Fast Edge Splitting and Edmonds' Arborescence Construction for Unweighted Graphs

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

Abstract

Given an unweighted undirected or directed graph with n vertices, m edges and edge connectivity c, we present a new deterministic algorithm for edge splitting. Our algorithm splits-off any specified subset S of vertices satisfying standard conditions (even degree for the undirected case and in-degree $\geq$ out-degree for the directed case) while maintaining connectivity c for vertices outside S in $\tilde{O}(m+nc^2)$ time for an undirected graph and $\tilde{O}(mc)$ time for a directed graph. This improves the current best deterministic time bounds due to Gabow [8], who splits-off a *single* vertex in $\tilde{O}(nc^2+m)$ time for an undirected graph and $\tilde{O}(mc)$ time for a directed graph. Further, for appropriate ranges of $n, c, |S|$ it improves the current best randomized bounds due to Benczúr and Karger [2], who split-off a single vertex in an undirected graph in $\tilde{O}(n^2)$ Monte Carlo time.

We give two applications of our edge splitting algorithms. Our first application is a sub-quadratic (in n) algorithm to construct Edmonds' arborescences. A classical result of Edmonds [5] shows that an unweighted directed graph with c edge-disjoint paths from any particular vertex r to every other vertex has exactly c edge-disjoint arborescences rooted at r. For a c edge connected unweighted undirected graph, the same theorem holds on the digraph obtained by replacing each undirected edge by two directed edges, one in each direction. The current fastest construction of these arborescences by Gabow [7] takes $O(n^2c^2)$ time. Our algorithm takes $\tilde{O}(nc^3+m)$ time for the undirected case and $\tilde{O}(nc^4+mc)$ time for the directed case. The second application of our splitting algorithm is a new Steiner edge connectivity algorithm for undirected graphs which matches the best known bound of $\tilde{O}(nc^2+m)$ time due to Bhalgat et al [3]. Finally, our algorithm can also be viewed as an alternative proof for existential edge splitting theorems due to Lovász [9] and Mader [11].

1 Introduction

In this paper we consider edge connectivity problems on unweighted directed and undirected graphs on n vertices and m edges. *Connectivity* in this paper is denoted by c and refers to *edge connectivity.* We present fast algorithms for certain problems in connectivity through efficient algorithms for *edge splitting.*

1.1 Edge Splitting.

Edge Splitting in a graph (also called the *splitting off* operation) involves replacing two edges ab and bd by the edge ad. The goal of this operation is to reduce the graph size while retaining certain connectivity properties, thus serving as an inductive/recursive tool for proving connectivity properties as well as obtaining algorithms for connectivity problems. The edge splitting operation was introduced by Lovász [9],[10] (exercise 6.53), who showed that edges incident on any even-degree vertex b of an undirected graph G could be paired up and split-off in such a way that the global connectivity of vertices excluding b is retained, provided the global connectivity of G is at least 2. Mader [11] generalized this theorem to maintain connectivity from any vertex r (also called r-connectivity[1]) for directed graphs while splitting off pairs of edges incident on vertices with in-degree $\geq$ out-degree.

Splitting Off Vertices. In this paper, we consider the task of removing any specified set of *eligible* vertices v while maintaining connectivity of the remaining graph. By an eligible vertex v, we mean the following conditions have to be satisfied: (1) if G is undirected, then v has even degree, and (2) if G is directed, then v has in-degree $\geq$ out-degree. We show that such a task of removing vertices while preserving the connectivity of the remaining graph can be accomplished via edge splitting as follows.

First, consider one eligible vertex v. Note that pairs of edges incident on v can be split off by successive appli-

*Google India Pvt Ltd, Bangalore, India; anandbhalgat@google.com Part of this work was done when the author was at IISc, Bangalore, India.

†Strand Life Sciences and House of Algorithms, Bangalore, India; ramesh@strandls.com Part of this work was done when the author was at IISc, Bangalore, India.

‡CSA Department, Indian Institute of Science, Bangalore, India; kavitha@csa.iisc.ernet.in

§Massachusetts Institute of Technology, Cambridge, MA; debmalya@mit.edu. Part of this work was done when the author was at IISc and Bell Labs, Bangalore, India.

[1] r-connectivity is defined as $\min_{v \neq r} \lambda(r,v)$ and $\lambda(r,v)$ is the connectivity from r to v.

cation of the edge splitting theorems without affecting the connectivity of the remaining graph until either v has no incident edge (for the undirected case) or v has no outgoing edge (for the directed case). Now, v can be removed from the graph without affecting the connectivity of the remaining graph. The pairs of edges which are split off correspond to a *matching* of the edges incident on v (for the undirected case) or of its incoming and outgoing edges (for the directed case). We call this the *splitting off* of vertex v. Next, consider the case of multiple eligible vertices. These can be split off successively without affecting the connectivity of the remaining graph. This corresponds to replacement of paths of length possibly more than 2 by single edges in the graph. For example, if vertices u and v are split-off and edge (s, u) is paired with (u, v) at u and (u, v) is paired with (v, t) at v, then the path from s to t through u and v is replaced by a single edge (s, t).

Background. Constructive versions of splitting-off have been the subject of much research [6, 8, 1, 13, 2] and its applications appear in [6, 8, 13, 2]. The current best time bounds are due to Gabow [8], who splits-off a single vertex in $\tilde{O}(nc^2 + m)$ time for an undirected graph and $\tilde{O}(mc)$ time for a directed graph, and due to Benczúr and Karger [2] who split-off a single vertex in an undirected graph in $\tilde{O}(n^2)$ Monte Carlo time.

