A Fast Algorithm for Computing Steiner Edge Connectivity[*]

Richard Cole
Courant Institute
New York University
NY, NY 10012
cole@cs.nyu.edu

Ramesh Hariharan [*]
Indian Institute of Science
Bangalore
ramesh@csa.iisc.ernet.in

ABSTRACT

Given an undirected graph or an Eulerian directed graph G and a subset S of its vertices, we show how to determine the edge connectivity $\mathcal{C}$ of the vertices in S in time $O(\mathcal{C}^3 n \log n + m)$. This algorithm is based on an efficient construction of tree packings which generalizes Edmonds' Theorem. These packings also yield a characterization of all minimal Steiner cuts of size $\mathcal{C}$ from which an efficient data structure for maintaining edge connectivity between vertices in S under edge insertion can be obtained. This data structure enables the efficient construction of a cactus tree for representing significant $\mathcal{C}$-cuts among these vertices, called $\mathcal{C}$-separations, in the same time bound. In turn, we use the cactus tree to give a fast implementation of an approximation algorithm for the Survivable Network Design problem due to Williamson, Goemans, Mihail and Vazirani.

Categories and Subject Descriptors

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

General Terms

Algorithms

Keywords

Steiner points, cactus trees, edge-connectivity

1. INTRODUCTION

The *global edge connectivity* of a directed graph is defined as the minimum number of edges whose removal destroys the strong connectivity of the graph. In many graphs, the edge connectivity of different vertices may vary considerably. An extreme example is provided by directed graphs with vertices u and v connected by $\mathcal{C}$ directed edge-disjoint paths; u and v have edge connectivity $\mathcal{C}$, but the graph has edge connectivity 1. This paper is concerned with *Steiner edge connectivity*, in which a set of *terminal* vertices is specified, and the minimum number of edges with the following property is sought: removing these edges splits the graph into at least 2 strongly connected components in such a way that the terminal vertices are spread over two or more components. Similarly, on an undirected graph, one seeks a minimum number of edges whose removal splits the graph into at least two connected components, spreading the terminal vertices over two or more components. This problem was previously studied by Dinitz and Vainshtein [5].

Global Connectivity. For general directed graphs, the best algorithm for determining global connectivity is due to Gabow [12] and takes $O(\mathcal{C}_{min} m \log \frac{n^2}{m})$ time, where $\mathcal{C}_{min}$ is the global connectivity. For undirected graphs, the same result holds (typically, one just converts an undirected graph to directed by orienting the edges in both directions) with the additional improvement that m can be held down to $O(\mathcal{C}_{min} n)$ using a construction due to Nagamochi and Ibaraki [20]. Gabow's result is based on an efficient construction of spanning tree packings, as opposed to previous approaches which were based on network flows and Menger's theorem; the best of these had running time $O(\min\{\mathcal{C}_{min}^2 n^2, mn\})$ on directed graphs and $O(\mathcal{C}_{min} n^2)$ on undirected graphs.

Thus, the spanning tree packing approach led to the first sub-quadratic (in n) algorithm for determining global connectivity. This approach revolves around two classical theorems by Edmonds [8, 7], stated below. Here, an *r-arborescence* is a directed spanning tree rooted at a specified root vertex r with all edges directed away from r, an *r-cut* is the set of edges directed from $V - S$ to S, where S is any subset of the vertices not containing r, and a *directionless r-spanning tree* is like an arborescence but with the weaker constraint that only edges incident on r must be directed away from the root.

Edmonds' Theorem[8]: The maximum number of edge disjoint r-arborescences equals the minimum cardinality of an r-cut.

Edmonds' Relaxed Theorem[7]: The maximum number of edge disjoint r-arborescences in a directed graph equals the maximum number of edge disjoint directionless r-spanning trees with the property that each vertex $v \neq r$ has total in-degree $\mathcal{C}_{min}$ over all these r-spanning trees.

*This work was supported in part by NSF grant CCR-0105678.
*Work partly done while visiting NYU and while on leave at Strand Genomics Pvt. Ltd.

STOC'03, June 9–11, 2003, San Diego, California, USA.

In [8], Edmonds gave an algorithmic proof of the first theorem above, which seems to need exponential time in the worst case. Subsequent proofs were given by Tarjan[21], Frank [10], Lovasz [19], Tong and Lawler [22], and Gabow [12], the last of which reduced the running time to $O(\mathcal{C}_{min}^2 n^2)$, where $\mathcal{C}_{min}$ is the size of the minimum r-cut. Note that all of these are quadratic in n. The advantage of the second relaxed theorem above is that it admits fast algorithms. Gabow [12] showed that, in contrast to arborescences, directionless spanning trees with the above description could be constructed in time sub-quadratic in n, i.e., in time $O(\mathcal{C}_{min} m \log \frac{n^2}{m})$. This construction was the basis of his fast global connectivity algorithm.

Steiner Connectivity. The goal of this paper is to generalize this packing approach to the Steiner case to get a fast (i.e., sub-quadratic in n) Steiner connectivity algorithm. To address the above problem, we need to go beyond packings aimed at capturing only the minimum r-cuts in G and capture higher order connectivity information, for instance, the minimum r-cut separating r from a particular vertex v; clearly, this cut could be larger than the minimum r-cut. Our generalized notion of tree packings is aimed at capturing this higher order connectivity information; however, the associated theorems and algorithms will apply only to directed graphs that are *Eulerian*. A special case of such a graph will be the Eulerian directed graph obtained from an undirected graph by directing the edges in both directions. Thus, our algorithms will indeed apply to all undirected graphs. In fact, as Lemma 1 notes, as regards cuts, undirected and Eulerian graphs are equivalent.

LEMMA 1. *Let $G = (V, E)$ be a Eulerian graph and let G' be the corresponding undirected graph in which each edge is made undirected. There is a cut of cardinality h in G separating vertex sets C and $V - C$ if and only if there is a cut of cardinality $2h$ in G' separating C and $V - C$.*

Let $con(v)$ denote the maximum number of edge disjoint paths from the special root vertex r to vertex v. The following theorem appears in [2], 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 the Edge-Splitting lemma of Lovasz [19] and does not immediately lead to an efficient algorithm.

The Tree Packing Theorem. Given a Eulerian directed graph G, there exist $\mathcal{C} = max_{v \neq r}\{con(v)\}$ edge-disjoint directed trees rooted at r such that each vertex $v \neq r$ in G appears in exactly $con(v)$ trees (these trees need no longer contain all vertices in G; the directions of edges in a tree are all away from the root).

The above theorem clearly implies its relaxed version stated below.

The Relaxed Tree Packing Theorem: Given a Eulerian directed graph G, there exist $\mathcal{C}$ edge-disjoint directionless trees, $T_1, T_2, \ldots, T_{\mathcal{C}}$, rooted at r such that each vertex $v \neq r$ in G appears exactly $\min\{\mathcal{C}, con(v)\}$ times over all trees (occurring multiple times in a tree possibly) and has in-degree exactly $\min\{\mathcal{C}, con(v)\}$ over these trees.

$T_1, T_2, \ldots, T_{\mathcal{C}}$ are called *packing trees* if they instantiate the relaxed tree packing theorem.

Our contribution. Our main contribution is a fast constructive proof of the above relaxed theorem. Our construction runs in time $O((\mathcal{C}^3 n + \mathcal{C}m) \log n))$. For undirected graphs $G = (V, E)$, the construction of Nagamochi and Ibaraki applies; with $O(m)$ preprocessing, this construction finds a subset $E' \subseteq E$ of $O(\mathcal{C}n)$ edges such that for each pair v, w of vertices, the connectivity of v and w in the subgraph $G' = (V, E')$ is $min\{\mathcal{C}, con(v, w)\}$. Thus it suffices to run the connectivity algorithms on G', improving the time bound to $O(\mathcal{C}^3 n \log n + m)$. This improvement can be extended to Euler graphs by applying Lemma 1.

Our algorithm could be viewed as a generalization of Gabow's algorithm [12] for directionless spanning tree generation mentioned above which runs in time $O(\mathcal{C}_{min} m \log \frac{n^2}{m})$. But, the generalization is non-trivial as we explain next. The crux of Gabow's algorithm is an iterative procedure which gathers a subset of vertices and finally claims that these vertices occur contiguously in all the packing trees being constructed. This argument breaks down when not all the vertices are present in each of these trees. Our algorithm manages to retain this property but only with the help of a key, non-trivial relaxation: we ensure that a vertex which does not appear in all the trees always appears with degree *no more than 2*, possibly at the cost of appearing *several times* in each tree. Allowing vertices to appear several times in a tree and thereby ensuring degree at most 2 turns out to be key in obtaining time sub-quadratic in n in several steps of the algorithm.

The Steiner connectivity $\mathcal{C}$ of a specified collection of vertices is an easy by-product of the above algorithm. Further, the trees constructed also give a characterization of the Steiner minimum cuts which will be useful in the following application.

