Efficiently Scanning All Spanning Trees of an Undirected Graph

Akiyoshi SHIOURA
Department of Information Sciences
Tokyo Institute of Technology
2-12-1 Oh-okayama, Meguro-ku
Tokyo 152, Japan

Akihisa TAMURA
Department of Computer Science
and Information Mathematics
The University of Electro–Communications
1-5-1 Chofugaoka, Chofu-shi
Tokyo 182, Japan

(July 1993; revised November 1994)

Abstract: Let G be an undirected graph with V vertices and E edges. We consider the problem of enumerating all spanning trees of G. In order to explicitly output all spanning trees, the output size is of $\Theta(NV)$, where N is the number of spanning trees. This, however, can be compressed into $\Theta(N)$ size. In this paper, we propose a new algorithm for enumerating all spanning trees of G in such compact form. The time and space complexities of our algorithm are $O(N+V+E)$ and $O(VE)$, respectively. The algorithm is optimal in the sense of time complexity.

1 Introduction

In this paper we consider the problem of enumerating all spanning trees of an undirected graph with V vertices, E edges and N spanning trees. Several algorithms for this problem have been proposed (see [7], [8], [4], [5], [6]). To explicitly enumerate all spanning trees, the total output size is $\Theta(NV)$. For this requirement, Gabow and Myers' algorithm [4], and Matsui's [6] which require $O(NV+V+E)$ time and $O(V+E)$ space are best in sense of both time and space complexities. The spanning tree sequence of $\Theta(NV)$ size, however, can be compressed into $\Theta(N)$ size. Recently, Kapoor and Ramesh [5] gave an algorithm for outputting such 'compact' form, which requires $O(N+V+E)$ time and $O(VE)$ space.

In this paper, we propose a new algorithm for outputting all spanning trees in a 'compact' form. The time and space complexities are $O(N+V+E)$ and $O(VE)$, the same as Kapoor and Ramesh's, but the structure is more simple. Our algorithm can be regarded as an application of the reverse search method proposed by Avis and Fukuda [3]. The reverse search method is a scheme for general enumeration problems (see [1], [2], [6]). From the standpoint of the reverse search scheme, our algorithm assumes that any spanning tree other than a specified spanning tree T^0 has a unique parent so that T^0 is the 'progenitor' of all spanning trees. It then outputs all spanning trees by reversely scanning all children of any spanning tree. In Section 3, we define a useful child-parent relation and propose a naive algorithm for scanning all children. In Section 4, we present an efficient implementation of our algorithm which attains the desired complexities.

2 Preliminaries

Let G be an undirected connected graph consisting of a vertex-set $\{v_1, \cdots, v_V\}$ and an edge-set $\{e_1, \cdots, e_E\}$. We consider the natural total orders (denoted by $<$) over the vertex-set and the edge-set according to subscripts. Let us call the smallest vertex v_1 the *root* of G. Each edge e has two incidence vertices, written as $\partial^+ e$ and $\partial^- e$. Here we assume that $\partial^+ e$ is smaller than or equal to $\partial^- e$, and call these the *tail* and *head* of e, respectively. A *spanning tree* of G is a connected subgraph of G which contains no circuit and which contains all vertices. Without loss of generality, we represent a spanning tree as its edge-set of size $(V-1)$. For any spanning tree T and any edge $f \in T$, the subgraph induced by the edge-set $T \setminus f$ has exactly two components (we write a singleton $\{f\}$ as f, whenever there is no confusion.) The set of edges connecting these components is called a *fundamental cut* associated with T and f, and here written as $C^*(T \backslash f)$. It is well-known that for any edge $f \in T$ and for an arbitrary edge $g \in C^*(T \backslash f)$, $T \backslash f \cup g$ is also a spanning tree. For any edge $g \notin T$, the edge-induced subgraph of G by $T \cup g$ has a unique circuit, called a *fundamental circuit* associated with T and g. We denote the set of edges of the circuit as $C(T \cup g)$. For any $g \notin T$ and for any $f \in C(T \cup g)$, $T \cup g \backslash f$ is a spanning tree. Relative to a spanning tree T of G, if the unique path in T from vertex v to the root v_1 contains a vertex u then u is called an *ancestor* of v and v is a *descendant* of u. Similarly, for two edges e and f in T, we call e an *ancestor* of f and f a *descendant* of e if the unique path in T from f to the root v_1 contains e. A 'depth-first spanning' tree of G is a spanning tree which is found by some depth-first search of G. It is known that a *depth-first spanning tree* is defined as a spanning tree such that for each edge of G, its one incidence vertex is an ancestor of the other.

3 Algorithm for finding all spanning trees

Given a graph G, let us consider the graph $\mathcal{S}(G)$ whose vertex-set $\mathcal{T}$ is the set of all spanning trees of G and whose edge-set $\mathcal{A}$ consists of all pairs of spanning trees which are obtained from each other by exchanging exactly one edge using some fundamental cut or circuit. For example, the graph $\mathcal{S}(G_1)$ of the left one G_1 is shown in Figure 1.

Our algorithm finds all spanning trees of G by implicitly traversing some spanning tree $\mathcal{D}$ of $\mathcal{S}(G)$. In order to output this sequence of all spanning trees, $\Theta(|\mathcal{T}| \cdot V)$ time is needed. However, if we output explicitly only the first spanning tree, and restrict the output of all others to the sequence of exchanged edge-pairs of G obtained by traversing $\mathcal{D}$, then $O(|\mathcal{T}| + V)$ time is enough, because $|\mathcal{D}| = |\mathcal{T}| - 1$ and exactly two edges of G are exchanged for each edge of $\mathcal{D}$. Furthermore, by scanning such a 'compact' output, one can construct all spanning trees. Since we adopt such a compact output, it becomes desirable to find the next spanning tree from a current one efficiently (in constant time.) In this section, we propose an algorithm for generating a compact sequence of spanning trees. The next section is devoted to an efficient implementation of our algorithm and its analysis.

Hereafter we assume that G whose vertex-set $\{v_1, \cdots, v_V\}$ and whose edge-set $\{e_1, \cdots, e_E\}$ satisfies the following conditions:

(1) a depth-first spanning tree T^0 of G is given ;
(2) $T^0 = \{e_1, \cdots, e_{V-1}\}$, and any edge in T^0 is smaller than its proper descendants ;
(3) each vertex v is smaller than its proper descendants relative to T^0;
(4) for two edges $e, f \notin T^0$, $e < f$ only if $\partial^+ e \leq \partial^+ f$.

For instance, graph G_1 of Figure 1 satisfies these conditions. In fact, one can find a depth-first spanning tree T^0 and sort vertices and edges of G in $O(V + E)$ time so that G satisfies the above conditions, by applying Tarjan's depth-first search technique [9]. We note that conditions (1) and (2) are sufficient in order to show the correctness of our algorithm. We, however, need further conditions (3) and (4) for an efficient implementation.