Note that splitting-off a single vertex and recursing leads to an overall quadratic (in n) time bound in applications. To get sub-quadratic algorithms, one needs to be able to split-off multiple vertices faster than successive applications of single vertex splitting. Motivated by this, we present the first algorithms which split off multiple vertices simultaneously.

1.1.1 New Results. Our splitting-off algorithms split off any subset of vertices in the same time as the current fastest algorithm to split off a single vertex due to Gabow [8]. In particular, we show the following constructive theorems. (Note that all the special conditions below, i.e., $c > 1$ and even degree for the undirected case, and in-degree $\geq$ out-degree for the directed case, apply to the corresponding existential theorems as well.)

Theorem 1.1. *Given an unweighted undirected graph and any vertex subset S comprising only even degree vertices, vertices in S can be split-off in $O((nc^2 + m)\log n)$ time so that the connectivity c of the remaining vertices is preserved, provided $c > 1$.*

Theorem 1.2. *Given an unweighted directed graph, a special root vertex r, and any vertex subset $S, r \notin S$ comprising only vertices with in-degree $\geq$ out-degree, vertices in S can be split-off in $O(mc \log n)$ time so that the r-connectivity (connectivity from r as defined earlier) c of the remaining vertices is preserved.*

Further, while Gabow's splitting-off algorithm [8] provides an efficient implementation of previous existential proofs of splitting, our algorithms are self-contained and provide a new alternative proof for edge splitting in the process. In addition, for appropriate ranges of $n, c, |S|$, our algorithm improves the previous best randomized bound due to Benczúr and Karger [2] who split-off a single vertex in an undirected graph in $\tilde{O}(n^2)$ Monte Carlo time.

Applications of our Edge Splitting theorems. A beautiful and classical result of Edmonds [5] shows that an unweighted directed graph with c edge-disjoint paths from any particular vertex r to every other vertex has exactly c edge-disjoint arborescences rooted at r. For a c edge connected unweighted undirected graph G (i.e., the global connectivity of G is c), the same theorem holds on the digraph obtained by replacing each undirected edge by two directed edges, one in each direction. The current fastest construction of these arborescences by Gabow [7] takes $O(n^2c^2)$ time. One of the main applications of our edge splitting theorem is a faster algorithm, more precisely an algorithm whose running time is sub-quadratic in n, to construct Edmonds' arborescences. We show the following results here.

Theorem 1.3. *Edmonds' Arborescences (rooted at any arbitrary vertex) for an unweighted undirected graph (after each undirected edge is replaced by two directed edges) with edge connectivity c can be constructed in $O(m + nc^3 \log^2 n)$ time.*

Theorem 1.4. *Edmonds' Arborescences (rooted at vertex r) for a directed graph with r-connectivity c can be constructed in $O(mc \log n + nc^4 \log^2 n)$ time.*

The above results are particularly appealing when $n >> c$, which is a realistic condition in fabricated networks since adding edges is costly (and therefore, c is small).

We use our efficient splitting-off algorithms for directed graphs to construct Edmond's arborescences as follows: we split-off a sizable fraction of the vertices while maintaining the connectivity of the remaining vertices. We then recurse on the remaining graph, and finally put back the split-off vertices into the resulting arborescences. The last step requires that the set of vertices split-off together at any stage of the recursion form an *independent set* (and therefore, speaking roughly, we split off $\Theta(n/c)$ vertices simultaneously). The algorithm, therefore, splits off vertices in phases, and we show that the total time spent on splitting is only a factor of c greater than the time for splitting-off in a single phase. A further factor of c for the directed case results from the requirement that the set of vertices split-off in a phase is independent and additionally, each split-off vertex has in-degree $\geq$ out-degree.

Another application of our efficient algorithm for splitting-off multiple vertices simultaneously is a fast algorithm for *Steiner edge connectivity.* This problem deals with determining the *edge connectivity* c, i.e., the value $\min_{s,t \in S} \lambda(s,t)$, of a given subset S of vertices. We get an algorithm with running time $O(m + nc^2 \log n)$ for this problem by simply splitting-off the vertices outside S. The running time of this new algorithm matches the running time of the current best algorithm [3] already known for this problem.

1.2 Our Techniques.

Below we summarize the main techniques in our algorithm that leads to Theorem 1.2 stated above. For now, we concentrate on our splitting-off algorithm for directed graphs since this result suffices for the faster construction of Edmonds' arborescences and the efficient algorithm for Steiner edge connectivity.

Our splitting-off algorithm is based on the following classical theorem of Edmonds [4]: *The maximum number of edge-disjoint directionless r-spanning trees in a directed graph is equal to the r-connectivity c*, where a directionless r-spanning tree is a spanning tree rooted at r with edges not constrained to be directed towards or away from r, but each vertex other than r is constrained to have in-degree c over all trees, and r is constrained to have in-degree 0. Note that for undirected graphs, each undirected edge is replaced by two directed edges, one in either direction, to yield a directed graph with r-connectivity equal to the connectivity of the original undirected graph; then the above theorem applies on this directed graph.

Gabow [7] showed a constructive version of the above theorem that constructs c edge-disjoint directionless r-spanning trees in subquadratic (in n) time (time complexity of $\tilde{O}(mc)$ for directed graphs and $\tilde{O}(nc^2 + m)$ for undirected graphs). We extend Gabow's algorithm for edge splitting as follows.

1. We initially perform an arbitrary pairing for all edges incident on each vertex in the set S of vertices to be split off. This yields a graph with vertex set $V - S$ and edges connecting vertices in $V - S$ (each edge is actually a path with endpoints in $V - S$ and internal vertices from S, with internal vertices ignored except as described below).