Cactus Trees. The cactus tree for an undirected graph G, devised by Dinits et al. [4], represents the $\mathcal{C}_{min}$-cuts of $G = (V, E)$, where G is $\mathcal{C}_{min}$-edge connected, but not $\mathcal{C}_{min} + 1$-edge connected. For odd $\mathcal{C}_{min}$, this cactus tree is a tree; for even $\mathcal{C}_{min}$, it can also include cycles.

We start by describing the structure for odd $\mathcal{C}_{min}$. Let r be an arbitrary vertex of G. Given a cut $F \subseteq E$ of cardinality $\mathcal{C}_{min}$ which partitions the vertices into sets $C, V - C$, with $r \in V - C$, we name the cut using vertex set C. Every node of the cactus tree except the root corresponds to a distinct cut and is labelled by the vertex set name for this cut. Cuts associated with pairs of nodes in this tree are either disjoint or contained one inside the other. Node ancestry in the cactus tree simply corresponds to set containment for the cut names. Oftentimes, the cactus tree edge from node C to its parent is labelled by the edge set (of cardinality $\mathcal{C}_{min}$) forming this cut.

For even $\mathcal{C}_{min}$, cuts of cardinality $\mathcal{C}_{min}$ can overlap, but if C and D are two such distinct non-disjoint cuts and neither is contained inside the other then $C-D, D-C, C\cap D, C\cup D$ are also cuts with cardinality $\mathcal{C}_{min}$. In general, this yields chains of cuts $C_1 \ldots C_r$ such that $\cup_{i=p}^{q} C_i$ is a cut of cardinality $\mathcal{C}_{min}$ for any $1 \leq p \leq q \leq r$. Nodes corresponding to $C_1, \ldots, C_r$ are created, forming a cycle together with the "parent" node to which C_1 and C_r are attached.

We turn to minimum cuts for a collection $S \subseteq V$ of $\mathcal{C}$-connected vertices. We are interested in restrictions of these cuts to subsets of S. As Dinitz and Vainshtein [5] note (and credit to each of Naor and Westbrook), these cuts have exactly the same structure as the global minimum cuts above and so can also be represented by cactus trees. They show how to construct such a cactus tree using $|S|-1$ maxflow computations. We give an algorithm that starts with $\mathcal{C}$ packing trees, and then runs in time $O((\mathcal{C}n+m)\log n)$; this is bounded by $O(\mathcal{C}n\log n+m)$ using the construction of Nagamochi and Ibaraki mentioned above.

An Application to Designing Survivable Networks. In the *The Uniform Survivable Network Problem*, each vertex v of a given undirected graph has an associated non-negative integral label r_v, which is usually a small constant in practice. 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. This problem has applications to the design of fiber-optic telecommunication networks [13], and a more complete discussion of the problem appears in Grotschel et al. [16].

Williamson, Goemans, Mihail and Vazirani (WGMV, for short) [23] gave a combinatorial algorithm for this problem with an approximation factor of $2\max_v\{r_v\}-1$. Actually, this algorithm also works for the non-uniform case, i.e., when each pair of vertices has an associated demand for a certain number of edge disjoint paths. However, our results will apply only to the uniform case. We mention that there are algorithms achieving better approximation factors even for the non-uniform case, notably the algorithm due to Jain [17]; however, this algorithm requires linear programming and is therefore much slower.

The WGMV algorithm performs several iterations of the Goemans-Williamson clustering procedure [15], a general technique (which builds on an earlier technique due to Agrawal, Klein and Ravi [1]) which forms the core of several algorithms and is described below. Broadly, an iteration will comprise several rounds, in each of which the algorithm will identify some subsets of vertices as *active* and some as *inactive* and choose two such subsets to merge into one using an edge addition step. Finally, only a subset of the edges added over all rounds in an iteration will be retained. To implement an iteration, three issues need to be addressed: which sets are active/inactive in each round, which two subsets need to be merged in each round, and which edges must be discarded in the final pruning step in each iteration.

The original Goemans-Williamson paper [15] addressed these issues only for the first iteration and the associated algorithm took $O(n^2\log n)$ time. This was improved to $O(n\sqrt{m}\log n)$ by Klein [18], to $O(n^2+n\sqrt{m\log\log n}))$ time by Gabow, Goemans and Williamson [13], to $O(n\sqrt{m})$ by Gabow and Pettie [14], and to $O((n+m)\log^2 n)$ time by Cole, Hariharan, Lewenstein and Porat [3] (the last algorithm also adds an arbitrarily small additive term to the approximation factor). Computing active sets for the first iteration of the WGMV algorithm is straightforward and therefore, all the above algorithms focus on determining which two subsets to merge at each round within an iteration. However, for subsequent iterations of the WGMV algorithm, determining active sets is more involved and therefore, the above results do not generalize to these iterations (though [3] does show how to do this for one more iteration with the same time bounds).

For subsequent iterations of the WGMV algorithm, the implementation of [23] took $O(\max_v\{r_v\}^3n^4)$ total time over all these iterations, and the implementation due to Gabow, Goemans and Williamson [13] took $O(\max_v\{r_v\}^2n^2 + \max_v\{r_v\}n\sqrt{m\log\log n}))$ total time. Thus, no subquadratic (in n) implementation was known for the WGMV algorithm prior to this work. Further, all these algorithms were based on network flows and Menger's theorem based techniques, and not on tree packings. However, both the above algorithms work for the non-uniform case of the Survivable Network Design problem as well.

We give an implementation of the WGMV algorithm for the uniform case which uses the cactus tree and runs in time $O((\max_v\{r_v\}^4n+\max_v\{r_v\}^2m)\log n+\max_v\{r_v\}m\log^2 n)$, which improves to $O((\max_v\{r_v\}^4n\log n+\max_v\{r_v\}m\log^2 n)$, using the construction of Nagamochi and Ibaraki mentioned above. Under the practical assumption that $\max_v\{r_v\}$ is a not too large constant, the above time complexity is close to linear. Our restriction to the uniform case comes from the fact that our packings are rooted at particular vertex v (which will be chosen to be the vertex with the largest requirement value r_v). Our algorithm maintains the cactus tree under edge insertions (a problem previously studied by Dinitz and Westbrook [6]).

2. OVERVIEW OF THE ALGORITHMS

In this section, we describe the broad frameworks for Gabow's algorithm for directionless spanning tree packing, for our directionless tree packing and cactus construction algorithms, and for the WGMV Survivable Network construction algorithm.

2.1 Gabow's Algorithm

The directionless spanning trees are constructed one at a time. Given the first $i-1$ spanning trees $T_1,\ldots,T_{i-1}$, the ith tree T_i is constructed in several rounds. T_i is built from a forest initially comprising n singleton vertices. Overloading our notation, we name this forest T_i. 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 ith forest T_i by at least half, leading to an overall time of $O((n+m)\log n)$ per tree, or $O(\mathcal{C}_{min}(m+n)\log n)$ time overall (the log factor can be improved for dense graphs).

Any particular round begins with several connected components, each of which has exactly one *deficient vertex*, i.e., a vertex whose total in-degree in $T_1\ldots T_i$ is $i-1$ (all other vertices have in-degree i in $T_1\ldots T_i$). Each of these connected components gets processed in this round. Consider one such component C with a deficient vertex v. Gabow's algorithm now computes the minimum set M containing edges satisfying at least one of the following properties:

- $e\notin T_1\ldots T_i$ and is directed into v.
- e is in one of $T_1\ldots T_i$ and is in the fundamental cycle formed by some edge f in M with respect to that tree.
- $e\notin T_1\ldots T_i$ and is directed into a vertex into which some other edge in M is directed.

Clearly, computing M needs a closure-like algorithm, and Gabow shows how to perform this efficiently, i.e., in time proportional to the number of edges and vertices involved in M. More importantly, Gabow shows that this closure like algorithm has one of two possible outcomes.

- Either there exists an edge $e\in M$ which is directed from a vertex in T_i outside C to a vertex inside C. In this case, there exists a sequence of swap adjustments to $T_1 \dots T_i$ culminating in the addition of e to T_i, which ensures that v is no longer deficient; further, no new deficient vertices are created in the process, and the number of connected components is reduced by 1.

- Or, when no such edge exists in M, Gabow shows that the set S of vertices into which edges of M are directed occur contiguously in $T_1 \dots T_i$, and further, $S, V-S$ actually forms an r-cut of size $i-1$.

In the former case, the algorithm has made progress towards reducing the number of connected components, and in the latter case, the algorithm terminates as $i-1$ trees have already been constructed and $i-1$ is also the size of the min r-cut.

Of critical importance is the proof that $S, V-S$ forms an r-cut of size $i-1$. This proof is based on the following facts.