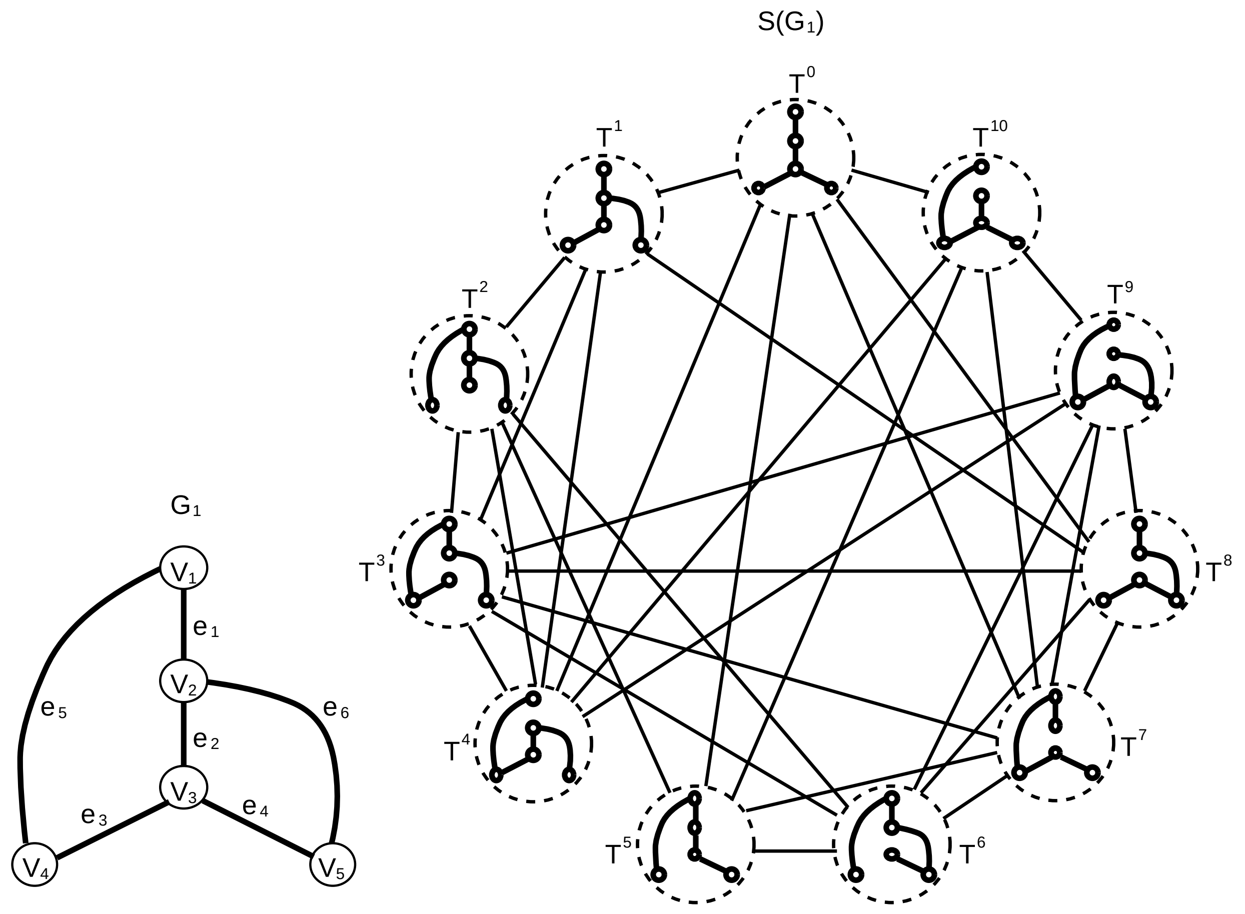

Figure 1: graph G_1 and graph $\mathcal{S}(G_1)$

Suppose that for any nonempty subset S of the edge-set, $\mathrm{Min}(S)$ denotes the smallest edge in S. For convenience, we assume that $\mathrm{Min}(\emptyset) = e_V$.

Lemma 3.1. *Let T^c be an arbitrary spanning tree other than T^0. Under conditions* (1) *and* (2), *if $f = \mathrm{Min}(T^0 \setminus T^c)$ then*

$$|C(T^c \cup f) \cap C^*(T^0 \backslash f) \setminus f| = 1,$$

where $|\cdot|$ denotes its cardinality.

Proof. Since $T^c \neq T^0$ and $f = \mathrm{Min}(T^0 \setminus T^c)$, $f \in T^0$ and $f \notin T^c$. Let us consider the unique path from $\partial^- f$ to $\partial^+ f$ in T^c. Obviously, the fundamental circuit $C(T^c \cup f)$ is the union of this path and f. The set $T^0 \setminus f$ has exactly two components such that one of these contains $\partial^- f$ and the other $\partial^+ f$. Thus the path passes through at least one edge in $C^*(T^0 \backslash f) \setminus f$, that is, $C(T^c \cup f) \cap C^*(T^0 \backslash f) \setminus f \neq \emptyset$. Since T^0 is a depth-first spanning tree, without loss of generality, we assume that the head of any edge is a descendant of its tail relative to T^0. Suppose that e is the first edge in the path from $\partial^- f$ with $e \in C^*(T^0 \backslash f) \setminus f$. Then, the tail $\partial^+ e$ of e is an ancestor of $\partial^+ f$ relative to T^0, and the head $\partial^- e$ is a descendant of $\partial^- f$. From condition (2) and the minimality of f, $\partial^+ e$ and $\partial^+ f$ are connected in $T^c \cap T^0$. Thus, e is a unique edge in $C(T^c \cup f)$ with $e \in C^*(T^0 \backslash f) \setminus f$, that is, $|C(T^c \cup f) \cap C^*(T^0 \backslash f) \setminus f| = 1$. ∎