2. We now start constructing directionless r-spanning trees using Gabow's algorithm, where r is any arbitrary vertex.

3. If the initial arbitrary pairing was good, then Gabow's algorithm will indeed succeed in constructing c directionless r-spanning trees. But, if the initial pairing led to a drop in connectivity below c, then Gabow's algorithm will get stuck before c trees are fully constructed. In this situation, Gabow's algorithm also provides a violating cut C. Our main contribution is the following:

   - We show that there exists an edge e inside C and an edge f partly or fully outside C with both edges containing an internal vertex $s \in S$.
   - We show that revising the initial pairing by *mating* edges e and f (Section 3 has the details) fixes the violating cut C.
   - Most importantly, we maintain the invariant that on repeated application of this procedure, the trees that we have built so far stay connected and the current tree being built makes further progress. Thus we obtain a splitting off of vertices in S while preserving the connectivity of the remaining graph.

For undirected graphs (see Theorem 1.1), note that the directionless trees are built on the directed graph obtained by directing each edge in both directions. This provides an additional challenge, namely that the mating of directed edges $e = (u, v)$ and $f = (u', v')$ has to be coupled with the mating of their corresponding reverse edges $e^{rev} = (v, u)$ and $f^{rev} = (v', u')$; this is necessary for the directed edge splitting to be a genuine one for the underlying undirected edges. Each mating operation then needs a corresponding *reverse mate* to be performed. This leads to several technical hurdles and our contribution lies in showing that such coupled reverse mates can also be executed efficiently.

Roadmap. Section 2 outlines Gabow's directionless tree construction algorithm which we use in Section 3 to outline a proof of Theorem 1.2. Sections 4 and 5 outline proofs of Theorem 1.3 and Theorem 1.4, respectively. For lack of space, we skip the description of a proof of Theorem 1.1 but provide a brief outline in Section 6.

2 Gabow's Algorithm for Directionless Spanning Trees

We outline Gabow's algorithm for constructing c directionless r-spanning trees. Recall that a directionless r-spanning tree is a spanning tree rooted at r with edges not constrained to be directed towards or away from r, but each vertex other than r is constrained to have in-degree c over all trees, and r is constrained to have in-degree 0.

Gabow's algorithm constructs trees one at a time. Suppose $k-1$ trees $T_1, T_2, \ldots, T_{k-1}$ have been constructed. Now, to construct the k^{th} tree T_k, one starts with the n vertices forming a forest with n components and no edges. Overloading our notation, we call this forest T_k as well. Now, T_k is augmented (i.e., the number of components is reduced) over several *rounds*. The following invariants hold at the beginning of each round.

1. The in-degree of the root vertex r over $T_1, \ldots, T_k$ is 0. All the other vertices in the component containing r in T_k have an in-degree of k over $T_1, \ldots, T_k$.

2. Let *Comp* be any component in T_k not containing r. Then, there is *exactly one* vertex in *Comp* (called the *deficient vertex of Comp*) whose total in-degree over $T_1, \ldots, T_k$ is $k-1$. All other vertices in *Comp* have total in-degree of k over $T_1, \ldots, T_k$.

A round aims to reduce the number of components by half, and therefore $O(\log n)$ rounds are sufficient to reduce the number of components in T_k to 1. Each round runs in $O(n+m)$ time (we describe a round below), leading to an overall time of $O((n+m)\log n)$ per tree, or $O(c(m+n)\log n)$ for constructing c trees.

A Single Round. Rounds run independently and in a non-interfering manner on each component of T_k, so we can focus on one particular component *Comp*. The goal of this round, as far as *Comp* is concerned, is to connect it to another component in T_k while incrementing the in-degree of the deficient vertex v in *Comp* by 1 and maintaining the in-degrees of all other vertices.

The simplest case is when we have an edge e with the following properties: e is not yet used in any of the trees, e is directed into the sole deficient vertex v in *Comp*, and e's other endpoint is outside *Comp*. In this case, simply adding e to T_k does the job. However, if no such edge exists, but an unused edge e directed into v exists (this edge is from another vertex in *Comp* to v) then we need to do more work. It may be possible to add edge e to one of the trees T_i and release an edge f belonging to the fundamental cycle created by adding e to T_i, which could join *Comp* to another component. Extending this further, there might be a sequence of such *swaps* which will eventually release a *joining edge* (i.e., an edge connecting *Comp* to another component while incrementing the in-degree of *Comp*'s deficient vertex). Even further, on each swap in this sequence, the edge released can be replaced by an edge that is currently not in any of the trees and that is *incident* on the same destination vertex, for the purposes of the next swap in the sequence. This is because total in-degrees are preserved if an edge is removed and it is replaced by another with the same destination. The algorithm for a round explores the entire search space of all *swap* and *incidence* possibilities until it locates a joining edge, or until the possibility of finding a joining edge is ruled out. This is done in two steps.

First, a *closure computation* process identifies the joining edge for *Comp* and specifies the series of transformations (swap and incidence operations) required to release this joining edge. Second, the series of transformations specified by closure computation are actually performed and the trees are readjusted accordingly and a joining edge is added to T_k. Note that the trees $T_1 \ldots T_k$ are themselves not modified by the closure computation process, rather this process just provides a list of modifications to be performed.

Closure Computation. The following data structures are used in the closure computation process.

1. A global set of edges F, initialized to all *unused edges* (edges not present in any of $T_1, \ldots, T_k$) directed into α. An edge in F has the following properties:

   (i) If it is currently used in any of the trees, then there is a sequence of swap and incidence operations which can be used to release it if needed.

   (ii) If it is currently unused then either it is incident on α or it is incident on some other vertex β and a used edge directed into β can be released via a sequence of swap and incidence operations.

