_i and v' to C_i in order of increasing distance from C_i and add the corresponding vertices to C in the same order.
    end for
  end for
  8. Identify all white vertices w which are newly incidence-ready, i.e., incidence-ready now but not at the end of the previous iteration of the repeat loop.
  9. Identify all vertices w' such that there exists an unused edge from w' directed into w, where w is newly incidence-ready; add w' to C if either w' is black or it is white and belongs to $Comp$, otherwise terminate the algorithm.
until C converges, i.e., does not change in an iteration of the repeat loop

Lemma 19. *For each tree T_i, the vertex occurrences in C_i are contiguous in T_i. Further C_i contains at least one white vertex.*

Proof. C_i starts with the single vertex occurrence u. For each subsequent addition to C_i in Steps 7, all intervening vertex occurrences are also added to C_i. □

Lemma 20. *The time taken by the above procedure is proportional to $O(||C|| + k)$ if C converges. And if termination happens in Steps 5, 6 or 9, the time taken summed over all components is $O(nk)$.*

Proof. The checks in Step 6 are easily implemented in $O(1)$ time by marking vertex occurrences appropriately (note each vertex occurrence can be in the C_i for at most one component by virtue of Step 6). The time for Step 8 is dominated by Steps 5, 7 because one can track incidence-readiness when two endpoints of an edge are added to C_i. So consider two cases for the remaining steps.

First, suppose C converges. Step 2 takes time $O(k)$. For Steps 1, 5, 7, 9, each unit of time spent in these steps can be charged to an edge completely within C, so this time is $O(||C||)$. Second, suppose C doesn't converge. Then, the times are as follows, aside from the $O(1)$ time spent on the terminating vertex. The time taken in steps 2, 5, 7 is $\sum_1^{k+1} O(|C_i|) + O(w(C))$ (each unit of time spent can be charged to an addition to C_i or to a white vertex in $Comp$). For Steps 1 and 9, the time taken is proportional to the number of edges directed into white vertices in $Comp$ plus the length of the seed superedge. By Lemma 22, $\sum_1^{k+1} |C_i|$ adds up to $O(nk)$ over all components. The other terms clearly add up to $O(nk)$.

This leaves Steps 3 and 4. It suffices to show that this step can be performed in time $O(|X|)$ (because at least $O(1)$ time will be spent in processing each item in X in Steps 5, 6, unless termination happens in these steps when processing X, in which case this cost can be charged to the entry into C of the corresponding vertices in Steps 7 or 9). To achieve $O(|X|)$ time, we keep two queues with each tree. Each time a vertex v not already in C is added to C it is put into the second queue for the next tree (in cyclic order) that has an occurrence of v. And each time a vertex v in X is processed in the inner for loop it is put into the first queue for the next tree (in cyclic order) that has an occurrence of v, provided v has not gone through a full cyclic round. X for a tree is then obtained simply by combining the contents of the two associated queues for that tree. By keeping the queues segregated by blacks and whites, the ordering required in Step 4 can be achieved. The lemma follows.

□

Lemma 21. *If the extended closure computation procedure for $Comp$ terminates because C converges, then C is a closure set.*

Proof. Recall properties 1-5 of closure sets from Section 5.2. These are shown as follows. Property 1 follows from the initialization of C. Properties 2 and 3 follows from the fact that Algorithm 2 adds only white vertices within $Comp$ to C. Property 4 comes from Lemma 19. Property 5 is shown below.

We need to show that all unused edges directed into C have both endpoints in C. Suppose this is not true for an unused edge directed into vertex $w \in C$. Then w never becomes incidence-ready and therefore, in each tree T_i, all tree edges directed into w have their other endpoints outside C_i. By Lemma 4, the first 4 properties above imply that there are only k edges directed into C in the trees. So the in-degree of w in the trees plus seed superedge (if any) must be k, i.e., w is deficient. Then, by initialization in Step 1, all unused edges directed into w have both endpoints in C, a contradiction.

Finally, we need to show that C is minimal, i.e., no subset of C satisfies all the properties of a closure set for $Comp$. This is easily seen because Algorithm 2 starts with vertices needed to satisfy Invariants 1 and 5, and adds vertices only if Invariant 4 is violated. □

It remains to show that if the extended closure computation procedure terminates in Steps 5, 6 or 9 for each of the components, then there exists a transformation sequence (see Section 5.3) that reduces the number of components in T_{k+1} by a constant fraction. We show how to algorithmically obtain such a transformation sequence next in time $O(nk)$.

8 Obtaining a Transformation Sequence

There are two phases to this procedure. Phase 1 considers each component $Comp$ independently and obtains a component-specific transformation sequence (the goal is to get within a few transformations of having one endpoint outside $Comp$). The time taken by Phase 1 will dominated by the closure computation time above. Phase 2 considers all components together and adds further transformations to decrease the number of components in T_{k+1} by a constant fraction. The time taken by Phase 2 will be $O(nk)$.

Phase 1 for component $Comp$ itself depends upon which step in Algorithm 2 causes termination. If termination is caused in Step 6 by some vertex occurrence v' belonging to $C_i^{Comp'}$ for some previously processed component $Comp'$ then we say that $Comp$ is *premature*. And if termination is caused either in Step 6 by some vertex occurrence v' belonging to the seed superedge for another component $Comp'$, or in Steps 5 or 9 due to a white vertex outside $Comp$, we say $Comp$ is *mature*. Phase 1 will be described separately for these two types of components, in Sections 8.1 and 8.2, respectively (the mature case is the simpler one, the premature case will need more processing). Section 8.3 describes Phase 2. We need the following useful lemma.

Lemma 22. *For all trees T_i, $1 \leq i \leq k+1$, and all pairs of components $Comp', Comp''$, the sets $C_i^{Comp'}$ and $C_i^{Comp''}$ are disjoint.*

Proof. This follows from the termination conditions in Steps 5 and 6 of Algorithm 2. □

8.1 Phase 1: Component-specific Transformation Sequence for Mature Component $Comp$

Note that mature components have one of the following termination criteria: either a white vertex outside $Comp$ is encountered in Steps 5 or 9, or an occurrence of a black vertex which is present on the seed superedge of some previously processed component $Comp'$ is processed in Step 6. The former components are labeled *white-terminal-5* and *white-terminal-9* respectively, and the latter are labeled *black-terminal*. First, we introduce the notion of a *trace sequence*, which is essentially a trace of Algorithm 2. Subsequently, we show how to convert a trace sequence to a component-specific transformation sequence for a mature component $Comp$.

8.1.1 Trace Sequences and Properties

The core of the extended closure computation in Algorithm 2 for $Comp$ is Step 7 which repeatedly performs traversals from a vertex occurrence v' towards the current C_i in tree T_i (recall from Lemma 19 that C_i is contiguous and therefore can be visualized as a single shrunk vertex for convenience). The *trace sequence* $\mathcal{T}$ for $Comp$ is a sequence of pairs

$$(v_0, j_0), (v_1, j_1), \ldots, (v_{r-1}, j_{r-1}), (v_r, j_r)$$

where each v_h is a vertex occurrence in tree T_{j_h} which is processed by Algorithm 2 (i.e., it plays the role of v' in the inner for loop in Algorithm 2).

Intuition. We start with the vertex occurrence v_r during whose processing Algorithm 2 encountered one of the termination cases. We then consider vertex occurrence v_{r-1} which when processed caused vertex v_r to be encountered for the first time in the algorithm. Then we consider vertex occurrence v_{r-2} which when processed caused vertex v_{r-1} to be encountered for the first time in the algorithm. Thus each vertex

occurrence v_h is the cause for Algorithm 2 encountering vertex v_{h+1}. And v_0 is one of the vertices whose cause is the initialization in Step 1.

Notation. Let C_i^h denote the set C_i just before vertex occurrence v_h in the trace sequence is processed. Let C^h denotes the set C at the same instant.

Trace Sequence Definition. We now define trace sequence formally.

- Vertex v_0 must have entered C by the initialization in Step 1.
- For $0 \leq h < r$, vertex occurrence v_h must be the *cause* of vertex v_{h+1}, i.e., vertex occurrence v_h is the first vertex occurrence to be processed for which vertex v_{h+1} is encountered in either Step 7 or Step 9, as made more precise below.
  - Either an occurrence of vertex v_{h+1} appears on the path π_h from vertex occurrence v_h to $C_{j_h}^h$ in T_{j_h} (*cause via Step 7*) (see Fig. 8 part a),
  - Or, this path contains an edge directed into a vertex w such that no edge directed into w in any tree T_i has both endpoints inside C_i^h, and there exists an unused edge between v_{h+1} and w that is directed into w (*cause via Step 9*) (see Fig. 8 part b).
- If $Comp$ is white-terminal-5 then vertex occurrence v_r in the item (v_r, j_r) is the cause of a white vertex outside $Comp$ via Step 5.
- If $Comp$ is white-terminal-9 then v_r is outside $Comp$ and vertex occurrence v_{r-1} in the item (v_{r-1}, j_{r-1}) is the cause of v_r via Step 9. In this case the item (v_r, j_r) is really a dummy item we add for convenience, i.e., it does not reflect an actual vertex occurrence processed in Algorithm 2 because termination happens as soon as v_r is encountered in Step 9; however, we imagine a dummy step added to Algorithm 2 which processes this v_r and stops as soon as this processing starts; accordingly we set j_r to undefined and note for later reference that the item $(v_r, undefined)$ has been introduced for notational convenience in this case.
- If $Comp$ is black-terminal then v_r is black and there is an occurrence of v_r in the seed superedge of a previously processed component $Comp'$. Further, when the vertex occurrence v_r in (v_r, j_r) is processed by Algorithm 2, the algorithm stops in Step 6.