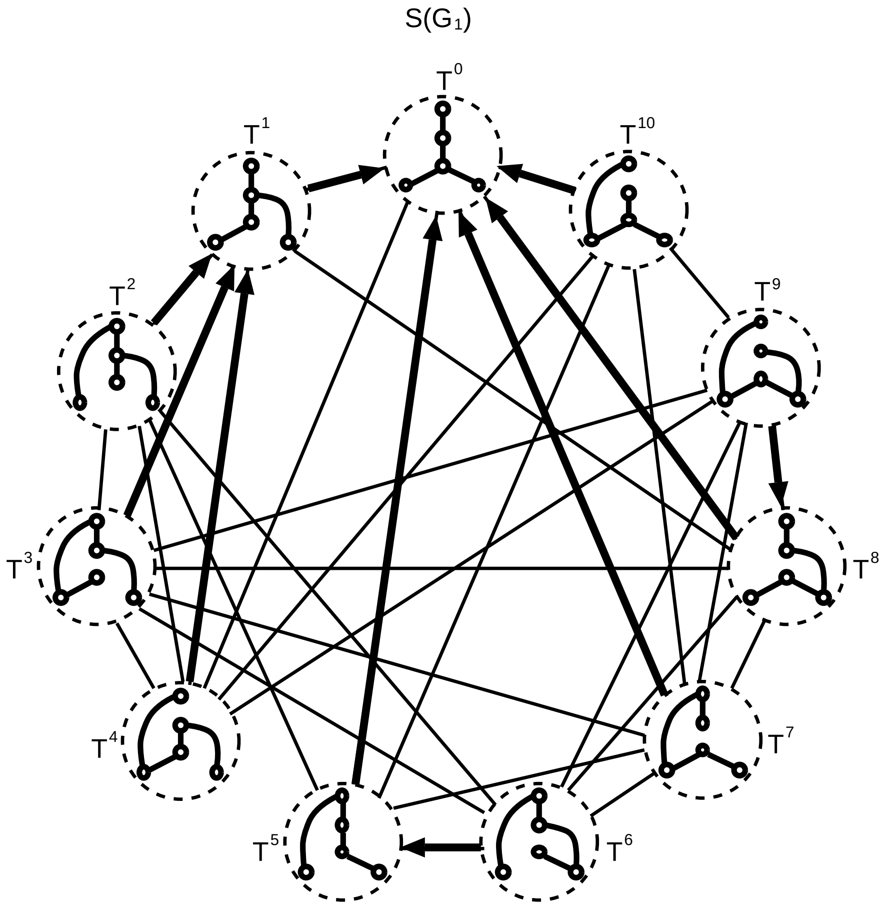

Figure 2: all child-parent relations in $\mathcal{S}(G_1)$

For graph G_1 of Figure 1, let $T^0 = \{e_1, e_2, e_3, e_4\}$ and let $T^c = \{e_1, e_4, e_5, e_6\}$. Then

$$
\begin{aligned}
&f = \mathrm{Min}\{e_2, e_4\} = e_2,\\
&C(T^c \cup f) = \{e_2, e_4, e_6\},\\
&C^*(T^0 \backslash f) = \{e_2, e_5, e_6\}.
\end{aligned}
$$

Therefore, $C(T^c \cup f) \cap C^*(T^0 \backslash f) \setminus f = \{e_6\}$.

Given a spanning tree $T^c \neq T^0$ and the edge $f = \mathrm{Min}(T^0 \setminus T^c)$, let g be the unique edge in $C(T^c \cup f) \cap C^*(T^0 \backslash f) \setminus f$. Clearly, $T^p = T^c \cup f \backslash g$ is a spanning tree. We call T^p a *parent* of T^c and T^c a *child* of T^p. For any spanning tree other than T^0, Lemma 3.1 guarantees the existence and uniqueness of its parent. Since $|T^p \cap T^0| = |T^c \cap T^0| + 1$ holds, T^0 is the 'progenitor' of all spanning trees. In Figure 2, arrows show such child-parent relations among the spanning trees of graph G_1. Each arrow points from a child to its parent.

Let $\mathcal{D}$ be a spanning tree of $\mathcal{S}(G)$ rooted at T^0, such that the edges of $\mathcal{D}$ consist of all child-parent pairs of spanning trees. Our algorithm implicitly traverses $\mathcal{D}$ from T^0 by recursively scanning all children of a current spanning tree. Thus we must find all children of a given spanning tree, if they exist. The next lemma gives a useful idea for this.

Lemma 3.2. *For any spanning tree T^p of G and for two arbitrary edges f and g, let $T^c = T^p \backslash f \cup g$. Under conditions* (1) *and* (2)*, T^c is a child of T^p if and only if the* *following conditions hold*

(3.1) $\qquad e_1 \le f < \mathrm{Min}(T^0 \setminus T^p)$ *and* $g \in C^*(T^p \backslash f) \cap C^*(T^0 \backslash f) \setminus f.$

Proof. We remark that the first condition of (3.1) guarantees $f \in T^p \cap T^0$ under condition (2), and that the second condition says $g \notin T^p \cup T^0$. Thus if (3.1) holds then T^c is a spanning tree different from T^0. On the other hand, this also holds if T^c is a child of T^p. The above equivalence is proved by the following implications:

$$\begin{aligned}
&T^c = T^p \backslash f \cup g \text{ is a child of } T^p \ , \\
&\qquad\qquad \Updownarrow \qquad \text{[from the definition]} \\
&f = \mathrm{Min}(T^0 \setminus T^c) \ \text{ and } \ g \in C(T^c \cup f) \cap C^*(T^0 \backslash f) \setminus f, \\
&\qquad\qquad \Updownarrow \qquad \text{[because } g \in C(T^c \cup f) \Leftrightarrow g \in C^*(T^p \backslash f)] \\
&f = \mathrm{Min}(T^0 \setminus T^c) \ \text{ and } \ g \in C^*(T^p \backslash f) \cap C^*(T^0 \backslash f) \setminus f \ , \\
&\qquad\qquad \Updownarrow \qquad \text{[because } T^c = T^p \backslash f \cup g] \\
&f = \mathrm{Min}(T^0 \setminus (T^p \backslash f \cup g)) \ \text{ and } \ g \in C^*(T^p \backslash f) \cap C^*(T^0 \backslash f) \setminus f \ , \\
&\qquad\qquad \Updownarrow \qquad \text{[because } f \in T^p \text{ and } g \notin T^0] \\
&e_1 \le f < \mathrm{Min}(T^0 \setminus T^p) \ \text{ and } \ g \in C^*(T^p \backslash f) \cap C^*(T^0 \backslash f) \setminus f.
\end{aligned}$$

∎

Let e_k be the largest edge less than $\mathrm{Min}(T^0 \setminus T^p)$. This lemma says that we can find all children of T^p if we know the edge-sets $C^*(T^p \backslash e_j) \cap C^*(T^0 \backslash e_j) \setminus e_j$ for $j = 1, 2, \cdots, k$. For example, consider the graph $G = G_1$ defined in Figure 1 and $T^p = T^1$. In this case, for all edges $\{e_1, e_2, e_3\}$ smaller than $\mathrm{Min}(T^0 \setminus T^1) = e_4$,

$$\begin{aligned}
C^*(T^1 \backslash e_1) \cap C^*(T^0 \backslash e_1) \setminus e_1 &= \{e_1, e_5\} \cap \{e_1, e_5\} \setminus e_1 = \{e_5\}, \\
C^*(T^1 \backslash e_2) \cap C^*(T^0 \backslash e_2) \setminus e_2 &= \{e_2, e_4, e_5\} \cap \{e_2, e_5, e_6\} \setminus e_2 = \{e_5\}, \\
C^*(T^1 \backslash e_3) \cap C^*(T^0 \backslash e_3) \setminus e_3 &= \{e_3, e_5\} \cap \{e_3, e_5\} \setminus e_3 = \{e_5\}.
\end{aligned}$$

Therefore, T^1 has three children $T^2 = T^1 \backslash e_1 \cup e_5$, $T^3 = T^1 \backslash e_2 \cup e_5$, and $T^4 = T^1 \backslash e_1 \cup e_5$.