- Each vertex in S other than v has in-degree i in $T_1 \dots T_i$ and vertex v has in-degree $i-1$ in $T_1 \dots T_i$.
- Vertices in S occur contiguously in each of $T_1 \dots T_i$.
- Edges not in $T_1 \dots T_i$ but directed into a vertex in S lie completely within S. Thus, edges directed into S from $V-S$ must all be in $T_1 \dots T_i$.

An easy consequence of the first two properties is that the in-degree of S in $T_1 \dots T_i$ is at most $i-1$. The third property ensures that the in-degree of S in the whole graph is also at most $i-1$.

2.2 Directionless Tree Packing

As in Gabow's algorithm, the packing trees are constructed one at a time. Given the first $i-1$ trees with the property that each vertex v occurs in exactly $\min\{con(v), i-1\}$ trees and has in-degree exactly equal to its number of occurrences in these trees, the ith tree is constructed in several rounds as follows. Each round runs in $O(i^2n+m)$ time and reduces the number of connected components in the ith forest by at least half, leading to an overall time of $O((i^2n+m)\log n)$ for constructing T_i, or $O((\mathcal{C}^3n+\mathcal{C}m)\log n)$ time overall.

Again, any particular round begins with several connected components, each of which has exactly one *deficient vertex*, i.e., a vertex v for which $con(v) \geq i-1$ and $con(v)$ has not yet been established as being equal to $i-1$. The aim now is as before: to consider a particular connected component with a deficient vertex v and either to find a sequence of swap adjustments to $T_1 \dots T_i$ culminating in the addition of an edge e to T_i (in fact, the addition of a path, as we shall see) increasing the in-degree of v, or to find a subset S of vertices such that S has in-degree $i-1$ and $v \in S$ (so $con(v) = i-1$). The differences begin here.

First, unlike Gabow's algorithm, the algorithm cannot stop if it finds such a set S, because $\mathcal{C}$ could be bigger than $i-1$ and this demands that more trees be constructed. Finding S as above only signifies that vertices in S have $con() = i-1$ and therefore these vertices need not occur in any more trees.

Second, since not all vertices occur in all trees, the proof that the vertices in S occur contiguously in each of $T_1 \dots T_i$ no longer holds. The proof is just the observation that the closure method for computing M maintains contiguity. To see the problem, consider Fig.1 in which portions of two trees, T_1 and T_2 say, are shown. Vertex z occurs only in the first tree. The edge directions are suppressed. Suppose edge $(x, v) \in M$; then the closure algorithm would add all the vertical edges shown in T_1 to M. Thus, vertices y and u will occur eventually in S. Next, consider the point when edges from T_1 newly added to M are considered relative to T_2 by the closure algorithm. Since w and z are not present in T_2, it does not follow that the edges on the path from y to u in T_2 get added to M at this point, nor those from v to u, and therefore vertices in S need not be contiguous in T_2. Note that if z had in fact been present in T_2 then the edges on the path from y to u in T_2 would indeed be added to M.

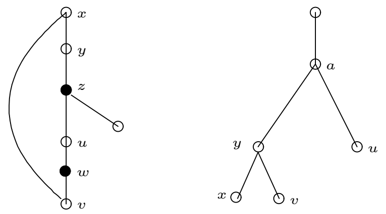

Figure 1: The Contiguity Problem: Trees T_1 and T_2

To solve the above problem, we will ensure the following critical constraint: vertices which do not occur in all trees (we call such vertices *black*) will have degree at most two (in and out combined) in each place they occur. Further, we will maintain M as a set of paths and not a set of edges (in other words, our swaps will swap out and swap in whole paths rather than single edges, where all internal vertices on such paths will be black). This will ensure the required contiguity property for vertices in S (for example, in Fig.1, assuming z has degree 2, the whole path (u, z, y) will be added to M, and when the closure algorithm considers this path with respect to T_2, both edges (y, a) and (a, u) will get added to M. To make the above machinery work, we will need that the graph be Eulerian. Further details of the algorithm appear in Section 3.

2.3 Cactus Construction

The cactus we aim to construct will have nodes representing equivalence classes of minimum cuts separating the given terminal vertices (so not all terminal vertices are on the same side of the cut). All cuts in an equivalence class split the terminal vertices in exactly the same way. Non-terminal vertices could be split in different ways by distinct cuts in the same equivalence class. We use the term *separation* to denote an equivalence class. As we will see, each equivalence class has a unique minimal cut associated with it. While describing the cactus construction algorithm, let $(i-1)$ be the cardinality of any minimum cut which separates the terminal vertices; the minimal cuts representing the various equivalence classes of such minimum cuts will be denoted by the term $(i-1)$-cuts.

The computation of the cactus tree proceeds in two phases. First, we determine those $(i-1)$-cuts found by the algorithm when computing the ith packing tree T_i. We call these cuts *visible* cuts. These visible cuts are arranged in a preliminary cactus tree according to their set containment. Unfortunately, $(i-1)$-cuts which are themselves unions of visible $(i-1)$-cuts may not be identified in this process; we call such cuts *invisible*. We need an additional procedure to find these invisible cuts and detect cycles formed by these cuts;

this procedure runs in time $O((in+m)\log n)$, as described in Section 4.

In the case that all the vertices are terminal, Gabow [11] has shown how to find the cactus from the $i-1$ packing trees in linear time.

2.4 The WGMV algorithm

The algorithm executes $\max_v\{r_v\}$ iterations, with each iteration removing a subset of the edges in the given undirected graph G and adding these edges to the final solution Q. Iteration i begins with several *violated* sets, i.e., subsets S of vertices satisfying the following properties: there exists $v \in S$ such that $r_v \geq i$ and $\delta_Q(S) = i-1$, where $\delta_Q(S)$ is the total number of edges between S and $V-S$ in Q. This iteration now has several rounds, each of which will identify exactly one edge. A violated set is *satisfied* when an edge in G connecting a vertex in S to a vertex outside S is identified (note that edges in Q are removed from G and therefore are no longer present in G). The aim of this iteration is to identify sufficiently many edges for addition to Q so that all violated sets are satisfied. However, only a minimal subset of these identified edges which can by itself satisfy all violated sets will actually be removed from G and added to Q.

Each round proceeds as follows. All currently minimal unsatisfied violated sets will be *active* in this round. These sets will then *expand* (as in the Goemans-Williamson algorithm [15]) until some edge becomes *tight*; this edge is the edge identified for this round. All active sets in a round are disjoint and the collection of all active sets over all rounds forms a hierarchical *laminar family*.

Assuming an oracle which provides the active sets in a suitable form in each round, all rounds in an iteration can together be performed in $O(m\log^2 n)$ time, using the algorithm in [3]. Further, determining the minimal subset of the identified edges in each iteration to add to Q can also be done in $O(n)$ time as shown in [13]. In this paper, we show how to implement the above oracle so that it runs in $O((i^3n + im)\log n)$ time over all the rounds in iteration i. The total time taken by the algorithm is thus $O((\max_{v\neq r}\{r_v\}^4 n + \max_{v\neq r}\{r_v\}^2 m)\log n + \max_{v\neq r}\{r_v\} m\log^2 n)$.

3. CONSTRUCTING THE PACKING TREES

Consider the ith iteration and suppose several rounds have been performed in this iteration resulting in trees $T_1 \dots T_{i-1}$ and a forest T_i.

Note that vertices in G can be partitioned into the following categories at this point: vertices v for which $con(v) < i-1$, vertices with $con(v) = i-1$ and which have already been discovered to have $con(v) = i-1$, vertices with $con(v) = i-1$ and which have not yet been discovered to have $con(v) = i-1$, and vertices for which $con(v) \geq i$. Vertices in the first two categories appear exactly $con(v) \leq i-1$ times over all trees, possibly occurring multiple times in a tree, while vertices in the last two categories occur exactly once each in each of $T_1 \dots T_i$. Vertices in the former two categories are organized hierarchically into supervertices, while vertices in the latter two categories will appear as singleton supervertices.

A supervertex will in turn be composed of other supervertices (each supervertex could be a singleton vertex by itself as well). $con(v)$ for a supervertex v is defined as the maximum of $con(w)$ over all supervertices w contained in v. At the outermost level of nesting, supervertices which occur in all trees (including r) are said to be *white* (these are exactly the category 3 and 4 singleton supervertices above) and the remaining supervertices are said to be *black*. At further levels of nesting, a supervertex w nested immediately within a supervertex v is said to be *white* if and only if every occurrence of v contains w. Supervertices satisfy the following properties.

P1. All supervertices nested within a particular supervertex occur contiguously in all the trees $T_1 \dots T_{i-1}$ (in fact, not all of these supervertices occur in all the trees, but those which do occur, occur contiguously).

P2. White supervertices are necessarily singleton and occur exactly once in each tree (at the outermost level) or exactly once in each occurrence of the *parent* supervertex, i.e., the supervertex at the next outer level of nesting. A white supervertex w nested inside a black supervertex v satisfies $con(v) = con(w)$ and a white vertex at the topmost level of nesting satisfies $con(v) = i$ or $con(v) = i-1$.

