

# An Algorithm for Enumerating All Spanning Trees of a Directed Graph

Sanjiv Kapoor\*      H. Ramesh†

## Abstract

We present an  $O(NV + V^3)$  time algorithm for enumerating all spanning trees of a directed graph. This improves the previous best known bound of  $O(NE + V + E)$  ([1]) when  $V^2 = o(N)$ , which will be true for most graphs. Here,  $N$  refers to the number of spanning trees of a graph having  $V$  vertices and  $E$  edges. The algorithm is based on the technique of obtaining one spanning tree from another by a series of edge swaps. This result complements the result in the companion paper ([2]) which enumerates all spanning trees in an undirected graph in  $O(N + V + E)$  time.

**Key words.** spanning tree, directed graph, enumeration.

## 1 Introduction

Spanning tree enumeration in directed graphs is an issue in some problems encountered in network and circuit analysis. Applications are given in ([3], pp 252-364).

The previously best known algorithm due to Gabow and Myers ([1]) used a searching technique based on depth first search and had a time complexity of  $O(NE + V + E)$  and a space complexity of  $O(V + E)$ . Here  $N$  refers to the number of spanning trees of a directed graph with  $V$  vertices and  $E$  edges.

The algorithm presented in this paper takes  $O(NV + V^3)$  time and  $O(V^2)$  space for enumerating all the spanning trees. While the space required is more than that in Gabow and Myers' algorithm ([1]), our algorithm is faster when  $V^2 = o(N)$ . Note that for most graphs,  $N$  will indeed be large.

Our algorithm uses the paradigm followed in a companion paper ([2]) for enumerating spanning trees in undirected graphs. The main fact used is that a spanning tree can be obtained from another spanning tree by replacing edges in it by edges outside it. We first obtain one spanning tree of the graph. Then edges external to it are characterized as back, cross and forward edges. A cross or forward edge may be exchanged for an edge in the spanning tree having the same tail, to result in a new spanning tree of the graph. Repeating this procedure for all cross and forward edges, gives all spanning trees which can be

---

\*Indian Institute of Technology, Delhi.

†Indian Institute of Science, Bangalore. Work done partly at the Indian Institute of Technology, Delhi.

obtained from the original spanning tree by an exchange of one edge. The entire procedure mentioned above is repeated with each of the new spanning trees obtained in order to generate all the spanning trees of the graph. To prevent repetitions, an inclusion/exclusion principle is used which eliminates certain edges from consideration while exchanging. The computation forest describing this procedure has a number of computation trees. Each tree in this forest enumerates arborescences rooted at one particular vertex of the graph. Each node in a particular computation tree represents an arborescence of the graph, and the arborescences associated with a node and its parent in the tree, differ in exactly one pair of edges.

Our algorithm can output either all the spanning trees explicitly or in an implicit form (as in our companion paper on enumerating spanning trees in undirected graphs ([2])). The implicit form corresponds to outputting the first spanning tree explicitly; the remaining spanning trees are then output as differences from the previous spanning tree in the sequence of spanning trees to be output. The advantage of the implicit representation is that it is concise, i.e., has size  $O(N)$ . In either case, our algorithm has the same performance, as does the algorithm of Gabow and Myers ([1]).

Section 2 gives some necessary definitions, section 3 gives an outline of the algorithm, section 4 describes the details of the scheme.

## 2 Definitions

Let  $G$  be a directed graph with  $V$  vertices,  $E$  edges and  $N$  spanning trees (arborescences). A *spanning tree of  $G$  rooted at  $v$*  is a spanning tree of  $G$  having a unique path from  $v$  to every other vertex with edges directed away from the root  $v$ . Let  $N(v)$  denote the number of spanning trees rooted at  $v$ . Then  $N = \sum_{v \in G} N(v)$ .