Note that in all three cases, the processing of items (v_{r-1}, j_{r-1}) does not conclude in termination while the processing of (v_r, j_r) does. Clearly, the trace sequence for $Comp$ can be determined in time bounded by the closure computation time by just recording a history of causes in the extended closure algorithm itself.

We capture the following simple properties for future reference.

Lemma 23. *The following hold for any item (v_h, j_h) in the above trace sequence (note that in cases where $h+1 = r$ and j_r is undefined, the relevant claims involving T_{j_r} will not apply).*

1. *For $0 \leq h < r$, vertex v_{h+1} is outside C^h.*
2. *For $0 \leq h < r$, all vertex occurrences on the path from vertex occurrence v_h to $C_{j_h}^h$ in T_{j_h} are in $C_{j_h}^{h+1}$ (see Fig. 8, both parts a and b) and the corresponding vertices are in C^{h+1}.*
3. *For $0 \leq h < r$, suppose vertex occurrence v_h in T_{j_h} is the cause of vertex v_{h+1} via Step 7, and suppose v_{h+1} is white. Let $w \neq v_{h+1}$ denote the first white vertex on the path from the occurrence of vertex v_{h+1} in π_h to $C_{j_h}^h$. Then the occurrence of w in $T_{j_{h+1}}$ is in $C_{j_{h+1}}^{h+1}$.*

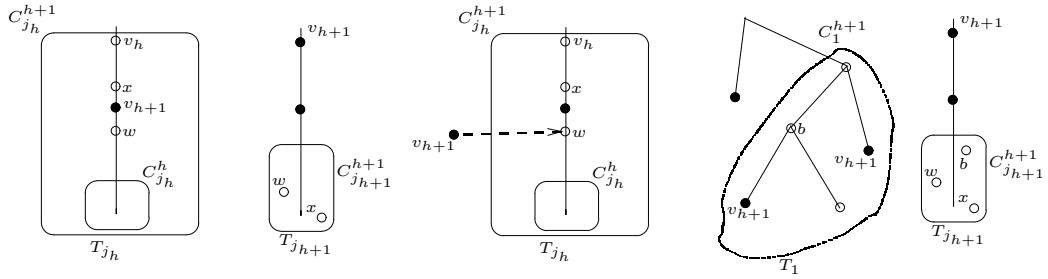

Figure 8: a) The first two figures show cause via Step 7. b) The last three show cause via Step 9. In these, once v_{h+1} enters C via Step 9, the processing of v_{h+1} in T_1 will result in the second of these three drawings before v_{h+1} is processed in $T_{j_{h+1}}$ as in the third drawing.

4. *For $0 \leq h < r$, suppose vertex occurrence v_h in T_{j_h} is the cause of vertex v_{h+1} via Step 7, and suppose v_{h+1} is black. Let w denote any white vertex on the path π_h or any white vertex in $C^h_{j_h}$. Then the occurrence of w in $T_{j_{h+1}}$ is in $C^{h+1}_{j_{h+1}}$ (see Fig. 8 part a).*

5. *If $Comp$ has a seed superedge, both white endpoints of this superedge (except v_0 if it is white) are in $C^0_{j_0}$. And if $Comp$ has a seed edge containing v_0 then the other white endpoint of this edge is in $C^0_{j_0}$. In any case, all endpoints of seed superedges/seed edges are in C^0.*

6. *For $0 \leq h < r$, suppose vertex occurrence v_h in T_{j_h} is the cause of vertex v_{h+1} via Step 9. Let w denote the white endpoint of the relevant unused edge. Then the occurrence of w in $T_{j_{h+1}}$ is in $C^{h+1}_{j_{h+1}}$ and both endpoints of the above unused edge are in C^{h+1} (see Fig. 8 part b).*

7. *For $0 \leq h < r$, suppose vertex occurrence v_h in T_{j_h} is the cause of vertex v_{h+1} via Step 9, and suppose v_{h+1} is black. Unless the vertex occurrence v_{h+1} represented in the pair (v_{h+1}, j_{h+1}) is a leaf, all leaf occurrences of vertex v_{h+1} in T_1 (and there exists at least one such leaf by Lemma 2) and all vertex occurrences on their respective partial superedges (the white endpoints inclusive) are in C^{h+1}_1; further these white endpoints are themselves in $C^{h+1}_{j_{h+1}}$ in tree $T_{j_{h+1}}$ (see Fig. 8 part b).*

8. *Suppose vertex v_0 is a black vertex on a seed edge. Unless the vertex occurrence v_0 represented in the pair (v_0, j_0) is a leaf in T_1 (i.e., $j_0 = 1$), all leaf occurrences of vertex v_0 in T_1 (and there exists at least one such leaf by Lemma 2) and all vertex occurrences on their respective partial superedges (the white endpoints inclusive) are in C^0_1; further these white endpoints are themselves in $C^0_{j_0}$ in tree T_{j_0}.*

Proof. Part 1 follows from the fact that v_h causes v_{h+1}. Part 2 follows from Step 7 of Algorithm 2. Part 3 follows from the ordering of entry into C in Step 7 and the corresponding order of vertex processing in Step 4. Part 4 follows from the order of vertex processing in Step 4, i.e., whites are processed before blacks. Consider Part 5: If v_0 is a white endpoint of a seed superedge/edge, then part 5 follows from the initialization in Steps 1 and 2 (note that v_0 cannot be the vertex u from Step 2); and if v_0 is black, then part 5 follows from the initialization in Step 1 and the order of vertex processing in Step 4, i.e., whites are processed before blacks. Part 6 stems from the fact that w enters C before v_{h+1} does, the order of vertex processing in Step 4 (i.e., whites are processed in order of entry into C and before the blacks), and the fact that v_{h+1} can be used in Step 4 only after entering C. The first claims in Parts 7 and 8 follow from the ordering in the for loop

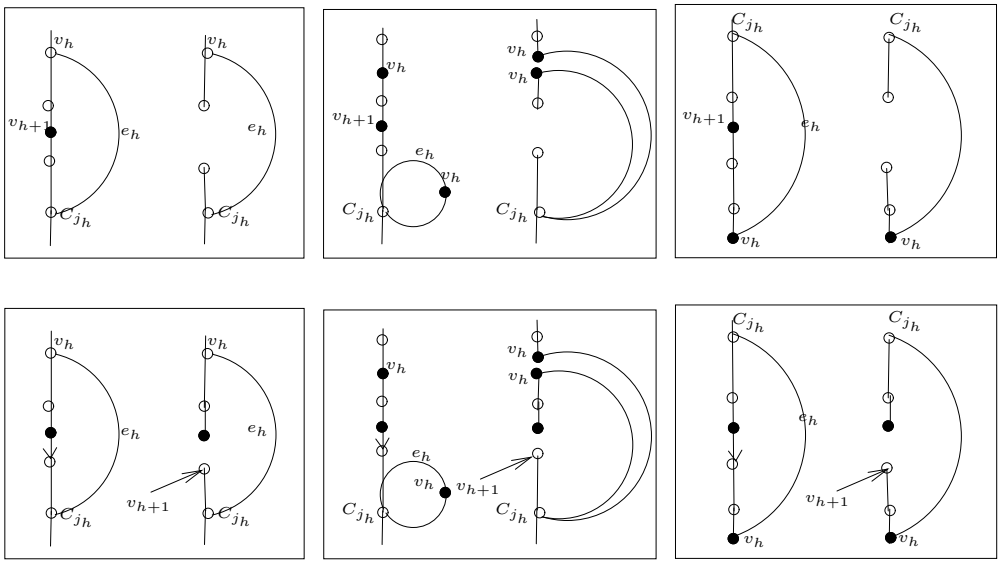

Figure 9: Swap, Mate-Swap, Join-Swap, Swap+Incidence, Mate-Swap+Incidence, Join-Swap + Incidence

after Step 4 (i.e, leaves first) along with Step 7 where all vertex occurrences on these partial superedges will be added to C_1 before the vertex occurrence v_{h+1} is processed in $T_{j_{h+1}}$. The last claims in Parts 7 and 8 follow from the order of vertex processing in Step 4, i.e., whites are processed before blacks. □

8.1.2 Converting the Trace Sequence to a Component-specific Transformation Sequence

Consider the trace sequence $\mathcal{T}$ for $Comp$ as above. The goal of this section is to convert $\mathcal{T}$ to a component-specific transformation sequence. We do this by processing the items in $\mathcal{T}$ in order, but skipping the very last item (v_r, j_r). That item will be handled in Phase 2. For each item (v_h, j_h), $0 \leq h < r$, we will do the following: given a free superedge/edge e_h containing v_h, we will show how one transformation operation can be performed using e_h to free another superedge/edge e_{h+1} containing v_{h+1}. The algorithm for this is shown in Algorithm 3, and unfortunately, there are several cases, with each case performing one of the transformation operations listed in Section 5.3. The main cases are shown in Fig. 9.

Correctness Overview. Note that proving correctness of Algorithm 3 requires an elaborate case analysis. We take one case for illustration. Suppose v_h is white, e_h is a superedge (with endpoint v_h) and v_{h+1} is caused via Step 7 (see the first drawing in Fig. 9). Then Step 1 does not apply. Step 2 applies and prescribes a swap. For a valid swap, we need to show that a) e_h has one endpoint in $C^h_{j_h}$, and b) there is a superedge containing v_{h+1} on the path from vertex occurrence v_h to $C^h_{j_h}$ in $T^h_{j_h}$. We will assume that the first part holds inductively, using what we call Invariant P1 below. For the second part, note that by the definition of a trace sequence, such a superedge indeed existed before any transformations were made. Why is it still in existence? We will assume inductively that these previous transformations leave the portion of T_{j_h} outside $C^h_{j_h}$ untouched, using what we call Invariant P2 below. We will then show that Invariants P1 and P2 hold through iterations of the for loop.