P3. Each black supervertex v appears exactly $con(v) \leq i-1$ times over all the trees, possibly occurring multiple times in a single tree. The total in-degree over all these occurrences of v is exactly $con(v)$.

P4. If a particular occurrence of a black supervertex v contains another supervertex w, then all occurrences of w occur within occurrences of v.

P5. Each black supervertex has at least one white supervertex at the next deeper level of nesting.

P6. A white supervertex v at the outermost level could have in-degree i or $i-1$ over all trees; further, if the in-degree is $i-1$ then the in-degree of v in T_i equals 0. White vertices at the outermost level of nesting with in-degree $i-1$ are called *deficient*. With the exception of the connected component containing the root r, each connected component in T_i has exactly one deficient vertex.

P7. Each black supervertex has degree at most 2. There are exactly $2 * con(w)$ edges in G that are incident on black supervertices w. On unnesting w one level, one sees it comprises one or more white vertices of connectivity $con(w)$, joined together in a tree by single edges and/or paths of black supervertices. The edges incident on w are either incident on one of these white vertices nested in w, or joined to such a nested white vertex by a path of nested black supervertices. Each of the black supervertices on this path has the same recursive structure. While w remains a top level black supervertex, each edge in G incident on w is associated permanently with such a path of black supervertices, which may be an empty path; this path is called the *attachment path*. Further, exactly two attachment paths will be associated with each instance of w. To allow an edge incident on w to change its w endpoint from one copy of w to another in $O(1)$ time, we maintain pointers to the two vertices, not supervertices, at the ends of each such path.

Our algorithms will ensure that these invariants can be maintained at the end of this round as well, with the number of components in T_i reduced by a factor of at least half. This round performs the following steps.

3.1 Algorithm for One Round

Step 1. This step will consider only supervertices at the outermost level of nesting. For each connected component C in T_i such that $r \notin C$, a set $S(C)$ of supervertices with the following properties will be determined, provided it exists.

- $S(C)$ comprises supervertices in C and possibly some black supervertices not present in T_i, as explained shortly.
- $S(C)$ contains the only deficient supervertex v in C.
- The in-degree of $S(C)$ in G equals $i - 1$.
- $S(C)$ is *white-maximal* and *black-minimal*, i.e., there is no set which satisfies the above 3 properties and has more white vertices, and removal of a black vertex from $S(C)$ causes a violation of one of the above conditions.

$S(C)$ is identified using the following criteria: Supervertices in $S(C)$ occur contiguously in the trees $T_1 \dots T_i$, and further, no edge outside of $T_1 \dots T_i$ is directed into $S(C)$ from outside. The following simple counting argument shows that these two conditions are necessary and sufficient for $S(C)$ to have in-degree $i - 1$ in G.

LEMMA 2. *Let S be a set of supervertices in G. The in-degree of S in $T_1 \dots T_i$ equals $i-1$ if the contiguity condition holds, and exceeds $i - 1$ otherwise.*

Proof. With the exception of v, each supervertex w in S has as many occurrences $\#w$ as its total in-degree over all trees. v's total in-degree is exactly one less than its number of occurrences. Thus the total in-degree over all supervertices w in S equals $(\sum_{w \in S} \#w) - 1$. Exactly $(\sum_{w \in S} \#w) - i$ of the edges which contribute to the above in-degree lie within S if the contiguity property holds; this number is even smaller if the contiguity property does not hold. This leaves exactly $i - 1$ edges directed into S in $T_1 \dots T_i$ if the contiguity property holds, and more otherwise, as required. □

The algorithmic details of finding $S(C)$ appear in Section 3.1.1. The time taken for this procedure will be $O(i^2 n + m)$. The main reasons why this computation is efficient is that by Eulerianness, the total degree (in and out combined) of $S(C)$ in G is $2(i - 1)$ and further, $S(C)$ occurs i times over all the trees $T_1 \dots T_i$; these two facts together imply that either $S(C)$ is a whole connected component in T_i or it is a leaf subtree in one of $T_1 \dots T_{i-1}$. That $S(C)$ is uniquely defined in spite of white-maximality, if it exists at all, follows from the fact that if there are two incomparable candidate sets A and B for $S(C)$ then $A \cup B$ is also a candidate. If $S(C)$ exists then it becomes a new black supervertex s, and supervertices comprising $S(C)$ are now nested within s; this is possible due to the contiguity property. The in-degree of s over all trees equals $i - 1$.

Step 2. For each component C, if $S(C)$ exists and contains all of C then C is just removed from T_i. This is fine because s needs to appear only $i - 1$ times, and it currently appears i times in $T_1 \dots T_i$. While if $S(C)$ does not exist or does not contain all of C (note the convoluted wording of this condition; this arises because $S(C)$ may contain black supervertices that do not occur in C), we will be able to connect it to another component so C no longer has a deficient vertex, as we will see. This may entail the restructuring of some or all of $T_1 \dots T_i$.

Next, we process the black supervertices to ensure that each occurrence has degree at most 2 and obeys Invariant P7. For each new black supervertex s, as s occurs i times in $T_1, \dots, T_i$, we will be left with at least two leaf occurrences of s. Also, as we will see, while this process may restructure some of the components, for all components in which $S(C)$ did not exist, it remains the case that $S(C)$ does not exist. This process takes $O(in)$ time.

The connecting of the remaining components entails finding a suitable set M of paths for each component C. These are found by means of a closure process starting from a collection of seed paths. The method for finding seed paths, in time $O(in)$, is described next.

For each new black supervertex s replacing $S(C)$, now reduced to degree 1 or 2, we form a seed path as follows. If s has degree 1 in C, we take the path from the remaining leaf occurrence of s to its nearest white ancestor and concatenate it with the path from s in C to the nearest white vertex; this forms the seed path. If s has degree 2 in C, we take the paths from the remaining leaf occurrences to their nearest white ancestors and concatenate these paths; this forms the seed path. Note that in either case both endpoints of the seed path are white and internal vertices are black.

Finally, if $S(C)$ does not exist, then we define a collection of one or more seed paths for $S(C)$. Each seed path begins with a distinct unused non-tree edge directed into vertex v. There must be such an edge for otherwise $\{v\} = S(C)$. For each such edge (x, v), if x is black, the seed path is extended to a white vertex as follows: from a leaf occurrence of x, of which there must be one because (x, v) is an unused edge, we take the path to its nearest white ancestor and concatenate this path with the edge (x, v) to form a seed path.

Step 3. Starting with the above seed path or collection of seed paths as the case may be, we find a sequence of swaps which will connect C to another component in T_i, and in the case when $S(C)$ does not exist, increase the in-degree of v by 1. A swap consists of the addition of a path to some packing tree T_h, forming a cycle in T_h, and the removal of a path in T_h breaking this cycle. The initial set of paths for swapping is provided by the seed path collection. The number of occurrences and in-degrees of all other vertices will be unaffected by this swap sequence and black vertices will continue to have degree at most 2. This process is repeated sequentially for each connected component of T_i which has not already joined up to another component. The total time taken over all these connected components will be $O(in+m)$.

The total time taken for the entire round is $O(in + m)$, which leads to $O((\mathcal{C}^3 n + \mathcal{C} m) \log n)$ time overall. Setting $m = O(\mathcal{C} n)$ using the construction of Nagamochi and Ibaraki, this becomes $O((\mathcal{C}^3 n \log n + m)$. Next, we elaborate on some of the above steps.

3.1.1 Computing $S(C)$

LEMMA 3. *$S(C)$ is unique.*

Proof. Let $\delta(X)$ denote the number of edges directed into vertex set X from outside X. We show that if A and B

are two distinct candidates for $S(C)$ then $A \cup B$ and $A \cap B$ would be also. It suffices to show that both $\delta(A \cap B)$ and $\delta(A \cup B)$ equal $i-1$ given that $\delta(A) = \delta(B) = i-1$. This follows from the fact that $\delta(A \cap B) + \delta(A \cup B) \leq \delta(A) + \delta(B)$ and the fact that both $\delta(A \cap B)$ and $\delta(A \cup B)$ are at least $i-1$ (as both contain white vertices). □

Given a collection of trees $T_1 \ldots T_i$ (the last of these is a forest), we find all subsets S of vertices satisfying:

- All supervertices in $S \cap T_h$ are contiguous in T_h for all h.
- Either $S \cap T_i$ is a full connected component in T_i, or $S \cap T_h$ is a leaf subtree in T_h, for some h, $1 \leq h < i$.
- All white vertices in S are present in all trees.
- No edge outside $T_1 \ldots T_i$ is directed into S from outside S.

$S(C)$, for each connected component C in T_i, is easily obtained from this computation.