By applying the characterization (3.1) of children, our algorithm scans all spanning trees. Procedure find-children( ) plays an important role in it. The procedure has two arguments T^p and k which represent a spanning tree in question and an edge e_k less than $\mathrm{Min}(T^0 \setminus T^p)$. It outputs all children T^c of T^p not containing e_k and recursively calls itself for two purposes: for outputting all children of T^c (i.e., all grandchildren of T^p not containing e_k) and for outputting all children of T^p containing e_k. For the first purpose, arguments are set as T^c and $k-1$ because if $k > 1$ then e_{k-1} is the largest edge less than $\mathrm{Min}(T^0 \setminus T^c)$. For the second purpose, the second argument decreases by 1. From the above explanation, by initially calling find-children( ) with arguments T^0 and $V-1$, one can output all spanning trees. For convenience, we shortly write $C^*(T^p \backslash e_j) \cap C^*(T^0 \backslash e_j) \setminus e_j$ as $Entr(T^p, e_j)$ on grounds that any edge in $C^*(T^p \backslash e_j) \cap C^*(T^0 \backslash e_j) \setminus e_j$ can be 'entered' into T^p in place of e_j. Our algorithm is formally described as below.

```
algorithm all-spanning-trees($G$) ;
  input: a graph $G$ with a vertex-set $\{v_1, \cdots, v_V\}$ and an edge-set $\{e_1, \cdots, e_E\}$ ;
begin
    by using a depth-first search, execute
        · find a depth-first spanning tree $T^0$ of $G$,
        · sort vertices and edges to satisfy assumptions (2), (3) and (4);
    output("$e_1, e_2, \cdots, e_{V-1}, tree,$") ;          {output $T^0$}
    find-children($T^0$,$V-1$) ;
end .

procedure find-children($T^p$,$k$) ;
  input: a spanning tree $T^p$ and an integer $k$ with $e_k < \mathrm{Min}(T^0 \setminus T^p)$ ;
begin
    if $k \le 0$ then return ;
    for each $g \in Entr(T^p, e_k)$ do begin {output all children of $T^p$ not containing $e_k$}
        $T^c := T^p \setminus e_k \cup g$ ;
        output("$-e_k, +g, tree,$") ;
        find-children($T^c$,$k-1$) ;           {find the children of $T^c$}
        output("$-g, +e_k,$") ;
    end ;
    find-children($T^p$,$k-1$) ;               {find the children of $T^p$ containing $e_k$}
end .
```

Theorem 3.3. *Algorithm* all-spanning-trees( ) *outputs each spanning tree exactly once.*
Proof. From Lemma 3.2, every spanning tree different from T^0 is output once for each time its parent is output. From Lemma 3.1, for any spanning tree T^c other than T^0, its parent always exists and is uniquely determined. Although the algorithm outputs the symmetric difference of T^0 and the last spanning tree at the end, T^0 is output only at the beginning. Since T^0 is the 'progenitor' of all spanning trees, the algorithm outputs each spanning tree exactly once. ∎

4 An efficient implementation

The performance of our algorithm is decided by the efficiency of the method of finding the edge-set $Entr(T^p, e_k)$. Since it can be found in $O(V+E)$ time, a naive implementation should require $O(E \cdot N + V + E)$ time, where N denotes the number of all spanning trees. $Entr(T^p, e_k)$, however, can be efficiently constructed by using the information from previous steps. By using this idea, we present an implementation whose time complexity is $O(N+V+E)$ and whose space complexity $O(VE)$ in this section. This implementation is optimal in the sense of time complexity.

The pair (T^p, k) of arguments of procedure find-children( ) expresses the state of our algorithm. At state (T^p, k), to output all children of T^p not containing e_k, only the entering edge-set $Entr(T^p, e_k)$ is required. After moving the current state to $(T^c, k-1)$ (or $(T^p, k-1)$,) the requirement of the entering edge-set $Entr(T^c, e_{k-1})$ (or $Entr(T^p, e_{k-1})$) occurs for the first time. Thus, when the current state moves to the next one, it is useless to update entering edge-sets $Entr(T^p, e_j)$ for $j = 1, \cdots, k-1$. Our implementation records and maintains sets $Can(e_j; T^p, k)$ defined below, instead of entering edge-sets $Entr(T^p, e_j)$. Let T^p be a spanning tree and k a positive integer with $e_k < \mathrm{Min}(T^0 \setminus T^p)$. For an edge e_j $(j = 1, \cdots, k)$, we define $Can(e_j; T^p, k)$ by:

$$Can(e_j; T^p, k) = Entr(T^p, e_j) \setminus \bigcup_{h=j+1}^{k} Entr(T^p, e_h). \tag{4.1}$$

Here we use this notation in the sense that $Can(e_j; T^p, k)$ is a set of 'candidates' of the entering edges $Entr(T^p, e_j)$ for a leaving edge e_j at state (T^p, k). We note that $Can(e_k; T^p, k) = Entr(T^p, e_k)$ from the definition (4.1). This says that it is enough to maintain $Can(*; *, *)$. The next two lemmas state the benefits of $Can(*; *, *)$.

Lemma 4.1. *Under conditions* (1), (2) *and* (3), *for* $j = 1, \cdots, V{-}1$,

$$Can(e_j; T^0, V{-}1) = \{e \mid e \notin T^0,\ \partial^- e = \partial^- e_j \ \textit{and}\ \partial^+ e \le \partial^+ e_j\}. \tag{4.2}$$

Proof. Since $Entr(T^0, e_j) = C^*(T^0 \backslash e_j) \setminus e_j$, $Can(e_j; T^0, V{-}1)$ can be written as:

$$Can(e_j; T^0, V{-}1) = \Big[C^*(T^0 \backslash e_j) \setminus e_j\Big] \setminus \bigcup_{h=j+1}^{V-1} \Big[C^*(T^0 \backslash e_h) \setminus e_h\Big].$$

Under conditions (1) and (3), an edge $e \notin T^0$ belongs to $C^*(T^0 \backslash e_j)$ if and only if $\partial^- e$ is a descendant of $\partial^- e_j$ and $\partial^+ e$ is an ancestor of $\partial^+ e_j$ relative to T^0. In addition, under condition (2), for $e \notin T^0$, e_j is the largest edge with $e \in C^*(T^0 \backslash e_j)$ if and only if $\partial^- e = \partial^- e_j$ and $\partial^+ e$ is an ancestor of $\partial^+ e_j$ relative to T^0, that is, if and only if $\partial^- e = \partial^- e_j$ and $\partial^+ e \le \partial^+ e_j$. Hence (4.2) holds. ∎

By using (4.2), one can construct sets $Can(e_j; T^0, V{-}1)$ for all $j = 1, \cdots, V{-}1$ in $O(V + E)$ time by using the depth-first search technique.