Working with Edges. Another complication to note is that e_h could be an edge or a superedge. Suppose it is an edge and not a superedge. If vertex occurrence v_h in T_{j_h} is a leaf (so $j_h = 1$) then leaving e_h as is is fine

Algorithm 3 Converting a trace sequence to a component-specific transformation sequence.

0. $e_0 \leftarrow$ Seed superedge/edge containing v_0.
for each item (v_h, j_h), $0 \leq h < r$ in order **do**
  if e_h is an edge, v_h is black, and vertex occurrence v_h in T_{j_h} is not a leaf occurrence **then**
    1. $e_h \leftarrow$ *JOIN* e_h with the partial superedge in T_1 incident on some leaf occurrence of v_h.
  end if
  if v_h causes v_{h+1} via Step 7 **then**
    $t \leftarrow$ Superedge containing v_{h+1} on the path from vertex occurrence v_h to $C^h_{j_h}$ in T_{j_h}.
  else if v_h causes v_{h+1} via Step 9 involving white vertex w **then**
    $t \leftarrow$ Edge directed into w on the path from vertex occurrence v_h to $C^h_{j_h}$ in T_{j_h}.
  end if
  if v_h is white **then**
    2. *SWAP* e_h for t in T_{j_h}.
  else if v_h is black and e_h is an edge **then**
    3. *JOIN-SWAP* e_h for t in $T_{j_h} = T_1$.
  else if v_h is black and e_h is a superedge **then**
    4. *MATE-SWAP* e_h for t in T_{j_h}.
  end if
  if v_h causes v_{h+1} via Step 7 **then**
    $e_{h+1} \leftarrow t$
  else if v_h causes v_{h+1} via Step 9 involving white vertex w **then**
    5. $e_{h+1} \leftarrow$ *INCIDENCE* t for the unused edge directed into w from v_{h+1}.
  end if
end for
if e_r is an edge, v_r is black, and vertex occurrence v_r is not a leaf occurrence **then**
  6. $e_r \leftarrow$ *JOIN* e_r with the partial superedge in T_1 incident on some leaf occurrence of v_r.
end if

because it can perform a join-swap operation with the partial superedge associated with above occurrence of v_h. However, if vertex occurrence v_h in T_{j_h} is not a leaf (so $j_h = 1$) then we would like to convert e_h to a superedge so it can perform swaps or mate-swaps; hence Steps 1 and 6 in Algorithm 3.

Invariant P1. e_h satisfies the following properties for all h, $0 \leq h \leq r$.

1. If vertex v_h is white then e_h is a superedge for which one endpoint has an occurrence $C^h_{j_h}$ and v_h is the other endpoint. The one special case is when $h = r$ and $Comp$ is white-terminal-9 (see the trace sequence definition for this special case), in which case j_r is undefined; in that case, e_r is a superedge for which one endpoint is in C^r and v_r is the other endpoint.

2. If vertex v_h is black then there are two cases:

   2.1. Either e_h is a superedge containing an occurrence of v_h and with both white endpoints having occurrences in $C^h_{j_h}$ (and therefore both endpoints are in C^h as well).

   2.2. Or, e_h is an unused edge directed from vertex v_h to some white vertex w with an occurrence in $C^h_{j_h}$ (vertex w is therefore in C^h as well); this case holds only when v_{h-1} causes v_h via Step 9, or if $h = 0$ and e_h is a seed edge.

Invariant P2. Consider transformations made to free e_h in Algorithm 3, for all h, $0 \leq h \leq r$.

- All changes made to tree T_i, $1 \leq i \leq k+1$, are localized within C_i^h, which stays contiguous even after these changes.
- If e_h is an edge with black endpoint v_h, then leaf occurrences of v_h (in T_1) are intact.
- The only unused edges affected by this process are those with both endpoints in C^h.
- Seed superedges for components other than $Comp$ are untouched.

Lemma 24. *Invariants P1 and P2 hold at the beginning of Algorithm 3, at the end of each iteration of the for loop, and at the very end.*

Proof. We show this by induction.

The Base Case. Consider Step 0. e_0 is defined by Step 0 and v_0 has to be in e_0 by the definition of a trace sequence. Invariant P1 follows from Lemma 23, part 5. Invariant P2 clearly holds because no changes are made to the trees, and the only unused edge affected, if any, is a seed edge with both endpoints in C^0 (Lemma 23, part 5).

The Inductive Step. Assuming Invariants P1 and P2 hold at the beginning of iteration h, $0 \leq h < r$. Consider iteration h. We show that the invariants hold at the end of Step 1, at the end of whichever of Steps 2, 3, 4 or 2+5, 3+5, 4+5 is performed, and at the end of Step 6 if $h = r-1$.

Consider Step 1 which converts an edge e_h to a superedge under the mentioned conditions. By Invariant P1, e_h is unused. The new e_h is a superedge in this case. The join operation is valid: v_h had a leaf occurrence in T_1 to begin with (Lemma 2), and this occurrence of v_h is still in T_1 (obvious for $h = 0$ because no transformations have been made; and by Invariant P2 for $h \geq 1$). Invariant P1 is maintained: the new e_h clearly contains v_h, and both endpoints of e_h are in $C_{j_h}^h$ (Lemma 23, parts 7 and 8). Invariant P2 is maintained: the relevant portions of T_1 modified are in C_1^h (Lemma 23, parts 7 and 8), which is still contiguous.

Consider Step 2. e_{h+1} is a superedge in this case. The swap operation is valid: e_h has endpoints v_h and in $C_{j_h}^h$ (Invariant 1), the path from vertex occurrence v_h to $C_{j_h}^h$ in T_{j_h} is untouched by previous transformations (Invariant 2), and vertex v_{h+1} is on the above path (by the definition of cause via Step 7). Invariant P1 is maintained: e_{h+1} clearly contains v_{h+1}, and all endpoints of e_{h+1} (other than possibly v_{h+1}) are inside $C_{j_{h+1}}^{h+1}$ (Lemma 23, parts 3 and 4). Invariant P2 is maintained: changes made to T_{j_h} are restricted to the path between v_h and $C_{j_h}^h$, which lies within $C_{j_h}^{h+1}$ (Lemma 23, part 2); these changes clearly maintain contiguity of $C_{j_h}^{h+1}$ as well.

Consider Step 3. e_{h+1} is a superedge in this case. By the condition for Step 1, the vertex occurrence v_h is a leaf occurrence, and therefore in T_1. Therefore $j_h = 1$. The join-swap operation is valid: the path from leaf occurrence v_h to $C_{j_h}^h$ in T_{j_h} is untouched by previous transformations (Invariant 2), and vertex v_{h+1} is on the above path (by the definition of cause via Step 7). Invariant P1 is maintained: e_{h+1} clearly contains v_{h+1}, and all endpoints of e_{h+1} (other than possibly v_{h+1}) are inside $C_{j_{h+1}}^{h+1}$ (Lemma 23, parts 3 and 4). Invariant P2 is maintained: changes made to T_{j_h} are restricted to the path between v_h and $C_{j_h}^h$, which lies within $C_{j_h}^{h+1}$ (Lemma 23, part 2); these changes clearly maintain contiguity of $C_{j_h}^{h+1}$ as well.

Consider Step 4. e_{h+1} is a superedge in this case. The mate-swap operation is valid: e_h contains black v_h and has both endpoints in $C_{j_h}^h$ (Invariant 1), the path from vertex occurrence v_h to $C_{j_h}^h$ in T_{j_h} is untouched by previous transformations (Invariant 2), and vertex v_{h+1} is on the above path (by the definition of cause

via Step 7). Invariant P1 is maintained: e_{h+1} clearly contains v_{h+1}, and all endpoints of e_{h+1} (other than possibly v_{h+1}) are inside $C^{h+1}_{j_{h+1}}$ (Lemma 23, parts 3 and 4). Invariant P2 is maintained: changes made to T_{j_h} are restricted to the path between v_h and $C^h_{j_h}$, which lies within $C^{h+1}_{j_h}$ (Lemma 23, part 2); these changes clearly maintain contiguity of $C^{h+1}_{j_h}$ as well.

Consider Steps 2+5. e_{h+1} could be a superedge or an edge depending upon whether or not v_{h+1} is white. The swap operation is valid as in Step 2. The incidence operation is valid: the unused edge e_{h+1} is not affected by previous transformations (Invariant P2 and Lemma 23, part 1). Invariant P1 is maintained: e_{h+1} is directed into a white vertex w whose occurrence in $T_{j_{h+1}}$ is in $C^{h+1}_{j_{h+1}}$ (Lemma 23, part 6). The one technical special case for Invariant P1 in subcase 1 is when $h+1 = r$ and j_r is undefined in the white-terminal-9 case; in this case, the endpoint w of e_{h+1} is in C^{h+1} (Lemma 23, part 2). Invariant P2 is maintained: changes made to T_{j_h} are restricted to the path between v_h and $C^h_{j_h}$, which lies within $C^{h+1}_{j_h}$ (Lemma 23, part 2), these changes clearly maintain contiguity of $C^{h+1}_{j_h}$ as well, if e_{h+1} is an edge then leaf occurrences of v_{h+1} (in T_1) are intact (Lemma 23, part 1), and e_{h+1} has both endpoints in C^{h+1} (Lemma 23, part 6).

