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 \( O(C^3 n \log n + m) \). 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 2 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 \( O(C_{min} m \log \frac{m}{\mathcal{C}_{min}}) \) time, where \( C_{min} \) 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 \( O(C_{min}, n) \) 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 \( O(\min[O(C_{min}, n^2), mn]) \) on directed graphs and \( O(C_{min}, n^2) \) 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

\( \copyright \) 2003 ACM 1-58113-674-9/03/0006 ... $5.00.
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], and Gabow [12], the last of which reduced the running time to $O(e^{3/2}m)$, where $C_m$ is the size of the minimum $r$-cut. Note that none 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 $O(C_m^2m \log \frac{n}{m})$. 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 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 $S$ and $V - C$ if and only if there is a cut of cardinality $2h$ in $G'$ separating $S$ 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 an Eulerian directed graph $G$, there exist $C = \max_{v \in V} \{con(v)\}$ 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 an Eulerian directed graph $G$, there exist $C$ edge-disjoint directionless trees, $T_1, T_2, \ldots, T_C$, rooted at $r$ such that each vertex $v \neq r$ in $G$ appears exactly $\min\{C, con(v)\}$ times over all trees (occurring multiple times in a tree) and has in-degree exactly $\min\{C, con(v)\}$ over these trees. $T_1, T_2, \ldots, T_C$ 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 $O((C^3 + Cm) \log n)$. 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 $O(Cn)$ edges such that for each pair $u, v$ of vertices, the connectivity of $v$ and $w$ in the subgraph $G' = (V, E')$ is $\min\{C, \frac{con(v)}{C}, \frac{con(w)}{C}\}$. Thus, it suffices to run the connectivity algorithms on $G'$, improving the time bound to $O(C^3n \log n + m)$. 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 $O(C_{min}^2m \log \frac{n}{m})$. 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 $\theta$, 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 $C$ 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 Dinitz et al. [1], represents the $C_{min}$-cuts of $G = (V, E)$, where $G$ is $C_{min}$-edge connected, but not $C_{min} + 1$-edge connected. For odd $C_{min}$, this cactus tree is a tree; for even $C_{min}$, it can also include cycles.

We start by describing the structure for odd $C_{min}$. Let $r$ be an arbitrary vertex of $G$. Given a cut $F \subseteq E$ of cardinality $C_{min}$ 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 $C_{min}$) forming this cut.

For even $C_{min}$, cuts of cardinality $C_{min}$ 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 $C_{min}$ in general, this yields chains of cuts $C_1, \ldots, C_r$ such that $U_i \subset C_i$ is a cut of cardinality $C_{min}$ 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 $C$-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 \(C\) packing trees, and then runs in time \(O((n + m) \log n)\); this is bounded by \(O(Cn \log n + m)\) 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 \(\min\{r_v, r_w\}\) as 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 \(2 \max\{r_v\} - 1\). 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, Hariri, Lowenstein 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 \(O(\max\{r_v\}^3 n^4)\) total time over all these iterations, and the implementation due to Gabow, Goemans and Williamson [13] took \(O(\max\{r_v\}^2 n^2 + \max\{r_v\}^2 n \sqrt{m} \log \log \sqrt{m})\) 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 \(O(\max\{r_v\}^3 n + \max\{r_v\}^2 m) \log n + \max\{r_v\} m \log^2 n)\), which improves to \(O(\max\{r_v\}^2 n \log n + \max\{r_v\}^2 (\log^2 n)\), using the construction of Nagamochi and Ibaraki mentioned above. Under the practical assumption that \(\max\{r_v\}\) 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 \(i\)th 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 \(i\)th forest \(T_i\) by at least half, leading to an overall time of \(O((n + m) \log n)\) per tree, or \(O(C_{\text{max}} (m + n) \log n)\) 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 such one 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:

- \(e \not\in T_1 \ldots T_i\) 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.
- \(e \not\in T_1 \ldots T_i\) 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...
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 $\{\text{con}(v), i-1\}$ trees and has in-degree exactly equal to its number of occurrences in these trees, the $i$th tree is constructed in several rounds as follows. Each round runs in $O(i^2n+m)$ time and reduces the number of connected components in the $i$th forest by at least half, leading to an overall time of $O((i^2n+m)\log n)$ for constructing $T_i$, or $O((C^2n+Cm)\log n)$ 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 $\text{con}(v) \geq i-1$ and $\text{con}(v)$ 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 $T_1 \ldots T_i$, 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 $\text{con}(v) = i-1$). The differences begin here.

First, unlike Gabow’s algorithm, the algorithm cannot stop if it finds such a set $S$, because $C$ 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 $\text{con}(v) = i-1$ 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 $T_1 \ldots T_i$ 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 $(x, y) \in M$; 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 $T_2$ are added to $M$ at this point, nor those from $v$ to $u$, and therefore vertices in $S$ need not be contiguous in $T_2$.