Lemma 4.2. *Let* T^p *be a spanning tree and* k *a positive integer with* $e_k < \mathrm{Min}(T^0 \setminus T^p)$. *Under conditions* (1), (2) *and* (3), *the following relation holds*

$$Can(e_j; T^p, k{-}1) = \begin{cases} Can(e_j; T^p, k) \cup [Can(e_k; T^p, k) \cap \{e | \partial^+ e {<} \partial^+ e_k\}] & \textit{if } \partial^- e_j = \partial^+ e_k, \\ Can(e_j; T^p, k) & \textit{if } \partial^- e_j \ne \partial^+ e_k, \end{cases} \tag{4.3}$$

and furthermore, for any edge $g \in Can(e_k; T^p, k)$ *and for a child* $T^c = T^p \backslash e_k \cup g$,

$$Can(e_j; T^c, k{-}1) = \begin{cases} Can(e_j; T^p, k) \cup [Can(e_k; T^p, k) \cap \{e | \partial^+ e {<} \partial^+ g\}] & \textit{if } \partial^- e_j = \partial^+ g, \\ Can(e_j; T^p, k) & \textit{if } \partial^- e_j \ne \partial^+ g. \end{cases} \tag{4.4}$$

Proof. Here, for two edges $e, f \in T^0$, we say that e is an ancestor of f or f is a descendant of e omitting the phrase 'relative to T^0,' for convenience. For an edge $g \notin T^0$, we also define its ancestors as ancestors of the edge $e \in T^0$ with $\partial^+ g = \partial^- e$ if e exists.

We show the first relation (4.3). From the definition (4.1),

$$Entr(T^p, e_j) = Can(e_j; T^p, k) \ \cup \bigcup_{h=j+1}^{k} (Entr(T^p, e_h) \cap Entr(T^p, e_j)).$$

This and (4.1) imply

$$\begin{aligned} Can(e_j; T^p, k{-}1) &= \left[Can(e_j; T^p, k) \ \cup \bigcup_{h=j+1}^{k} (Entr(T^p, e_h) \cap Entr(T^p, e_j))\right] \setminus \\ &\qquad \bigcup_{h=j+1}^{k-1} Entr(T^p, e_h) \\ &= Can(e_j; T^p, k) \ \cup \\ &\qquad \left[(Entr(T^p, e_k) \cap Entr(T^p, e_j)) \setminus \bigcup_{h=j+1}^{k-1} Entr(T^p, e_h)\right]. \end{aligned}$$

because $Can(e_j;T^p,k) \cap \bigcup_{h=j+1}^{k-1} Entr(T^p,e_h)=\emptyset$. Since T^0 is a depth-first search tree, if e_j is not an ancestor of e_k then $C^*(T^0\backslash e_j)\cap C^*(T^0\backslash e_k)=\emptyset$, that is, the set in the brackets of the above equation is empty. Thus $Can(e_j;T^p,k{-}1)=Can(e_j;T^p,k)$. Suppose that $\partial^- e_j=\partial^+ e_k$. Since e_h $(h=j{+}1,\cdots,k{-}1)$ is not an ancestor of e_k, from conditions (2) and (3), the set in the brackets equals to

$$Entr(T^p,e_k)\cap Entr(T^p,e_j)=Can(e_k;T^p,k)\cap\{e|\partial^+ e{<}\partial^+ e_k\}.$$

Suppose that e_j is an ancestor of e_k and $\partial^- e_j\neq\partial^+ e_k$. Then $Entr(T^p,e_k)\cap Entr(T^p,e_j)\subseteq Entr(T^p,e_t)$ where e_t is an ancestor of e_k with $\partial^- e_t=\partial^+ e_k$. Since $j<t<k$, in this case, $Can(e_j;T^p,k{-}1)=Can(e_j;T^p,k)$. This concludes the proof of (4.3).

Before proving the second relation (4.4), we show the following claim:

$$Entr(T^c,e_j)=\begin{cases}Entr(T^p,e_j) & \text{if } e_j\in A\\ Entr(T^p,e_j)\setminus Entr(T^p,e_k) & \text{if } e_j\notin A\end{cases}\tag{4.5}$$

for $j=1,\cdots,k{-}1$, where A denotes the set of ancestors of g. A vertex is a descendant of $\partial^- e_k$ relative to T^p if and only if it is a descendant of $\partial^- g$ relative to T^c. Then, if e_j is an ancestor of g then the entering edge-set is unchanged, i.e., $Entr(T^c,e_j)=Entr(T^p,e_j)$. If e_j is an ancestor of e_k but not of g then $Entr(T^c,e_j)\subseteq Entr(T^p,e_j)$. More precisely, any edge $e\in Entr(T^p,e_j)$ such that $\partial^- e$ is a descendant of $\partial^- e_k$ relative to T^p does not belong to $Entr(T^c,e_j)$, and the other edges obviously belong to $Entr(T^c,e_j)$. If e_j is an ancestor of neither e_k nor g, $Entr(T^c,e_j)=Entr(T^p,e_j)$ holds, however, in this case, $Entr(T^p,e_j)\cap Entr(T^p,e_k)=\emptyset$.

We finally prove (4.4). By combining the definition (4.1) with the relation (4.5), we obtain the equation:

$$\begin{aligned}Can(e_j;T^c,k{-}1) &= Entr(T^c,e_j)\setminus\\ &\quad\left[\bigcup_{h=j+1,\,e_h\notin A}^{k-1}(Entr(T^p,e_h)\setminus Entr(T^p,e_k))\cup\bigcup_{h=j+1,\,e_h\in A}^{k-1}Entr(T^p,e_h)\right]\\ &= Entr(T^c,e_j)\setminus\\ &\quad\left[\bigcup_{h=j+1}^{k}Entr(T^p,e_h)\setminus\left(Entr(T^p,e_k)\setminus\bigcup_{h=j+1,\,e_h\in A}^{k-1}Entr(T^p,e_h)\right)\right]\\ &= \left[Entr(T^c,e_j)\setminus\bigcup_{h=j+1}^{k}Entr(T^p,e_h)\right]\cup\\ &\quad\left[Entr(T^c,e_j)\cap\left(Entr(T^p,e_k)\setminus\bigcup_{h=j+1,\,e_h\in A}^{k-1}Entr(T^p,e_h)\right)\right]\\ &= \left[Entr(T^c,e_j)\setminus\bigcup_{h=j+1}^{k}Entr(T^p,e_h)\right]\cup\\ &\quad\left[(Entr(T^c,e_j)\cap Entr(T^p,e_k))\setminus\bigcup_{h=j+1,\,e_h\in A}^{k-1}Entr(T^p,e_h)\right].\end{aligned}\tag{4.6}$$

In either case $e_j\in A$ or $e_j\notin A$, the first term of the right-hand side of (4.6) is equal to $Can(e_j;T^p,k)$. If $e_j\notin A$, then the second term is empty from (4.5). Let us assume that