2. A global set of vertices C, initialized to α, the deficient vertex in *Comp*. This is the set of endpoints of edges in F. Eventually, when the closure computation procedure gets stuck, Gabow shows that $(V-C, C)$ forms a minimum r-cut (i.e., r is on the source side).

3. Vertex sets $C_i \subseteq C$ of vertices in tree T_i, $1 \le i \le k$, initialized to α. Vertices in C_i are exactly the endpoints of edges in $F \cap T_i$ along with the vertex α. An important property of C_i is that vertices in C_i appear contiguously in T_i.

The closure computation process adds edges to F and vertices to C and C_i's via a set of two rules, the *swap* and *incidence* rules, described below.

Swap Rule. For a given tree T_i, if edge $(u,v) \in F$ is such that $u \in C_i$ while $v \notin C_i$ (or vice-versa), then:

1. add all edges in T_i on the path joining v and C_i (recall that since C_i is contiguous in T_i, a unique

path is present in T_i between v and C_i).[2]

2. add all vertices on the above path in T_i to C_i and to C.

Note that the condition that exactly one of u, v is in C_i ensures that C_i expands and that it stays contiguous.

Incidence Rule. If edge $(u, v) \in F$ and (w, v) is unused, then add (w, v) to F. Correspondingly add w to C.

The rules themselves are applied by processing $T_1, \ldots, T_k$ in a cyclic order. At a particular tree T_i, all edges added to F when T_{i-1} (or T_k if $i = 1$) was processed are first considered for application of the swap rule. Gabow ensures that when an edge is considered for the swap rule exactly one of its endpoints is in C_i (recall from above that this is essential for making progress and for maintaining contiguity of C_i). This is done via an ordering on these edges, namely edges closer to C_{i-1} are considered before edges farther away, along with the following lemma.

Lemma 2.1. *During this round-robin process, if T_i is being currently considered, then $C_i \subseteq C_{i+1} \subseteq \ldots \subseteq C_k \subseteq C_1 \subseteq \ldots \subseteq C_{i-1}$.*

Now, edges with one endpoint outside C_i lead to addition of new edges to F by the swap rule. If no new edge is added to F, then the cyclic round-robin process is stopped and the procedure terminates. (We call this *termination with failure.*) On the other hand, if at least one new edge is added to F, then the incidence rule is applied on each new edge added to F and the cyclic process moves on to the next tree. If the process does not terminate with failure, then it terminates when an edge with one endpoint outside *Comp* is added to F. (We call this *termination with success.*) The total time taken before termination is proportional to the number of vertices and edges incident on the current component, giving $O(n + m)$ time over all components.

Termination with Failure. If even one component *Comp* terminates with failure, the procedure is said to have terminated with failure. In this case, Gabow shows that $C_1 = C_2 = \cdots = C_k = C$ and that $C, V - C$ actually forms an r-cut of size $k - 1$. In this case, the algorithm terminates as $k - 1$ trees have already been constructed. The proof that $C, V - C$ forms an r-cut of size $k - 1$ is based on the following facts.

1. Each vertex other than α in C has in-degree k in $T_1, \ldots, T_k$ and vertex α has in-degree $k - 1$ in $T_1, \ldots, T_k$.

2. Vertices in C occur contiguously in each of $T_1, \ldots, T_k$.

3. Edges not in $T_1, \ldots, T_k$ but directed into a vertex in C lie completely within C. Thus, edges directed into C from $V - C$ must all be in $T_1, \ldots, T_k$.

An easy consequence of the first two properties is that the in-degree of C in $T_1, \ldots, T_k$ is $k - 1$. The third property ensures that the in-degree of C in the whole graph is also $k - 1$.

Termination with Success. For any component *Comp*, let $g \in F$ be the edge which connects a vertex in *Comp* to a vertex outside *Comp*. In this case, Gabow shows how to find a sequence of swap and incidence adjustments[3] to $T_1, \ldots, T_k$ culminating in the addition of g to T_k without violating any of the original invariants. Since all components have met with success, the number of connected components in T_k reduces by half.

Note that finding the above sequence of adjustments is non-trivial because the closure computation procedure is run on static trees $T_1, \ldots, T_k$ without accounting for the fact that initial swaps in this sequence can change the trees potentially rendering later swaps invalid. Gabow shows that this never happens because each swap works on a portion of the trees which previous swaps have not touched. More specifically, if a swap happens in one of the trees T_i and C_i^{prev} is the value of C_i when T_i was last processed in the round robin scheme, all changes to T_i thus far are localized within C_i^{prev} which continues to be contiguous in T_i, and the edges participating in the swap at this stage are not internal to C_i^{prev}. This ensures that previous changes do not affect the current swap.

3 Splitting Off Multiple Vertices in Directed Graphs

Vertices which are to be split-off are termed *black* vertices while the remaining vertices are termed *white*. We assume that black vertices have in-degrees equal to their respective out-degrees (otherwise, we can balance the

[2] The set of edges added are traversed from the lower end-point to the upper end-point of the added path if v and a vertex in C_i share an ancestor-descendant relationship and from v and the appropriate end-point of C_i to their *least common ancestor* (or, lca) otherwise. Since the above rules are applied on static trees $T_1, \ldots, T_k$ without making any changes to the trees themselves, determining the traversal sequence is a constant time operation (by storing the depth of each vertex in each tree and the lca's in each tree using a pre-processing step).