An *exchange* for a spanning tree  $T$  of  $G$  rooted at  $v$  is a pair of edges  $(e, f)$ , where  $e \in T$ ,  $f \in E - T$  and  $T - \{e\} \cup \{f\}$  is a spanning tree rooted at  $v$ . An edge not in a spanning tree  $T$  is a *back edge* relative to  $T$  if its tail is an ancestor of its head in  $T$ , a *forward edge* if its tail is a descendant of its head in  $T$ , and a *cross edge* otherwise. Forward and cross edges are collectively referred to as *non-back edges*. The *least common ancestor (lea)* of two vertices in a tree is the last common vertex on the paths from the root to the two vertices. A vertex in a tree is considered to be an *ancestor* but not a *proper ancestor* of itself. *Tail( $f$ )* and *head( $f$ )* refer to the tail and head vertices of edge  $f$  in  $G$ , respectively.

## 3 Algorithm Outline

In this section, we present an outline of the algorithm for generating all spanning trees of a directed graph. As in the first part of our paper ([2]), a computation tree is generated as follows. The algorithm starts off with a directed spanning tree,  $T$ , rooted at particular vertex,  $v$ . All other spanning trees rooted at  $v$  are generated from  $T$  by exchanging non-tree edges with edges in  $T$ . All possible directed spanning trees may be generated by starting with directed spanning trees at each of the vertices. However note that the

exchanges in the directed case are more specific due to the nature of the non-tree edges as illustrated by the following properties.

**Property 1:** Every non-back, non-tree edge,  $f$ , relative to a spanning tree  $T$ , may replace exactly one tree edge,  $e$ , in the spanning tree, namely the edge having the same tail, to result in a new spanning tree rooted at the same vertex as  $T$ .

**Property 2:** A back edge can not be exchanged for any edge in the spanning tree to get a new spanning tree.

To construct all spanning trees rooted at some particular vertex  $v$ , we consider the computation tree obtained by a search procedure using the principle of inclusion and exclusion. The search procedure starts off with a tree at the root of the computation tree. It then constructs sons of the node by the exchange property. From properties 1 and 2, it follows that unlike the undirected case, the exchanges are limited to a single exchange for each non-back non-tree edge.

More formally, let  $CD(G, v)$  be the computation tree which generates all spanning trees of the directed graph  $G$  with root  $v$ . At every node  $a$  of  $CD(G, v)$ , there is a spanning tree  $SD_a$  rooted at  $v$ . Node  $a$  has two sons  $b1$  and  $b2$ .  $SD_{b1}$  is obtained from  $SD_a$  by exchanging  $f$  with  $e$ , where  $f$  is a non-tree non-back edge and  $e$  is the unique tree edge with the same tail as  $f$ .  $SD_{b2}$  is the same as  $SD_a$ . The significance of  $b2$  is that the subtree rooted at  $b2$  will not include  $f$  in any spanning tree. This is captured by maintaining two sets,  $IN$  and  $OUT$ , with every node in the computation tree.  $IN_a$  is the set of all edges which must be present in the rooted spanning trees that correspond to nodes in the subtree rooted at the node  $a$  and  $OUT_a$  represents all edges that are not present in the same spanning trees. The  $IN$  and  $OUT$  sets at sons  $b1$  and  $b2$  are obtained from the parent node  $a$  as follows:

$$\begin{aligned} IN_{b1} &= IN_a \cup \{f\} \\ OUT_{b1} &= OUT_a \cup \{e\} \cup \{\text{all edges in } E \text{ incident upon } tail(e)\} - \{f\} \\ IN_{b2} &= IN_a \\ OUT_{b2} &= OUT_a \cup \{f\} \end{aligned}$$

The  $IN$  and  $OUT$  sets for the root node are empty.

We show next that  $CD(G, v)$  suffices to generate all directed spanning trees of  $G$ , rooted at vertex  $v$ .

