An Algorithm for Finding All the Spanning Trees in Undirected Graphs

Tomomi MATSUI

(March 1993)

Department of Mathematical Engineering and Information Physics
Faculty of Engineering, University of Tokyo
Bunkyo-ku, Tokyo 113, Japan

Abstract: In this paper, we propose an algorithm for finding all the spanning trees in undirected graphs. The algorithm requires $O(n + m + \tau n)$ time and $O(n + m)$ space, where the given graph has n vertices, m edges and τ spanning trees. For outputting all the spanning trees explicitly, this algorithm is optimal.

1 Introduction

This paper considers a problem for finding all the spanning trees in undirected graphs. This problem has a long history and a lot of algorithms have been proposed (e.g., [4, 5, 9]). In 1975, Read and Tarjan presented an algorithm by using a technique called *backtracking* [7]. Their algorithm requires $O(n + m + \tau m)$ time and $O(n + m)$ space, where the given graph has n vertices, m edges and τ spanning trees. In [3], Gabow and Myers refined the backtracking approach and obtained an algorithm with $O(n+m+\tau n)$ time and $O(n+m)$ space. For outputting all the spanning trees explicitly, this algorithm is optimal.

In this paper, we propose an algorithm which generates all the spanning trees explicitly and it requires $O(n + m + \tau n)$ time and $O(n + m)$ space. Both the time complexity and the memory requirement are the same as those of Gabow and Myers' algorithm.

Different from the backtracking approach, our algorithm finds a new spanning tree by exchanging two edges. It means that our algorithm traverses 1-dimensional faces of the spanning tree polytope (the convex hull of the characteristic vectors of spanning trees). More precisely, our algorithm follows a tree structure (a rooted spanning tree) on the skeleton graph of the spanning tree polytope. Our algorithm can be considered as a specialization of the reverse search method proposed by Avis and Fukuda [1], which is a general scheme for solving enumeration problems.

Our algorithm has an advantage in that the tree search rule by which we traverse the tree structure on the skeleton graph is irrelevant to the time complexity. Thus, one can easily construct various algorithms with $O(n + m + \tau n)$ time complexity by using different tree search rules. For example, employing the depth-first search rule and a recursive structure, we obtain an algorithm whose space complexity is $O(n + m)$. A breadth-first implementation requires as much as $O(n\tau + m)$ space, but when a parallel computer is available, this might have an advantage. One can even combine the two tree search rules to construct an algorithm which runs parallel but requires a fixed amount of memory storage. In this sense, our algorithm is flexible.

In [3], Gabow and Myer proposed an efficient algorithm by using a specialized data structure. In our algorithm, we do not require any specialized data structure, but the pre-procedure plays an important role. In the pre-procedure, we partition all the edges into a specialized set of forests by employing the algorithm proposed by Nagamochi and Ibaraki [6] for generating a sparse k-connected graph. Based on this set of forests, we decide the order of edges to exchange.

2 Main Framework

Let us consider an undirected graph $G = (V, E)$ with the vertex-set V and the edge-set E. In this paper, we assume that the graph G is connected and it contains neither self-loops nor parallel edges. We denote $|V|$ by n and $|E|$ by m. An edge-subset $F \subseteq E$ is called a *forest*, when the graph (V, F) does not contain any cycle. If a forest F satisfies that the graph (V, F) is connected, we say F is a *spanning tree* (of G). Denote the set of all the spanning trees of G by $\mathcal{T}$ and $|\mathcal{T}|$ by τ. For any spanning tree T and for any edge $f \notin T$, $\text{cycle}(T, f) \subseteq E$ denotes the unique cycle in $(V, T \cup \{f\})$. For any edge $g \in T$, the graph $(V, T \setminus \{g\})$ contains exactly two components. Thus the set of edges in E connecting these two components becomes a cut-set and we denote the cut-set by $\text{cut}(T, g)$.