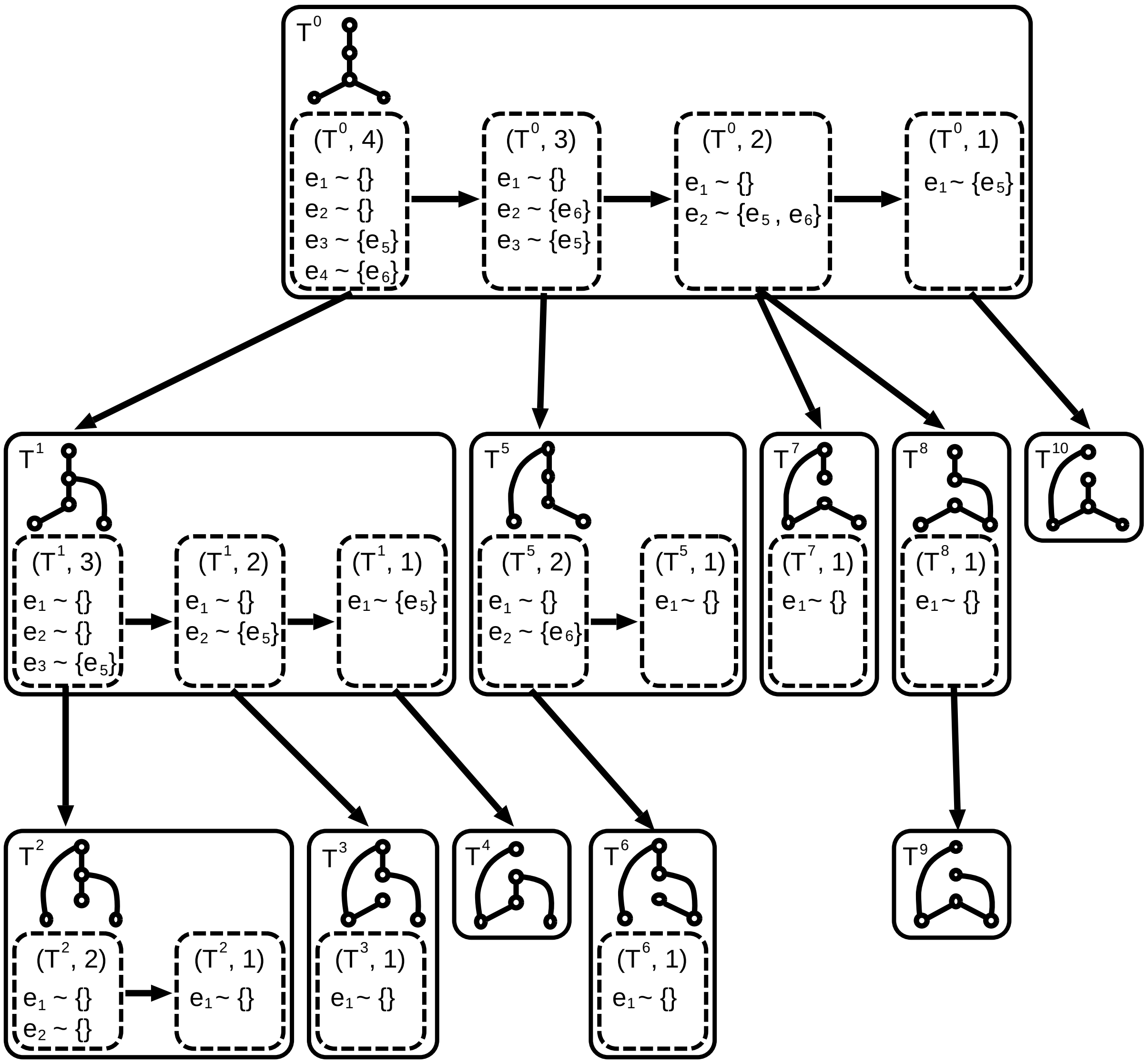

Figure 3: movement of the state and $Can(*;*,*)$

e_j is an ancestor of g. If $\partial^- e_j = \partial^+ g$ then the second term of (4.6) is equivalent to

$$Entr(T^p, e_j) \cap Entr(T^p, e_k) = Can(e_k; T^p, k) \cap \{e | \partial^+ e < \partial^- e_j\}.$$

If $\partial^- e_j \neq \partial^+ g$ then the second term is empty because $Entr(T^p, e_j) \cap Entr(T^p, e_k) \subseteq Entr(T^p, e_t)$ for the ancestor e_t of g with $\partial^- e_t = \partial^+ g$ and because $t > j$. We conclude the proof. ∎

Lemma 4.2 guarantees that at most one of sets $Can(*; T^p, k)$ is updated when state (T^p, k) moves. Figure 3 shows how the state and edge-sets $Can(*;*,*)$ change during the algorithm with input G_1 in Figure 1. For example, at the initial state $(T^0, 4)$,

$$\begin{aligned} Can(e_1; T^0, 4) &= \emptyset, \\ Can(e_2; T^0, 4) &= \emptyset, \\ Can(e_3; T^0, 4) &= \{e_5\}, \\ Can(e_4; T^0, 4) &= \{e_6\}. \end{aligned}$$

At the succeeding states $(T^1, 3)$ and $(T^0, 3)$, $Can(*;*,*)$ become

$$\begin{aligned} Can(e_1;T^1,3) &= \emptyset, \\ Can(e_2;T^1,3) &= \emptyset, \\ Can(e_3;T^1,3) &= \{e_5\}, \end{aligned}$$

and

$$\begin{aligned} Can(e_1;T^0,3) &= \emptyset, \\ Can(e_2;T^0,3) &= \{e_6\}, \\ Can(e_3;T^0,3) &= \{e_5\}. \end{aligned}$$

We note that relations (4.3) and (4.4) are identical except for the difference between e_k and g. Thus, one can use the same procedure for maintaining sets $Can(*;*,*)$ when state (T^p, k) moves to either $(T^c, k-1)$ or $(T^p, k-1)$. We note that none of sets $Can(*;T^p,k)$ is updated if and only if either $Can(e_k;T^p,k) = \emptyset$ or the tail of $\hat{g}$ is less than or equal to the tail of the smallest edge in $Can(e_k;T^p,k)$, where $\hat{g}$ is e_k or g.

