Algorithms for Enumerating All Spanning Trees of Undirected and Weighted Graphs *

Sanjiv Kapoor *H. Ramesh* †
Department of Computer Science & Engg.
Indian Institute of Technology
Hauz Khas, New Delhi 110016
India.

Abstract

In this paper, we present algorithms for enumeration of spanning trees in undirected graphs, with and without weights.

The algorithms use a search tree technique to construct a computation tree. The computation tree can be used to output all spanning trees by outputting only relative changes between spanning trees rather than the entire spanning trees themselves. Both the construction of the computation tree and the listing of the trees is shown to require $O(N+V+E)$ operations for the case of undirected graphs without weights. The basic algorithm is based on swapping edges in a fundamental cycle. For the case of weighted graphs (undirected), we show that the nodes of the computation tree of spanning trees can be sorted in increasing order of weight, in $O(N \log V + VE)$ time. The spanning trees themselves can be listed in $O(NV)$ time. Here N, V, and E refer respectively to the number of spanning trees, vertices, and edges of the graph.

Key words. spanning tree, undirected graph, weighted graph, enumeration.

1 Introduction

Spanning tree enumeration in undirected graphs is an important issue in many problems encountered in network and circuit analysis. Applications are given in [Ma72]. Weighted spanning tree enumeration in order would find application in a subroutine of a generate-and-test procedure for connecting together a set of points with the minimum amount of wire, where the connection satisfies some additional constraint, e.g., a minimum distance to be maintained between 2 wires.

*An extended abstract of this work appeared in the proceedings of WADS 91, Ottawa, LNCS 519

†Presently at Courant Institute of Mathematical Sciences, New York University, New York, N.Y. 10012

Spanning tree enumeration has a long history (See references). Previous techniques employed for solving the problem include depth first search [GM78, TR75], selective generation and testing [Ch68], and edge exchanging [Ga77]. Of these, Gabow and Myers' algorithm [GM78] seems to be the fastest with a time complexity of $O(NV)$ on a graph with V vertices, E edges and N spanning trees. Their algorithm requires $O(NV)$ time for generating the trees themselves and not merely for outputting them. Their algorithm is optimal up to a constant factor if all spanning trees of the graph need to be explicitly output. For many practical applications, the spanning trees need not be explicitly output and only a computation tree which gives relative changes between spanning trees is required. We note that from this computation tree, the spanning trees can be listed out explicitly in $O(NV)$ operations, if required.

In this paper, we enumerate spanning trees by listing out differences between them. Each node of the computation tree that describes this procedure represents a spanning tree of the graph. The spanning trees represented by a node and its parent in the computation tree differ in exactly one pair of edges, i.e., the spanning tree at any node is obtained by exchanging an edge in the spanning tree at its parent for an edge not present in that spanning tree. This exchange is obtained from the fundamental cycles of the graph. An edge external to a spanning tree can be exchanged with any edge in its fundamental cycle to give a spanning tree which differs from the original spanning tree in exactly one pair of edges. By repeating this for all external edges, all spanning trees which differ from the original spanning tree in one pair of edges can be obtained. The computation tree is generated by repeatedly applying this procedure. Repetition of the same spanning tree is avoided by following a search tree inclusion-exclusion strategy. The algorithm presented here outputs, for each node of the computation tree, the difference between the spanning trees associated with that node and its predecessor in a preorder scan of the computation tree. This is done by traversing the computation tree in a depth first manner. We describe two algorithms each requiring $O(N + V + E)$ time. The first requires $O(V^2E)$ space and the second requires $O(VE)$ space. The first algorithm has a more general methodology and may be more useful in certain applications. It is used in the weighted case in this paper.

An $O(N \log V + VE)$ algorithm for sorting the nodes of the computation tree in increasing order of weight is also presented here and is based on the fact that there are a bounded number of exchanges that change one spanning tree into another. To output the spanning trees in sorted order however requires $O(NV)$ operations. The scheme presented betters the $O(N \log N)$ time heapsort used by Gabow [Ga77] which results in a total time complexity of $O(NE + N \log N)$.

In a companion paper [KR92], we use similar techniques to enumerate all spanning trees of a directed graph in $O(NV)$ time, improving upon the previous best known bound of $O(NE)$ time [GM78].

Section 2 describes the generation of spanning trees in undirected, unweighted graphs and Section 3 describes a way of ordering the spanning trees in the computation tree for weighted graphs. Each of the sections contains a description of the algorithms and proofs of their correctness and complexity.

2 Undirected Spanning Tree Enumeration

Let G be an undirected graph with V vertices, E edges and having N spanning trees. $E(G)$ refers to the set of edges of the graph G.

2.1 Algorithm Outline

In this section, an outline of the algorithm for generating all spanning trees of an undirected graph is presented.

The algorithm starts off with a spanning tree T, and generates all other spanning trees from T by replacing edges in T by edges outside T. For undirected graphs, an edge in a fundamental cycle of the graph can be replaced by its corresponding non-tree edge to result in a new spanning tree. Thus, a number of spanning trees can be generated from a single spanning tree by exchanging edges in a fundamental cycle with the corresponding non-tree edge. This computation can be represented by a computation tree with spanning tree T at its root and the spanning trees resulting from these exchanges at its sons. To generate other spanning trees, these sons are expanded recursively in the same manner as the root. Thus each node in the computation tree is associated with a spanning tree of G.

We need to ensure that each spanning tree is generated exactly once. This is done by a search tree type computation tree which uses the inclusion/exclusion principle. To aid the construction of the computation tree, at every node in the computation tree 2 sets with the following classification are maintained. For a node x in the computation tree, the set IN_x consists of edges which are always included in all spanning trees at x and its descendants in the computation tree. The set OUT_x contains edges which are not included in any spanning tree at x or at its descendants in the computation tree. We let S_x denote the spanning tree generated at x and G_x denote the current graph obtained by contracting edges in IN_x and removing edges in OUT_x. Note that G_x may be a multigraph. We define $CYCLE_x$ to be the set of fundamental cycles of non-tree edges (with respect to S_x) which are in G_x. We now formally define the computation tree $C(G)$ with respect to the graph G. $C(G)$ has a spanning tree of G associated with every node. The computation tree starts off with an arbitrary spanning tree at the root. Let A be a node in the computation tree and let S_A be the spanning tree associated with A. Let f be an edge not in OUT_A or S_A and let $c_f = (e_1, e_2, \ldots, e_k)$ be the fundamental cycle in G_A formed by f with respect to S_A. Then A has as its sons B_i, $1 \leq i \leq k+1$ (see Fig. 1; each edge is labeled with the pair of edges exchanged). For $1 \leq i \leq k$, B_i corresponds to the spanning tree obtained by the exchange (e_i, f). Note that e_i is not already in IN_A because edges in IN_A are contracted in G_A. And B_{k+1} corresponds to a node in the computation tree such that no descendant of the node has f in the spanning trees generated. Note that the tree at B_{k+1} is the same as the one at its parent.