Throughout this paper, we assign a linear ordering on the edge-set by setting $E = \{e_1, e_2, \ldots, e_m\}$. For any edge e_j, we say that the *index* of the edge e_j is j. We denote the index of an edge e by $\text{Index}(e)$. Given an edge-subset $E' \subseteq E$, the edge in E' with the smallest index is called the *top edge* of E' and denoted by $\text{top}(E')$. Similarly, we call the edge in E' with the largest index the *bottom edge* of E' and denote the edge by $\text{btm}(E')$. For any pair of edge-subsets E' and E'', we say E' is lexicographically greater than E'', denoted by $E' >_{\text{lex}} E''$ or $E'' <_{\text{lex}} E'$, when there exists an integer j satisfying that (1) for any k with $0 \leq k < j \leq m$, $e_k \in E'$ if and only if $e_k \in E''$ and (2) $E' \ni e_j \notin E''$.

Throughout this paper, we assume that the following property holds

Assumption 1 *The edge-subset* $T^* = \{e_1, e_2, \ldots, e_{n-1}\}$ *is a spanning tree of* G.

Clearly, T^* is the lexicographically maximum spanning tree of G.

For any spanning tree T' $(\neq T^*)$, $\phi(T')$ denotes the spanning tree $(T' \setminus \{f\}) \cup \{g\}$, where f is the bottom edge of T' and g is the top edge of the cut-set $\text{cut}(T', f)$.

Lemma 1 *Let* T' *be a spanning tree satisfying* $T' \neq T^*$. *If* $\phi(T') \setminus T' = \{g\}$, *then the edge* g *is contained in* T^*.

Proof. The definition of ϕ implies that the edge g is the top edge of $\text{cut}(T', f)$, where f is the bottom edge of T'. Since T^* is a spanning tree, $\text{cut}(T', f) \cap T^* \neq \emptyset$ and $\text{Index}(g) = \text{Index}(\text{top}(\text{cut}(T', f))) \leq \text{Index}(\text{top}(\text{cut}(T', f) \cap T^*)) \leq \text{Index}(\text{btm}(T^*)) = n-1$. Then, Assumption 1 implies that $g \in T^*$. □

The proof of the above lemma implies that if $T' \; (\neq T^*)$ is a spanning tree of G, then $|\phi(T') \cap T^*| = |T' \cap T^*| + 1$, $\phi(T') >_{\text{lex}} T'$ and $1 \leq \exists k \leq n-1$, $\phi^k(T') = T^*$.

Given a spanning tree $T'(\neq T^*)$, we say T' is a *child* of $\phi(T')$ and $\phi(T')$ is the *parent* of T'. Then, we can construct a tree structure, denoted by $(\mathcal{T}, \phi)$, with the node set $\mathcal{T}$ and the arc set $\{(T, T') \in \mathcal{T} \times \mathcal{T} \mid T' \neq T^*,\ T = \phi(T')\}$. Clearly, the spanning tree T^* corresponds to the root of this tree structure. Our algorithm searches this tree structure by using a tree search rule and outputs all the spanning trees of the given graph.

3 Finding All the Children

In this section, we propose a procedure for finding all the children of a spanning tree.

3.1 Naive Algorithm

Let T be a spanning tree of G and $T' = (T \setminus \{g\}) \cup \{f\}$ be a child of T. It is obvious that g is the top edge of $\text{cut}(T', f)$. Since $\text{cut}(T', f) = \text{cut}(T, g)$, g is the top edge of $\text{cut}(T, g)$. We denote the edge-subset $\{e' \in T \mid e' = \text{top}(\text{cut}(T, e'))\}$ by $H(T)$. Then, $g \in H(T)$ and the edge f joins two different components of the graph $(V, T \setminus H(T))$.

Given a spanning tree T of G, we say that an edge f is a *pivot edge* of T, if there exists a child T' of T such that $T' \setminus T = \{f\}$. Then the following lemma characterizes the pivot edges.

Lemma 2 *Let T be a spanning tree of G. An edge f is a pivot edge of T if and only if f satisfies the conditions that (1) $\mathit{Index}(f) > \mathit{Index}(\mathit{btm}(T))$, and (2) the edge f joins a pair of distinct components of the graph $(V, T \setminus H(T))$.*