[3] A swap adjustment adds a released edge to a tree and releases an edge in the fundamental circuit formed by the added edge. An incidence adjustment simply replaces a released edge by an unused edge directed into the same vertex.

excess in-degree by additional edges to the root vertex r without affecting connectivity from r). Incoming and outgoing edges on black vertices are now paired up arbitrarily to yield white-white edges, each of which could have black vertices internally. Note that a single black vertex has as many occurrences as an internal vertex as its in-degree/out-degree. Pairing incoming and outgoing edges on black vertices could also lead to some cycles comprising black vertices alone. (We call these *self loops.*) These self loops have the following property which is simple to show.

LEMMA 3.1. *Consider the set of all black vertices B which have at least one occurrence on a self loop. There exists some black vertex $b \in B$ such that b has at least one occurrence on a white-white edge.*

Using the above lemma, we insert an entire self loop containing b into a white-white edge containing b. (Note that a black vertex is allowed to have multiple occurrences on a white-white edge.) We repeatedly use the above procedure until no self-loop is left. This entire process takes $O(m)$ time.

We now run Gabow's algorithm for constructing directionless trees on the graph comprising white vertices, ignoring black vertices for the moment. Consider the closure computation procedure run for a particular component *Comp*. On termination with failure, while we can claim that white vertices involved in C are contiguous in all trees constructed so far, we cannot claim the same of black vertices hidden inside the various edges, and therefore we cannot claim that we have found a $k-1$-sized cut. To handle black vertices, we introduce the notion of *mating*, as described below.

3.1 Modified Closure Computation Details. The closure computation procedure is now run for the white-white edges (with internal black vertices). For a particular component *Comp* in T_k with deficient vertex α, this new closure computation process has the following notable differences from the corresponding procedure described in Section 2.

1. Set C and the sets C_i for $i = 1, \ldots, k$, initialized to α, will comprise white vertices and black vertex occurrences (each occurrence is treated as a separate vertex; unlike white vertices black vertices can occur multiple times (or never at all) in a tree and this requires us to treat each black vertex occurrence separately).

2. When applying the swap rule or incidence rule, all black occurrences and white vertices on edges added to F are added to C and C_i.

3. A *mating* rule described below is used in addition to the *swap* and *incidence* rules.

Before defining the mating rule, let us try to get some intuition into it. Recall that if we get stuck, we would like to claim that C is contiguous in all the trees, in which case we can claim that there is a $k-1$ cut in the graph using Gabow's proof. However, though this is true for white vertices (which participate in swap and incidence rule invocations exactly as earlier), this might not be true for the black vertices lying on the white-white edges. Contiguity demands that the presence of one occurrence of a black vertex inside C requires the presence of all occurrences of this black vertex inside C. In other words, there could be a black vertex b which has one occurrence inside C (on an edge (u, v), say) and one outside it (on an edge (u', v'), say). Another way of looking at this is that the edge pairings have erroneously produced a $k-1$ cut and we need to revise the pairing to increase the cardinality of this cut. One way of doing this is by re-adjusting the white-white edges $u-b-v$ and $u'-b-v'$ to produce two new edges $u-b-v'$ and $u'-b-v$. As shown in Figure 1 (ignore the label regarding substitute edge for the moment), this adds two new edges to the $k-1$-sized cut, thus increasing its cardinality. This re-adjustment of the pairing is called *mating*. We give a more formal definition of the mating rule below.

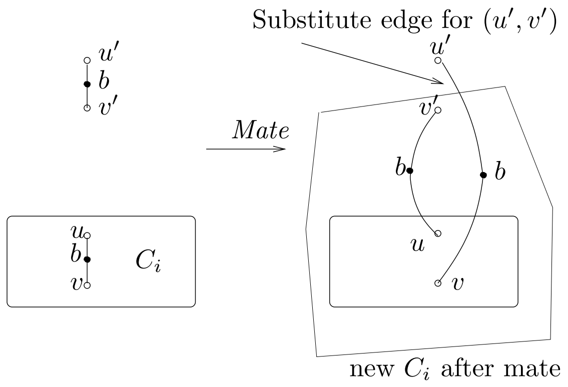

Figure 1: Mating between edges (u, v) and (u', v').

Mating Rule. This rule is applied only when neither the swap rule nor the incidence rule is applicable. If $(u, v) \in F$ is an edge containing a black vertex occurrence b and (u', v') is an edge containing another occurrence of b not present in C, then we do the following:[4].

[4]Note that edges in F are releasable from the existing trees, so (u, v) should be considered as a released edge in the description of the mating rule. Further, the description of the mating rule corresponds implicitly to the destruction of e, f and the creation of

- if (u', v') is unused, add vertices from u' to b on (u', v') to C and add the full edge formed by joining (u', b) (from (u', v')) and (b, v) (from (u, v)) to F.
- if (u', v') is used in tree T_i then let x be the nearer of u', v' to C_i (since C_i is contiguous, this is well defined). If $x = v'$ then add the full edge formed by joining (u, b) (from (u, v)) and (b, v') (from (u', v')) to F. And if $x = u'$ then add the full edge formed by joining (u', b) (from (u', v')) and (b, v) (from (u, v)) to F. Also, add all edges on the path from x to C_i in T_i to F. Finally, add the vertex occurrences on the path from b to C_i in (u', v') to C and to C_i.

Recall that these rules run on static trees $T_1, \ldots, T_k$. So the computation of nearer endpoint in the description above takes constant time (using a pre-processing step to store the depth of each vertex and lca's in each tree initially).

Implementation of Closure Computation. The closure computation procedure is now divided into stages. Each stage (other than the first one) begins with an instance of the mating rule, followed by one or more swap and incidence rule applications. Swap and incidence rule applications are performed by processing $T_1, \ldots, T_k$ in a cyclic order as in Gabow's algorithm. If processing T_i leads to no new additions to F (because none of the edges had an applicable swap rule), then the cyclic round-robin process is stopped and the stage ends. Lemma 3.2 holds when a stage ends.