The IN and OUT sets are formally defined as follows.

$$
\begin{aligned}
IN_{B_j} &= IN_A \cup \{e_1, e_2, \ldots, e_{j-1}\} \cup \{f\}, \text{ for } 1 \leq j \leq k \\
OUT_{B_j} &= OUT_A \cup \{e_j\}, \text{ for } 1 \leq j \leq k
\end{aligned}
$$

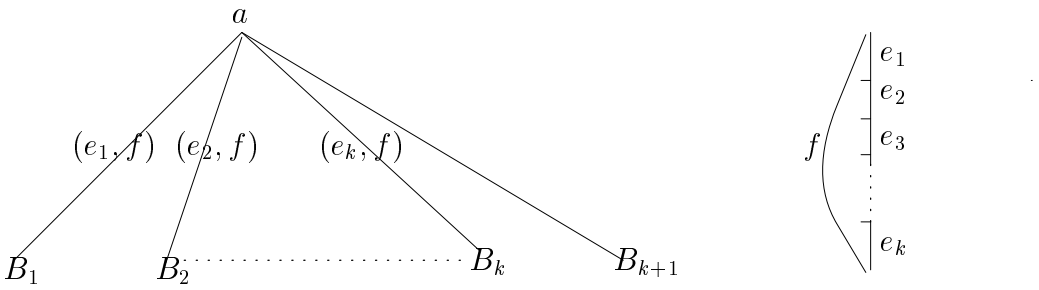

Figure 1: $C(G)$

$IN_{B_{k+1}} = IN_A$
$OUT_{B_{k+1}} = OUT_A \cup \{f\}$
$S_{B_j} = S_A - \{e_j\} \cup \{f\}$, for $1 \leq j \leq k$
$E(G_{B_j}) = E(G_A) - \{e_j\}$ *with edges* $e_1, e_2, \ldots, e_{j-1}, f$ *contracted*, for $1 \leq j \leq k$
$CYCLE_{B_j} = CYCLE_A$ **com** c_f *with edges* $e_1, e_2, \ldots, e_{j-1}$ *contracted in each resultant cycle* , for $1 \leq j \leq k$
{ Here the **com** operation combines (see Fig. 3) all cycles in $CYCLE_A$ which contain edge e_j with c_f and leaves the other cycles intact}
$S_{B_{k+1}} = S_A$
$E(G_{B_{k+1}}) = E(G_A) - \{f\}$
$CYCLE_{B_{k+1}} = CYCLE_A - \{c_f\}$

The IN and OUT sets for the root x of the computation tree are both empty. S_x is any spanning tree, G_x is the original graph G and $CYCLE_x$ is the set of fundamental cycles in G with respect to S. Before we show how to generate the computation tree, we show that $C(G)$ suffices to generate all the spanning trees of G.

Lemma 2.1 *The computation tree has at its internal nodes and leaves all the spanning trees of* G.

Proof: The proof follows from induction and the inclusion/exclusion principle. The inclusion/exclusion is implemented in the computation tree as follows: Let A be the root node of the computation tree. The subtrees rooted at B_1 through B_k together form the computation tree of the spanning trees which have the edge f in them. B_{k+1} is the root of the computation tree which computes all spanning trees not containing the edge f. Within the set of spanning trees which contain f, the subtree rooted at B_1 generates spanning trees without e_1 whereas $B_2, \ldots B_k$ generate subtrees with e_1. A similar inclusion-exclusion process is repeated at each of the nodes $B_2, \ldots B_k$, i.e., the computation subtree rooted at B_j corresponds to the set of spanning trees which contain the edges $e_1, \ldots, e_{j-1}$ but not e_j. Moreover, the computation subtree corresponding to the inclusion

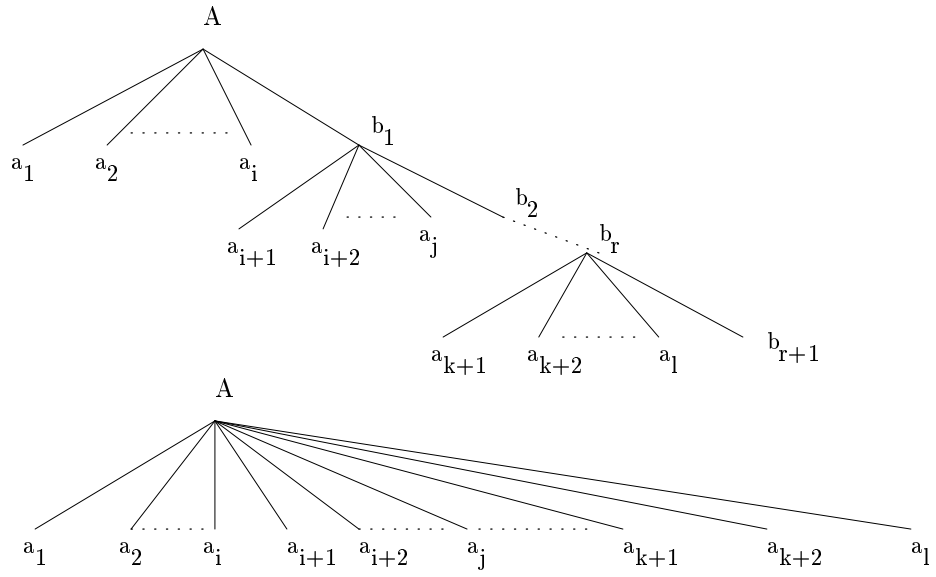

Figure 2: $C'(G)$

of $e_1, \ldots, e_k$ and f need not be included in the computation tree since this choice of edges forms a cycle and will not lead to any spanning tree. □

In order to analyze the algorithms, it is convenient to define a compressed form $C'(G)$ of $C(G)$. Having generated node A with spanning tree S_A, we find the fundamental cycle corresponding to some non-tree edge $f \in G_A$ and then generate the sons $B_1, \ldots, B_{k+1}$ according to the above description. However note that the sons of B_{k+1} will be obtained by using another non-tree edge relative to S_A in a graph where f is absent. And this is repeated along the entire rightmost branch of the computation tree. Since all these computations are with respect to S_A alone, we can obtain a compressed version of the computation tree called $C'(G)$ by considering all the fundamental cycles at node A and applying the inclusion/exclusion principle over the non-tree edges. Figure 2 illustrates the compression. The compressed computation tree $C'(G)$ has the advantage that each node in the tree corresponds to a unique spanning tree. Since the compression does not eliminate nodes with distinct spanning trees in $C(G)$, $C'(G)$ generates all spanning trees of G. Each node of $C(G)$ is associated with exactly one node of $C'(G)$ and we refer to both nodes by the same name.

To achieve the construction of $C(G)$ in linear time, we outline schemes for finding the fundamental cycles and generating the sons of a node. An important issue in the generating $C(G)$ is the computation of the set of fundamental cycles at each node of $C(G)$. Note that the current tree T has been obtained by replacing an edge e in the previous tree T' by a non-tree edge f. This affects all the fundamental cycles containing the edge e. Each fundamental cycle containing edge e must now be *combined* with the

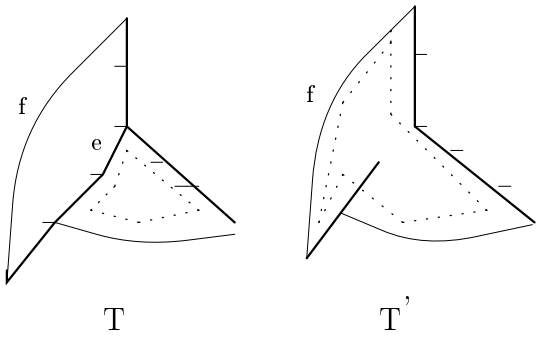

Figure 3: Cycle Combination

fundamental cycle of f, i.e., a new cycle is obtained by removing edges common to both cycles (see Fig 3.).

In Algorithm 1, at each node A in $C(G)$, an arbitrary non-tree edge f in G_A is chosen for exchanging. After each exchange is performed, each of the fundamental cycles affected by that exchange is combined with the fundamental cycle of f. This is done by scanning the affected cycles and changing them in time proportional to the sum of the lengths of the *resulting* cycles. Each fundamental cycle generates a number of spanning trees equal to the number of tree edges in it by exchanging with the corresponding non-tree edge. This ensures that the computation tree is generated in linear time. This approach requires $O(V^2E)$ space and the data structure for maintaining the fundamental cycles is slightly elaborate. Further, this involves repeated scanning of an edge which occurs in more than one cycle.