The validity proofs for Steps 3+5 and 4+5 are easily seen to be combinations of the proofs for Steps 3 and 4, respectively, along with the proof for Steps 2+5. The proof for Step 9 is identical to that for Step 1. The lemma follows. □

Corollary 1 will be the starting point for Phase 2 in Section 8.3.

Corollary 1. *For any mature component $Comp$, e_r satisfies the following properties.*

1. *If vertex v_r is white then e_r is a superedge for which one endpoint has an occurrence in $C^r_{j_r} = C_{j_r}$ and v_r is the other endpoint. For the white-terminal-9 case, one endpoint is in C and the other is v_r which is outside $Comp$.*

2. *If vertex v_r is black then there are two cases.*

   2.1. *Either e_r is a superedge containing an occurrence of v_r and with both endpoints having occurrences in $C^r_{j_r} = C_{j_r}$.*

   2.2. *Or, only in the event that $j_r = 1$ and the vertex occurrence v_r referred to in the item (v_r, j_r) is a leaf occurrence in T_1, e_r is an unused edge directed from vertex v_r to some white vertex w with an occurrence in $C^r_{j_r} = C_{j_r}$.*

3. *All changes made to tree T_i, $1 \leq i \leq k+1$, to free e_r are localized within C^r_i, which stays contiguous even after these changes.*

4. *The only unused edges affected are those with both endpoints in C^r.*

5. *Seed superedges for components other than $Comp$ are unaffected.*

Proof. Note that $C^r_{j_r} = C_{j_r}$ and $C^r = C$ because no changes happen to these sets when the last item (v_r, j_r) is processed. The claims follow from Lemma 24, Invariants P1 and P2, and Step 6 of Algorithm 3 (the latter for item 2.2 above). □

Corollary 2. *Phase 1 can be performed independently for all mature components. The total time taken for $Comp$ is bounded by the time for extended closure computation which adds up to $O(nk)$ over all components.*

Proof. By Lemma 22, for a given i, $1 \leq i \leq k+1$, C_i^r's for the various components are disjoint. Also note that the C^r's are disjoint as well when restricted to white vertices. The independence claim follows from parts 3, 4 and 5 of Corollary 1. The time spent essentially mimics the trace sequence which is a subset of the time spent in extended closure computation. □

8.2 Phase 1: Component-specific Transformation Sequence for Premature Component $Comp$

The challenge with premature components is that the above procedure for obtaining a component-specific transformation sequence may not yield a superedge which will connect $Comp$ to another component or even a superedge which can be made to connect to another component with $O(1)$ more transformations. One option is to not stop but continue with Algorithm 2 until one of the termination conditions for maturity applies (so $Comp$ terminates as mature). However, Lemma 22 gets violated then. So stopping prematurely or continuing to maturity both pose problems.

Our solution in this case is to observe that we can effectively combine the trace sequences of $Comp$ with that of $Comp'$ (where $C_i^{Comp'}$ contained the vertex occurrence causing premature termination of Algorithm 2 for $Comp$) to obtain a trace sequence that terminates with the same conditions that the trace sequence for a mature component would terminate with. We will need to then prove Lemma 23 for this trace sequence in order to claim that the construction of transformation sequences in Section 8.1.2 continues to work. To this effect, we devise an extension of Algorithm 2 described in Algorithm 4 below. The latter algorithm kicks in when Algorithm 2 terminates prematurely and it starts off exactly where Algorithm 2 left off. The goal of Algorithm 4 is the same as that of Algorithm 2 with the following notable difference: Algorithm 4 will consider only those black vertex occurrences in Step 5 which are in $C_j^{Comp'}$ for the tree T_j being processed. For tree T_1, we will process all black leaf occurrences as well in addition. And Steps 8 and 9 are not needed any longer We will show that this resulting computation will indeed terminate in maturity. The resulting trace sequence will comprise a prefix from Algorithm 2 and a suffix from Algorithm 4; and any item (v_h, j_h) in the latter will have the property that v_h is either white, a black leaf, or it is a black vertex occurrence in $C_{j_h}^{Comp'}$. We will then show that these trace sequences indeed satisfy Lemma 23. Lemma 22 is still violated though; but that is easily fixed by keeping one of $Comp, Comp'$ away from further processing, so no transformation sequences are obtained for that component; we will still be left with a constant fraction of the components after we discard such components.

Algorithm. We start by taking pairs $Comp, Comp'$ as above and dropping a constant fraction of components from all future processing so at most one item from each pair survives. For all remaining items $Comp$ we run Algorithm 4 starting with the states of C and C_i's as they were when Algorithm 2 for $Comp$ terminated, and starting by processing precisely the same vertex occurrence that caused the above termination (though this initialization has not been explicitly spelled out in Algorithm 4). Algorithm 4 then evolves these sets further. Note the key changes in Algorithm 4: Steps 8 and 9 are gone as well as the condition for premature termination; but most importantly, the handling of blacks in the inner for loop is different: only blacks in C with either a leaf occurrence of an occurrence in $C_i^{Comp'}$ are processed. Lemma 25 shows that the algorithm does indeed terminate resulting in a mature component.

Lemma 25. *Algorithm 4 for a premature component $Comp$ terminates; the time taken is $O(nk)$ over all components.*

Proof. Since $Comp$ is premature, Algorithm 2 provides a black vertex b, some occurrence of which is present in C, and an occurrence b_{r+1} of which is also present in $C_{j_{r+1}}^{Comp'}$ for some tree $T_{j_{r+1}}$ (the seemingly unnecessary subscripts $r+1, j_{r+1}$ are for future convenience).

Algorithm 4 Sequel to Extended Closure Computation for Premature Component $Comp$

repeat
  for each tree $T_i, 1 \leq i \leq k+1$ in order **do**
    3. Let X comprise white vertices in C outside C_i, and black vertices in C which have vertex occurrences outside C_i that are either leaf occurrences or that are inside $C_i^{Comp'}$.
    4. Order vertices in X so whites come first, blacks later, and each set is further sorted in increasing time order of entry into C.
    for each vertex $v \in X$ in order and every occurrence v' of v in T_i (if v is black then provided v' is a leaf or v' is in $C_i^{Comp'}$; occurrences are considered inn arbitrary order except in T_1 where leaf occurrences, if any, are considered first) **do**
      5. If the path between C_i and v' contains a white vertex outside $Comp$ terminate the algorithm.
      6. If v is black and has an occurrence in the seed superedge for some component $Comp'$ then terminate the algorithm.
      7. Add all white and black vertex occurrences between C_i and v' to C_i in order of increasing distance from C_i and add the corresponding vertices to C in the same order.
    end for
  end for
until eternity

Now consider Algorithm 2 for component $Comp'$ and the trace sequence (for component $Comp'$) which leads to vertex b_{r+1} entering $C^{Comp'}$. Let this sequence be $(b_0, j_0), (b_1, j_1), \ldots, (b_r, j_r)$, where b_0 enters $C^{Comp'}$ by initialization, and vertex occurrence b_h in tree T_{j_h} causes vertex b_{h+1} for $0 \leq h \leq r$, (see the definition of *cause* in the description of a trace sequence in Section 8.1.1). In this sequence, consider the largest value of l, $0 \leq l \leq r$, for which vertex occurrence b_l in tree T_{j_l} causes vertex b_{l+1} via Step 9 of Algorithm 2; and if no such l exists set $l = -1$. And let z denote the white vertex with which Algorithm 2 initialized $C_i^{Comp'}$'s. Then, for vertex occurrences b_h, $l+1 \leq h \leq r$, there exists an occurrence of b_{h+1} on the path to vertex z in T_{j_h}, and this entire path is in $C_{j_h}^{Comp'}$.

Now switching to Algorithm 4 on $Comp$. Since $b_{r+1} \in C$ at the beginning of Algorithm 4, it follows that vertices $b_{r+1}, b_r, \ldots, b_{l+1}$ will eventually enter C in Algorithm 4 (unless Algorithm 4 terminates before that, in which case the lemma holds anyway). If any of these vertices is white then Algorithm 4 terminates in Step 7, because these whites belong to $Comp'$. So assume all these vertices are black. There are two cases now. If $l = -1$ and $Comp'$ had a seed superedge, then that superedge contains $b_{l+1} = b_0$ and Algorithm 4 terminates in Step 6. So consider the case that $l >= 0$, or $l = -1$ and $Comp'$ has no seed superedge (in which case $b_{l+1} = b_0$ comes from a seed edge for $Comp$). In either case, we show below that a leaf occurrence of b_{l+1} in T_1 and its nearest white ancestor both belong to $C_1^{Comp'}$, which will cause termination in Step 5 when b_{l+1} is processed in T_1.

To see this, switch back to Algorithm 2 on $Comp'$ and recall the leaf first order for T_1 in the inner for loop. Once b_{l+1} enters $C^{Comp'}$ (which it does because an occurrence of b_{l+1} is in $C_{j_{l+1}}^{Comp'}$, even if $l = r$), it will first be processed in tree T_1. If the first leaf occurrence of b_{l+1} processed in T_1 succeeds in its traversal to the then $C_1^{Comp'}$, then we are done. Otherwise, if this traversal fails, then an occurrence of b_{l+1} cannot be in $C_i^{Comp'}$ for any tree T_i, which is a contradiction.