Proof. When f is a pivot edge, it is clear that f satisfies the above two conditions. Conversely, suppose that the edge f satisfies above two properties. It is obvious that $\text{cycle}(T, f) \cap H(T) \neq \emptyset$. Let g be an edge in $\text{cycle}(T, f) \cap H(T)$. Clearly, the edge-subset $T' = (T \setminus \{g\}) \cup \{f\}$ is a spanning tree of g. From the condition (1), f is the bottom edge of the spanning tree T'. Since $g \in H(T)$ and $\text{cut}(T, g) = \text{cut}(T', f)$, g is the top edge of $\text{cut}(T, g) = \text{cut}(T', f)$. Thus, T' is a child of T and so f is a pivot edge of T. □
Consider the following problem:

<u>`label`$((V, T), H)$</u>

input: a graph (V, T) with a vertex-set V and a spanning tree (an edge-set) T, and an edge-subset $H \; (\subseteq T)$.

output: labels of vertices satisfying that two vertices have same labels if and only if the two vertices are connected in the graph $(V, T \setminus H)$

When vertex labels are obtained by solving the problem `label`$((V, T), H(T))$, it is clear that an edge f is a pivot edge of T if and only if $\text{Index}(f) > \text{Index}(\text{btm}(T))$ and the end vertices of f have different labels. Thus, for each edge f, we can check whether f is a pivot edge or not in $\text{O}(1)$ time.

When we obtain a pivot edge f of a spanning tree T, the following lemma gives an idea for finding all the children of T with bottom edge f.

Lemma 3 *Let T be a spanning tree of G and f be an edge which satisfies the condition $Index(f) > Index(btm(T))$. Then an edge-subset T' with the bottom edge f is a child of T if and only if there exists an edge g satisfying that $g \in cycle(T,f) \cap H(T)$ and $T' = (T \setminus \{g\}) \cup \{f\}$.*

Proof. Let g be an edge in $\mathrm{cycle}(T,f) \cap H(T)$. Since $g \in \mathrm{cycle}(T,f)$, the edge-subset $T' = (T \setminus \{g\}) \cup \{f\}$ is a spanning tree. From the definition of T', g is the top edge of $\mathrm{cut}(T,g) = \mathrm{cut}(T',f)$ and it implies that T is the parent of T'.

The only if part is easy. □

Now we propose a naive algorithm for generating all the children of a spanning tree T. At first, we solve the problem `label`$((V,T),H(T))$. Next, we find all the pivot edges of T by comparing the end vertices of every edge whose index is larger than that of the bottom edge of T. When a pivot edge f is obtained, we construct the edge-subset $\mathrm{cycle}(T,f) \cap H(T)$ and output the spanning tree $(T \setminus \{g\}) \cup \{f\}$ for every $g \in \mathrm{cycle}(T,f) \cap H(T)$. In the worst case, this algorithm checks $\mathrm{O}(m-n+1)$ edges and finds that the spanning tree T has no child. In the next subsection, we give an idea for finding all the pivot edges efficiently.

3.2 Pre-procedure

In the rest of this section, we assume that the following condition holds.

Assumption 2 *There exists a sequence of integers $(j_0, j_1, \ldots, j_r)$ satisfying that (1) $0 = j_0 < j_1 < \cdots < j_r = m$, and (2) the edge partition $(F_1, F_2, \ldots, F_r)$, where $F_s = \{e_{j_{s-1}+1}, e_{j_{s-1}+2}, \ldots, e_{j_s}\}$, satisfies the condition that:*

$$\forall s \in \{1,2,\ldots,r\},\ F_s \text{ is a maximal forest of the graph } (V, F_s \cup F_{s+1} \cup \cdots \cup F_r).$$

Since the given graph $G = (V,E)$ is connected, the forest F_1 is a spanning tree of G and so the edge indices satisfying the above assumption also satisfy Assumption 1. In the paper [6], Nagamochi and Ibaraki proposed an $\mathrm{O}(n+m)$ time algorithm for finding a partition $(F_1, F_2, \ldots, F_r)$ of the edge-set E which satisfies the above condition. Thus, when we employ their algorithm in a pre-procedure, we can assign the integer numbers $\{1,2,\ldots,m\}$ to E so that Assumption 2 holds. The above assumption directly implies the following property.

Claim 4 *Let s be an integer satisfying $1 < s \le r$. For any edge $\{u,v\} \in F_s$, the vertices u and v are connected in the graph (V, F_{s-1}).*

From the above claim, we can show the following.