To reduce space and to simplify and speed up the data structures, we describe Algorithm 2. In this algorithm, the tree S_A is always a depth-first-search tree (d.f.s tree) of G_A at each node A of $C(G)$. This is ensured in the following manner. We start at the root of $C(G)$ with the d.f.s tree of G. Further, at each node A of $C(G)$, the non-tree edges in G_A are considered for exchanges in a particular order. The order is given by the increasing post-order number of the upper (i.e., closer to the root) end points of the non-tree edges. Note that since S_A is a d.f.s tree of G_A, all the non-tree edges are back edges with respect to S_A. Clearly, finding the fundamental cycle of a non-tree edge f is in now straightforward; it simply involves marching up S_A from the lower end point of f to its upper end point. As we shall describe later, combining fundamental cycles now becomes a matter of simply changing the endpoints of some edges. The total space used in this scheme is $O(VE)$.

We remark that while Algorithm 2 is more efficient, Algorithm 1 is more general. In particular, it can be used as the base scheme in generating the spanning trees of a weighted graph in order (Section 4) while Algorithm 2 fails in that application because of its depth-first restriction.

2.2 Algorithm 1 Description

The main algorithm *Main* has as input a graph G. It generates a spanning tree T of G corresponding to the root of the computation tree and computes the fundamental cycles with respect to T. *Main* also initializes the data structures which will be described shortly.

ALGO Main(G);
    Find a spanning tree, T, of G;
    Initialize data structures;
    $Gen(T)$;
End Main;

The heart of the algorithm is the generation scheme *Gen* which generates the sons of a node in the computation tree and recursively generates the subtrees rooted at them. The entire computation tree is thus generated in an pre-order traversal of the tree. The following is an outline of the scheme.

Gen picks an edge f from F, the data structure which stores all non-tree multiedges in the current graph. It then determines the fundamental cycle $c_f = (e_1, \ldots, e_k)$ of f with respect to T, the current spanning tree. Note that $e_1, \ldots, e_k$ are only the tree edges in c_f, the actual fundamental cycle being formed by these edges along with the edge f.

Each edge in c_f in chosen for exchange with f in turn. When a tree edge e_i is chosen for exchange, it is removed from the current graph (i.e., put in the OUT set). As a result, all fundamental cycles containing this edge are now modified, i.e., they now have to be combined with c_f. Note that by this point, the edges $e_1, \ldots, e_{i-1}$ are already added to the IN set and thus are already contracted in the graph and in c_f. Further, the edge f must be contracted in the current graph as it is added to the IN set. These changes are made by procedure *Prepare–for–son*.

After the computation subtree rooted at the node corresponding to the exchange (e_i, f) is constructed recursively, the edge e_i must now be contracted in the current graph (i.e., added to the IN set). This is done in the procedure *Prepare–for–sons'–sibling–branch*.

Finally, the last branch of $C(G)$ involves removal of the edge f from the current graph; this is done in procedure *Prepare–for–final–son*. Note that before returning from *Gen*, the state of the data structures is restored to that at the time its invocation.

The output of *Gen* is derived from the variable $CHANGES$. $CHANGES$ accumulates the exchanges used to derive the current spanning tree from the previous spanning tree generated. $CHANGES$ is initialized to ϕ by *Main* and reset whenever a spanning tree is output. It is modified whenever an edge of $C(G)$ is traversed in the downward direction and this modification is reversed while backtracking upwards along this edge.

ALGO Gen(T);
    $mf \leftarrow$ A multiedge in F;
    $f \leftarrow$ An edge in mf;
    $F \leftarrow F - mf$;
    $c_f \leftarrow$ the fundamental cycle of f w.r.t T in G;

/* $c_f \leftarrow (e_1, e_2, \ldots, e_k)$ in the order they occur in the cycle ; */
For $i = 1$ to k **do**
$T' \leftarrow T + f - e_i$;
$CHANGES \leftarrow CHANGES + \{(e_i, f)\}$;
Output $CHANGES$;
/*These are the difference from the last tree generated; */
$CHANGES \leftarrow \phi$;
Prepare–for–son(e_i);
If $F \neq \phi$ **then** *Gen(T)*;
$CHANGES \leftarrow CHANGES + \{(f, e_i)\}$;
If $i < k$ **then** *Prepare–for–sons'–sibling–branch(e_i)*;
End For
Restore all changes made to the graph and the data
structures in the above *For* loop;
Restore multiedge $mf - f$ to F;
Prepare–for–final–son();
If $F \neq \phi$ **then** *Gen(T')* ;
Restore changes made to the graph and the data
structures in *Prepare–for–final–son*;
Restore edge f to multiedge mf in F;
End Gen;

We describe the subprocedures used by *Gen* next. Their description and performance is linked to the data structures used. Clearly, the following data structures suffice.

1. The list F of non-tree multiedges.

2. A data structure AG to maintain the current graph which supports the operations of contracting and deleting edges. As described later, we store only non-tree edges in AG. The tree edges are stored in the following cycle data structure.

3. A data structure C storing all fundamental cycles of the current graph, which allows for determining the fundamental cycle of a particular non-tree edge, for determining all fundamental cycles which contain a particular tree edge, and for combining all cycles containing a particular tree edge with a given cycle.

We describe the subprocedures used by *Gen* in terms of the operations performed upon the data structures mentioned above. Each operation will be described in detail later.

PROCEDURE Prepare-for-son(e_i);
If $i = 1$ **then** contract non-tree edge f in AG;
Combine all cycles in C which contain tree edge
e_i with cycle in C corresponding to edge f;
End Prepare-for-son;

PROCEDURE Prepare-for-sons'-sibling-branch(e_i);
Contract e_i in all cycles in C containing e_i;
Modify AG in order to reflect the contraction of e_i;
End Prepare-for-sons'-sibling-branch;

PROCEDURE Prepare-for-final-son;
Delete non-tree edge f from AG;
Modify C to reflect the deletion of non-tree edge f;
End Prepare-for-final-son;

Data Structures. Next, we give a high level description of these data structures and the operations performed upon them by the subprocedures. As we will show later, the time spent in each of these operations can be amortized to nodes of $C'(G)$ and $C(G)$ in a manner such that each node gets charged a constant amount.

The Graph Data Structure. The graph (which is a multi-graph, in general) is maintained as an adjacency list structure AG of multi-edges in the usual manner with just the following difference: only edges not in the current spanning tree are maintained. Note that edges constituting a multi-edge are clubbed together in this structure. The edges in the current spanning tree are maintained as part of the data structure storing the fundamental cycles.

Operations on the Graph. Contraction and deletion of edges in AG is done as follows. Deleting a non-tree edge from AG is straightforward and takes constant time, given a pointer to that edge. Next, consider the contraction of edges. Note that only tree edges are contracted in the algortihm. Contraction of a tree edge involves merging the adjacency lists of the two end points of the edge and takes time proportional to the number of multi-edges in the two adjacency lists. While performing this merger, one of the multi-edges may become a self-loop; this multi-edge is removed. This ensures that each non-tree edge in AG has a fundamental cycle with at least one tree edge at all times.

Lemma 2.2 *Data structure AG allows deletion and contraction of non-tree edges in constant time and contraction of a tree edge in time proportional to the number of multiedges incident upon the two end points of that edge.*

The Data Structure F. The list F is maintained in the obvious way as a list of lists, each list storing a non-tree multi-edge. These multi-edges are linked to the corresponding multi-edges in AG, so changes in AG can be reflected in F in the same time bounds.