It suffices to find for each leaf subtree Q in T_h, $h < i$, and each connected component Q in T_i, whether these exists a set $S \supseteq Q$ of supervertices such that (i) $S - Q$ has only black supervertices, and (ii) supervertices in $S \cap T_g$ occur contiguously in T_g, $1 \leq g \leq i$. Without loss of generality, we consider the case when $h = 1$ and consider all leaf subtrees of T_1 with respect to the above creteria.

The first step is to perform the contiguity check on white vertices alone, ignoring intervening black supervertices. Each leaf subtree Q_x of T_1 rooted at x with the property that white vertices in Q_x are contiguous in all other trees (possibly with intervening black supervertices) are determined in this step. For this we consider each tree T_g, $g \neq 1$, in turn. For each white vertex v in T_g, the pair (v, w) is placed in T_1 at the node $lca(v, w)$, where w is the nearest white ancestor of v in T_g. Note that if we now process the vertices x of T_1 in order of decreasing distance from the root and for each pair (v, w) placed at x, union the sets containing v and w, then the number of sets over which white vertices in Q_x are spread is exactly the number of connected components of these vertices in T_g. This number has to be 1 for all T_g, $g \neq 1$, for Q_x to stay in consideration.

Next, we bring black supervertices into play. This processing is actually interleaved with the processing for white vertices described above. When processing node x, it is easy to also obtain the list of intervening black supervertices which separate the white vertices in Q_x in each of $T_1 \ldots T_i$. This is made possible by the fact that black supervertices have degree at most 2. We pool together the set of all such black supervertices over all trees T_g. Finally, we check for each such black supervertex b whether all its occurrences in all T_gs are connected to an occurrence of a white vertex in Q_x through only black (hence degree 2) supervertices; if not, b is called a violating supervertex. This check is done by determining for each occurrence of b in each tree T_g, whether or not any of the (up to) two white supervertices reachable via paths of black supervertices are in Q_x, and if not, then determining which of these two white vertices lies on the path between b and white vertices in Q_x (call this white vertex y). This is easily done using LCA queries. If the check succeeds then all black supervertices between b and the nearest white vertex in T_g are added to the pool of black supervertices being processed (taking care not to repeatedly add the same black supervertex to this pool). On the other hand, if the check fails and results in a violating black supervertex, then we stop processing x, transfer the list of partly processed and unprocessed black supervertices to $z = lca(x, y)$ in T_1; the processing of these black supervertices continues when z is being processed. Note that Q_w ceases to be in consideration for all white vertices $w \neq z$ on the path from x to z in T_1. Q_x stays in consideration only if a violating black supervertex is not found. If it stays in consideration then the set of black supervertices computed at x plus the white vertices in Q_x together form a set denoted by S_x.

Finally, if Q_x is still in consideration then we check for the presence of non-tree edges which are directed into a white vertex in Q_x from outside S_x (a non-tree edge from outside Q_x to inside cannot be incident on a black supervertex, as all edges directed into black supervertices are already in the trees $T_1 \ldots T_i$). This check is also interleaved with the above computation. The total time taken in $O(i^2 n)$.

3.1.2 Maintaining and Restoring Invariant P7

There are two parts to Invariant P7, the first concerns attachment paths into black supervertices and the second concerns the degree of each black supervertex. To maintain the first part of Invariant P7, two changes to edge incidence need to be considered. The first arises when edges e_1 and e_2 incident respectively on instances x_1 and x_2 of black supervertex x are switched in the swapping process, so e_1 is now incident on x_2 and e_2 is incident on x_1. It suffices to also switch the attachment paths for these edges. Clearly, this takes $O(1)$ time. The second arises if a non-tree edge e is made incident on a black leaf supervertex x. Suppose the attachment path p for e is currently associated with another leaf instance x' of this black supervertex. Then the unused attachment path associated with x is switched with attachment path p, in $O(1)$ time, and now edge e can be made incident on x.

It remains to explain how to maintain black nodes at degree 2 or less when a set $S(C)$ is contracted to form a new black supervertex s (recall that the swaps in Step 3 will ensure that black vertices continue to have degree at most 2 in the packing trees thereby preserving P7). Before performing the contraction, we redistribute paths of black supervertices leading to non-tree edges incident on $S(C)$. Each such path has one incident edge. The paths are redistributed so that all these paths are incident on copies of $S(C)$ that contract to a leaf instance of s. These paths, concatenated with the attachment paths at their two ends, will form attachment paths for the new supervertices s. The attachment paths for tree edges are obtained similarly.

Suppose an instance of s has degree 3 or more, its degree is reduced by the following reattachment process. As s has degree 3 or more, there is a leaf instance s' of s. The path from s' to its nearest white ancestor x is traversed and cut at x; then the path $s'x$ is attached to the instance of x in s's packing tree, called the reattachment point, and finally for a neighbour y of s the edge ys is replaced by ys', where y is chosen so that x is not in the subtree rooted at y (treating s as a root). Note that there is still one instance of each white vertex in each packing tree.

In order to perform all needed reattachments efficiently, we proceed as follows. In each packing tree, for each supervertex s of degree d, we choose $d-2$ paths for the reattachment process. It remains to determine which subtrees of s

to attach to which paths. Let $s_1, \ldots, s_r$ be the supervertices in T_h needing degree reductions, reductions of total amount d. Choose s_1 to be the temporary root of T_h and consider the subtrees of $s_1 \ldots s_r$. There will be exactly $d+1$ leaf subtrees, subtrees containing none of $s_1, \ldots, s_r$. One at least of these subtrees does not contain a reattachment point; this is the subtree chosen to be reattached first. Its reattachment point is declared used and the above process is iterated with the unused reattachment points. When a supervertex s_g has its degree reduced to 2, it is removed from consideration and a new leaf subtreee is brought into play for its nearest ancestral black supervertex $s_{g'}$. This takes time $O(in)$ for all the degree reductions in one iteration.

3.1.3 Finding the Swap Sequence.

Let w denote the vertex v if $S(C)$ is not defined or the vertex s if it is indeed defined (recall s is the new black supervertex obtained by condensing $S(C)$).

We now perform a cyclic scanning process in which we scan $T_1 \ldots T_i$ repeatedly in round-robin order. We start with the set S comprising all vertices which lie on any of the seed paths constructed above. The set M is initialized to comprise all these seed paths and all initial and final portions of the seed paths, which therefore have one white endpoint. As the process evolves, new paths will be added to M and each path in M will have the following property: all strictly internal nodes are black and at least one of the terminal vertices is white. For each of $T_1 \ldots T_i$, this process will also maintain contiguous portions $T_1' \ldots T_i'$, respectively, which will expand over time.

Cyclic Scanning Process. Given a collection of paths in M, a set of supervertices S, and a tree $T = T_h$, this step redefines S and M as follows. Paths for which at least one of the terminal vertices is absent from T continue to be in M. All other paths are removed from M; for each such path p, we consider the edges $g \in T$ which lie in the fundamental cycle formed by p. These edges are organized into maximal paths in T with internal vertices being black. For each such path, all prefixes of that path which do not lie completely in $T' = T_h'$ are added to M, where a prefix is defined from the white node end. For reasons that will become clear later, paths p with both endpoints being white are processed first. One complication arises from the fact that T could have several occurrences of the same supervertex. In this case, a path p could define not one but several fundamental cycles. All of these cycles are considered while defining M above. All portions of these cycles not yet in T' are added to T' (apart from the paths p themselves). Further, all vertices on these cycles which are not yet in S are added to S. Clearly, T' remains a contiguous portion of T.

Finally, we add some more paths to M as follows. For each path newly added to M, if its end edge is directed into its white endpoint w, then all unused edges (those which do not appear in any of $T_1, \ldots, T_{i-1}, T_i$) directed into w are added to M. Further, for each such edge, (x, w), if x is a black supervertex, the following paths are added to M. For each leaf instance of x, consider the path p from x to its nearest white ancestor, and let q be the concatenation of p and (x, w). All prefixes of q starting at w are added to M. All suffixes of q beginning at a vertex outside S and ending at w are added to M. Note that black supervertices will not have any unused edges directed into them.

The cyclic scanning process is continued until M has a path π which goes out of C to another connected component in T_i (i.e., one endpoint lies outside C).

LEMMA 4. *A path π which goes out of C will ultimately appear in M.*

Proof. Suppose the cyclic scanning procedure does not find a path π connecting C to another component in T_i. Then the following claims yield a contradiction.

Claim 1. The set S of vertices is contiguous in all the trees.

This follows from the fact that each path in M has a white endpoint and that a path is not removed from M unless it lies completely within T_h' (when T_h is being processed) or when it is applied to a tree T_h in which both its endpoints occur, in which case it gets replaced by another sequence of paths between these endpoints, or between T_h' and the one endpoint outside T_h'.

The following invariant is key.