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

2.3 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 $i$th 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. $\{r_v\}$ 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 $\delta_G(S) = i - 1$, where $\delta_G(S)$ 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 lamina 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 $O((i^3 n + im)\log n)$ time over all the rounds in iteration $i$. The total time taken by the algorithm is thus $O((\max_{v \in V} \{r_v\} + \max_{w \in W} \{r_w\})m \log n + \max_{v \in V} \{r_v\} m \log^2 n)$.

3. CONSTRUCTING THE PACKING TREES

Consider the $i$th iteration and suppose several rounds have been performed in this iteration resulting in trees $T_1 \ldots T_{i-1}$ 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 $T_1 \ldots T_i$. 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 $v$) 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 $v$ occur contiguously in all the trees $T_1 \ldots T_{i-1}$ (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 two $\con(w)$ 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 contiguous in the trees $T_1 \ldots T_i$ and further, no edge outside of $T_1 \ldots T_i$ 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 $T_1 \ldots T_i$ 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 $T_1 \ldots T_i$ if the contiguity property holds, and more otherwise, as required. □

The algorithmic details of finding $S(C)$ appear in Section 3.1. The time taken for this procedure will be $O(i^2 n + m)$. The main reason 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 $T_1 \ldots T_i$; 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 $T_1 \ldots T_{i-1}$. 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 component 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 $T_1 \ldots T_i$. 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 $T_1 \ldots T_i$.

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 $T_1 \ldots T_i$, 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 $S(C)$, we reduce 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 $s$ 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 $O(in)$, which leads to $O(C^n n \log n)$ time overall. Setting $m = O(Cn)$ using the construction of Nagamochi and Ibaraki, this becomes $O(C^n n \log n + m)$.

Next, we elaborate on some of the above steps.

3.1.1 Computing $S(C)$

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 $h$.
3.1.2 Maintaining and Restoring Invariant P7

For each connected component $C$ in $T_i$, it is easily maintained and restored. The complete component $C$ is cut out from $S$ in $T_i$, and its edges are labeled as follows: each leaf edge is labeled as $L_j$, $0 < j < n_i$, $1 < L_j$, and each non-leaf edge is labeled as $N_j$, $0 < j < n_i$, $1 < N_j$. Each edge $E_j$ (either $L_j$ or $N_j$) is labeled with the label of the node it connects to. Each component $C$ is cut out from $S$ in $T_i$ and restored to $S$ in $T_{i+1}$.

The edges in $C$ are labeled as follows: each leaf edge is labeled as $L_j$, $0 < j < n_i$, $1 < L_j$, and each non-leaf edge is labeled as $N_j$, $0 < j < n_i$, $1 < N_j$. Each edge $E_j$ (either $L_j$ or $N_j$) is labeled with the label of the node it connects to. Each component $C$ is cut out from $S$ in $T_i$ and restored to $S$ in $T_{i+1}$.
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 $T_1, \ldots, T_t$, 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. 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 $T_1, \ldots, T_t$, this process will also maintain contiguous portions $T'_1, \ldots, T'_t$, 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 $y \in T$ 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 $T' = T_h'$ and whose black endpoint is not already in $S$ are added to $M$, where a prefix is defined from the white node end. 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 vertices on paths in $M$ which are not yet in $S$ are now added to $S$. $T'$ is now redefined to be the original $T'$ augmented with the vertices newly added to $S$. We claim that $T'$ 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 $v$, then all unused edges (those which do not appear in any of $T_1, \ldots, T_{i-1}, T_i$) directed into $v$ are added to $M$. Further, for each such edge, $(x, v)$, 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 $(x, v)$. All prefixes of $q$ starting at $w$ are added to $M$.

Black supervertices will not have any unused edges directed into them. Also note that the fact that at least one endpoint of each path in $M$ is white can be shown from the above description.

The cyclic scanning process is continued until $M$ has a path $\pi$ which goes out of $C$ to another connected component in $T_i$ (i.e., one end point lies outside $C$). Let $\pi_1, \pi_2, \ldots, \pi_k = \pi$ be the sequence of paths this process produces, where the application of $\pi_i$ frees $\pi_{i+1}$ from the tree containing it, for $1 \leq i < k$. We would like to apply this sequence of paths to the packing trees $T_1, \ldots, T_t$, but we face several difficulties.