**Lemma 3.1**  $CD(G, v)$  has at its nodes all directed spanning trees of  $G$  rooted at vertex  $v$ .

**Proof:** The proof follows from induction and the inclusion/exclusion principle. Let  $a$  be the root node of the computation tree and let  $b1$  and  $b2$  be its left and right sons. Let  $SD_{b1} = SD_a - \{e\} \cup \{f\}$ . Then the computation subtree rooted at  $b2$  forms the computation tree  $CD(G - \{f\}, v)$ . The computation subtree rooted at  $b1$  generates all

directed spanning trees rooted at  $v$ , which include the edge  $f$ . Note that all edges in  $G$  with the same tail as  $f$  have been removed in this subtree. This removal is valid since no spanning tree rooted at  $v$  and including the edge  $f$  may include any of these edges  $e$ .  $\square$

The algorithm for constructing the computation tree follows a recursive strategy. At the root we start with a directed spanning tree. In general at each node,  $a$ , we have a directed spanning tree. A non-back edge,  $f$ , is chosen to construct a new spanning tree by the exchange  $(e, f)$  where  $e$  is the corresponding exchange edge given by property 1. The same procedure is repeated at the sons  $b1$  and  $b2$  of the node  $a$  in the computation tree as described above. Note that to construct the entire sub-computation tree rooted at each node  $a$ , one needs to find the set of non-back, non-tree edges since every such edge leads to an exchange resulting in a new spanning tree. An exchange results in changing this set since some back edges may be converted to non-back or vice-versa. However we show that if we use a depth-first tree as the starting spanning tree at the root node and order non-back edges by the postorder number of their tail vertices in the spanning tree then no non-back edges are converted to back-edges thus simplifying the changes that need to be determined. We use this strategy in our algorithm in the next section.

## 4 Algorithm Description.

This section describes in detail an algorithm for generating all directed spanning trees of  $G$  rooted at a particular vertex  $r$ .

The algorithm begins with a DFS tree of  $G$  (rooted at  $r$ ) at the root of the computation tree  $CD(G, r)$ . For each node  $a$  of the computation tree, two sets  $NB$  and  $B$  are maintained.  $NB$  is a set of those non-tree edges which are non-back w.r.t the directed spanning tree  $SD_a$  and which are not included in  $OUT_a$ .  $NB$  is maintained as a list of non-empty lists, with each list containing edges incident upon a particular vertex. The lists in  $NB$  are arranged in postorder number of the corresponding tail vertex in  $SD_a$ . Note that insertion of a new edge in  $NB$  takes  $O(V)$  time.  $B$  is a set of those back edges w.r.t  $SD_a$  which are not included in  $OUT_a$  and which may be useful, i.e., converted to non-back, at some proper descendant of  $a$  in the computation tree. At the root of  $CD(G, r)$ ,  $B$  contains all back edges w.r.t the spanning tree at the root. The data structure for  $B$  will be detailed later. Some of the back edges w.r.t  $SD_a$  which have been identified at node  $a$  to be not useful in the above manner, constitute the set  $RB_a$ .

At every node in the computation tree, the first edge in the first list in  $NB$  is used as an exchange edge. The sets  $NB$  and  $B$  are updated at every exchange. The updates to these sets involve transferring edges from  $B$  to  $NB$  and removal of edges from  $B$  and  $NB$ . It is proved later that these changes suffice and no changes from  $NB$  to  $B$  need to be made. The change of edges from  $B$  to  $NB$  is characterized by the following easily seen property:

**Property 3:** Let spanning tree  $T'$  be obtained from spanning tree  $T$  by applying the exchange  $(e, f)$ . If  $x$  is a non-tree edge which is back w.r.t  $T$  and non-back w.r.t  $T'$ , then

$head(x)$  lies in the subtree of  $T$  rooted at  $tail(f)$ , and  $tail(x)$  is a vertex which is a proper ancestor of  $tail(f)$  and a proper descendant of  $lca(head(f), tail(f))$  in  $T$ .

Note that at a node  $a$  in the computation tree, the first edge in the first list in  $NB$  will have the tail with the least postorder number (with respect to the spanning tree at the root) among all edges in  $NB$ . The following claim will be important. It follows from the following 2 facts. First, the spanning tree at  $a$  differs from the spanning tree at the root in only that subtrees rooted at tails of edges used for exchanges at proper ancestors of  $a$  in the computation tree have been moved around. Second, none of the edges in  $NB$  at node  $a$  are incident on vertices in the above subtrees.

**Claim 1:** At node  $a$  of the computation tree, the first edge in the first list in  $NB$  will have the tail with the least postorder number (with respect to the spanning tree at  $a$ ) among all the edges in  $NB$ .