Invariant 1:

(i) Let T_{h-1} be the most recently processed tree. Then, for $0 \le a < b < i$,

$$T'_{h+a(mod\, i)} \cap T_{h+b(mod\, i)} \subseteq T_{h+a(mod\, i)} \cap T'_{h+b(mod\, i)}$$

Loosely speaking, on their pairwise common vertices, $T'_{h+a} \subseteq T'_{h+b}$, $0 \le a < b < i$ (the $(mod\, i)$ terms are implicit).

(ii) For each T_g, $1 \le g \le i$, T_g' is connected in T_g and a subset of S.

(iii) For each vertex v in $S \cap (T_g - T_{g'})$, there exists a sequence of paths in M with the following properties:

- When concatenated, these paths form a path from a white vertex in T_g' to v.
- For all except the last of these paths, both endpoints are white.

As every point on the paths in M lies in S, once there are no paths in M, the invariant implies $T_g \cap S = T_g'$ for all g, which immediately yields Claim 1.

The invariant is clearly true initially. We show that it is maintained when the tree T_h is processed. After the scanning step $S \cap T_h = T_h'$, thus clearly the first part of the invariant is maintained.

When T_h is processed the following changes occur to paths in M. First, each path with both endpoints in T_h' is removed from M. But, by the first part of the invariant, for each T_g', $g \neq h$, each such path either has both endpoints in T_g' or one endpoint, which must be a black endpoint, is not in T_g at all. Thus, at this point, the second part of the invariant still holds. Second, each remaining path P in M with both endpoints in T_h is removed from M and replaced by a sequence of new paths which connect the endpoints of P, if both endpoints are outside T_h', otherwise, these new paths connect the endpoint of P outside T_h' to a white vertex in T_h'. Clearly, this maintains the second part of the invariant for points already S. But points on $P - S$ are added to S; the new paths connect each such point to the white endpoint(s) of P, or to a white vertex in T_h', if the white endpoint of P is in T_h'. Thus for these points too the second part of the invariant will hold.

Claim 2. The number of edges directed into vertices of S from outside S is at most $i-1$.

By Claim 1 and Lemma 2, there are at most $i-1$ used edges into S. To show that there are no unused edges into S, it suffices to show that each unused edge directed into a vertex v in S must appear as part of some path in M at some point. By properties P3 and P7, there are no such unused edges for black vertices. For white deficient vertices v, all the above edges comprise the first edges of their respective seed paths. This leaves the case of white non-deficient vertices v. If a used edge directed into v becomes part of M, then all unused edges directed into v will also become part of M (by construction). Thus it suffices to show that some used edge directed into v will become part of M at some point. This is true, for otherwise, if $v \in S$ and none of the i used edges ever enter M, then set S has in-degree i, a contradiction. This proves Claim 2.

The proof is completed by showing that S would contradict the maximality of $S(C)$. First, S has exactly $i-1$ edges directed into it (by Claim 2, it has at most $i-1$ such edges, and as it contains a white vertex it has at least $i-1$ such edges). If $S(C)$ does not exist then S contains the deficient supervertex for C and hence S satisfies the conditions for $S(C)$. While, if $S(C)$ exists, imagine uncontracting $S(C)$ to form an expanded S; the expanded S still has in-degree $i-1$, and contains the deficient supervertex for C, so here too S satisfied the condition for $S(C)$. □

Let $\pi_1, \pi_2, \cdots, \pi_k = \pi$ be the sequence of paths produced by the cyclic scanning process, where the application of π_i frees π_{i+1} from the tree containing it, for $1 \le i < k$. Note that if π_{h+1} ends with a non-tree edge into vertex v, then on the cycle formed by π_h in T_g (the tree containing π_h), (one of) the edge(s) directed into v on the cycle is removed from T_g, thereby maintaining the overall indegree of v and the "treeness" of T_g. We would like to apply this sequence of paths to the packing trees $T_1, \cdots, T_k$ but we face several difficulties.

First, note that the cyclic scanning procedure always works with the original trees $T_1 \dots T_i$. Once some swaps are applied these trees will change. Therefore, if a path $\widetilde{\pi}$ in T_h enters M because of the application of path π' to T_h in the cyclic scanning process, then it is not obvious that π' can be swapped in for $\widetilde{\pi}$, once the previous swaps have been executed. The reason why it will still be possible to perform this swap is that π' is not completely in T'_h when it enters M, and all the previous swaps on T_h would have happened completely within T'_h.

Second, for each path added to M between white vertices having intermediate degree 2 black vertices, all prefixes of this path will enter M and conceivably several of these prefixes could be involved in pulling other paths into M. We need to show that the actual swap sequence will not involve the successive use of several of these prefixes (since the removal of one of these prefixes affects the other prefixes).

Third, and most problematic, as some of the swaps are made, black nodes with high degree could result and then it may not be possible to free up a path between a black node and a white node in a tree because there is now an intervening black node of high degree. Thus, in general, one cannot mimic the swaps obtained implicitly from the cyclic scanning procedure (in which all black nodes had degree at most 2) once some swaps have been made.

We address these problems by identifying a sequence of swaps which will indeed connect two connected components in T_i with the property that as each swap is made, all black nodes retain the degree at most 2 restriction. Specifically, we show there exist paths $\pi'_k, \pi'_{k-1}, \dots \pi'_1$ with the following properties:

- The π's run from white vertices to white vertices possibly with intervening black vertices.
- *Swapping in* π'_j and *swapping out* π'_{j+1} in sequence for each each j from 1 to $k-1$ will free π'_k, which connects two components, possibly but not necessarily those that π_k was supposed to connect.
- Each swap maintains the degree 2 restriction on black nodes.

Our algorithm will need to maintain two versions of the tree T_h, one which remains unchanged and is used for the scanning process, and one which changes and reflects the restructuring due to swapping paths in and out; the latter is denoted by $\widetilde{T}_h$. Likewise, we maintain T'_h and $\widetilde{T}'_h$, which is the restructured T'_h. As we will see, $T_h - T'_h = \widetilde{T}_h - \widetilde{T}'_h$.

Next, we show how to construct π'_j. Suppose π_j is added to M during iteration I of the cyclic scanning process by being taken from tree T_h. π_j has two parts, a portion $\pi_{j,1}$ which consists of the edges with one or both endpoints outside T'_h (there is at least one such edge), and a possibly empty portion $\pi_{j,2}$, which comprises those edges with both endpoints in T'_h. If π_j is a path in T_h, then π'_j comprises $\pi_{j,1}$ (which lies in $T_h - T'_h = \widetilde{T}_h - \widetilde{T}'_h$) extended at one or both ends in $\widetilde{T}_h$, if need be, to reach the nearest white vertices. Otherwise, π_j comprises a path whose last edge is a non-tree edge (x, y) into white vertex y. If the other endpoint of π_j is black, recall that π_j was obtained by traversing a portion of the path p from a leaf instance of x to its nearest white ancestor w in some tree T_g (possibly $T_g = T_h$). The definition of π'_j is much as in the previous case. Now $\pi_{j,1}$ consists of those edges with one or both endpoints outside T'_g, and includes edge (x, y). $\pi_{j,2}$, which may be empty, comprises those edges with both endpoints in T'_g. π'_j comprises $\pi_{j,1}$ extended, if need be, at its endpoint in $\widetilde{T}_g$ (i.e. not at endpoint y) to reach the nearest white vertex.

Note that if all of π_j lies outside T'_h, then π_j is a prefix of π'_j. Further, if an endpoint of π_j outside T'_h is white then this white vertex will also be an endpoint of π'_j.

π'_j is swapped into $\widetilde{T}_g$ as follows, where T_g is the tree into which π_j is swapped. If π_j has two white endpoints, then π'_j simply connects its two white endpoints in $\widetilde{T}_g$. If π_j has a black endpoint, x say, then π'_j may be swapped in in one of two ways. Let the instance of x to which π_j connects in T_g lie on the path of black supervertices between white vertices a and z. Let π'_j connect vertices y and w. If, on rooting $\widetilde{T}_g$ at a, w lies in the subtree rooted at z, then π'_j simply connects y and w; if not, the neighbours of the two copies of x are switched as shown in Fig.2. In every case, if $\pi_{j+1,1}$ lies in $T_g - T'_g = \widetilde{T}_g - \widetilde{T}'_g$, it will also lie on the cycle formed by the swapping in of π'_j. Consequently, $\widetilde{T}_g$ is maintained as a tree, either by removal of π'_{j+1} from $\widetilde{T}_g$ if π_{j+1} lies in T_g, or by removal of the cycle edge into b, where b was the white endpoint of π_{j+1} with the last edge of π_{j+1} being a

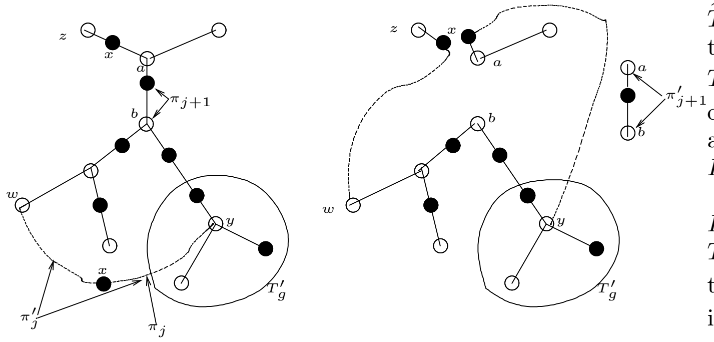

Figure 2: Swapping on tree T_j

non-tree edge into b. Note that in the latter case, the cycle edge into b lies in $\widetilde{T}_g - \widetilde{T}'_g$, and hence is also in $\widetilde{T}_g - \widetilde{T}'_g$.