It remains to determine the time complexity. The main change from Algorithm 2 is in Step 3 and it suffices to show how this step can be performed in time $O(|X|)$. This is done in a manner analogous to the queues in Lemma 20; we need to access queues in the next tree T_i which has an occurrence of the relevant

black within $C_i^{Comp'}$. This can be accomplished with appropriate book-keeping. The lemma follows. □

Trace sequences can be defined as before. We show below that Lemma 23 holds for these trace sequences.

Lemma 26. *Lemma 23 holds for trace sequences of premature components $Comp$ obtained by running Algorithm 4 after Algorithm 2.*

Proof. Let the trace sequence for $Comp$ be

$$(v_0, j_0), (v_1, j_1), \ldots, (v_{r-1}, j_{r-1}), (v_r, j_r), \ldots, (v_{t-1}, j_{t-1}), (v_t, j_t)$$

where (v_{r-1}, j_{r-1}) is the last item that comes from Algorithm 2 and all subsequent items starting with (v_r, j_r) come from Algorithm 4. By Lemma 23, parts 1, 2, 3, 4, 6 and 7 in Lemma 23 hold for h, $0 \leq h < r-1$ (we use $r-1$ and not r here because the case of $h = r-1$ can be impacted by the processing of both v_{r-1} and v_r, which are done by different algorithms), and parts 5 and 8 hold for $h = 0$ provided $r > 0$. So it remains to show parts 1, 2, 3, 4, 6 and 7 for h, $r-1 \leq h < t$, and parts 5 and 8 for $h = 0$ in the event that $r = 0$.

The same proofs as before are easily seen to hold in Parts 1, 2, 3 for all h, $r-1 \leq h < t$. The same is true for Part 5 in the case $h = r = 0$. Parts 6 and 7 do not apply to $r-1 < h < t$ because Step 9 is missing in Algorithm 4; so these parts need to be shown only for $h = r-1$. The proof of part 6 for $h = r-1$ is the same as before as well. So that leaves only parts 7 and 8 to be shown for cases $h = r-1$ and $h = r = 0$, respectively. The same proof below handles both of these parts.

Note that these parts apply when v_r is black, caused via Step 9 or via seed edge initialization, and the vertex occurrence v_r in T_{j_r} is not a leaf occurrence. After Algorithm 2 adds vertex v_r to C, it will process all leaf occurrences of v_r in T_1 before processing vertex occurrence v_r in T_{j_r}. If Algorithm 2 terminates before this happens then since Algorithm 4 continues where Algorithm 2 left off and since Algorithm 4 is also allowed to process leaf black occurrences and processes those first over other occurrences of v_r in T_1, all leaf occurrences of v_r in T_1 will indeed be processed before processing vertex occurrence v_r in T_{j_r}. If Algorithm 4 also terminates before this happens then vertex occurrence v_r in T_{j_r} will never be processed, a contradiction. The first claim in parts 7 and 8 follow. The second claim in parts 7 and 8 follow since whites are still processed before blacks by the ordering in Step 4.

The lemma follows. □

Lemma 27. *Lemma 22 continues to hold after Algorithm 4 has run on all surviving premature components.*

Proof. Suppose sets $C_i^{Comp'}$ and $C_i^{Comp''}$ are not disjoint, i.e., they share a black vertex occurrence b in common. Then only two cases are possible. Either b is a leaf or Algorithm 2 for one of these components, say $Comp'$, terminated prematurely because it processed a black vertex occurrence inside $C_j^{Comp''}$ for some tree T_j. In the latter case, only one of these two components survives. So consider the former case. By virtue of Step 5 in both algorithms, Algorithm 2 and Algorithm 4, the leaf occurrence b will enter both $C_1^{Comp'}$ and $C_1^{Comp''}$ only if the the white endpoint of the partial superedge incident on b is in both these components, a contradiction. The lemma follows. □

To complete the description, note that Lemma 19 is easily seen to hold. Since Lemmas 22, 19 and 23 all hold, the machinery in Section 8.1.2 applies as well. We are thus left with a constant fraction of the original set of components, all of which are mature, and Corollaries 1 and 2 apply to these.

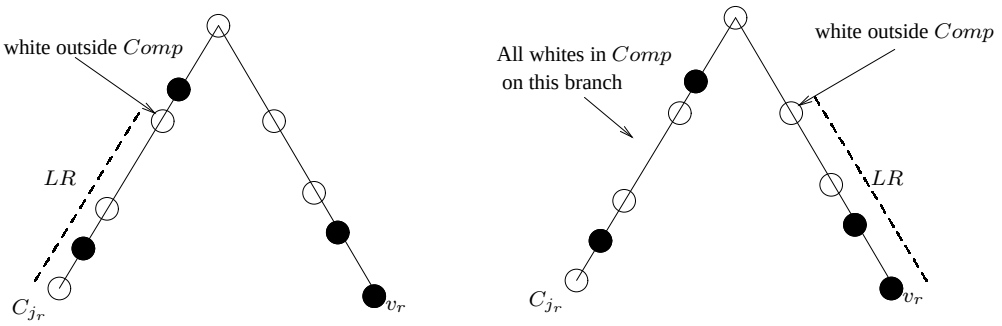

Figure 10: Two cases for the definition of a last stretch; the former is c-biased and the latter v-biased.

8.3 Phase 2: Completing the Transformation Sequence

This section considers all surviving components which comprise a constant fraction of the original set of components. To control interactions between these components, we will need to further prune away this surviving set of components in $O(nk)$ time to retain a further constant fraction, as described in Section 8.3.1. Phase 1 will be performed only *after* this pruning step and only on the surviving components, resulting in a component-specific transformation sequence for each of these components with the properties laid out in Corollary 1. Note that by Corollary 2, all of these Phase 1 computations can proceed independently. The goal next is to perform further transformations so the number of connected components in T_{k+1} reduces by a constant fraction, as shown in Section 8.3.2. In fact, each surviving component will be connected to some other component as a result of the transformations performed in Section 8.3.2. This will take time proportional to the number of components in play.

8.3.1 Pruning Components

We need the following definitions. Recall (v_r, j_r) is the last item in the trace sequence for $Comp$. As usual, we use explicit superscripts to denote components other than $Comp$ (for instance $v_r^{Comp'}$ is vertex occurrence v_r for $Comp'$).

Last Stretches. For a white-terminal-5 component, define the last stretch LR as follows. Traverse up from v_r and from C_{j_r} towards their least common ancestor, stopping each traversal when the first white vertex outside $Comp$ is encountered (which will happen by the definition of a white-terminal-5 component; also recall no transformations have been performed yet). If both traversals encounter such white vertices then pick the traversal that starts with a vertex with lower postorder number, otherwise pick the only traversal that encounters a white vertex outside $Comp$; the stretch of vertex occurrences traversed by this traversal, both endpoints inclusive, is denoted by LR. See Fig. 10. Note that LR terminates on a white vertex outside $Comp$. We say that LR (and component $Comp$ as well) is v-biased if the corresponding traversal began at v_r, and c-biased otherwise. Note that computing these last stretches requires knowing the least common ancestor, which can be determined in $O(1)$ time after linear time precomputation on trees $T_1, \ldots, T_{k+1}$.

The goals of pruning are listed in Lemmas 28, 29, 30, 31 and 32. We use the following pruning rules which still retain a constant fraction of the components; the time taken is linear in the number of components.

Pruning Rule 1. We say that two components $Comp, Comp'$ *clash* if either the last stretch of $Comp$ terminates on a white vertex in $Comp'$, or the vertex occurrence v_r for component $Comp$ (from the last

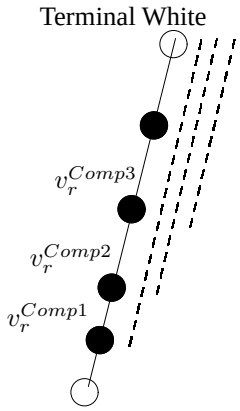

Figure 11: Overlapping last stretches are within a superedge.

item in its trace sequence) is part of a superedge with an endpoint in $Comp'$, or vertex occurrence v_r is part of $C_{j_r}^{Comp'}$ in tree T_{j_r}, or $Comp$ is black-terminal and $Comp'$ is the component whose seed superedge has an occurrence of v_r causing the termination of extended closure computation on $Comp$. Since the number of clashing pairs is linear in the number of components, we can find a constant fraction of clash-free components in time linear in the number of components.

Pruning Rule 2. We partition surviving white-terminal-5 components into two groups, those which have v-biased last stretches and those which have c-biased last stretches; we retain only components in the larger group.

Pruning Rule 3. If the last stretches for multiple components terminate at the same white vertex occurrence, we discard the one with the longest stretch.

Lemma 28. *For all white-terminal-5 components $Comp$ with last stretch LR in tree, say T_i, edges in LR are outside $C_i^{Comp'}$, as is the terminal white vertex of LR, for any component $Comp'$ (inclusive of $Comp$). Further, the terminal*

Proof. The lemma follows from pruning rule 1 and the definition of a last stretch. □

Lemma 29. *For any pair of white-terminal-5 components, if their respective last stretches have an edge in common then both stretches must be strictly contained within one superedge and both must have the same terminal white vertex (see Fig. 11).*

Proof. Follows from pruning rule 1, pruning rule 3, and the fact that two stretches sharing an edge can have at most one white vertex in common. □

Lemma 30. *Either all white-terminal-5 components are v-biased or all are c-biased.*

Proof. Follows from pruning rule 2. □

Lemma 31. *Vertex occurrence v_r for a white-terminal-5 component $Comp$ is not part of the last stretch for any other white-terminal-5 component unless all surviving components are v-biased.*

Proof. By Lemma 30, it suffices to consider the case when all white-terminal-5 components are c-biased. Suppose for a contradiction that v_r is part of the last stretch of white-terminal-5 component $Comp'$. Then the superedge containing v_r must have an endpoint in $Comp'$. The lemma then follows by pruning rule 1. □

Lemma 32. *Consider a white-terminal-5 component $Comp$ that is v-biased. If the post-order number of C_{j_r} is less than that of v_r in T_{j_r}, then the last stretch for any other white-terminal-5 component $Comp'$ cannot terminate on a white vertex on the path π from C_{j_r} to the least common ancestor of C_{j_r}, v_r in T_{j_r}.*

Proof. By the definition of a last stretch, all white vertices on the path π are in $Comp$. The lemma now follows from pruning step 1. □

8.3.2 Completing the Transformation Sequence

We do this in two steps. The first step considers a subset of the white-terminal-5 components and performs transformations on these. The second step considers all components to complete the procedure. At the end, each component will be connected to some other component as a result of the transformations performed here. The time taken is proportional to the number of components.