We now give the algorithm. The algorithm *ALGO Main* computes a DFS tree  $T$  of the graph along with the sets  $NB$  and  $B$ . It then calls algorithm *Gen* to generate the entire computation tree in a depth-first manner. Dropping subscripts, we use global sets  $IN$  and  $OUT$ . Their values at a node  $a$  of the computation tree will equal  $IN_a$  and  $OUT_a$ , respectively.

**ALGO Main**( $G, r$ )

```

 $T \leftarrow$  DFS tree of  $G$  rooted at vertex  $r$ ;
Initialize all data structures;
Compute  $NB$  and  $B$  w.r.t  $T$ ;
/*  $NB$  ( $B$ ) contains all edges that are non-back (back) w.r.t  $T$  */
 $IN \leftarrow \phi$ ;
 $OUT \leftarrow \phi$ ;
 $CHANGES \leftarrow \phi$ ;
If  $NB \neq \phi$  then  $Gen(T)$ ;

```

**End ALGO Main;**

As in ([2]),  $CHANGES$  is used to store the differences from the last spanning tree generated.

**ALGO Gen**( $T$ )

```

/* The data structures  $B, NB$  and  $CHANGES$  are global to this
procedure,  $FL$  is local*/
While  $NB \neq \phi$  do
   $FL \leftarrow$  First list in  $NB$ ;
   $f \leftarrow$  First edge in  $FL$ ;
  Remove  $FL$  from  $NB$ ;
   $e \leftarrow$  Unique exchange edge for  $f$  in  $T$ ;
   $T' \leftarrow T \cup \{f\} - \{e\}$ ;
   $IN \leftarrow IN \cup \{f\}$ ;
   $OUT \leftarrow OUT \cup \{e\} \cup \{FL\} - \{f\}$ ;
   $CHANGES \leftarrow CHANGES \cup \{(e, f)\}$ ;

```

```

Output CHANGES;
    /*This outputs the differences from the last tree generated*/
CHANGES  $\leftarrow \phi$ ;
    Compute-back-to-non-back(f, T);
        /*Computes edges in B which become non-back w.r.t T'
        by virtue of the exchange (e, f), removes them from B,
        inserts them into NB, stores the changes made to NB
        and B on STACK.*/
    If NB  $\neq \phi$  then Gen(T');
    Undo changes to NB and B stored in STACK by the above
        call to Compute-back-to-non-back;
    If {FL}  $\neq \phi$  then NB  $\leftarrow \{FL\} - \{f\} \cup NB$ ;
        /* FL with edge f removed is restored as the first list of NB*/
    IN  $\leftarrow IN - \{f\}$ ;
    OUT  $\leftarrow OUT - \{e\} - \{\text{edges in } FL\} \cup \{f\}$ ;
    CHANGES  $\leftarrow CHANGES \cup \{(f, e)\}$ ;

End While;
End ALGO Gen;

```

Before we describe the procedure *Compute-back-to-non-back* which updates the sets *NB* and *B* when an exchange (*e*, *f*) is made on *T*, we describe the data structures required to facilitate these updates. The data structure for *NB* was described earlier. We now describe the data structure for *B*. We maintain at each node *v* of *G*, a list *B*[*v*] of non-tree back edges in *B* having head vertex *v*. This list is ordered by decreasing depth of the tail vertices (i.e., in order of increasing distance from *v*) of these edges in *T*. Each list is indexed by an array *A*[*v*] such that the  $p^{\text{th}}$  element of the array points to the first edge in its list, if any, which is incident upon a proper ancestor of node *p*; if no such edge exists then *A*[*v*][*p*] is undefined. Each array has a base vertex *BASE*[*v*] associated with it. Any edge in the list *B*[*v*] which is incident upon a descendant of *BASE*[*v*] is redundant for future computations in the current computation subtree. In the initialization step, *ALGO Main* constructs the data structures *B* and *A* and sets *BASE*[*v*] to *v*, for all vertices *v*.