To determine whether to apply the restructuring of Fig.2, one can simply traverse the paths from w and y in $\widetilde{T}_g$, using LCA queries on T_g to guide one (for $T_g - T'_g = \widetilde{T}_g - \widetilde{T}'_g$). The Fig.2 scenario applies if node x is not encountered in this traversal. As all traversed nodes are added to S at the end of the current iteration of the scanning process, this has cost $O(ni)$ over all iterations.

Remark. At the end of each iteration, all nodes in $(S \cap \widetilde{T}_g) - \widetilde{T}'_g$ are added to $\widetilde{T}'_g$, as for the construction of T'_g.

Invariant 2. Let $\widetilde{T}g$ be the tree into which π'_j is swapped. Let $I_{j'}$ be the iteration of the scanning process in which this swap occurs. Then:

(i) At the start of iteration I'_j, $T_g - T'_g = \widetilde{T}_g - \widetilde{T}'_g$. Also, T'_g and $\widetilde{T}'_g$ are connected subtrees of T_g and T'_g, respectively.

(ii) Let C_j and $\widetilde{C}_j$ be the cycles formed by π_j and π'_j in T_g and $\widetilde{T}_g$, respectively. Then $(C_j - \pi_j) \cap (T_g - T_{g'}) = (\widetilde{C}_j - \pi'_j) \cap (\widetilde{T}_g - \widetilde{T}'_g)$.

(iii) At the end of iteration $I_{j'}$, all vertices on π'_{j+1} have been added to S.

LEMMA 5. *Invariant 2 holds.*

Proof. The result is proved by induction on j. Clearly, at the start of the first iteration $T'_g = \emptyset = \widetilde{T}'_g$; thus (i) holds. Also the portion of π'_1 forming $\widetilde{C}_1$ is exactly π_1; consequently $C_1 = \widetilde{C}_1$ and (ii) holds for iteration $I_{1'}$. Initially, (iii) refers to the case $j = 0$, and it suffices to note that π'_1 is a seed path whose vertices are indeed in S. So suppose (i) holds at the start of iteration $I_{j'}$, (ii) is true for C_j and $\widetilde{C}_j$, and (iii) holds at the end of the preceding iteration.

First, we show (i) holds for $j + 1$. The only trees that change during iteration $I_{j'}$ are T'_g, $\widetilde{T}'_g$, and $\widetilde{T}_g$. The changes to $\widetilde{T}_g$ are due to the addition of π'_j and the removal of either $\widetilde{\pi}'_{j+1}$ or a single edge from $\widetilde{C}_j$, the resulting cycle. By (iii), all vertices on π'_j are already in S and consequently are added to $\widetilde{T}'_g$ by the end of iteration $I_{j'}$. By construction, all of $(C_j - \pi_j) - T'_g$ is added to T'_g, by (ii), this is all of $(\widetilde{C}_j - \pi'_j) - \widetilde{T}'_g$, which is added to $\widetilde{T}'_g$. The structure of $\widetilde{T}_g - \widetilde{T}'_g - (\widetilde{C}_j - \pi'_j)$ is unchanged and hence remains equal to the corresponding portion of T_g. The remaining additions to T'_g and $\widetilde{T}'_g$ are of vertices in $T_g - C_j$ and have the same effect on $T_g - T'_g$ and $\widetilde{T}_g - \widetilde{T}'_g$. Consequently, $T_g - T'_g = \widetilde{T}_g - \widetilde{T}'_g$ at the end of iteration $I_{j'}$ and thus at the start of iteration $I_{(j+1)'}$.

T'_g is connected by construction. At the start of iteration $I_{j'}$, $T_g - T'_g$ comprises one or more subtrees attached to T'_g. T'_g grows by adding neighboring portions of one or more of these subtrees. Since $T_g - T'_g = \widetilde{T}_g - \widetilde{T}'_g$ at the start of iteration $I_{j'}$, these same subtrees are also attached to $\widetilde{T}'_g$. At the end of iteration $I_{j'}$, $\widetilde{T}_g$ is still connected, and as $T_g - T'_g = \widetilde{T}_g - \widetilde{T}'_g$, it must be that $\widetilde{T}'_g$ is connected.

We turn to the proof of (ii). Suppose that π_{j+1} is swapped into tree T_h. There are four cases to consider.

Case 1. π_{j+1} connects two white vertices outside T'_h. Then $\pi'_{j+1} = \pi_{j+1}$. Thus by (i) for iteration $I_{(j+1)'}$, π'_{j+1} connects two white vertices in $\widetilde{T}_h - \widetilde{T}'_h$. Consequently $C_{j+1} = \widetilde{C}_{j+1}$ and (ii) holds for $j + 1$.

Case 2. π_{j+1} connects a white vertex outside T'_h to a black vertex outside T'_h. Then $\pi'_{j+1} = \pi'_{j+1,1}\pi_j$, where π'_{j+1} is extended by $\pi'_{j+1,1}$ from π_j's black endpoint. Further, the cycle $\widetilde{C}_{j+1}$ will use only π_j. As π_{j+1} connects two points in $T_h - T'_h = \widetilde{T}_h - \widetilde{T}'_h$, and as π'_{j+1} connects the same two points, $C_{j+1} = \widetilde{C}_{j+1}$ and again (ii) holds for $j + 1$.

Case 3. π_{j+1} connects a white vertex inside T'_h to a white vertex w outside T'_h. Then π'_{j+1} connects white vertex w to a white vertex inside $\widetilde{T}'_h$. Since $T_h - T'_h = \widetilde{T}_h - \widetilde{T}'_h$, the paths from w to T'_h in T_h and from w to $\widetilde{T}'_h$ in $\widetilde{T}_h$ are identical. Consequently (ii) holds for $j + 1$.

Case 4. π_{j+1} connects a white vertex inside T'_h to black vertex b outside T'_h Then π'_{j+1} connects a white vertex w' inside $\widetilde{T}'_h$ to another white vertex with a path passing through x. Further, the cycle $\widetilde{C}_{j+1}$ formed by π'_{j+1} uses the portion of π'_{j+1} from w' to x. The argument now proceeds as in Case 3, with x playing the role of w.

It remains to prove (iii). There are two possibilities. Either π'_{j+1} lies on $\widetilde{C}_{j+1}$; then all vertices on π'_{j+1} will be added to S by the end of iteration $I_{j'}$. Otherwise π'_{j+1} ends in a non-tree edge, and by construction all vertices in π'_{j+1} not already in S are added to S by the end of iteration $I_{j'}$

Next, we consider which two components are connected by π'_k. If π_k had two white endpoints then π'_k connects the same two components as π_k. If π_k has a black endpoint x in some component $C' \neq C$, π'_k need not have an endpoint outside C. If π'_k has an endpoint in component $C'' \neq C$, then it is used to connect C'' and C. If not, a crossover, as in Fig.2, is performed at x, and this will connect C and C'.

This process is applied in some arbitrary order to all components that have not been joined to another component, thereby at least halving the number of components.

The total time complexity for the cyclic scanning process for component C is the number of edges scanned plus i times the number of vertices scanned. The scanning processes for distinct components C operate on distinct edges and vertices and therefore the complexity of cyclic scanning over all components in T_i is $O(in + m)$. The complexity bound relies on making sure that each path in M is processed just once. But it suffices, for paths with a black endpoint b, to know which is the next tree T_h containing b. This information can be determined by presorting, for each black point b, the indices of the trees holding b. Then, as there are $O(ni)$ paths added to M, moving the paths out of M takes $O(ni)$ time. Note that it suffices to store a path by recording its endpoints.

We have shown:

THEOREM 6. *In an Eulerian or undirected graph G, the edge connectivity $\mathcal{C}$ of a subset of its vertices can be found in time $O(\mathcal{C}^3 n \log n + m)$.*

4. THE CACTUS CONSTRUCTION ALGORITHM

Recall the notion of visible and invisible cuts from Section 2.3. Visible $(i-1)$-cuts are explicitly identified in the process of constructing T_i. Since we always compute black-minimal cuts, a visible cut cannot correspond to several consecutive nodes on a cycle in the cactus tree, for it can be shown that such a cut would span several components in T_i. Therefore, each visible cut computed will form a node in the cactus tree. Set containment for these cuts is readily determined, first within individual components in a single iteration (note, we find all $(i-1)$-cuts containing the deficient vertex and not only the largest one, in each component) and second between iterations by keeping track of black supervertices into which $(i-1)$-cuts from the previous iteration have been contracted.