Lemma 3.2. *The set of whites in all the C_i's is identical.*

To begin the next stage, we look for a black vertex b with the properties described in the mating rule. If such a vertex is found along with the requisite edges (u', v') and (u, v), then the mating rule is applied. This mate is guaranteed to add at least one new vertex occurrence to C. If this adds a white vertex to C then the cyclic round-robin process is resumed, starting from T_1 if (u', v') is not in any tree and starting from T_i if $(u', v') \in T_i$. In either case, the edge considered for swapping is the lone edge added to F by the mating rule. On the other hand, if only black vertices are added to C by the mating, then this stage ends as well and the next stage begins (with another application of the mating rule).

Termination. Termination happens when an edge connecting *Comp* to another component gets added to F via a swap or incidence rule (*success*), or when a stage ends and none of the three rules applies (*failure*). If any of the components of T_k results in failure, then Lemma 3.3 holds based on arguments analogous to Gabow's algorithm.

two new edges $e' = (u', v)$ and $f' = (u, v')$, as shown in Figure 1; however, closure computation does not actually destroy edges as it runs on static trees $T_1, \ldots, T_k$. Actual destruction of edges happens later in Section 3.2

Lemma 3.3. *On termination with failure, for a black vertex b with some occurrence in C, all occurrences of b are in C, and therefore, by Lemma 3.2, the cut size of C in G is $k - 1$.*

3.2 Termination with Success

In this case we need to demonstrate a sequence of swap, incidence and mating steps which would release an edge connecting *Comp* to another component. As before, the issue is that the closure computation works on static trees while actually performing swaps and mates might change these trees rendering future swaps and mates invalid. Our solution to these problems is along the lines of Gabow's algorithm, i.e., we can show that the portions of a tree involved in a swap or mate are outside the contiguous portion that could have been changed by previous operations. However, there are two additional challenges.

Cross Component Interference. First, recall that the mating operation requires determination of the endpoint nearer to C of one of the participating edges and that the closure process does this computation statically. However mating *destroys* edges (mating edges e and f destroys the edges e and f), so when the modification sequence is actually applied, mates for one component cause edge destruction which in turn could cause change of the nearer endpoint for mates in another component. (We call this *cross component interference.*) To overcome this problem, we show the following.

Lemma 3.4. *For any component Comp, only the very last mate in the swap/incidence/mate sequence for Comp can be affected by cross component interference due to a mate in some other component.*

We can also give a procedure which performs last mates for a constant fraction of the components all together in one global pass in overall time $O(n + m)$.

Mating Destroys Edges. Second, mating destroys edges and a swap might use an edge which has been destroyed previously in a mating step. To handle this, recall that (1) the edges added to F by the swap rule applied on an edge $e = (u, v)$ (let $u \in C_i$) are the edges on the path joining C_i to v in tree T_i (let us call this path P_i), (2) all modifications in the trees due to the swap/incidence/mates performed previously are restricted to the C_i's in the trees, and

(3) C_i's might have changed internally in the trees, but are still contiguous in the trees after the previous swaps/incidences/mates.

From observations (1), (2) and (3), it follows that the path P_i is not affected by the previous swaps/ incidences/mates. In other words, the path from C_i to v has the same set of edges as in the static trees (that were used during closure computation). From observation (1), we also conclude that the fundamental circuit formed by adding to T_i any white-white edge *which is identical to e outside* C_i *and possibly different from e inside* C_i contains the path P_i. Such an edge can act as a substitute for e in the swap rule applied to T_i. This is illustrated in Figure 2 where the edge (u', v) can act as a substitute for the edge (u, v).

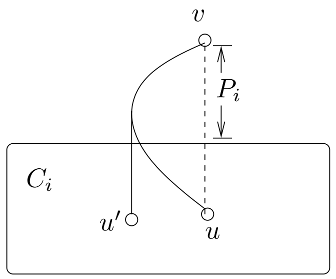

Figure 2: The edge $(u'v)$ can act as a substitute for the edge (u, v).

But, are we always guaranteed to find such a substitute edge for an edge destroyed by a mating? It turns out that one of the edges resulting out of this mating itself can serve as the substitute edge. More formally, we define a *substitute* edge $s(g)$ for each edge g which is destroyed via mating before it enters F as the edge resulting from this mating which is not added to F. (For an illustration, refer to Figure 1). Now, in the event that $s(g)$ itself is further destroyed by mating before g enters F, we redefine the substitute of g to be the substitute of $s(g)$, and so on. Define $C(g)$ to be the value of C just before g gets added to F. Now, Lemma 3.5 asserts that $s(g)$ is exactly identical to g outside $C(g)$ and may be different inside $C(g)$. But, recall from the previous discussion that g's role in a swap can be performed by any edge which is identical to g outside $C(g)$ and possibly different inside $C(g)$. Thus, $s(g)$ serves as a substitute for g (Theorem 3.1).

Lemma 3.5. *At any point of time, $s(g)$ is identical to g in parts outside the current C and possibly different from g in the parts inside C. When g enters F, the final $s(g)$ satisfies the above property with C replaced by $C(g)$.*

Theorem 3.1. *Given the sequence of rules aimed at releasing an edge e, and given substitutes (as defined above) for each of the edges involved in this sequence, operations in this sequence can legitimately be performed on these substitute edges. Further, these operations will affect only portions of trees within $C(e)$ in addition to the edge e itself.*

Thus, these substitute edges overcome the problem of edges being destroyed by mating. We now analyze the running time. Prior to the last mate, the total time taken for a component is proportional to the number of vertices and edges incident on the component. The last set of mates involves a global pass of time $O(m+n)$ because of cross-component effects. This gives $O(m+n)$ overall time over all components in a round, thus showing Theorem 1.2.