The procedure for updating *B* is as follows: Let *f* be the exchange edge (*u*, *v*). Let *a* be the *lca* of *u* and *v* in *T*. To compute all edges which have their heads in a subtree rooted at node *v* and tails on the path from *a* to *v* (*a* and *v* excluded) in *T*, the vertices in the subtree rooted at *v* are scanned and all the back edges satisfying the above property are removed from the back edge lists. This is done by accessing the array *A*[*w*] for each vertex *w* in the subtree rooted at *v* which satisfies the property that *BASE*[*w*] is a descendant of *a* in *T*. For each such vertex *w*, let  $j_w$  be one of the two vertices, *BASE*[*w*] or *v*, whichever is closer to the root. Then *B*[*w*] is traversed starting from *A*[*w*][ $j_w$ ] till a back edge whose tail is an ancestor of *a* is found. All but the last of the edges traversed above need to be removed from *B* and transferred to *NB*. The removal of these edges is simply effected by setting *BASE*[*w*] to *a* if *BASE*[*w*] is not an ancestor of *a*. Note that in this process, back edges which occur before *A*[*w*][*v*] in the list *B*[*w*] are also implicitly removed from *B*. We note that these edges have both their head and tail in the subtree of *T* rooted at *v*. Since future descendant nodes of the computation tree are constructed by a postorder scan of

the spanning tree  $T$ , the subtree of  $T$  rooted at  $v$  is never used again for exchanges of tree edges. These edges are thus redundant and constitute the set  $RB$  at the current node of the computation tree. We further note that the edges in  $B$  are not removed explicitly but only by updating the base pointer. This helps in restoring the back edges at the end of the recursive step in the algorithm. Restoration is now done by simply resetting the base vertex to its previous value which is stored in  $STACK$  before recursion. To update  $NB$  whenever an exchange is made, each edge removed from  $B$  and having its tail on the path from  $v$  to  $a$  (both endpoints excluded) is added to  $NB$  in  $O(V)$  time per edge; this insertion is performed by searching  $NB$  in the obvious way for the tail vertex of this edge. The consequent changes made to  $NB$  are also stored in  $STACK$ .

We now give the procedure *Compute-back-to-non-back*.

```

ALGO Compute-back-to-non-back( $f, T$ );
/* Let  $f$  be the edge  $(u, v)$  */
 $a \leftarrow lca(u, v)$  in  $T$ ;
For every vertex  $w$  in the subtree of  $T$  rooted at  $v$  do
  If  $BASE[w]$  is a descendant of  $a$  in  $T$  then
     $j \leftarrow$  higher numbered of the two vertices  $v, BASE[w]$  in  $T$ 
    in the postorder scheme;
    Let  $e$  be the edge in list  $B[w]$  that is pointed to by  $A[w][j]$ ;
    While  $e$  is defined and  $tail(e)$  is not ancestor of  $a$  do
      Insert  $e$  into  $NB$ ;
       $e \leftarrow$  Next edge in  $B[w]$ ;
    End While;
     $BASE[w] \leftarrow$  One of  $a, BASE[w]$ , whichever has the higher
    postorder number;
  /* This deletes the new non-back edges from  $B$  */
  End If;
  Store changes made above to  $B$  and  $NB$  on  $STACK$ ;
End For;
End Compute-back-to-non-back;

```

**A Remark on the Data Structures.** Note that for each vertex  $v$ , the array  $A[v]$  and the list  $B[v]$  does not change explicitly over the course of the algorithm, only the variable  $BASE[v]$  does. It can also be seen that  $BASE[v]$  is always an ancestor of  $v$  in the spanning tree at every node  $x$  in the computation tree. Lemma 4.2 will show that just changing  $BASE[v]$  suffices in maintaining  $B$  correctly.

## 4.1 Correctness

We need the following property to prove correctness.

**Property 4:** If  $f = (u, v)$  is an edge in  $NB$  and  $b$  is the  $lca$  of  $u$  and  $v$  in  $SD_a$  then no edge in  $NB$  has its head in the subtree of  $SD_a$  rooted at  $v$  and its tail on the path from  $b$

to  $u$ . ( $b$  excluded)

**Proof.** This property is essentially true because the depth first nature of the tree is maintained at every node of  $CD(G, r)$  due to the order of selection of the exchange edge from  $NB$ . Define an edge to be *eligible* at node  $x$  of the computation tree if it is not in  $SD_x$  or  $OUT_x$  and is non-back w.r.t  $SD_x$ . It suffices to show that at any node  $x$  of the computation tree, all eligible edges must connect a vertex with higher postorder number (w.r.t  $SD_x$ ) to a vertex with lower postorder number.