The Cycle Data Structure. The data structure C for storing fundamental cycles is as follows. The tree edges in each fundamental cycle are stored in a circular doubly linked list in the order in which they appear. This list is accessible by the corresponding non-tree multi-edge. Note that all edges which constitute a multi-edge share the same fundamental cycle. For each tree edge e which is not in the IN set, to find all fundamental cycles containing e, we maintain a list of fundamental cycles containing e. Further, given a tree edge e which is not in the IN set, we need to be able to delete e from all cycles

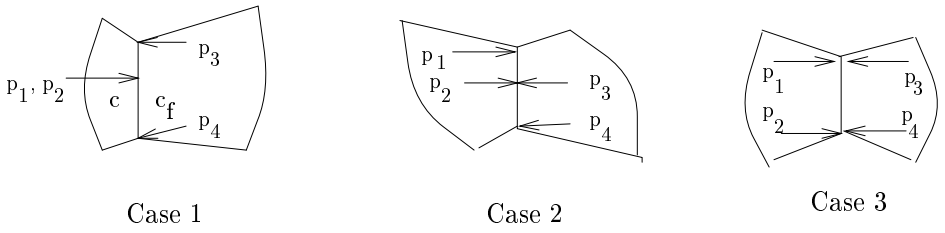

Figure 4: Three Cases in the Cycle Combination Algorithm

containing e in time proportional to the number of these cycles. Therefore, for each tree edge e, the list containing fundamental cycles also stores pointers to the location of e in each of the cycles.

Operations on the Cycles. Consider deletion of non-tree edge f first. If f is not part of a multi-edge then its fundamental cycle must be removed from C and C must then be modified appropriately. This is easily accomplished in time proportional to the size of this fundamental cycle. If f is part of a multi-edge then no changes need be made to C. Next, consider contractions of tree edges. When a tree edge e_i is contracted, the change must be reflected in each of the cycles containing e_i. This takes time proportional to the number of such cycles.

It remains to describe the operation of combining cycles. Consider the generation of son B_i of A. All fundamental cycles containing e_i in the current graph must be combined with c'_f, the fundamental cycle of f with respect to S_A with the edges $e_1, \ldots, e_{i-1}$ contracted.

The Cycle Combination Algorithm. We show how to combine cycles c and c_f in time proportional to the size of the resulting cycle. Note that the resulting cycle does not have any tree edges common to both c and c_f. Therefore, the main aim of the combining operation is to combine c and c_f while avoiding the chain of edges common to both c and c_f. This is done as follows. Let a_1 and a_2 be the end points of the non-tree edge g associated with fundamental cycle c. Let a_3 and a_4 be the end points of the non-tree edge f associated with fundamental cycle c_f. We show how to determine the end points of the chain D of tree edges, which constitutes the portion common to c and c_f in time proportional to $|c| + |c_f| - |D|$. Clearly, knowing these two end points, the resulting cycle can easily be obtained in same time bound. Thus the time required to combine two cycles is proportional to the size of the resulting cycle.

We traverse c and c_f using 4 pointers $p_1 \ldots p_4$, two per cycle, with p_i pointing to a_i initially. There are a number of rounds; in each round, each pointer traverses one edge of the cycle moving towards the other end of the cycle. Pointer p_i stops moving when either it reaches an edge which has been traversed previously (by some pointer) or it meets another pointer p_j. The latter conditions holds when either p_i and p_j point to the same vertex or they cross each other while traversing an edge. The procedure stops when all four pointers have stopped moving. The number of rounds is at most $\max\{|c|, |c_f|\} - |D|$

because as long as the above procedure continues, at least one of the four pointers must be outside D.

We show that the end points of D can be inferred from the final positions of the four pointers. Clearly, some two pointers must meet each other during the above procedure. There are three cases to consider depending upon which pointers meet (see Figure 4; pointers are shown in their final positions in each case). First, suppose that the pointers which traverse c meet each other. Then the pointers which traverse c_f must be finally located at the two endpoints of D. The case when the two pointers which traverse c_f meet can be handled similarly. Second, suppose a pointer which traverses c meets a pointer which traverses c_f but the remaining two pointers do not meet. These two remaining pointers must be finally located at the end points of D. Third, suppose each pointer which traverses c meets at least one pointer which traverses c_f. Then, it can easily be verified that each pointer which traverses c meets exactly one of the pointers traversing c_f. In this case, two of the pointers, one which traverses c and one which traverses c_f, must be finally located at one end point of D while the other two pointers must be located at the other end point.

It follows that the cycle combination can be achieved in time proportional to the number of edges in the resulting cycle.

Lemma 2.3 *The data structure C allows for*

1. *Deletion of non-tree edge f in time proportional to the size of its fundamental cycle.*
2. *Contraction of tree edge e_i in time proportional to the number of cycle it is contained in.*
3. *Combining two cycles in time proportional to the size of the resulting cycle.*

This ends the description of the data structures.

2.2.1 Analysis

We start with the analysis of the time complexity of Algorithm 1. First, we show that the total output size of the algorithm is $O(N)$.

Lemma 2.4 *The number of exchanges output by Gen is at most $2N$.*

Proof: Consider internal node x of $C(G)$. For each son y of x, except the last, one exchange is added to $CHANGES$ when y is generated and its opposite exchange ((e, f) is the exchange opposite to (f, e)) is added to $CHANGES$ after the sub-tree rooted at y has been generated. Therefore, the number of exchanges added to $CHANGES$ is at most $2N$. Further, every time $CHANGES$ is output, it is reset to ϕ immediately. The lemma follows. □

The following lemma is the key one in obtaining the final complexity.

Lemma 2.5 *The work done at each node A of $C(G)$ is $O(|s(A)| + |g(A)| + r(A))$, where $s(A)$ is the set of sons of A in $C(G)$, $g(A)$ is the set of sons in $C'(G)$ of nodes in $s(A)$, and $r(A)$ is the number of exchanges output at node A.*

Proof. The work done to output exchanges at A is at most $O(r(A))$. We consider the other operations performed at node A and show how to amortize the time spent in these operations to nodes in $s(A)$ and $g(A)$

Clearly, the operations performed in *Gen* (excluding those performed in the subprocedures) take time proportional to the number of sons of A. We look at the operations performed in the subprocedures next.

First, consider *Prepare-for-son(e_i)*. By Lemma 2.2, contracting f takes constant time. By Lemma 2.3, the time to combine all cycles containing e_i with the fundamental cycle for f takes time proportional to the sum sizes of the resulting cycles. Each resulting cycle leads to a number of sons of B_i in $C'(G)$ equal to its size. Thus the time spent in combining cycles can be charged to the sons of node B_i in $C'(G)$. Consequently, each node in $g(A)$ gets charged once for cycle combinations over all calls to *Prepare-for-son*.

Next, consider *Prepare-for-sons'-sibling-branch(e_i)*. First, consider contraction of edge e_i in C. By Lemma 2.3, contracting e_i in C takes time proportional to the number of cycles it is contained in. Consider each of these cycles following the contraction of e_i. At most one of these, c_j say, will not contain a tree edge, as edges corresponding to a multi-edge are clubbed together and self-loops are removed as they are formed. Therefore, each of these cycle except c_j and c'_f, the fundamental cycle of f with respect to S_A with the edges $e_1, \ldots, e_{i-1}, e_i$ contracted, will lead to at least one son of B_{i+1} in $C'(G)$. Next, consider the contraction of edge e_i in AG. By Lemma 2.2, this takes time proportional to the number of multiedges incident upon the two end points of e_i. Consider the set M of these multiedges, excluding the one multiedge which is converted to a self-loop upon contraction, if any. Following the contraction, each multiedge $g \in M$ with tree edges other than the one contracted has a fundamental cycle containing at least one tree edge (recall self-loops are removed as they are formed). Therefore, every edge in each such multiedge g, with the possible exception of the multiedge having the same end points as f, gives rise to at least one son of B_{i+1} in $C'(G)$. Thus, the time spent in *Prepare-for-sons'-sibling-branch(e_i)* can be charged to the sons of node B_{i+1} in $C'(G)$. Therefore, each node in $g(A)$ gets charged at most twice over all calls to *Prepare-for-sons'-sibling-branch*.

Finally, consider *Prepare-for-final-son*. By Lemma 2.2, deletion of f from AG takes constant time. By Lemma 2.3, deletion of f from C takes time proportional to the size of its fundamental cycle, which, in turn, equals the number of sons of A in $C(G)$. The lemma follows. □

Next, we obtain the time complexity of *Gen* and *Main*.