Next, we describe the procedure for finding invisible $(i-1)$-cuts. Consider a visible cut D with visible children cuts $C_1, \ldots, C_k$. We show how to find invisible cuts which are contained within D and which contain two or more children of D.

To this end, we form subtrees of the packing trees by contracting $C_1 \ldots C_k$ as well the portions of the packing trees outside D to single nodes. Each node in the reduced packing trees has combined degree at most $2i-2$, for all white vertices in D are contained in some C_h, $1 \leq h \leq k$. Thus every node occurs as a leaf in some tree or has degree two in every tree. We seek cuts in this reduced collection of packing trees which do not include vertices outside D. These correspond exactly to cuts lying between D and its children in the final cactus tree.

The invisible cuts are found by means of a *closure* process, which given a vertex set R, finds the smallest vertex set S, if any, such that $R \subseteq S$, S is contiguous in each reduced packing tree, and no unused edge (which is not present in the packing trees) is directed into S. Let m_S denote the number of unused edges with both endpoints in S. This procedure runs in time $O(i|S| + m_S)$, if S exists, and in time $O(i|D| + m_D)$, otherwise. This closure procedure is very similar to the processing of black supervertices in the computation of $S(C)$ and is left to the reader.

We find invisible cuts using the closure process as a black box as follows. Choose some reduced packing tree T_1, say. Let v be the centroid of T_1 (so every subtree of v in T_1 has size at most $2/3 * |T_1|$). First, we determine if there is an invisible cut containing v and if so find the minimal such cut S in time $O(i|S| + m_S)$; if yes, then we contract S to a single vertex in all reduced packing trees and again seek the minimal invisible cut containing this shrunk vertex, iterating this process until no further cuts containing the shrunk vertex exist within D. Once all such cuts have been found we return to tree T_1 and find invisible cuts not containing v by recursing on each of the subtrees obtained by removing vertex v. Clearly, the iterative step takes time $O(i*n_D+m_D)$ and the recursion adds a further $\log n$ factor. Summing over all subproblems created by different nodes of the preliminary cactus tree yields a overall running time of $O((in+m)\log n)$.

To find S, we proceed as follows. If v occurs as a leaf in reduced tree T_h, then R, the initial set for the closure algorithm is set to $\{v, w\}$, where w is the parent of v in T_h. Otherwise, let x and y be v's neighbours in T_1. The closure process is run twice, in parallel, with initial sets $\{v, x\}$ and $\{v, y\}$, respectively; whichever process ends first provides the set S.

The invisible cuts found in this process need to be incorporated into the preliminary cactus tree. Further, note that not all $(i-1)$-cuts have been detected yet. In particular, chains of cuts leading to cycles in the cactus tree get parenthesised above and not all pairs of cuts on a chain will be identified as new cuts. Details of these cases are omitted.

5. APPLICATIONS TO THE UNIFORM SURVIVABLE NETWORK DESIGN PROBLEM

Recall that the WGMV algorithm runs on an undirected graph. We describe how the above machinery can be used to get an efficient implementation of this algorithm. In particular, we show how iteration i can be implemented. Previous iterations would have ensured through edge additions that every vertex v for which $r_v \leq i-1$ now has $con(v) = \min\{r_v, i-1\}$ and vertices with $r_v \geq i$ have $con(v) \geq i-1$. In the graph G_Q restricted to the edges already added to Q (see Section 2.4) and made directed by directing edges in both directions, we first construct the cactus tree as in Section 4.

A vertex r_v for which $r_v \geq i$ is called a *demand* vertex. Those cactus tree nodes whose subtrees contain a demand vertex represent violated sets. The minimal cuts corresponding to minimal violated sets provide the active sets. As edges are added to Q, the number of $(i-1)$-cuts and of distinct nodes in the cactus tree are reduced. The active sets are updated accordingly. One detail is that addition of an edge may not change a violated set but it may nonetheless expand the corresponding minimal cut by drawing in more black supervertices. In fact, this growth, which is computed by a closure procedure, may require working into black supervertices recursively.

The changes to the cactus tree resulting from edge addition are standard apart from the effect of black supervertices (see [6] for the effect on a standard cactus tree).

The overall time taken for iteration i turns out to be $O((i^3 n + im)\log n)$, giving a total time of $O((\max_v\{r_v\}^4 n + \max_v\{r_v\}^2 m)\log n + \max_v\{r_v\} m \log^2 n)$, which improves to $O((\max_v\{r_v\}^4 n \log n + \max_v\{r_v\} m \log^2 n)$, using the construction of Nagamochi and Ibaraki mentioned earlier.

Acknowledgements

We thank Hal Gabow for helpful discussions and pointers to the tree packing theorem [2].

6. REFERENCES

[1] A. Agrawal, P. Klein, R. Ravi. When trees collide: an approximation algorithm for the generalized

Steiner problem on networks. Proceedings of the *Twenty-third Annual ACM Symposium on Theory of Computing*, pp. 134–144, 1991.

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

[3] R. Cole, R. Hariharan, M. Lewenstein, E. Porat. A Faster Implementation of the Goemans-Williamson Clustering Algorithm. Proceedings of the *12th Annual Symposium on Discrete Algorithms*, 2001.

[4] E.A. Dinits, A.V. Karzanov, M.V. Lomonosov. On the structure of a family of minimal weighted cuts in a graph. *Studies in Discrete Optimization*, A.A. Fridman, Ed., pp. 240–306, 1976. (Original article in Russian, translation available from National Translation Center, Library of Congress, Cataloging Distribution Center, Washington D.C, 20541 (NTC 89-20265)).

[5] Y. Dinitz and A. Vainshtein. The Connectivity Carcass of a Vertex Subset in a Graph and its Incremental Maintenance. Proceedings of the *26th Annual Symposium on Theory of Computing*, 1994, pp. 716–725.

[6] Y. Dinitz and J. Westbrook. Maintaining the classes of 4-edge-connectivity in a graph on-line. *Algorithmica*, 20, pp. 242–276, 1998.

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

[8] J. Edmonds. Edge Disjoint Branchings. *Combinatorial Algorithms*, R. Rustin, editor, Algorithmics Press, NY, 1972, pp. 91–96.

[9] L. Fleischer. Building chain and cactus representations of all minimum cuts from Hao-Orlin in the same run time. Journal of Algorithms, 33, 1, pp. 51–72.

[10] A. Frank. Kernel systems of directed graphs. *Acta Sci. Math.*, 41, 1979, pp. 63–76.

[11] H. Gabow. Applications of a poset representation to edge connectivity and graph rigidity. Proceedings of the *32th IEEE Symposium on Foundations of Computer Science*, 1991, pp. 812–821.

[12] H. Gabow. A matroid approach to finding edge connectivity and packing arborescences. Proceedings of the *23th ACM Symposium on Theory of Computing*, 1991, pp. 112–122.

[13] H. Gabow, M. Goemans, D. Williamson. An efficient approximation algorithm for the survivable network design problem, Proceedings of *IPCO,* pp. 57–74, 1993. To appear in *Math. Programming.*

[14] H. Gabow and S. Pettie. The dynamic vertex minimum problem and its application to clustering-type approximation algorithms. *Scandinavian Workshop on Algorithm Theory*, 2002.

[15] M. Goemans, D. Williamson. A general approximation technique for constrained forest problems, *SIAM Journal on Computing*, 24(2), pp. 296–317, 1995.

[16] M. Grotschel, C.L. Monma, M. Stoer. *Design of Survivable Networks*, Handbook of Operations Research and Management Science, 1993.

[17] K. Jain. A Factor 2 Approximation Algorithm for the Generalized Steiner Network Problem. *Combinatorica*, Vol 21-1, 39-60, 2001.

[18] P. Klein. A data structure for bicategories with applications to speeding up an approximation algorithm, *Information Processing Letters*, 52, pp. 303–307, 1994.

[19] L. Lovasz. On two minimax theorems in graph theory. *Journal of Combinatorial Theory*, B, 21, 1976, pp. 96–103.

[20] H. Nagamochi, T. Ibaraki. Linear time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7, 1992, pp. 583–596.

[21] R. Tarjan. A good algorithm for edge-disjoint branchings. *Information Processing Letters*, 3, 1975, pp. 51–53.

[22] P. Tong, E. Lawler. A faster algorithm for finding edge disjoint branchings. *Information Processing Letters*, 17, 2, 1983, pp. 73–76.

[23] D. Williamson, M. Goemans, M. Mihail, V. Vazirani. A primal-dual approximation algorithm for generalized steiner network problems, *Combinatorica*, 15, pp. 435–454, 1995.