We show this by induction on the level of  $x$ . This is true for the root node of the computation tree because the spanning tree at that node is the DFS tree of  $G$ . Assume that this is true for any node  $x$  of the computation tree.

First consider the left son  $b1$  of  $x$ . Let  $f = (u, v)$  be the exchange edge at  $x$  and  $e$  be the edge in  $SD_x$  incident upon  $v$ . By the choice of  $f$  and Claim 1,  $tail(f)$  has the smallest postorder number (with respect to  $SD_x$ ) among the tails of all eligible edges at  $x$ . Hence, none of these edges can be incident upon the subtree of  $SD_x$  rooted at  $v$ . After the exchange, the DFS tree changes since the subtree rooted at  $v$  in  $SD_x$  moves its root. This affects the DFS numbering but leaves intact the relative ordering of all non-back edges since none of these edges are incident on vertices in the subtree of  $SD_x$  rooted at  $v$ . Thus in  $SD_{b1} = SD_x - \{e\} \cup \{f\}$ , all edges that are eligible at both  $b1$  and  $x$  satisfy the property that the postorder number of their tail is lower than the postorder number of their head w.r.t  $SD_{b1}$ . Further, the only edges that are eligible at  $b1$  but not at  $x$  are those whose head lies in the subtree of  $SD_x$  rooted at  $v$  and whose tails lie on the path  $P$  from  $v$  to  $lca(u, v)$  (Property 3). By the induction hypothesis,  $tail(f)$  has lower postorder number than  $head(f)$  in  $SD_x$ . After  $e$  is exchanged for  $f$ , the postorder numbers of all vertices in the subtree of  $SD_{b1}$  rooted at  $v$  remains greater than the postorder number of the vertices in  $P$ .

For the right son  $b2$  of  $x$ , this is trivially true. This proves the property.  $\square$

In order to prove correctness of our algorithm, we first prove that the set of crucial edges, i.e, the set of non-back non-tree edges  $NB$  is maintained correctly at each node in the computation. This is shown by the following lemma:

**Lemma 4.1** *During the construction of the computation tree the following changes to  $NB$  and  $B$  suffice after an exchange at a node  $a$  in the computation tree:-*

- (a) *changes from  $B$  to  $NB$  by property 3.*
- (b) *deletions from  $NB$  according to  $IN$  and  $OUT$  definitions.*
- (c) *removal of  $RB_a$  from  $B$ .*

**Proof:** Property 4 implies that no edge needs to be transferred from  $NB$  to  $B$  following an exchange. To complete the proof of the lemma, observe that all possible changes to be made to  $B$  and  $NB$ , except the ones ruled out by property 4, are made in (a) (b) and (c).  $\square$

The following property follows from the above lemma and the fact that at each node, the edge in  $NB$  with smallest postorder number of its tail vertex is chosen for swapping in. It justifies the removal of  $RB_x$  from  $B$  at any node  $x$  of the computation tree.

**Property 5:** If an exchange  $(e, f)$ ,  $f = (u, v)$  is made at a node  $x$  of the computation tree then the subtree of  $SD_x$  rooted at  $v$  is preserved as such in each of the trees generated

at descendant nodes of  $x$  in the computation tree. So at node  $x$ , any edge in  $B$  incident upon a vertex in that subtree, is redundant for future computations at descendant nodes of  $x$  and may be removed from  $B$ .