Theorem 2.6 *All spanning trees can be correctly generated in $O(N + V + E)$ time by Main.*

Proof: Firstly note that *Gen* correctly computes the spanning trees at sons of a node A of the computation tree. This follows from the fact that the cycles and the graph are

correctly updated after the inclusion of edges into the IN and OUT sets. The correctness of the updates is evident from the operations on the data structures discussed in detail before. Also note that Gen correctly maintains $CHANGES$ which stores the difference between the current tree being output and the last spanning tree generated. Thus Gen correctly generates the computation tree and outputs the tree differences.

We next compute the time complexity. The preprocessing steps in $Main$ before calling Gen is called require $O(V+E)$ time, in addition to the time required for setting up the cycle data structure. The latter can be charged to the sons of the root node of $C'(G)$. We show next that the time taken by the call to Gen in $Main$ is $O(N)$.

The total time for outputting exchanges over all invocations of Gen is $O(N)$ by Lemma 2.4. Next, consider the time spent in a particular invocation of Gen, minus the time for outputting exchanges in that invocation. Let this invocation correspond to the creation of the sons of node x in $C(G)$. By Lemma 2.5, the time taken by this invocation of Gen is at most $O(|s(x)| + |g(x)|)$. Summing over all nodes of $C(G)$, it follows that the time taken by the call to Gen in $Main$ is $O(N)$ The lemma follows. □

Next, we now analyze the space complexity.

Theorem 2.7 *The space required by the spanning tree enumeration algorithm, Main, is* $O(V^2E)$.

Proof: At each node of $C(G)$, changes to the data structures need to be stored to enable restoration later. These changes take $O(VE)$ space and this dominates the space requirement. The space taken to store changes when the last son of any node in $C(G)$ is generated is $O(1)$ (only a non-tree edge is deleted). On any root to leaf path in $C(G)$, the number of nodes which are not the rightmost sons of their respective parents (note that this number is bounded by the height of $C'(G)$) is at most V. This is because any two spanning trees can differ in at most $V-1$ pairs of edges. The theorem now follows from the fact that Gen generates $C(G)$ in a depth first manner. □

2.3 Algorithm 2 Description

We describe a second algorithm based on the use of depth first search to construct $C(G)$.

As before we generate the computation tree, $C(G)$ recursively: The details are as follows. At the root of the computation tree the spanning tree is constructed by a depth first scan of the graph. In fact we maintain the following invariant: at each node A of the computation tree, S_A is a d.f.s spanning tree of G_A. Consequently, all non-tree edges in G_A are back edges with respect to S_A. This property makes the task of determining and combining fundamental cycles much easier. In particular, it is no longer necessary to maintain all the fundamental cycles in a separate data structure. Only the spanning tree S_A itself needs to be stored, with the edges in IN_A contracted. Given S_A, the fundamental cycle corresponding to a particular non-tree edge $f \in G_A$ can now be found by marching along S_A from the end point farther from the root to the end point closer to the root. This is instrumental in reducing the space bound to $O(VE)$. The combination of cycles is also simplified, as we will describe below.

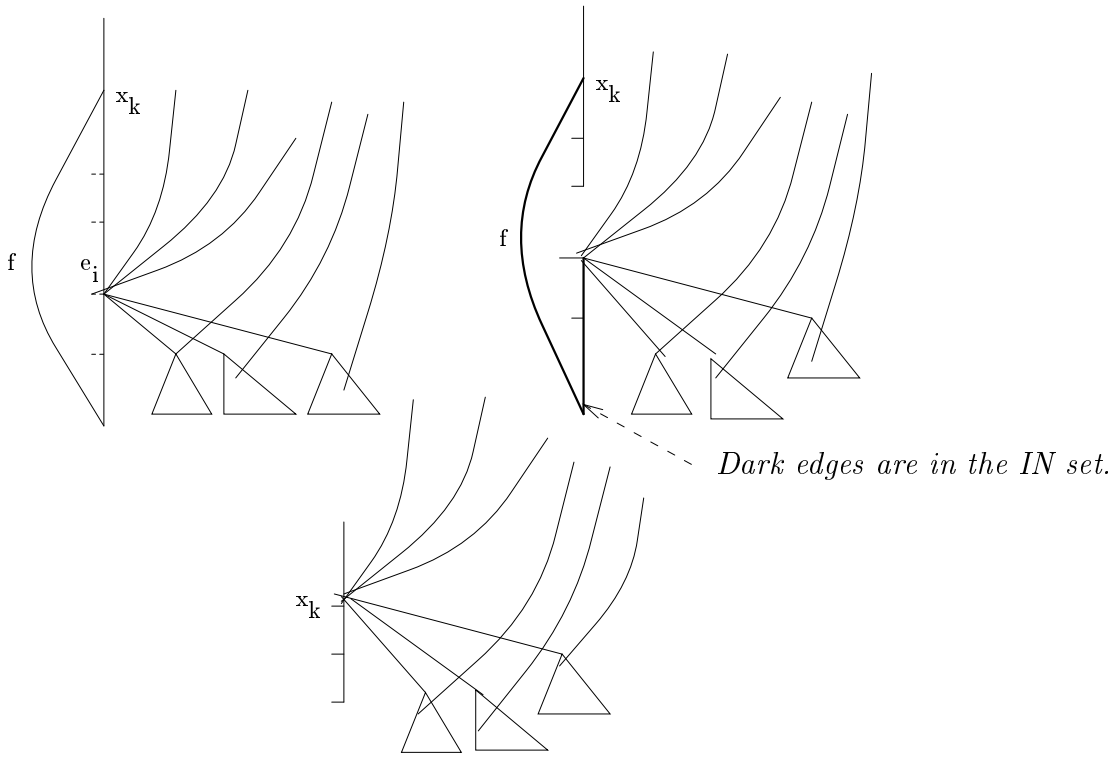

Figure 5: Generation of B_i

Moreover, in order to ensure efficiency we must ensure that all useless tree edges are not to be considered in the replacement process. Useless edges are those edges that do not give rise to an exchange and hence to a spanning tree. These are thus those edges that do not occur in any cycles, i.e., they are the bridges of the graph. We describe their removal below.

For convenience, let *up* and *down* denote the directions towards and away from the root, respectively. For each non-tree edge, its *upper* endpoint is the one closer to the root.

2.3.1 Maintaining the DFS invariant.

We show how to maintain the invariant regarding the d.f.s nature of each spanning tree. Suppose node A of $C(G)$ has been generated and S_A is a d.f.s tree of G_A. Assume that the vertices of S_A are numbered by a postorder traversal of S_A. At node A, we select the non-tree edge f whose upper end point has the least postorder number among all the non-tree edges in G_A. Edge f is used as the replacement edge. Further, the tree edges in the fundamental cycle of f with respect to S_A are replaced in order, starting from the edge farthest from the root and proceeding upwards. Let $e_1, \ldots, e_k$ be these tree edges in that order, and let $e_i = (x_{i-1}, x_i)$. We claim that for each of the sons $B_1, \ldots, B_k$ of A, the

spanning trees at these nodes are d.f.s trees of G_A. This is shown in Lemma 2.8. Before we prove this lemma, we describe the various operations that take place when these sons are generated. This is helpful in describing the proof of Lemma 2.8.

Let T be the data structure storing the spanning trees. T is implemented in the usual manner as a set of parent pointers and child pointers. Before any of the sons of A have been generated, T stores S_A. Consider the generation of son B_i of A. The edges $e_1, \ldots, e_{i-1}, f$ are all in the IN set and therefore, must be contracted in T. As a result of this contraction, the sons of $x_0, \ldots, x_{i-1}$ in T now become sons of x_k. Further all non-tree edges in G_A which have one of $x_0, \ldots, x_{i-1}$ as their lower end points must have their lower end point changed to x_k. Since edges $e_1, \ldots, e_{i-2}, f$ would already have been contracted before son B_{i-1} is generated, only the sons of x_{i-1} need be made sons of x_k and only the lower end points of those non-tree edges which currently have lower end point x_{i-1} need be set to x_k when son B_i is generated (See Figure 5). Note that with this, we have effectively achieved the task of cycle combination.