Step 1. By Lemma 29, we form equivalence classes of white-terminal-5 components, where a non-singleton equivalence class is contained within the same superedge. We pick a representative from each class, namely the one with the longest last stretch; we then process all class representatives together in this step. Note that the last stretches of these representatives are completely edge-disjoint by this construction and by Lemma 29.

Consider a class representative component $Comp$ and let e denote the superedge/edge released for $Comp$ in Phase 1 as per Corollary 1. The additional transformations to be performed for $Comp$ are illustrated in Fig. 12, depending upon which of the cases 1, 2.1, 2.2 from Corollary 1 holds for e. By the definition of a last stretch, the resulting superedge freed in each case connects $Comp$ to another component. The following lemma shows that these additional transformations are indeed valid, in spite of transformations performed in Phase 1 and in spite of other class representative components processed before $Comp$ in Step 1. The time taken will be linear in the number of components.

Lemma 33. *After Phase 1 and after all previous representative components have been processed in Step 1, LR for $Comp$ is still intact and is present on the path from vertex occurrence v_r to C_{j_r} in T_{j_r}.*

Proof. By Lemma 19, Corollary 1 and Lemma 28, LR is intact after Phase 1 and still appears on the path from v_r to C_{j_r} in T_{j_r}. LR is still intact after all previous representative components have been processed in Step 1 above because the last stretches of all representative components chosen here are edge-disjoint and because of Lemma 31 and Lemma 28. It remains to show that LR is still on the path from v_r to C_{j_r} in T_{j_r} after all previous representative components have been processed in Step 1 above.

To see this, consider processing representative components in the order defined below (clearly, components with last stretches on trees other than T_{j_r} are uninteresting). For any particular component $Comp'$, let x denote the terminal white vertex of $LR^{Comp'}$ and let y denote its child such that $LR^{Comp'}$ has the edge (x, y). Order components in increasing post-order number of their corresponding y's. The description below assumes that $Comp'$ is the first component in this order (see Fig. 13).

We show in the next paragraph that the subtree rooted at y contains neither $C_{j_r}^{Comp''}$ nor $v_r^{Comp''}$, for any component $Comp'' \neq Comp'$. It follows that we can remove the edge (x, y) and the whole subtree of T_{j_r} rooted at y to get a smaller tree without affecting the processing for any of the components other than

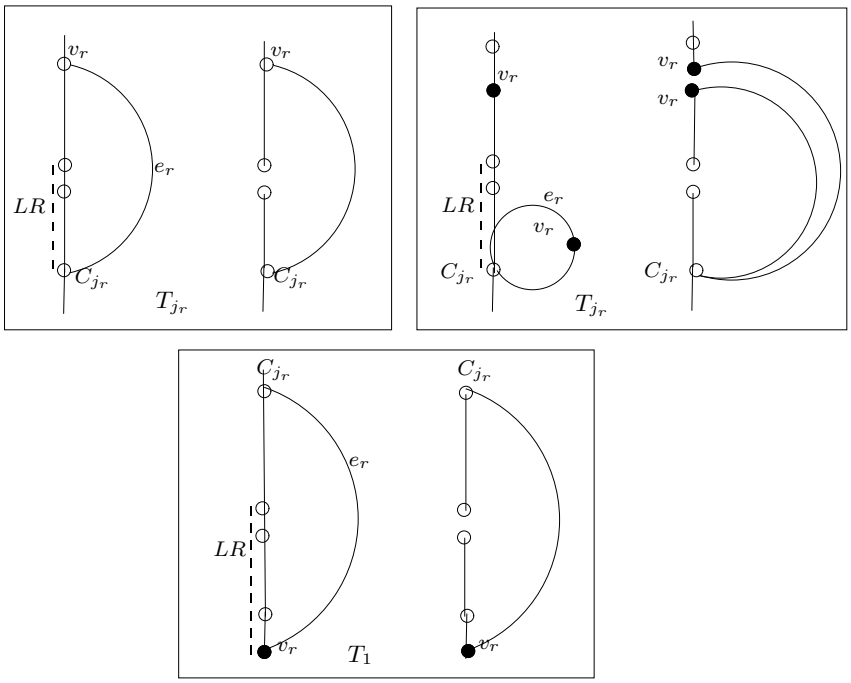

Figure 12: Additional transformations for $Comp$: Swap, Mate-Swap and Join-Swap, based on which of cases 1, 2.1 and 2.2 hold for e_r in Corollary 1.

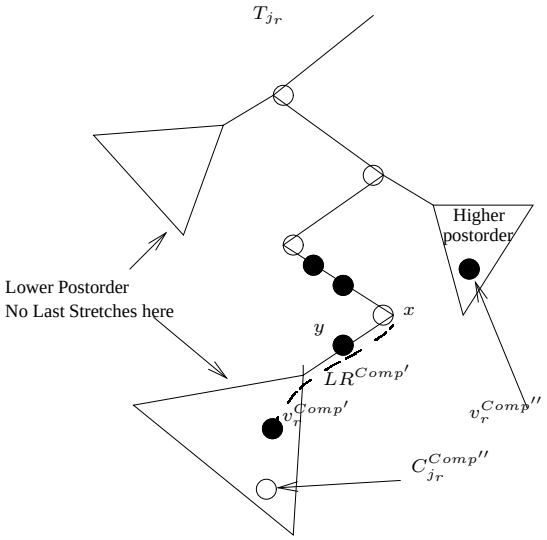

Figure 13: $LR^{Comp'}$, x and y

$Comp'$. We process these components recursively to get a new tree T'_{j_r}. Now we put back edge (x, y) and the subtree rooted at y; clearly $LR^{Comp'}$ is still on the path from $v_r^{Comp'}$ to $C_{j_r}^{Comp'}$ in this resulting tree, as required.

It remains to show that the subtree rooted at y contains neither $C_{j_r}^{Comp''}$ nor $v_r^{Comp''}$, for some component $Comp'' \neq Comp'$. Without loss of generality, assume all representative components are v-biased (see Lemma 30; a similar argument holds when all these components are c-biased). Then $v_r^{Comp''}$ cannot be in the subtree rooted at y, otherwise $LR^{Comp''}$ is present completely in this subtree as well and then $Comp''$ would precede $Comp'$ in the order established above. So suppose for a contradiction that $C_{j_r}^{Comp''}$ is in this subtree. There are two cases based on the post-order number of $v_r^{Comp''}$. If this number is less than that of x, then either $Comp''$ must precede $Comp'$ in the above specified order or $LR^{Comp''}$ must extend beyond the least common ancestor of $v_r^{Comp''}, C_{j_r}^{Comp''}$, both of which present contradictions. So consider the case that the post-order number of $v_r^{Comp''}$ is greater than that of x, and therefore that of $C_i^{Comp''}$ as well. $v_r^{Comp''}$ cannot be an ancestor of y, otherwise $Comp''$ is not v-biased. By the definition of a last stretch, it follows that all white vertices on the path from $C_{j_r}^{Comp''}$ to the least common ancestor of $v_r^{Comp''}, C_{j_r}^{Comp''}$ must be in $Comp''$. In particular, x is in $Comp''$, which contradicts Lemma 32. This completes the proof of the lemma. □

Step 2. Now consider any equivalence class of white-terminal-5 components, and all non-representative components in this class. Let $e_1, \ldots, e_h$ denote their respective edges/superedges as defined in Corollary 1, and let $a_1, \ldots, a_h$ denote the vertex occurrences involved in the last items in their respective trace sequences. Note that each e_i has a white endpoint in its respective component. By Lemma 29, $a_1, \ldots, a_h$ are all black. And by Corollary 1, e_i has an instance of a_i, for all i, $1 \leq i \leq h$. Also define e_0 to be the free superedge obtained for the representative component of this class in Step 1 above, which connects that component to another. Note that by the definition of a representative, e_0 has an instance of each of $a_1, \ldots, a_h$. Also note that some of the e_i's ($i \geq 1$) could be edges and not superedges; for each such edge e_i, there must be an associated leaf occurrence of the corresponding a_i in some tree (by Lemma 2); we can do a join of the partial superedge incident on this leaf occurrence with e_i to convert e_i to a superedge containing a_i. Now we have only superedges left. First, some of these superedges could have endpoints in distinct components, so those are done. Second, pair up and mate as many remaining e_i's ($i \geq 1$) as possible so the a_i's are identical within a pair; each of the resulting superedges has endpoints in distinct components; these are done as well. This leaves us with e_0 and a subset of other e_i's with distinct a_i's. We now perform mates among these as in Fig. 14. The resulting superedges connect together all of these components in question.