Lemma 5 *Let T be a spanning tree of G and k the index of the bottom edge of T.*
(1) If T has at least one child, then there exists a pivot edge f satisfying the condition that $Index(f) \le k + 2n - 3$.
(2) For any pivot edge f of T, either $Index(f) \le k + 2n - 3$ or there exists a pivot edge f' satisfying the condition that $Index(f) - 2n + 3 \le Index(f') < Index(f)$.

Proof. (1) Let $f = \{u, v\}$ be the pivot edge of T with the smallest index. Suppose that $\text{Index}(f) > k + 2n - 3$. Let F_s be the forest containing the edge f. Then the forest F_{s-1} satisfies that $k < \text{Index}(\text{top}(F_{s-1}))$ and $\text{Index}(\text{btm}(F_{s-1})) < \text{Index}(f)$. From Claim 4, there exists a path P in the graph (V, F_{s-1}) connecting u and v. Since f is a pivot edge, u and v are not connected in the graph $(V, T \setminus H(T))$. Thus the path P contains an edge which joins a pair of distinct components of the graph $(V, T \setminus H(T))$. From Lemma 2, P contains at least one pivot edge and it is a contradiction. We can show the property (2) analogously. $\square$

The above lemma gives a condition to stop searching the pivot edges. Denote the bottom edge of the current spanning tree T by e_k. In our algorithm, we compare the labels of two end vertices of the edges $(e_{k+1}, e_{k+2}, \ldots, e_m)$ in this order. When consecutive $2n - 3$ edges are not pivot edges, we can stop finding pivot edges.

3.3 Algorithm for Finding All the Children

Now we describe an algorithm for finding all the children. In the previous sub-sections, we discussed a method for finding all the children of a spanning tree T under the assumption that the edge-subset $H(T)$ is obtained. However, it seems time consuming to construct the edge-subset $H(T)$ directly from T. In our algorithm, we construct the edge-subset $H(T')$, for each child T' of T. The following lemma gives an idea to obtain $H(T')$ efficiently.

Lemma 6 *Let T be a spanning tree of G and $T' = (T \setminus \{g\}) \cup \{f\}$ a child of T. Then, $H(T') = H(T) \setminus \{e' \in \mathit{cycle}(T, f) \cap H(T) \mid \mathit{Index}(e') \geq \mathit{Index}(g)\}$*

Proof. Clearly, $T' \cap T^* = (T \cap T^*) \setminus \{g\}$.
Case (1): Let e' be an edge in $(T' \cap T^*) \setminus (\text{cycle}(T, f))$. Then $\text{cut}(T, e') = \text{cut}(T', e')$, and so $e' \in H(T')$ if and only if $e' \in H(T)$.
Case (2): Let e' be an edge in $(T' \cap T^*) \cap \text{cycle}(T, f)$. Then it is clear that $\text{cut}(T', e') = \text{cut}(T, e') \bigtriangleup \text{cut}(T, g)$, where $\bigtriangleup$ denotes the symmetric difference.

Case (2-i): Suppose that $e' \notin H(T)$. Denote the top edge of $\text{cut}(T, e')$ by e''. When $e'' \in \text{cut}(T, g)$, then $\text{Index}(g) < \text{Index}(e'') < \text{Index}(e')$, $g \in \text{cut}(T', e')$, and so $e' \notin H(T')$. If $e'' \notin \text{cut}(T, g)$, then $\text{Index}(e'') < \text{Index}(e')$, $e'' \in \text{cut}(T', e')$, and we have $e' \notin H(T')$. Therefore, if $e' \in (T' \cap T^*) \cap \text{cycle}(T, f)$ and $e' \notin H(T)$, then $e' \notin H(T')$.

Case (2-ii): Consider the case that $e' \in H(T)$. Clearly, $e' = \text{top}(\text{cut}(T, e'))$ and $g = \text{top}(\text{cut}(T, g))$. It implies that the top edge of $\text{cut}(T', e')$ is either e' or g. Thus, e' is the top edge of $\text{cut}(T', e')$ if and only if $\text{Index}(e') < \text{Index}(g)$.

Summerizing the above, we obtain the required result. $\square$

The above lemma says that if T' is a child of a spanning tree T, then $H(T') \subset H(T) \subseteq H(T^*) = T^*$.