Lemma 2.8 *For each son B_i, $1 \leq i \leq k+1$ of A, all non-tree edges in G_{B_i} are back edges with respect to S_{B_i}, assuming all non-tree edges in G_A are back edges with respect to S_A.*

Proof: For B_{k+1}, the lemma is easily seen to be true. So consider son B_i, $1 \leq i \leq k$. Consider any non-tree edge g in G_{B_i}. Note that $g \in G_A$. The upper end point of g in S_A must be either x_k or an ancestor of x_k in S_A. From the previous paragraph, when S_{B_i} is obtained from S_A, the lower end point of g either remains unchanged or is changed to x_k. Further, any descendant of x_k in S_A remains a descendant of x_k is S_{B_i}. Thus all non-tree edges in G_{B_i} are back edges with respect to S_{B_i}. The lemma follows. □

2.3.2 Detecting and Removing Bridge Edges in T.

We now show how bridge edges in T are detected and removed as they are formed. This is initially accomplished by dividing the graph into its biconnected components and generating the spanning trees for each component separately. The spanning trees for the entire graph can easily be obtained from the spanning trees of its biconnected components.

Next, we show how bridges are detected at node A of $C(G)$ when its sons $B_1, \ldots, B_{k+1}$ are being generated, assuming that, at A, T has no bridge edges. The following observations are key. Clearly, the only edges in T which can be converted to bridge edges are the edges $e_1, \ldots, e_k$. This can happen when either one of the e_i's is deleted or when f is deleted. The condition that characterizes the situation when e_i becomes a bridge edge is as follows. Let $j \geq i-1$ be the smallest number such that x_j has either a branch (i.e., at least two sons) or an incident non-tree edge (note that x_j would be the lower end point of such a non-tree edge). If $j \geq i$ then e_i is a bridge edge and so are the edges $e_{i+1}, \ldots, e_j$. But $e_{j+1}, \ldots, e_k$ remain non-bridge edges.

The bridge detection procedure which results from the above observations is as follows. First, consider the deletion of edge e_i. By this point, edges $e_1, \ldots, e_{i-1}$ are already contracted. Among the edges $e_{i+1}, \ldots, e_k$, those edges which are converted to bridge edges by this deletion are ascertained by traversing these edges in the above sequence until a

vertex x_j, $j \geq i$, is found such that x_j has either a branch (i.e., at least two sons before the deletion of e_i) or a an incident non-tree edge. The edges traversed are converted to bridge edges by the deletion of e_i. These edges are removed from T by simply removing the vertex x_{j-1} from T. Further, note that when the edges $e_{i+1}, \ldots, e_j$ are deleted subsequently, bridge removal can be accomplished in each case by the deletion of x_{j-1} from T. Later, when e_{j+1} is deleted, bridge detection will be done by traversing the edges $e_{j+2}, \ldots, e_k$ in that order. Next, consider the deletion of edge f. Bridge detection is done by traversing the edges $e_1, \ldots, e_k$ until a vertex x_j, $j \geq 0$, is found such that x_j has either a branch or an incident non-tree edge. Bridge removal involves removing x_j from T.

Lemma 2.9 *The total time taken for bridge detection over deletions of all of $e_1, \ldots, e_k$ is $O(k)$. The time taken for bridge detection when f is deleted is also $O(k)$.*

2.3.3 Data Structures.

Next, we describe details of the data structures used as well as the operations on them along with the time complexity of executing them. Later, we will show that a result similar to Lemma 2.5 holds for this algorithm too.

Storing Tree Edges. The data structure T stores the current spanning tree in the form of a parent pointer and a list of child pointers for each node, with only edges not currently in the IN set present. Determining the fundamental cycle of a particular non-tree edge and deletion of a tree edge are trivial.

Consider edge contraction next. When edge $e_i = (x_{i-1}, x_i)$ is contracted, the sons of x_i are made sons of x_k. This takes time proportional to the number of sons of x_i. In order to account for this time, we use the invariant that each edge in T is a non-bridge edge in the current graph, i.e., it occurs in at least one fundamental cycle.

Lemma 2.10 *Data structure T allows for*

1. *Determining the fundamental cycle of a non-tree edge in time proportional to the length of the cycle.*
2. *Contracting tree edge e_i in time proportional to the number of sons of vertex x_i in T.*

Storing Non-Tree Edges. Recall that non-tree edges have to be ordered by their upper end points. Further, given a vertex v, we need to be able to determine all those non-tree edges which have lower end point v. These edges are required during the contraction of edges.

The following lemma is important for maintaining non-tree edges in the requisite order. It shows that this order is independent of the particular node of $C(G)$ being considered. Therefore, we can initially order non-tree edges in an order given by a postorder traversal of the spanning tree at the root of $C(G)$; this order will hold through the algorithm.

Lemma 2.11 *Suppose for edges $e_a, e_b \in G_{B_i}$, the upper end point of e_a appears before the upper end point of e_b in the postorder ordering of vertices in S_{B_i}, $1 \leq i \leq k+1$. Then the upper end point of e_a appears before the upper end point of e_b in the postorder ordering of vertices in S_A.*

Proof: This is clearly true for $i = k+1$. So suppose $i < k+1$. Edge $f = (x_0, x_k)$ has the least upper end point in the postorder ordering of all vertices in S_A. When B_i is generated, only the portion of S_A consisting of descendants of x_k is modified to give S_{B_i}. No descendants of x_k can have a greater postorder number than x_k. The lemma follows. □

As before, the data structure for storing non-tree edges actually stores multi-edges. This data structure has two components, *REP-LIST* and *ADJ-LIST*. Modifications to one of them can be reflected in the other without any extra time overhead by keeping pointers between the corresponding multi-edges in the two structures.

REP-LIST is simply a list of non-tree multi-edges eligible for replacement, with the multi-edges appearing in the requisite order. At each node A of $C(G)$, an edge f from the first multi-edge in this list is picked for replacement and this edge is deleted from this list to generate the last son of A; selecting the replacement edge and deleting an edge thus takes constant time.

For each vertex v, *ADJ-LIST* stores a list of non-tree multi-edges which ha ve v for their lower end point. Clearly, for vertex x_i, the lower end points of all edges which currently have lower end point x_i can be changed to x_k by merging the multi-edge list for x_i with that for x_k. Any self-loop formed in removed (note that since edges are organized into multi-edges, at most one self loop is formed per merger). This can be done in time proportional to the number of multi-edges in the two lists.

Lemma 2.12 *Data structures* REP-LIST *and* ADJ-LIST *can be maintained such that*

1. *The replacement edge can be selected and deleted in constant time.*
2. *The lower end points of all non-tree edges with lower end point x_i can be transfered to x_k in time proportional to the number of non-tree multi-edges incident upon the two vertices.*
3. *The ordering of non-tree edges remains unchanged.*

2.3.4 Algorithm 2 Pseudo-code

For detail and clarity, we present the pseudo-code of Algorithm 2. The main procedure *Main2* sets up the initial data structures and then uses *Gen2* to generate all spanning trees of G. The basic framework of *Main2* and *Gen2* is similar to that of *Main* and *Gen* but with one notable difference. *Main2* splits G into its biconnected components and generates all spanning trees of G by using *Gen2* to generate all spanning trees of each biconnected component. *Gen2* uses the procedures *Combine-Cycles*, *Remove-Bridges1*, and *Remove-Bridges2*. The first of these makes the changes required when edge edge

e_{i-1} is contraction and edge e_i is deleted in order to generate son B_i of A. The last two perform the detection and removal of bridges from the graph resulting from deletion of edges $e_1, \ldots, e_k$ and the deletion of edge f, respectively. A stack *STACK* is used to store changes made to data structures before recursing so as to undo these changes after the recursion completes.