Second, consider any equivalence class of black-terminal components so that the associated free superedges/edges specified by Corollary 1 all have black vertices in common with the seed superedge of the same component $Comp$. Let e_0 denote that superedge. e_0 is still intact at the end of Phase 1 by pruning step 1 in Section 8.3.1 and by Corollary 1. The same process as in the previous paragraph suggests transformations which will connect all of these components together with $Comp$.

Finally, for white-terminal-9 components, nothing further need be done because their associated superedges as specified by Corollary 1 connect across components. This completes the algorithm.

9 A fast Gomory-Hu tree algorithm

In this section we present our fast algorithm for computing a Gomory-Hu tree. The algorithm uses submodularity of cuts (Fact 1) and Theorem 7 which follows from Fact 1.

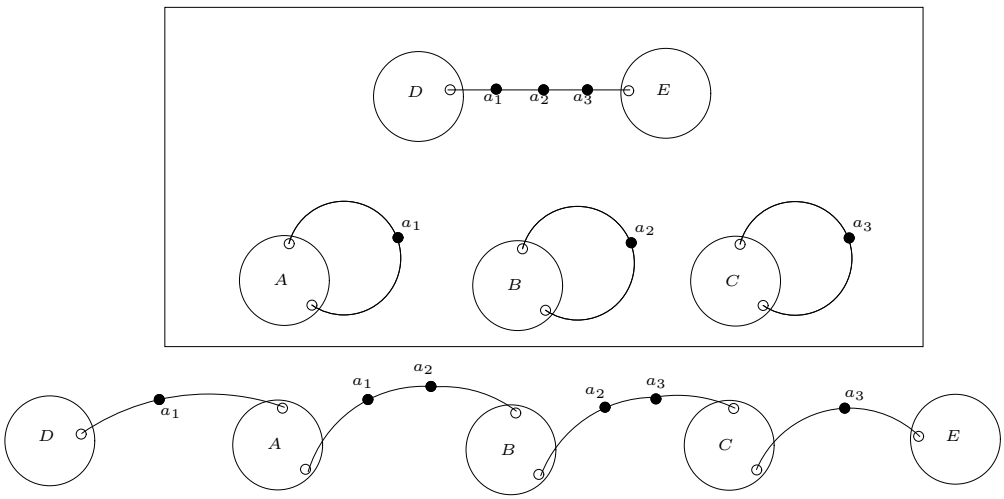

Figure 14: A Cascade of Mates.

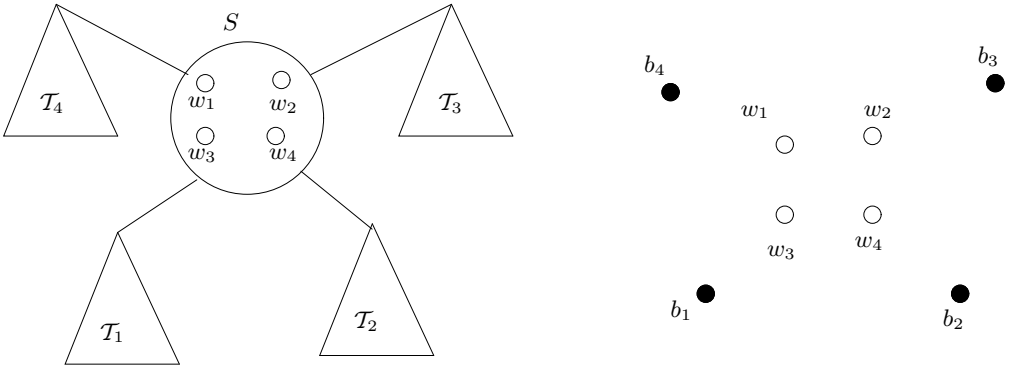

Figure 15: The current (partial) Gomory-Hu tree $\mathcal{T}$ on the left and $G(S)$ for a node S in $\mathcal{T}$ on the right; note only vertices in $G(S)$ are shown, edges are not shown.

The Gomory-Hu tree construction algorithm [GH61] initializes the cut tree $\mathcal{T}$ to a single node that contains the entire vertex set. As we proceed, nodes of $\mathcal{T}$ will partition the vertices of G. Each node S in $\mathcal{T}$ represents a collection of vertices $v(S)$ from G. Consider node S and the subtrees $\mathcal{T}_1 \ldots \mathcal{T}_h$ subtended at the h neighbors of node S in $\mathcal{T}$. We define $v(\mathcal{T}_i) = \cup_{S' \in \mathcal{T}_i} v(S')$. Obtain a new graph $G(S)$ (called the *relevant graph* for S) from G by compressing all vertices in $v(\mathcal{T}_i)$ for each i (see Fig. 15); this graph has h compressed vertices and $|v(S)|$ singleton vertices; the latter comprise the set of terminal vertices; for simplicity, we will use S itself to denote the set $v(S)$.

The algorithm now proceeds as follows. Repeatedly pick a node S of $\mathcal{T}$ containing more than one vertex from G and consider $G(S)$. Pick any two terminal vertices s, t in $G(S)$ and find the s-t min-cut (possibly via a max flow computation) in the graph $G(S)$. Theorem 7 ensures that the s-t min-cut thus obtained (we call it C) is also a s-t min-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.

Our Approach. We use the same framework as above but modify the processing of $G(S)$ as follows. Instead of repeatedly choosing s-t pairs as above, we use Theorem 10 to obtain the following in one shot: for each terminal v in $G(S)$, the minimal min-cut in $G(S)$ separating v from a root vertex r, where r is chosen uniformly at random from the terminals in $G(S)$. We now refine S in $\mathcal{T}$ by inserting each of the minimal min-cuts from the above family, as described below. Once all these minimal min-cuts have been inserted, the procedure continues by picking another node of $\mathcal{T}$ with more than one vertex, until all nodes are singleton.

Minimal Min-Cut Witnesses. For every minimal min-cut $(B, V(G(S)) - B)$ found above with respect to root r, there exists a terminal vertex v in $G(S)$ such that $v \in B$, $r \in V(G(S)) - B)$ and no subset of B containing v is a min-cut separating v from r. All such terminal vertices v are said to be *witnesses* for B and are denoted by $w(B)$.

Laminar Families. Note that any two minimal min-cuts are either disjoint or contained one within the other by Fact 1. So a collection of minimal min-cuts forms a laminar family $\mathcal{F}$, where the outermost cut is the trivial cut which has all vertices inside it and all further nested cuts are the actual minimal min-cuts with respect to the chosen root r. For instance, in Fig. 16, the outermost cut has all vertices $w_1 \ldots w_4$ and $b_1 \ldots b_4$, and there are 3 nested cuts, each a minimal min-cut with respect to root w_4. In general, the level of nesting could be arbitrary.

Refining Node S in $\mathcal{T}$. In the current Gomory-Hu tree $\mathcal{T}$, we replace the node S with the following tree structure obtained from the laminar family $\mathcal{F}$ as follows. Create a new tree node $\beta(B)$ for each cut B in $\mathcal{F}$. Associate with this node, vertices in the set $w(B)$. For the outermost cut B, $w(B)$ is defined to contain only the root r. For each non-terminal vertex b immediately inside a cut B, add an edge from node $\beta(B)$ to the root of the $\mathcal{T}_i$ (where b was obtained by compressing vertices associated with nodes in $\mathcal{T}_i$); this edge has the same weight as the edge from S to $\mathcal{T}_i$ in $\mathcal{T}$. If B has a parent cut B' which contains B in the laminar family then $\beta(B)$ has an edge to $\beta(B')$ of weight equal to the size of the cut $(B, V(G(S) - B)$ in $G(S)$. Fig. 16 illustrates the resulting tree. The figure on the left shows a cut B consisting of one terminal w_4 and one non-terminal b_4 and 3 child cuts, each consisting of one terminal w_i and a non-terminal b_i, for $i = 1, 2, 3$. On the right we have the corresponding laminar tree. It is easy to see that this new Gomory-Hu tree is the old tree $\mathcal{T}$ augmented with exactly the minimal min-cuts found above, as required.

Analysis. Let us partition the various subproblems (these are elements of the queue Q) that we spawn in our algorithm into *layers*. $Layer(0)$ consists of the problem (V, G). $Layer(1)$ consists of the subproblems $(B, G(B))$ that correspond to all the minimal min-cuts generated by the problem (V, G) in $Layer(0)$. Recursively, $Layer(i)$ consists of the subproblems corresponding to all the minimal min-cuts generated by all the problems in $Layer(i-1)$. Note that corresponding to each layer $Layer(i-1)$, there exists a corresponding (partial) Gomory-Hu tree; refining each node in this tree yields the Gomory-Hu tree corresponding to $Layer(i)$.

Our first claim is that the total time complexity of the problems involved in the same layer is $O(mc \log n)$, where $c = max_{u,v \in V} c(u, v)$. Such a claim would be immediate if it were the case that all the problems involved in the same layer were edge-disjoint. But that is not the case and the same edge might be present in various subproblems in the same layer. However we will show that the total number of occurrences of all edges summed over all the subproblems in the same layer is $O(m)$.