Now we describe our algorithm.

Algorithm A

input: a spanning tree T and the edge-subset $H = H(T)$

output: all the pairs $\{(T', H(T')) \mid T' \text{ is a child of } T\}$

```
 A1: begin
 A2:   solve the problem label $((V,T),H)$ ;
 A3:   set $k :=$ Index(btm($T$)) ; set $j' := k$ ; set $j := k+1$ ;
 A4:   while $j \leq j' + 2n - 3$ do
 A5:     begin
 A6:       if labels of two ends of $e_j$ are same then set $j := j+1$;
 A7:       else
 A8:         begin
 A9:           set $f := e_j$ ; set $j' := j$ ; set $j := j+1$ ; set $D :=$ cycle$(T,f) \cap H$ ;
A10:           for $g \in D$ do
A11:             begin
A12:               set $T' := (T \setminus \{g\}) \cup \{f\}$ ;
A13:               set $H' := H \setminus \{e' \in D \mid \text{Index}(e') > \text{Index}(g)\}$ ;
A14:               output $(T', H')$ ;
A15:             end;
A16:         end;
A17:     end;
A18: end
```

Now, we discuss the time complexity of the above algorithm. In our algorithm, we maintain the current spanning tree T by the adjacency list of the graph (V,T). Then, for any pair of vertices, we can find a unique path in T connecting the pair in $O(n)$ time by employing a tree search algorithm, e.g., depth-first search and/or breadth-first search (see [8]). For each edge e of the current spanning tree T, we assume that we can check whether $e \in H$ or $e \notin H$ in $O(1)$ time. For example, it is accomplished by a 1-dimensional array indexed by the edges T which represents the characteristic vector of H. Then, we can solve the problem `label` $((V,T),H)$ in $O(n)$ time by applying a tree search algorithm to the graph $(V, T \setminus H)$. Thus, the time complexity of every line in the above algorithm is bounded by $O(n)$. Especially, Lines A4 and A6 require $O(1)$ time. Let τ' be the number of children of T. Lemma 5 says that Lines A4 and A6 are executed at most $2(2n-3)\tau'$ times. All the other lines are executed at most τ' times. From the above, the overall time complexity becomes $O(n + m + \tau' n)$. Clearly, the memory requirement is bounded by $O(n+m)$.

4 Algorithms for Finding All the Spanning Trees

Here, we discuss some algorithms for finding all the spanning trees in undirected graphs.

In the previous section, we described an algorithm for finding all the children of a spanning tree. By using the algorithm as a subprocedure, we can construct a spanning tree enumeration algorithm which searches the tree structure $(\mathcal{T}, \phi)$ by using a tree search rule, e.g., depth-first search rule and/or breadth-first search rule. As described in the previous section, we can assign the integer numbers $\{1, 2, \ldots, m\}$ to E so that Assumption 2 holds by applying Nagamochi and Ibaraki's algorithm in $\mathrm{O}(n+m)$ time. Thus, the overall time complexity of the algorithm becomes $\mathrm{O}(n+m+\tau n)$.

Now we discuss the memory complexity. Each subprocedure requires a memory space to maintain the doublet $(T', H(T'))$ for each child T' of the current spanning tree. Clearly, any spanning tree has $\mathrm{O}(nm)$ children. It implies that if we use the depth-first search rule, the algorithm maintains at most $\mathrm{O}(n^2 m)$ doublets, since the height of the tree structure $(\mathcal{T}, \phi)$ is $\mathrm{O}(n)$. Thus the algorithm requires $\mathrm{O}(n^3 m)$ memory space. If we employ both the depth-first search rule and the recursive structure, each subprocedure maintains only one doublet and so the over all memory complexity becomes $\mathrm{O}(n^2)$.