Corollary 4.3. *Suppose that T is a spanning tree and that k is a positive integer with $e_k < \mathrm{Min}(T^0 \setminus T)$. For each e_j $(j \le k)$, let g_j denote either e_j or an arbitrary edge in $Can(e_j;T,k)$. Under conditions* (1), (2) *and* (3), *$T' = T \setminus \{e_1, \cdots, e_k\} \cup \{g_1, \cdots, g_k\}$ is a spanning tree.*

Proof. Let $T^j = T \setminus \{e_j, \cdots, e_k\} \cup \{g_j, \cdots, g_k\}$ for $j = 1, \cdots, k$. Obviously, T^k is a spanning tree. We suppose that T^j is a spanning tree. If $j \ge 2$, from Lemma 4.2, $Can(e_{j-1};T,j-1) \subseteq Can(e_{j-1};T^j,j-1)$. Thus, $T^{j-1} = T^j \setminus e_{j-1} \cup g_{j-1}$ is a spanning tree. ∎

Before formally writing our implementation, we briefly explain its outline. Procedure find-children( ) is implemented by splitting it into two procedures find-child( ) and sub-child( ) which mutually call each other. Roughly speaking, find-child( ) is a natural implementation of find-children( ) and sub-child( ) is a sub-procedure of find-child( ) for maintaining $Can(*;*,*)$. Procedure find-child( ) explicitly outputs all children of T^p not containing e_k, and calls sub-child( ) for two purposes: the first is to find all children of $T^c = T^p \setminus e_k \cup g$ for each $g \in Entr(T^p, e_k)$, and the second to find all children of T^p containing e_k. Procedure sub-child( ) maintains data according to Lemma 4.2, and calls find-child( ) for outputting all children of a current tree. Our implementation uses global variables `leave` and `candi` for representing a current state (T^p, k) and sets $Can(*;T^p,k)$. Variables `candi`(e_j) for $j = 1, \cdots, k$, represent $Can(e_j;T^p,k)$, and variable `leave` the set $\{e_j \mid j \le k$ and `candi`$(e_j) \ne \emptyset\}$. That is, the last entry of variable `leave` corresponds to k. We note that our implementation does not have a data structure which explicitly represents a current tree. Our algorithm is written as below.

```
algorithm all-spanning-trees($G$) ;
  input: a graph $G$ with a vertex-set $\{v_1, \cdots, v_V\}$ and an edge-set $\{e_1, \cdots, e_E\}$ ;
begin
ℓ0: by using a depth-first search, (simultaneously) execute
          · find a depth-first spanning tree $T^0$ of $G$,
          · sort vertices and edges to satisfy assumptions (2), (3), (4),
          · for each $e{\in}T^0$,  $\texttt{candi}(e) := Can(e; T^0, V{-}1)$,
          · $\texttt{leave} := \{e{\in}T^0 | \texttt{candi}(e) \neq \emptyset\}$,
     output(“$e_1, e_2, \cdots, e_{V-1}, tree,$”) ;                 {output $T^0$}
     find-child( ) ; {of $T^0$}
end .

procedure find-child( ) ;    {$T^p$:current spanning tree}
begin
     if  leave $= \emptyset$ return ;
     $Q := \emptyset$ ;
     $e_k$ := the last entry of leave;
     delete $e_k$ from leave;
     while  $\texttt{candi}(e_k) \neq \emptyset$  do begin
          $g$ := the last entry of $\texttt{candi}(e_k)$ ;
          delete $g$ from $\texttt{candi}(e_k)$, and add $g$ to the beginning of $Q$ ;
          output(“$-e_k, +g, tree,$”) ;                  {output $T^c := T^p \backslash e_k \cup g$}
          sub-child($e_k$,$g$) ;                          {find children of $T^c$}
          output(“$-g, +e_k,$”) ;                        {reconstruct $T^p := T^c \cup e_k \backslash g$}
     end ;
     move all entries of $Q$ to $\texttt{candi}(e_k)$ ;
     sub-child($e_k$,$e_k$) ;                             {find children of $T^p$ containing $e_k$}
     add $e_k$ to the end of leave;
end .

procedure sub-child($e_k$,$\hat{g}$) ;    {$T$:current spanning tree}
begin
     if  [$\texttt{candi}(e_k) = \emptyset$]  or  [$\partial^+\hat{g} \leq \partial^+$(the first entry of $\texttt{candi}(e_k)$)]  then  {Case 0.}
          find-child( ) ; {of $T$}
          return ;
     endif ;
     $f$ := the edge in $T^0$ with $\partial^- f = \partial^+\hat{g}$ ;
     if  $\texttt{candi}(f) \neq \emptyset$  then                                   {Case 1.}
ℓ1:       $S := \{e{\in}\texttt{candi}(e_k) | \partial^+ e < \partial^+\hat{g}\}$ ;
ℓ2:       merge $S$ into $\texttt{candi}(f)$ ;
          find-child( ) ; {of $T$}
ℓ3:       delete all entries of $S$ from $\texttt{candi}(f)$ ;
          $S := \emptyset$ ;
     else {$\texttt{candi}(f) = \emptyset$}                                       {Case 2.}
ℓ4:       $\texttt{candi}(f) := \{e{\in}\texttt{candi}(e_k) | \partial^+ e < \partial^+\hat{g}\}$ ;
ℓ5:       insert $f$ to leave;
          find-child( ) ; {of $T$}
          delete $f$ from leave;
          $\texttt{candi}(f) := \emptyset$ ;
     endif ;
end .
```

Finally, we analyze the time and space complexities of the above implementation. The

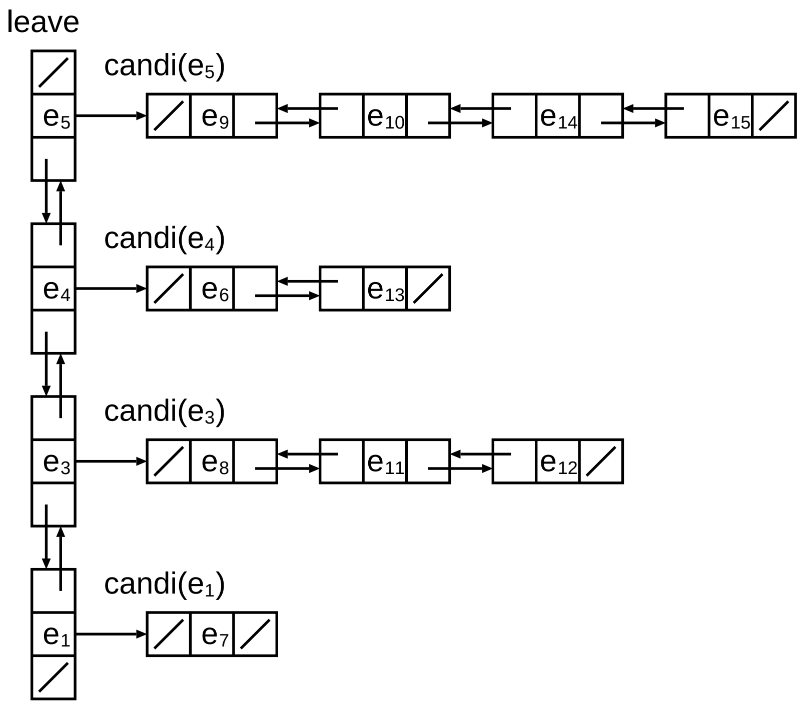

Figure 4: data structures for `candi`$(*)$ and `leave`

complexities are dependent on data structures. We use a data structure for a given graph G such that two incidence vertices of any edge are found in constant time, and one for the initial spanning tree T^0 such that for any vertex v other than root, the unique edge e with $\partial^- e = v$ is found in constant time. For representing a subset of the edges, we use an ascending ordered list L realized by a double linked list. Then the following operations are executable in constant time:

· check whether L is empty,
· find the first (or last) entry of L if $L \neq \emptyset$,
· delete an entry from L,
· add an element which is smaller than all entries of L to the beginning of L,
· add an element which is greater than all entries of L to the end of L,
· clear L i.e., $L := \emptyset$,
· move all entries of L to an empty list.