4 Arborescence Packing Algorithm for Undirected Graphs

We now apply Theorem 1.2 (splitting in directed graphs) to show Theorem 1.3 (packing arborescences in undirected graphs). We start by applying the Nagamochi-Ibaraki construction [12] to reduce the number of edges in the graph to at most $(n-1)c$, while maintaining a connectivity of c, in time $O(m)$. Next, we replace each undirected edge by two directed edges, one in each direction, to create an Eulerian directed graph with strong edge connectivity c. Note that we construct arborescences on this directed graph, so our splitting-off theorem for directed graphs, Theorem 1.2, suffices here.

The Recursion. We find a maximal independent set using a simple greedy algorithm, split off all the vertices in the independent set simultaneously using Theorem 1.2, and recurse on the remaining graph. We are guaranteed both Eulerianness and a connectivity of c in each step of the recursion.

Subsetting Edges. Amongst the edges in the graph for the current recursive sub-problem, we use only the following subset for invoking Theorem 1.2.
(i) All edges incident on the vertices in the independent set (i.e., vertices which are being split-off in this sub-problem).
(ii) Edges which belong to the c directionless trees that are constructed on the parent Eulerian graph by the splitting-off procedure at the parent.

The former ensures in-degree equals out-degree for vertices being split out and the latter ensures connectivity at least c from r. Together these satisfy the conditions needed (for splitting-off vertices in a directed graph) by Theorem 1.2. The total time complexity of the splitting-off process follows from the following lemma.

Lemma 4.1. *Vertex cardinalities add up to $O(nc \log n)$ and edge subset cardinalities add up to $O(nc^2 \log n)$, over all recursive sub-problems. The total time taken is therefore $O(m + nc^3 \log^2 n)$.*

Proof. Any recursive sub-problem with n/x vertices and at most $2nc$ edges has an independent set of size $\Omega(n/x^2c)$. The next recursive sub-problem would then have $O((1 - 1/xc)n/x)$ vertices. Splitting x into doubling ranges and adding up yields the lemma. For edges, consider the two categories of edges described above. The first adds up to $O(nc)$ over all sub-problems. The second is clearly c times the number of vertices in a sub-problem, so that adds up to $O(nc^2 \log n)$.

Putting vertices back. The recursion bottoms out when we have 2 vertices remaining. Constructing c arborescences on the c-connected pair of vertices is trivial. Once we have built these arborescences, we put back the vertices that we had split off at each step in reverse order of splitting-off to obtain the final set of arborescences. If an instance of the vertex v is contained on an arborescence edge, it is simply made explicit. After this, if vertex v does not occur c times in the arborescences, then there is an unused edge which contains v internally (since v must be contained on at least c edges). The prefix of this edge up to v can be attached to any of the arborescences (we use independence here, i.e., the other end-point of this prefix is present in each of the arborescences returned). The only problem remaining is to convert multiple occurrences of a vertex in a tree to single occurrences in multiple trees. This is done by a transformation which moves subtrees rooted at distinct occurrences of vertex v in a tree to one of the occurrences thereby converting all the other occurrences to leaves which can be migrated to any other arborescence (again, we use independence here). We show that this can be done using a single bottom-up pass on each tree, leading to $O(nc^2 \log n)$ time overall.

5 Arborescence Packing Algorithm for Directed Graphs

For directed graphs, the Nagamochi-Ibaraki construction does not apply, even initially, and the initial set of m edges cannot be limited to $O(nc)$ edges. We use Gabow's directionless tree construction procedure [7] to obtain a subgraph with $O(nc)$ edges in $O(mc \log n)$ time, and work with this new graph instead. Since each vertex (other than the root) has in-degree exactly c in this subgraph, a simple counting argument shows that there are at least $\Omega(n/c)$ vertices with out-degree $\leq c$ in this subgraph. All these vertices satisfy in-degree $\geq$ out-degree and have total degree (sum of in-degree and out-degree) of at most $2c$. Using a greedy algorithm, we can identify an independent set of size $\Omega(n/c^2)$ satisfying in-degree $\geq$ out-degree among these vertices. Our algorithm splits-off this independent set using Theorem 1.2 and recurses. So each recursive step involves running Gabow's directionless tree algorithm followed by a splitting-off step, each of which takes $O(n'c^2 \log n)$ time, where n' is the number of vertices in the current graph. Vertices are then put back exactly as before. The total time over the $O(c^2 \log n)$ levels of recursion, along with $O(mc \log n)$ pre-processing time, becomes $O(mc \log n + nc^4 \log^2 n)$.

6 Splitting-off in Undirected Graphs

As mentioned in the introduction, the additional complication in this case is that the (forward) mates and reverse mates have to be performed in a coupled manner. We perform exactly the same closure computation as before. The same procedure is used to determine the actual sequence of swaps/mates/incidence rules to be applied. The changes are only when this sequence is actually applied. For each mate, first the forward mate is performed, followed by all the swaps/incidences in the closure computation seeded by the edge resulting from this mate which goes into F. Finally, the reverse mate is performed when the stage comes to an end and a new mate has to be performed to start the next stage. Thus each stage has a forward mate, a sequence of swaps/incidences and finally, the reverse mate corresponding to the initial forward mate.