ALGO Main2(G)

Determine bi-connected components of $G = (G_1, G_2, \ldots, G_k)$;
$T_i \leftarrow$ D.F.S tree of G_i, $1 \leq i \leq k$;
For each i, $1 \leq i \leq k$ do
    Compute *REP-LIST* and *ADJ-LIST*;
Use $Gen2$ to output all spanning trees of G in the
  lexicographic ordering of the biconnected components.
End Main2;

ALGO Gen2(T)

If *REP-LIST*$= \phi$ **then** return;
$f = (u, v) \leftarrow$ First edge in *REP-LIST*;
    /* u represents the lower endpoint of the
    back edge in the DFS tree */
$c_f \leftarrow (x_1 = u, x_2, \ldots, x_{k+1} = v)$;
    /*the sequence of vertices in the fundamental cycle
    of f w.r.t T from bottom upwards */
REP-LIST $\leftarrow$ *REP-LIST* $- f$;
LAST $\leftarrow$ The first ancestor of x_1 whose
    father either has a branch or a back edge;
      /* Useful for bridge elimination, *LAST* is local to $Gen2$ */
For $i = 1$ to k **do**
    $e \leftarrow (x_i, x_{i+1})$;
    LAST $\leftarrow$ *Remove-bridges1(LAST,* x_i*)*;
    $T \leftarrow T - e \cup f$;
    $IN \leftarrow IN + \{f\}$; /* Update the IN set*/
    $OUT \leftarrow OUT + \{e\}$; /* Update the OUT set*/
    CHANGES $\leftarrow$ *CHANGES* $+\{(e, f)\}$;
    Output *CHANGES*;
        /*Output differences from the last tree generated */
    CHANGES $\leftarrow \phi$;
    If $i < k$ *Combine-cycles(*i, x_k, f*)*;
    Gen2(T);
    Restore *LAST* to T;
    CHANGES$\leftarrow$ *CHANGES* $+\{(f, e)\}$;
    $IN \leftarrow IN - \{f\}$;
    $OUT \leftarrow OUT - \{e\}$;
    $T \leftarrow T - f + e$; /* restoring T */
    $IN \leftarrow IN + \{e\}$; /* adding edge removed to OUT set */

```
        inlist ← inlist ∪ e;
    End For;
    IN ← IN − inlist;
    OUT ← OUT + {f};
    Restore-changes;
        /* Undo all changes made by all calls to COMBINE-CYCLES
          in the above for loop using STACK to retrieve changes */
    ADJ-LIST[x_1] ← ADJ-LIST[x_1] − f;
        /* Prepares for last recursion associated with c_f */
    LAST ← Remove-bridges2();
    Gen2(T);
    Restore vertex LAST to T;
    Restore f to the front of REP-LIST and to ADJ-LIST [x_1];
    OUT ← OUT − {f};
End Gen2;

Procedure Combine-cycles(i, x_k, f);
    For each son y of x_i make father(y) ← x_k;
    Store the changes made above on STACK;
    For each sublist, l, of multiple back edges in ADJ-LIST(x_i) do
        If edges in l have upper endpoint v then
            Remove l from REP-LIST and ADJ − LIST[x_i];
                /*Edges in l are now loops and hence removed */
            Store the changes made above on STACK;
        else
            Transfer l to ADJ-LIST(v);
            Store the changes made above on STACK;
End Combine-cycle;

Procedure Remove-bridges1(LAST, x_i);
    If x_i = father of vertex LAST in T then
        LAST ← The first (i.e., closest to x_i) ancestor of x_i in c_f
            whose father has a back edge or a branch;
                /* LAST is x_k if no such vertex exists */
    Remove vertex LAST from T;
    Return(LAST);
End Remove-bridges1;

Procedure Remove-bridges2();
    If ADJ − LIST[x_1] is empty and x_1 has no son in T then
        LAST ←The first (i.e., closest to x_1) ancestor of x_1 in c_f
            whose father has a back edge or a branch;
    else
        LAST ← φ;
```

Remove vertex $LAST$ from T;
Return($LAST$);
End Remove-bridges2;

2.3.5 Time and Space Complexity

Lemma 2.4 holds for this algorithm too. Next, we show that Lemma 2.5 holds too.

Lemma 2.13 *The work done at each node A of $C(G)$ is $O(|s(A)|+|g(A)|+r(A))$, where $s(A)$ is the set of sons of A in $C(G)$, $g(A)$ is the set of sons in $C'(G)$ of nodes in $s(A)$, and $r(A)$ is the number of exchanges output at node A.*

Proof. The work done to output exchanges at A is $O(r(A))$ We consider the other operations performed at node A and show how to amortize the time spent in these operations to nodes in $s(A)$ and $g(A)$.

Clearly, the operations performed in $Gen2$ (excluding those performed in the subprocedures) take time proportional to the number of sons of A. We look at the operations performed in the subprocedures next.

First, consider all invocations of procedure *Remove-Bridges1* and *Remove-Bridges2*. By Lemma 2.9, the time spent in these procedures is proportional to the number of sons of A in $C(G)$.

Second, consider the call to *Combine-Cycle*(i, x_k, f). The two major operations done in this procedure are changing the parent pointers of sons of x_i to x_k and changing the lower endpoints of some non-tree edges from x_i to x_k. We consider each of these in turn.

First, consider the operation of changing the parent pointers of the sons of x_i. By Lemma 2.10, this takes time proportional to the number of sons of x_i in T. Since bridges are removed from T as they are formed, for each son y of x_i in T (note that x_{i-1} is not in T currently as the edge (x_{i-1}, x_i) is in the IN set), there exists at least one non-tree edge f' such that the edge (x_i, y) belongs to the fundamental cycle of f'. The exchange of f' for (x_i, y) leads to a son of B_{i+1} in $C'(G)$ to which the time for changing the parent pointer of y can be charged. Thus, over all calls to *Combine-Cycle*, each node in $g(A)$ gets charged at most once for changing parent pointers.

Second, consider the operation of changing the lower endpoints of some non-tree edges from x_i to x_k. By Lemma 2.12, this takes time proportional to the number of non-tree multi-edges incident upon x_i and x_k. At most one of the multi-edges incident upon x_i, i.e., the multi-edge with upper end point x_k, is converted to a self-loop in the above process. All other multi-edges which are incident upon any of the two vertices have a fundamental cycle containing at least one tree edge; each such fundamental cycle leads to at least one son of B_{i+1} in $C'(G)$. Therefore, the time spent in modifying lower end points in *Combine-Cycle*(i, x_k, f) can be charged to the grandsons of A through B_{i+1} in $C'(G)$. Thus, over all calls to *Combine-Cycle*, each node in $g(A)$ gets charged at most once for changing lower endpoints.

The lemma follows □

Theorem 2.14 *Algorithm 2 generates all spanning trees of T is $O(N+V+E)$ time correctly.*

Proof: We first consider correctness. From the discussions in the previous sections it is easy to see that *Gen* generates the computation tree such that S_A is a d.f.s. tree of G_A. Thus the cycle and exchange generation process is correct. Note that *Gen* maintains G_A as a bi-connected graph. Since *Main* calls *Gen* for each bi-connected component all spanning trees can be generated by taking all possible combinations in the individual components.

We next consider the time complexity. Without loss of generality assume that G is bi-connected. If this is not the case, then for the ith biconnected component of G having N_i spanning trees, V_i vertices and E_i edges, the time taken will be $O(N_1+V_1+E_1)$ and the time to put the computation trees of the various components together in the obvious manner will be $O(\Pi_i N_i + \Sigma_i(V_i+E_i)) = O(N+V+E)$.