For example, we use data structures like the ones in Figure 4 for representing `candi`$(*)$ and `leave`.

By using such a data structure, one can execute in constant time all lines except the six lines $\ell 0$, $\ell 1$, $\ell 2$, $\ell 3$, $\ell 4$ and $\ell 5$. The line $\ell 0$ initially requires $O(V+E)$ time. Hence if the total time of five other lines is proportional to the number N of all spanning trees, the time complexity of our implementation is $O(N+V+E)$. Since `leave` is an ordered list, the line $\ell 5$ is completed in $O(\,|\{e \in \texttt{leave} | e < f\}|\,)$ time. We recall that graph G satisfies

(4) for two edges $e, f \notin T^0$, $e < f$ only if $\partial^+ e \leq \partial^+ f$.

Under this condition, $\ell 1$ and $\ell 4$ require $O(\,|\{e \in \texttt{candi}(e_k) | \partial^+ e < \partial^+ \hat{g}\}|\,)$ time, and $\ell 2$ and $\ell 3$ $O(\,|\{e \in \texttt{candi}(e_k) | \partial^+ e < \partial^+ \hat{g}\}| + |\texttt{candi}(f)|\,)$ time. Procedure sub-child( ) deals with three cases. If none of $Can(*;*,*)$ is updated (this is recognized in constant time,) the procedure just calls find-child( ). From the above consideration, the two other cases respectively require:

Case 1 ($\texttt{candi}(f) \neq \emptyset$): $O(\,|\{e{\in}\texttt{candi}(e_k) | \partial^+ e{<}\partial^+\hat{g}\}| + |\texttt{candi}(f)|\,)$ time,
Case 2 ($\texttt{candi}(f) = \emptyset$): $O(\,|\{e{\in}\texttt{candi}(e_k) | \partial^+ e{<}\partial^+\hat{g}\}| + |\{e{\in}\texttt{leave} | e{<}f\}|\,)$ time.

We split these two cases into the next four:

Case A.1: $e_k \notin T$ (i.e., $T = T^c$) and $\texttt{candi}(f) \neq \emptyset$,
Case A.2: $e_k \notin T$ (i.e., $T = T^c$) and $\texttt{candi}(f) = \emptyset$,
Case B.1: $e_k \in T$ (i.e., $T = T^p$) and $\texttt{candi}(f) \neq \emptyset$,
Case B.2: $e_k \in T$ (i.e., $T = T^p$) and $\texttt{candi}(f) = \emptyset$.

In Case A.1, $|\{e{\in}\texttt{candi}(e_k) | \partial^+ e{<}\partial^+\hat{g}\}| + |\texttt{candi}(f)|$ is bounded by the number of children of T^c not containing f. In Case A.2, $|\{e{\in}\texttt{candi}(e_k) | \partial^+ e{<}\partial^+\hat{g}\}|$ is bounded by the number of children of T^c not containing f. Moreover, for each $e{\in}\texttt{leave}$ with $e{<}f$, there is a child of T^c not containing e. Thus, if $e_k \notin T$ (in Case A) then the time complexity of sub-child( ) other than calling find-child( ) is O(the number of children of T). In Case B.1, from Corollary 4.3, T has at least $(\,|\{e{\in}\texttt{candi}(e_k) | \partial^+ e{<}\partial^+\hat{g}\}| \times |\texttt{candi}(f)|\,)$ grandchildren which contain neither e_k nor f. Thus, in this case, the required time of sub-child( ) is bounded by the number of grandchildren of T not containing e_k. Similarly, in Case B.2, if $\{e{\in}\texttt{leave} | e{<}f\} \neq \emptyset$ then sub-child( ) requires (at most) time proportional to the number of grandchildren of T not containing e_k; otherwise it requires (at most) time proportional to the number of children of T not containing e_k. From the above discussion, the total time complexity of our implementation is $O(N{+}V{+}E)$.

For each state (T^p, k) of find-child( ) and sub-child( ), sets $\texttt{candi}(e_j)$ $(j = 1, \cdots, k-1)$ have no intersection with each other. However, these may have a nonempty intersection with $\texttt{candi}(e_h)$ $(h = k, \cdots, V{-}1)$). Thus, we need $O(VE)$ space for $\texttt{candi}$. Obviously, the cardinality of $\texttt{leave}$ is at most V. Since the size of local variable S is at most E and the depth of recursive call is at most $V{-}1$, the total space for local variables is $O(VE)$. Hence, the space complexity of our implementation is $O(VE)$.

Theorem 4.4. *The time and space complexities of our implementation are $O(N{+}V{+}E)$ and $O(VE)$, respectively.*

In this paper, we have proposed an efficient algorithm for scanning all spanning trees. This is optimal in sense of time complexity. The remaining question is whether the space complexity can be reduced to $O(V{+}E)$.

acknowledgments

We would like to thank the referees and Dr. Yoshiko T. Ikebe for their valuable comments which improved the presentation of this paper.

References

[1] Avis, D. and Fukuda, K.: A Basis Enumeration Algorithm for Linear Systems with Geometric Applications, *Applied Mathematics Letters*, Vol.4 (1991), 39–42.

[2] Avis, D. and Fukuda, K.: A Pivoting Algorithm for Convex Hulls and Vertex Enumeration of Arrangements and Polyhedra, *Discrete and Computational Geometry*, Vol.8 (1992), 295–313.

[3] Avis, D. and Fukuda, K.: Reverse Search for Enumeration, to appear in *Discrete Applied Mathematics*.

[4] Gabow, H. N. and Myers, E. W.: Finding All Spanning Trees of Directed and Undirected Graphs, *SIAM Journal on Computing*, Vol.7 (1978), 280–287.

[5] Kapoor, S. and Ramesh, H.: Algorithms for Enumerating All Spanning Trees of Undirected and Weighted Graphs. *SIAM Journal on Computing*, Vol.24 (1995), 247–265.

[6] Matsui, T.: An Algorithm for Finding All the Spanning Trees in Undirected Graphs, Technical Report METR 93-08, Dept. of Mathematical Engineering and Information Physics, University of Tokyo, Tokyo, 1993.

[7] Minty, G. J.: A Simple Algorithm for Listing All the Trees of a Graph, *IEEE Transactions on Circuit Theory*, Vol. CT–12 (1965), 120.

[8] Read, R. C. and Tarjan, R. E.: Bounds on Backtrack Algorithms for Listing Cycles, Paths, and Spanning Trees, *Networks*, Vol.5 (1975), 237–252.

[9] Tarjan, R.: Depth-First Search and Linear Graph Algorithms, *SIAM Journal on Computing*, Vol.1 (1972) 146–160.

Akihisa TAMURA:
Department of Computer Science
and Information Mathematics,
The University of Electro-Communications,
Chofu, Tokyo 182, Japan.
e-mail: `tamura@im.uec.ac.jp`