**Lemma 4.2** *Given  $NB$  and  $B$  for any node  $a$  in the computation tree,  $ALGO\ Gen$  correctly computes the sets  $NB$  and  $B$  for the 2 sons,  $b1$  and  $b2$ , of  $a$ .*

**Proof:** We consider the sons  $b1$  and  $b2$  separately. For  $b1$  it follows from Lemma 4.1 that the following changes need to be made to  $NB$  and  $B$

- (a) Changes from  $NB$  to  $B$  by property 3.
- (b) Removal of edges from  $NB$  according to  $IN$  and  $OUT$  definitions.
- (c) Removal of  $RB_{b1}$  from  $B$ .

Let  $SD_{b1} = SD_a - \{e\} \cup \{f\}$ . Let  $f = (u, v)$ . We note that  $f$  has been chosen as the first edge in  $NB$ , i.e. the first edge in the postordering of the edges by their tail vertex.

*Compute-back-to-non-back* uses property 3 to identify and remove from  $B$  the edges which are either converted from  $B$  to  $NB$  or are redundant for further computations. This is done correctly by the scan of the back edge list at each of the vertices in the subtree rooted at  $v$ . For the head vertex  $s$  of every edge which goes from  $B$  to  $NB$ , the value of  $BASE[s]$  is set to the  $lca$  of  $u$  and  $v$ , if it is an ancestor of the current value of  $BASE[s]$ . This implicitly deletes those edges from the data structure  $B[s]$  which either go into  $NB$  or into  $RB_{b1}$ . This is justified by properties 4 and 5 and Lemma 4.1 which ensure that no back edges lead from the subtree rooted at  $v$  to a vertex on the path from  $u$  to the  $lca$  of  $u$  and  $v$ .

By the same justification, also, the array  $A[w]$  is correctly maintained for index values  $\geq BASE[w]$  for all  $w$ . Note that only vertices which are ancestors of  $BASE[w]$  are considered when scanning back edges and  $BASE[w]$  only changes for vertices in the subtree rooted at  $v$ . Furthermore, a back edge incident onto a vertex which is an ancestor of  $BASE[w]$  remains a back edge. And, no new back edges are introduced. Thus no changes are required to  $A[w]$ .

Also note that edges which switch from  $B$  to  $NB$  at node  $b1$  are inserted into their correct postorder positions in  $NB$ . This is because the exchange  $(e, f)$  does not change the relative postorder number of vertices outside the subtree of  $SD_{b1}$  rooted at  $v$ , and the edges inserted are not incident on a vertex in that subtree.

*Gen* also implements the other changes required, i.e., the removal of those edges from  $NB$  which go into the  $OUT$  set of  $b1$ .

For node  $b2$ , the only changes are deletions according to the  $IN$  and  $OUT$  definitions which are correctly implemented.  $\square$

We can now state that *ALGO Gen* correctly computes all spanning trees without repetitions.

**Theorem 4.3** *ALGO Main generates the entire computation tree  $CD(G, r)$  correctly.*

**Proof:** Follows from Lemma 4.2.  $\square$

## 4.2 Complexity

For the purpose of this section we define a compressed version  $CD'(G, r)$  of  $CD(G, r)$ . This is done to take care of the fact that the son  $b2$  does not explicitly generate a tree but is used as a node which eliminates a non-back edge. In fact, starting at a node  $x$  of  $CD(G, r)$ , the entire rightmost path has nodes of this nature. We can thus compress this path so that the left sons of the nodes along this path are now the sons of node  $x$ . We thus obtain at each node  $x$ , sons corresponding to all trees which can be obtained by one exchange from  $SD_x$ , maintaining the  $IN$  and  $OUT$  set restrictions as required by the path.

**Lemma 4.4** *ALGO Main outputs the changes corresponding to the compressed computation tree  $CD'(G, r)$  in  $O(N(r)V + V^2)$  operations.*

**Proof:** At each node  $x$  of the computation tree, the major work that *ALGO Gen* performs is the conversion of back edges to non-back and removal of back edges. We let  $NBC_x$  be the number of edges that are converted from back to non-back at node  $x$ . The time for manipulating the data structures  $NB$ ,  $B$  and  $STACK$  (which includes the time for undoing changes after recursion) at node  $x$  is  $O(V * (NBC_x + 1))$ . This is because the procedure *Compute-back-to-non-back* takes just constant time for each vertex in the subtree rooted at  $v$  (for  $f = (u, v)$ ), apart from the time to determine edges that are converted from back to non-back. The time for determining these edges is proportional to their number. The time for inserting each of these into  $NB$  is  $O(V)$ .