From Lemma 2.13, each node of $C(G)$ and $C'(G)$ is charged a constant amount for work done over all nodes of $C(G)$ and by Lemma 2.4, the total work done to output exchanges is $O(N)$. *Main* takes $O(V+E)$ time to construct the initial data structures. The theorem follows. □

Theorem 2.15 *Algorithm 2 takes $O(VE)$ space.*

Proof: The space required for *REP-LIST* and *ADJ-LIST* and T is $O(V+E)$. Stack space for storing changes to be undone after returning from a recursive call is shown to be $O(VE+V^2)$ as follows. The parent vertex of any vertex is changed at most $O(V)$ times along any path of $C(G)$. The lower end point for any non-tree edge is changed at most V times along any path in $C(G)$. An edge is deleted at most once along any path in $C(G)$. Consequently storing changes to *ADJ-LIST*, *REP-LIST* and T require $O(VE)$ space. So the total space is $O(VE)$. □

3 Generating the Computation Tree in Increasing Weight Order

Next, assuming that the edges of G are weighted, we present an algorithm to generate the nodes of $C(G)$ in increasing order of weight.

The algorithm follows a branch and bound strategy on the computation tree. The root of the computation tree is now associated with the minimum spanning tree (MST) of the graph. The sons of the root are obtained as before by exchanging non-tree edges with tree edges. The exchange are made according to an order that ensures that the tree resulting from the exchange is the minimum spanning tree of the updated graph at the corresponding son. To ensure this the non-tree edges are considered for replacement in increasing order of weight. This is repeated at descendant nodes of the computation tree. The entire computation tree is generated in a Branch and Bound fashion. To ensure efficiency, we characterize each spanning tree generated by the exchange pair that generates it in the computation tree. The final algorithm is as follows: The generation

algorithm first generates the tree at the root and the sons at the root are input to a queue indexed by the exchange pair. The actual sorted order is generated by selecting the minimum tree from amongst all queues. This is done by maintaining a priority queue containing the first element of each queue.

The algorithm that we describe is similar to *Gen* but instead of traversing the nodes of the computation tree in a depth first fashion, a branch and bound strategy is used where the node corresponding to the spanning tree to be output next is expanded.

ALGO Genwt
Find min spanning tree, MST;
Repeat
Generate sons of node corresponding to MST
by considering non tree edges in increasing weight order;
Put each spanning tree generated into the
queue indexed by the exchange using which it was
obtained from its parent;
Pick minimum weighted spanning trees, MST, from
priority queue;
Until all queues are empty.
END Genwt

3.1 Correctness and Complexity

Lemma 3.1 *For node A in the computation tree, S_A is the Minimum Spanning Tree of G_A.*

Proof: The proof is by induction on the level of the tree. At the root the claim is true by construction. Assume that the claim is true for a node A. The sons of the node are generated by considering non-tree edges in increasing order of weight. Let the ordered set of non-tree edges at A be $f_1, f_2, \ldots, f_m$. Let non-tree edge f_i be used to generate sons $B_1, \ldots, B_k$ of A by exchanging with tree edges $e_1, \ldots, e_k$. Then edges $f_1, f_2, \ldots, f_{i-1}$ are absent from each of $G_{B_1}, G_{B_2}, \ldots, G_{B_k}$. Thus f_i is the smallest edge that is present in each of these graphs but not in S_A. For each j from 1 to k, since e_j is absent from G_{B_j}, it follows that $S(G_{B_j})$ is the MST of G_{B_j}. □

Lemma 3.2 *The number of exchanges is at most $(V-1)(E-V+1)$.*

Proof: The first entry in the exchange pair has $V-1$ values as the only edges allowed are those in the spanning tree associated with the root. This is true because the second entry in the exchange associated with any node is also in the IN set of that node and it's descendants. The second entry in the exchange pair cannot be an edge in the spanning tree associated with the root, because first entry in the exchange associated with any node is in the OUT set of that node. The result follows. □

The next lemma follows from the branch and bound generation of the computation tree.

Lemma 3.3 *The spanning trees enter the queue for each of the exchanges in sorted order.*

We thus have the following theorem:

Theorem 3.4 *Genwt correctly sorts the nodes of the computation tree in* $O(N \log V + VE)$ *time.*

Proof : The correctness follows from Lemmas C.1, C.2 and C.3. The time may be divided into generation of nodes in the computation tree and processing of the queues. Generation of the nodes in the computation tree requires $O(N + V + E)$ time. Processing the queues requires $O(N \log VE)$ steps since there are at most $O(VE)$ queues. An initial sorting of non-tree edges may require $O(E \log E)$ steps resulting in the required time bound. □

Theorem 3.5 *Genwt requires* $O(N + VE)$ *space.*

Proof: Follows from the space requirements of the computation tree and the queues. □

Output Complexity The output of the algorithm can be the computation tree itself in which the nodes are numbered according to the order in which they are sorted. Pointers to these nodes are maintained from a list whose indices give the index of the spanning tree. The space complexity is $O(N)$ words where each word has $O(\log N)$ bits. The trees can also be explicitly listed out in $O(NV)$ time. Note that the output time thus exceeds the time for sorting.

4 Conclusions

This paper presents a methodology for enumerating subgraphs of a given graph and illustrates this with the spanning tree problem. A companion paper describes enumeration of directed graphs. Efficient enumeration of cycles may also be possible using a similar scheme.

5 Acknowledgements

We thank P.C.P. Bhatt, N.C. Kalra, S.N. Maheshwari and S. Arun-Kumar for comments and pointers to references.

References

[Ch68] J. P. Char, Generation of trees, 2 trees and storage of master forests, IEEE Trans. Circuit Theory, vol. CT-15, pp. 128-138,1968.

[Ga77] H. N. Gabow, Two algorithms for generating weighted spanning trees in order, SIAM J. Comput., vol. 6, pp. 139-150, Mar 77.

[GM78] H. N. Gabow and E. W. Myers, Finding all spanning trees of directed and undirected graphs, SIAM J. Comput., vol. 7, no. 3, Aug 78.

[Ja89] Ann James, A study of algorithms to enumerate all stable matchings and spanning trees, M.Tech thesis, Dept. of Mathematics, Dec 1989, IIT Delhi.

[Jay81] R. Jayakumar, Analysis and study of a spanning tree enumeration algorithm, MS thesis, Dept. of Comp. Sc., IIT Madras, India, 1980. Also reported in Combinatorics and graph theory, Springer-Verlag lecture notes in Mathematics, no. 885, pp. 284-289, 1981.

[JTS84] R. Jayaraman, Thulasiraman, M. N. S. Swamy, Complexity of Computation of a spanning tree enumeration algorithm, IEEE Trans. Circuits and Systems, vol. CAS-31, pp. 853-860,1984.

[KJ89] N. C. Kalra and S. S. Jamuar, Microprocessor based Char's tree enumeration algorithm, JIETE, vol. 35, no. 5, Sep-Oct 89.

[KR92] S. Kapoor and H. Ramesh, An algorithm for generating all spanning trees of directed graphs, manuscript, submitted to *SIAM J. Comput.*.

[Ma72] W. Mayeda, Graph Theory, John Wiley, pp. 252-364, NY 1972.

[Mi65] G.J. Minty, A simple algorithm for listing all trees of a graph, IEEE Trans. Circuit Theory, vol. CT-12, pp. 120, 1965.

[Ra90] H. Ramesh, An algorithm for enumerating all spanning trees of an undirected weighted graph in increasing order of weight, manuscript, also presented at All India Student Seminar - Tryst-90, IIT Delhi.

[TR75] R. E. Tarjan, R. C. Read, Bounds on backtrack algorithms for listing cycles, paths and spanning trees, Networks, 5, 1975, pp. 237-252.