First, note that the cyclic scanning procedure always works with the original trees $T_1, \ldots, T_t$. Once some swaps are applied these trees will change. Therefore, if a path $\pi$ in $T_h$ enters $M$ because of the application of path $\pi'$ to $T_h$ in the cyclic scanning process, then it is not obvious that $\pi'$ can be swapped in for $\pi$, once the previous swaps have been executed. The reason why it will still be possible to perform this swap is that $\pi'$ 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 $\pi'_1, \pi'_{i+1}, \ldots, \pi'_1$ with the following properties:

- The $\pi'_i$ run from white vertices to white vertices possibly with intervening black vertices.
- Swapping in $\pi'_i$ and swapping out $\pi'_{i+1}$ in sequence for each $i$ from 1 to $k - 1$ will free $\pi'_k$, which connects two components, possibly but not necessarily those that $\pi_k$ 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 $\hat{T}_h$. Likewise, we maintain $T'_h$ and $T''_h$, which is the reststructured $T''_h$. As we will see, $T_h - T_h'' = T_h - T_h''$. 

Next, we show how to construct $\pi'_i$. Suppose $\pi_i$ is added to $M$ during iteration $i$ of the cyclic scanning process by being taken from tree $T_h$. $\pi_i$ has two parts, a portion $\pi_{i,1}$ 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 $\pi_{i,2}$, which comprises those edges with both endpoints in $T_h'$. If $\pi_i$ is a path in $T_h$, then $\pi'_i$ comprises $\pi_{i,1}$ (which lies in $T_h - T_h' = T_h - T_h''$) extended at one or both ends in $\hat{T}_h$, if need be, to reach the nearest white vertices. Otherwise, $\pi_{i,1}$ comprises a path whose last edge is a non-tree edge $(x, y)$ into white vertex $y$. If the other endpoint of $\pi_{i,1}$ is black, recall that $\pi_i$ was obtained by traversing a portion of the path $p$ from a leaf instance of $x$ to its nearest white ancestor $w$; $\pi'_i$ is simply the concatenation of all of $p$ with edge $(x, y).

Note that if all of $\pi_i$ lies outside $T''_h$, then $\pi'_i$ is a prefix of $\pi'_k$. Further, if an endpoint of $\pi_i$ outside $T_h'$ is white then this white vertex will also be an endpoint of $\pi'_k$.

$\pi'_i$ is swapped into $\hat{T}_h$ as follows, where $\hat{T}_h$ is the tree into which $\pi_i$ is swapped. If $\pi_i$ has two white endpoints, then $\pi'_i$ simply connects its two white endpoints in $\hat{T}_h$. If $\pi_i$ has a black endpoint, $x$, say, then $\pi'_i$ may be swapped in one of
two ways. Let the instance of $x$ to which $\pi_j$ connects in $T_2$ lie on the path of black supervertices between white vertices $a$ and $z$. Let $\pi'_j$ connect vertices $y$ and $w$. If, on rooting $T_2$ at $a$, $w$ lies in the subtree rooted at $z$, then $\pi'_j$ 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 $T_2 = T_2 = T_2 - T_2$, it will also lie on the cycle formed by the swapping in of $\pi'_j$. Consequently, $T_2$ is maintained as a tree, either by removal of $\pi_{j+1,1}$ from $T_2$ if $\pi_{j+1,1}$ lies in $T_2$, or by removal of the cycle edge into $b$, where $b$ was the white end-point of $\pi_{j+1,1}$ with the last edge of $\pi_{j+1,1}$ being a non-tree edge into $b$. Note that in the latter case, the cycle edge into $b$ lies in $T_2 = T_2$, and hence is also in $T_2 = T_2$.

To determine whether to apply the restructuring of Fig.2, one can simply traverse the paths from $w$ and $y$ in $T_2$, using LCA queries on $T_2$ to guide one (for $T_2 = T_2 - T_2$). 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 searching process, this has cost $O(m)$ over all iterations.

Next, we consider which two components are connected by $\pi'_j$. If $\pi_1$ had two white endpoints then $\pi'_j$ connects the same two components as $\pi_1$. If $\pi_1$ has a black endpoint $x$ in some component $C' \neq C$, $\pi'_j$ need not have an endpoint outside $C$. If $\pi_1$ 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 $C''$.

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 $C_i$, $1 \leq h \leq k$. 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 $R \subseteq S$ 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 $O(|D| + m_D)$, 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 $T_1, \ldots$. 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 recurcuing on each of the subtrees obtained by removing vertex $v$. Clearly, the iterative step takes time $O(i|S| + m_s)$ 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 $O((i + m) \log n)$.

To find $S$, we proceed as follows. If $v$ occurs as a leaf in reduced tree $T_k$, then $R$, the initial set for the closure algorithm is set to $\{v, w\}$, where $w$ is the parent of $v$ in $T_k$. Otherwise, let $x$ and $y$ be $v$'s neighbours in $T_k$. 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.
6. REFERENCES

Acknowledgements

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

6. REFERENCES