In the rest of this section, we describe an $\mathrm{O}(m+n)$ space algorithm. We modify the algorithm obtained by employing the depth-first search rule and a recursive structure as follows. When we apply the depth-first search to the tree structure, we have to maintain a sequence of spanning trees $(T^0 = T^*, T^1, T^2, \ldots, T^l)$ satisfying that T^j is a child of T^{j-1} and the length l is less than n. To save the memory space for the doublets $\{(T^j, H(T^j)) \mid j = 1, 2, \ldots, l\}$, we maintain the relative differences $\{f^j\} = T^j \setminus T^{j-1}$, $\{g^j\} = T^{j-1} \setminus T^j$, and $\Delta H^j = H(T^{j-1}) \setminus H(T^j)$. Since $l < n$, we need to maintain at most $\mathrm{O}(n)$ edge pairs (f^j, g^j). From Lemma 6, $T^* = H(T^0) \supset H(T^1) \supset \cdots \supset H(T^l)$ and so the overall memory space required for the edge-subsets $\{\Delta H^1, \Delta H^2, \ldots, \Delta H^l\}$ is bounded by $\mathrm{O}(n)$. At last, we show an idea to save the memory space for the edge-subsets $D^j = \mathrm{cycle}(T^{j-1}, f^j) \cap H(T^{j-1})$ $(j = 1, 2, \ldots, l)$. Clearly, $T^j(g) = (T^{j-1} \setminus \{g\}) \cup \{f^j\}$ is a child of T^{j-1} for every $g \in D^j$. If we construct the children $\{T^j(g) \mid g \in D^j\}$ in the order that the edge index of g is decreasing, we do not need to maintain the edge-subset D^j. When we update the edge g, we construct the edge-subset D^j again in $\mathrm{O}(n)$ time and replace the edge g by the bottom edge of $\{e' \in D^j \mid \mathrm{Index}(e') < \mathrm{Index}(g)\}$. The above modification saves the memory complexity to $\mathrm{O}(m+n)$ without decreasing the time complexity. However, it seems that this modification makes the algorithm slower in practice.

In the following, we describe the above algorithm in a pseudo code.

Algorithm B

```
B1:  **begin**
B2:    **procedure** find-children $(T, H)$
B3:    (**comment:** $T$ is the current spanning tree and $H = H(T)$ )
B4:      **begin**
B5:        output $T$ ;
B6:        solve the problem label $((V, T), H)$ ;
B7:        set $k := \mathrm{Index}(\mathrm{btm}(T))$ ; set $j' := k$ ; set $j := k + 1$ ;
B8:        **while** $j \leq j' + 2n - 3$ **do**
B9:          **begin**
B10:           **if** labels of two ends of $e_j$ are different **then** $j := j + 1$ ;
B11:           **else**
B12:             **begin**
B13:               set $f := e_j$ ; set $j' := j$ ; set $j := j + 1$ ; set $g := e_n$ ; set $\Delta H := \emptyset$ ;
```

```
B14:                do
B15:                  begin
B16:                    set $g :=$ btm($\{e' \in$ cycle$(T,f) \cap H \mid$ Index$(e') <$ Index$(g)\}$) ;
B17:                    set $T := T \setminus \{g\} \cup \{f\}$ ; set $H := H \setminus \{g\}$ ; set $\Delta H := \Delta H \cup \{g\}$ ;
B18:                    find-children $(T, H)$ ;
B19:                    set $T := T \setminus \{f\} \cup \{g\}$ ;
B20:                  end;
B21:                while $g$ is not the top edge of cycle$(T,f) \cap H$ ;
B22:                set $H := H \cup \Delta H$
B23:              end;
B24:          end;
B25:      end; (comment: end of procedure find-children)
B26:    begin (comment: start of the main routine)
B27:      set $T = T^*$ ; set $H = T^*$ ;
B28:      find-children $(T, H)$ ;
B29:    end; (comment: end of the main routine)
B30: end
```

5 Discussions

In this section, we give some ideas which are useful for practical implementations.

In Algorithms A and B, we solve the problem `label`$((V,T), H(T))$. However, from the vertex labels of the current spanning tree, we can efficiently construct the vertex labels of every child in practice. Assume that a pivot edge f is obtained. If we construct the child $(T \setminus \{g\}) \cup \{g\}$ for each edge $g \in D =$ cycle$(T,f) \cap H(T)$ in the order that the index of the edge g is decreasing, then the components of the graph $(V, T \setminus H(T))$ are merged one by one. So, we can employ a comfortable data structure for disjoint sets (see [2] Section 22 for example). In the above modification, we have to sort the edge-subset D by the edge indices and it requires $\mathrm{O}(|D| \log |D|)$ time. Since the current tree has $|D|$ children with bottom edge f, it does not increase the time complexity.