All other operations in *ALGO Gen*, except the output operation, require  $O(V)$  time. So the total time required by *ALGO Gen* minus the output operations equals  $O(\sum(V * NBC_x))$ , the summation being over all nodes of the computation tree. Note that at any node  $x$  in the compressed computation tree, an edge which is converted from back to non-back is used as an exchange edge at some descendant of  $x$  and hence leads to a new spanning tree. Therefore, the the above summation gives an  $O(N(r)V)$  time bound.

Total output operations are  $O(N(r))$  since only exchanges are output and the number of exchanges output is proportional to the size of the computation tree. Computing  $B$  and  $NB$  initially require  $O(V^2)$  time. Computing the DFS tree at the root of the computation tree requires  $O(V+E)$  time. The time bound follows.  $\square$

**Lemma 4.5** *ALGO Main requires  $O(V^2)$  space.*

**Proof:** Follows from the size of the data structures involved.  $B$  requires  $O(V^2)$  space (in particular, the arrays  $A$  require this much space).  $NB$  requires  $O(V+E)$  space.  $CHANGES$  requires  $O(V)$  space at most since any spanning tree may be obtained by any other spanning tree by at most  $V - 1$  exchanges.  $NB, B$  and  $CHANGES$  are global to procedure *Gen*, hence are not stacked on recursion. At every node  $x$  of the computation tree, the size of the changes stored on  $STACK$  is  $O(V + NBC_x)$  space. We note that the changes at a recursive step are undone and this eliminates need of copying data structures at the expense of a constant factor increase in number of operations. Since the computation tree  $CD(G, r)$  is generated in a depth first manner and changes are added to  $STACK$

only at the left branches, which are  $O(V)$  in number along any path of the computation tree, the total stack space required is  $O(V^2 + \sum NBC_x)$  where the summation is taken over all nodes on the path to a leaf node. From Lemma 4.1 it follows that  $\sum NBC_x$  is  $O(E)$  since no edge moves back from  $NB$  to  $B$  along the path. The theorem thus follows.  $\square$

From the above lemmas the following result follows if the above procedure is repeated with each vertex in turn as root.

**Theorem 4.6** *All rooted directed spanning trees can be output in  $O(NV + V^3)$  operations and  $O(V^2)$  space.*

## 5 Conclusion

An improved algorithm has been obtained for enumerating spanning trees in directed graphs. This algorithm takes  $O(NV + V^3)$  time and  $O(V^2)$  space for generating the computation tree and outputting relative changes between spanning trees of a directed graph. This betters the previous best time of  $O(NE)$  achieved by Gabow and Myers' algorithm ([1]). Explicit enumeration, if desired, can be done by traversing the computation tree in an optimal  $O(NV)$  time. Our algorithm is not optimal for constructing the computation tree itself because no lower bound other than the trivial  $O(N)$  bound is known. The existence of an  $O(N)$  algorithm for the same remains an open question. An improved algorithm was presented in a paper by the two authors along with another co-author in the Workshop on Algorithms and Data Structures, 1995 ([5]). However, this algorithm seems to have a bug which has not been removed as of now.

## References

- ([1]) H. N. Gabow and E. W. Myers: Finding all spanning trees of directed and undirected graphs, *SIAM J. Comp.*, Vol 7, No. 3, Aug 1978.
- ([2]) S. Kapoor and H. Ramesh: Algorithms for generating all spanning trees of undirected and weighted graphs, *SIAM J. Comp.*, Vol 24, No. 2, 1995.
- ([3]) W. Mayeda: *Graph Theory*, John Wiley, NY 1972.
- ([4]) S. Shinoda: Finding all possible directed trees of a directed graph, *Electron Commun.*, Japan, 51-A, 1968, pp. 45-47.
- ([5]) S. Kapoor, V. Kumar, H. Ramesh: An algorithm for generating all spanning trees of directed graphs, *Proceedings of Workshop on Algorithms and Data Structures*, 1995.