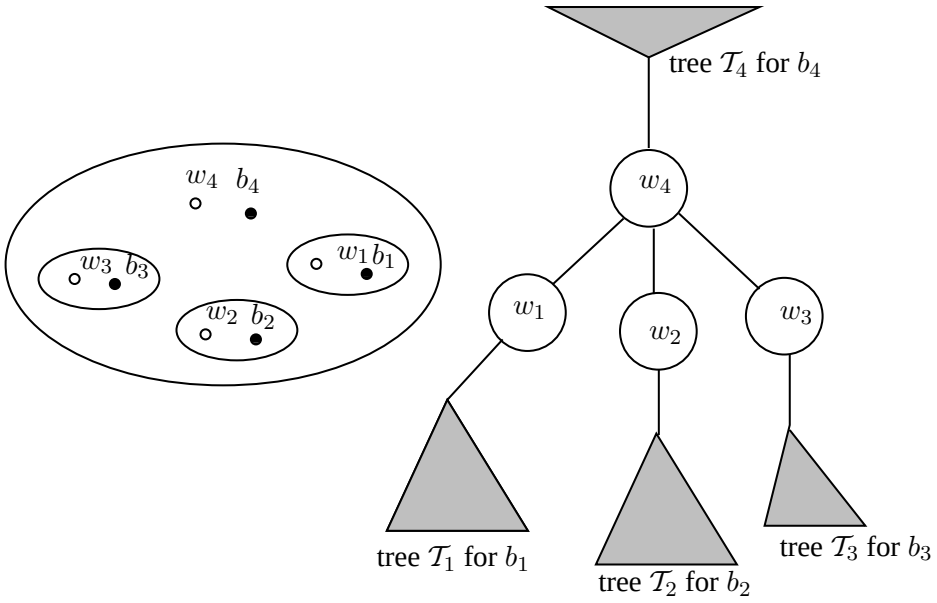

Figure 16: The laminar family of a cut and the corresponding laminar tree.

Algorithm 5 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 set (V, G). {*Any element in the queue is a pair (set of terminals, the relevant graph).*}
while the queue Q is not empty **do**
  – delete the first element $(S, G(S))$ from Q.
  if $|S| > 1$ **then**
    – pick a vertex in S uniformly at random as the root r.
    – identify the laminar family of all minimal min-cuts with respect to r in $G(S)$.
    – refine S in $\mathcal{T}$ by inserting each of these cuts as described earlier.
    – for each of the above cuts B, add the element $(B, G(B))$ to the Q, where $G(B)$ is obtained from $G(S)$ by contracting the vertices of $G(S) - B$ to a single vertex.
  end if
end while

Lemma 34. *The total number of all edges involved in all subproblems in the same layer is $O(m)$.*

Proof. The edges that are present in more than one problem in $Layer(i)$ correspond to edges that cross the cuts present in the (partial) Gomory-Hu tree $\mathcal{T}'$ after all $Layer(i-1)$ problems have been processed. We need to sum over all edges e, the number of cuts of $\mathcal{T}'$ that any edge e crosses. This is equivalent to summing the total sizes of the cuts in $\mathcal{T}'$. This sum is bounded from above by $c_1 + c_2 + \cdots + c_{n-1}$, where $c_1, \ldots, c_{n-1}$ are the weights of the $n-1$ edges of the final 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 endpoints of e in $\mathcal{T}$. It is easy to see that l is an one-to-one mapping. Now, for any edge $e = (u, v) \in \mathcal{E}$, the weight, $w(e)$, of e in the Gomory-Hu tree $\mathcal{T}$ is $c(u, v)$. Without loss of

generality, assume that $l(e) = u$. Now, since $(u, V \setminus u)$ represents a u-v cut of size $\deg(u)$, it follows that $w(e) = c(u, v) \leq \deg(u) = \deg(l(e))$. Summing over all the edges in $\mathcal{E}$ and noting that the function l is one-to-one, we have $\sum_{e \in \mathcal{E}} w(e) \leq \sum_{v \in V} \deg(v) = 2m$. □

Since the cost of computing all minimal min-cuts with respect to the root in any subproblem is $O(m'c \log n)$ where m' is the number of edges in that subproblem (by Theorem 10), Lemma 34 immediately implies that the total complexity of the problems involved in the same layer is $O(mc \log n)$. Now we show that with probability $1 - 1/n$, the number of layers is $O(\log n)$. We first prove the following lemma.

Lemma 35. *Let $(C, G(C))$ be any particular subproblem. Let $(P, G(P))$ be its parent problem. Let $|P|$ (similarly, $|C|$) denote the number of terminal vertices in the set P (resp., C). Then $Pr(|C| \geq |P|/2) \leq 1/2$.*

Proof. In the subproblem $(C, G(C))$, C is a minimal min-cut that separates the root r in the set P (of the parent problem $(P, G(P))$) from some terminal vertex $v \in C$. Since this is a minimal min-cut, the side containing v in C has connectivity at least one higher than the size of this r-v cut. If all the terminals of P are arranged in non-decreasing order of their connectivity from v, then the probability that the root r is one of the first $|P|/2$ vertices in this order (only then can the number of terminals in C be at least $|P|/2$) is at most $1/2$, since r was chosen uniformly at random from the $|P|$ terminals. This proves the lemma. □

Theorem 11. *With probability $1 - 1/n$, the number of layers is $O(\log n)$.*

Proof. Consider the path π from a leaf subproblem $(C_\ell, G(C_\ell))$ in $Layer(\ell)$ to the root problem ((V, G) in $Layer(0)$). Each edge in this path π is from a subproblem $(C_j, G(C_j))$ in $Layer(j)$ to its parent problem $(C_{j-1}, G(C_{j-1}))$ in $Layer(j-1)$. Define a random variable X_j as follows: X_j is 1 if $|C_j| \geq |C_{j-1}|/2$, 0 otherwise.

First, observe that X_j's are independent random variables since the root is chosen independently in each layer. Thus using Chernoff bound, the probability that a given subproblem is in layer $\ell > 4 \log n$ is at most $1/n^2$. Since there are at most n leaves, using the union bound, the number of layers is $O(\log n)$ with probability $1 - 1/n$. □

This leads to the following theorem, which gives the time complexity of the algorithm.

Theorem 12. *With probability $1-1/n$, the time complexity of our Gomory-Hu tree algorithm is $O(mc \log^2 n)$.*

10 Conclusion and Future work

We gave a deterministic algorithm for the Steiner edge connectivity problem in undirected/Eulerian directed graphs that runs in $\tilde{O}(m + nc^2)$ time, where c is the edge connectivity of the Steiner set $S \subseteq V$ and n, m are the number of vertices and edges in the input graph. We applied this algorithm to design a faster algorithm for the Gomory-Hu tree problem in undirected graphs. This algorithm has a running time of $\tilde{O}(mF)$ with high probability, where F is the maximum u-v edge connectivity, over all pairs of vertices u, v.

One obvious challenge is to derandomize our Gomory-Hu tree algorithm and achieve similar time bounds. Another question is whether the Steiner edge connectivity problem can be solved in Monte Carlo near-linear time (independent of the Steiner connectivity value). Another important question is that of extending our approach to design faster algorithms for building Gomory-Hu trees in *weighted* graphs. Our Gomory-Hu tree algorithm indeed works for integer weighted graphs too, however a running time of $\tilde{O}(mF)$, where F is the maximum u-v edge connectivity, over all pairs of vertices u, v, is not interesting (and is not even polynomial time) for weighted graphs.

11 Acknowledgments

We thank the anonymous referees for their feedback.

References

[Ben95] András A. Benczúr. Counterexamples for directed and node capacitated cut-trees. *SIAM J. Comput.*, 24(3):505–510, 1995.

[BHKP07] Anand Bhalgat, Ramesh Hariharan, Telikepalli Kavitha, and Debmalya Panigrahi. An õ(mn) gomory-hu tree construction algorithm for unweighted graphs. In *STOC*, pages 605–614, 2007.

[BHKP08] Anand Bhalgat, Ramesh Hariharan, Telikepalli Kavitha, and Debmalya Panigrahi. Fast edge splitting and edmonds' arborescence construction for unweighted graphs. In *SODA*, pages 455–464, 2008.

[BJFJ95] Jørgen Bang-Jensen, András Frank, and Bill Jackson. Preserving and increasing local edge-connectivity in mixed graphs. *SIAM J. Discrete Math.*, 8(2):155–178, 1995.

[CH03] Richard Cole and Ramesh Hariharan. A fast algorithm for computing steiner edge connectivity. In *STOC*, pages 167–176, 2003.

[DV94] Yefim Dinitz and Alek Vainshtein. The connectivity carcass of a vertex subset in a graph and its incremental maintenance. In *STOC*, pages 716–725, 1994.

[Edm69] Jack Edmonds. Submodular functions, matroids, and certain polyhedra. In *Calgary International Conference on Combinatorial Structures and their Application*, pages 69–87, 1969.

[Edm72] Jack Edmonds. Edge disjoint branchings. pages 91–96, 1972.

[Gab95] Harold N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. *J. Comput. Syst. Sci.*, 50(2):259–273, 1995.

[GH61] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *J. Soc. Indust. Appl. Math.*, 9(4):551–570, 1961.

[GT01] Andrew V. Goldberg and Kostas Tsioutsiouliklis. Cut tree algorithms: An experimental study. *J. Algorithms*, 38(1):51–83, 2001.

[Gus90] Dan Gusfield. Very simple methods for all pairs network flow analysis. *SIAM J. Comput.*, 19(1):143–155, 1990.

[HKP07] Ramesh Hariharan, Telikepalli Kavitha, and Debmalya Panigrahi. Efficient algorithms for computing all low $s - t$ edge connectivities and related problems. In *SODA*, pages 127–136, 2007.

[KL02] David R. Karger and Matthew S. Levine. Random sampling in residual graphs. In *STOC*, pages 63–66, 2002.

[WGMV95] David P. Williamson, Michel X. Goemans, Milena Mihail, and Vijay V. Vazirani. A primal-dual approximation algorithm for generalized steiner network problems. *Combinatorica*, 15(3):435–454, 1995.