For each child of the current spanning tree, we can enumerate their children independently in our algorithm without increasing the time complexity. In the previous section, we proposed an algorithm which searches the tree structure $(\mathcal{T}, \phi)$ by the depth-first search rule. However, if a parallel computer is available, it seems that the breadth-first search rule provides an efficient speedup over the sequential algorithm.

When we have the lexicographically minimum spanning tree, we can find the pivot edge with the largest index in $\mathrm{O}(n)$ time by using the following property.

Lemma 7 *Let T be a spanning tree with at least one child. Then the pivot edge of T with the largest index is contained in the lexicographically minimum spanning tree of G.*

Proof. Let f be the pivot edge of T with the largest index. Then there exists an edge g in cycle$(T,f) \cap H(T)$. Clearly, f is the bottom edge of the cut-set cut(T,g). For any cut-set $C' \subseteq E$, the bottom edge of C' is contained in the lexicographic minimum spanning tree. Thus f is contained in the lexicographically minimum spanning tree. □

The above lemma implies that when we check the edges in the lexicographically minimum spanning tree, we can find the pivot edge of T with the largest index in $O(n)$ time, if T has a child. Let e_l be the pivot edge of the current spanning tree with the largest index and e_k the bottom edge of the current spanning tree. Then, in Algorithms A and B, we can apply the 'pivot edge test' for each edge in $\{e_{k+1}, e_{k+2}, \ldots, e_l\}$ in any order without changing the time complexity. When we use a parallel computer, we can apply the above pivot edge tests independently.

Lastly, we consider the case that the given graph is weighted. Let $w(e)$ be the weight of the edge $e \in E$.

Lemma 8 *Let* $T^* = \{e_1, e_2, \ldots, e_{n-1}\}$ *be a minimum weight spanning tree. If the edge-set* $\{e_1, e_2, \ldots, e_m\}$ *satisfies Assumption 1, then for any spanning tree* $T'(\neq T^*)$, $\sum_{e \in T'} w(e) \geq \sum_{e \in \phi(T')} w(e)$.

Proof. Denote the spanning tree $\phi(T')$ by $T = (T' \setminus \{f\}) \cup \{g\}$. Assume the contrary that $\sum_{e \in T'} w(e) < \sum_{e \in T} w(e)$. Since $\sum_{e \in T} w(e) = \sum_{e \in T'} w(e) - w(f) + w(g)$, $w(f) < w(g)$. Then it is clear that $w(g) > w(f) \geq \min\{w(e) \mid e \in \mathrm{cut}(T, g)\}$. It contradicts with the property that g is contained in a minimum weight spanning tree T^*. □

Then the weight of a spanning tree is greater than or equal to that of its parent, if it exists. When we need to solve a minimum weight spanning tree problem with additional constraints, we can construct an enumerative algorithm which searches the tree structure $(\mathcal{T}, \phi)$ by using the best-first search rule.

References

[1] D. Avis and K. Fukuda, Reverse search for enumeration, Research report 92-5, Graduate School of Systems Management, The University of Tsukuba, (1992).

[2] T.H. Cormen, C.H. Leiserson and R.L. Rivest, *Introduction to Algorithms* (The MIT-Press, Cambridge, Massachusetts, 1990).

[3] H.N. Gabow and E.W. Myers, Finding all spanning trees of directed and undirected graphs, *SIAM J. Computing* 7 (1978) 280–287.

[4] W. Mayeda and S. Seshu, Generation of trees without duplications, *IEEE Trans. on Circuit Theory* CT-12 (1965) 181–185.

[5] G.J. Minty, A simple algorithm for listing all the trees of a graph, *IEEE Trans. on Circuit Theory* CT-12 (1965) 120.

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

[7] R.C. Read and R.E. Tarjan, Bounds on backtrack algorithms for listing cycles, paths, and spanning trees, *Networks* 5 (1975) 237–252.

[8] R.E. Tarjan, Depth-first search and linear graph algorithms, *SIAM J. Comput.* 1 (1972) 146–160.

[9] H. Watanabe, A computational method for network topology, *IRE Trans. on Circuit Theory* CT-7 (1960) 296–302.