Reverse Mates. Consider a stage that begins with a forward mate on edges e, f and terminates in the reverse mate on edges e^{rev} and f^{rev}. A challenge now is to show that the reverse mate can be performed without disconnecting any of the existing trees, affecting any of the vertex in-degrees or impacting any future operation. We show the following lemma in this case.

Lemma 6.1. *For $k \geq 2$, one of the following two options will ensure that $T_1, \ldots, T_k$ do not get disconnected (for T_k, this means no new connected components are created) and see no change of total in-degrees, and further, all operations in future stages are unaffected except for appropriate changes in the actual values of edge substitutes $s()$ defined in Lemma 3.5.*
(i) Mate on edges e, f and reverse mate on edges e^{rev} and f^{rev}.
(ii) Mate on edges e^{rev}, f and reverse mate on edges e and f^{rev}.

Algorithmically, the (forward) mate always uses the first option in Lemma 6.1, and which of the two options actually works is determined at the time of the reverse mate. A change of option at the time of the reverse mate may require the forward mate and consequent swaps and incidences in this stage to be redone. We need the following definition and lemma to determine the time complexity of this whole process.

Definition. For the current stage, let C^1 denote the set C at the beginning of the previous stage, let C^2 denote the corresponding quantity at the beginning of the current stage, and let C^3 denote the corresponding quantity at the beginning of the next stage. In the directed case, C^1, C^2, C^3 would be contiguous in all trees. Reverse mates, however, could cause a disruption in the contiguity of these vertex sets. The following lemma puts a limit on this disruption.

LEMMA 6.2. *C^2 and C^3 are contiguous in all trees while C^1 might not be contiguous in at most one tree, where it could have two connected components. Further, if the edge e^{rev} is used, it does not have both end-points in the same component of C^1 in the tree where it is used.*

We show using an elaborate case analysis that the right option in Lemma 6.1 can be determined during the reverse mate by a tree structure exploration which stays within C^3 and does not get into C^1; by Lemma 6.2 this takes time proportional to $|C^3| - |C^1|$, adding up to the overall component size over all stages. Further, since C^2 is contiguous, we show that a change of option during the reverse mate can be handled simply by flipping endpoints and directions for the two edges resulting from the forward mate.

The proof of Lemma 6.1 requires the condition that when a reverse mate is performed, the two edges involved in a reverse mate and an edge resulting from the corresponding forward mate or both edges resulting from the forward mate and e^{rev} should not be in the same tree. If this does indeed happen, we show how one of these edges can be moved out to another tree in $|C^3| - |C^1|$ time. But this requires that $k \geq 2$ when running the above procedure. In other words, the above procedure does not work for constructing the first tree. We provide another procedure for constructing the first tree using post-order traversal of the DFS tree in $\tilde{O}(n+m)$ time; this procedure requires that $c \geq 2$ (recall c is edge connectivity of the vertices that remain after splitting).

Conclusions and Open problems. In this paper we showed the first sub-quadratic (in n) construction of Edmonds' arborescences for directed and undirected graphs on n vertices. Our algorithm took $O(m + nc^3 \log^2 n)$ time for the undirected case and $O(mc \log n + nc^4 \log^2 n)$ time for the directed case, where m is the number of edges and c is the edge connectivity. This improves the previous fastest construction by Gabow which took time $O(n^2c^2)$ time. Our algorithm constructed these arborescences via a new deterministic algorithm for edge splitting. We presented efficient edge splitting algorithms ($O(mc \log n)$ time for directed graphs and $O((nc^2 + m) \log n)$ time for undirected graphs) to split off any specified subset S of vertices (satisfying standard conditions) while maintaining the edge connectivity c of vertices outside S. An important open question here is to extend this procedure to maintain not just global connectivity but local connectivities for all pairs of vertices outside S.

References

[1] A.A. Benczúr, *Cut structures and randomized algorithms in edge-connectivity problems*, Ph.D. Dissertation, M.I.T, Cambridge, 1997.

[2] A. Benczúr and D. R. Karger, *Approximating s-t Minimum Cuts in $\tilde{O}(n^2)$ Time*, J. Algorithms 37(1), pp. 2-36, 2000.

[3] A. Bhalgat, R. Hariharan, T. Kavitha, D. Panigrahi. *An $\tilde{O}(mn)$ algorithm for Gomory-Hu tree construction for unweighted graphs*, Proc. of ACM STOC, pp. 605-614, 2007.

[4] 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, pp. 69–87, 1969.

[5] J. Edmonds, *Edge-disjoint branchings*, Combinatorial Algorithms, Algorithmics Press, New York, pp. 91-96, 1972.

[6] A. Frank, *Augmenting Graphs to Meet Edge-Connectivity Requirements*, SIAM J. Discrete Math. 5(1), pp. 25-53, 1992.

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

[8] Harold N. Gabow, *Efficient splitting off algorithms for graphs*, Proc. of ACM STOC, pp. 696-705, 1994.

[9] L. Lovász, Lecture, Conference of Graph Theory, Prague, 1974.

[10] L. Lovász, *Combinatorial Problems and Exercises*, 2nd Ed., North-Holland, New York, 1993.

[11] W. Mader, *Konstruktion aller n-fach kantenzusammenhängenden Digraphen*, European J. Combin. 3 1, pp. 63-67, 1982.

[12] Hiroshi Nagamochi and Toshihide Ibaraki, *A Linear-Time Algorithm for Finding a Sparse k-Connected Spanning Subgraph of a k-Connected Graph*, Algorithmica 7(5&6), pp. 583-596, 1992.

[13] H. Nagamochi and T. Ibaraki, *Deterministic $O(nm)$ time edge-splitting in undirected graphs.* Journal of Combinatorial Optimization, 1(1), pp. 5–46, 1997.