An Optimal Enumeration of Spanning Trees in an Undirected Graph

Akiyoshi SHIOURA [*]   Akihisa TAMURA [†]   and   Takeaki UNO [‡]

(July, 1994)

Abstract: Let G be an undirected graph with V vertices and E edges. Many algorithms have been developed for enumerating all spanning trees in G. Most of the early algorithms use a technique called 'backtracking'. Recently, several algorithms using a different technique have been proposed by Kapoor and Ramesh (1992), Matsui (1993), and Shioura and Tamura (1993). They find a new spanning tree by exchanging one edge of a current one. This technique has the merit of enabling us to compress the whole output of all spanning trees by outputting only relative changes of edges. Kapoor and Ramesh first proposed an $O(N+V+E)$ time algorithm by adopting such 'compact' output, where N is the number of spanning trees. Another algorithm with the same time complexity was constructed by Shioura and Tamura. These are optimal in the sense of time complexity, but not in terms of space complexity, because they take $O(VE)$ space. We refine Shioura and Tamura's algorithm, and decrease the space complexity from $O(VE)$ to $O(V+E)$ while preserving time complexity. Therefore, our algorithm is optimal in the sense of both time and space complexities.

Keywords: Optimal Algorithm, Spanning Trees, Undirected Graphs.

1 Introduction.

Let G be an undirected graph with V vertices and E edges. A spanning tree of G is defined as a connected subgraph of G which contains all vertices, but no cycle. In this paper we consider the enumeration of all spanning trees in an undirected graph. Many algorithms for solving this problem have been developed, e.g. [7, 8, 4, 5, 6, 9], and these may be divided into several types.

The first type [7, 8, 4], to which belong many of the early algorithms use a technique called 'backtracking'. This is a useful technique for listing the kinds of subgraphs, e.g. cycles, paths, and so on. Gabow and Myers [4] refined Minty's algorithm [7] and Read and Tarjan's [8]. Their algorithm uses $O(NV+V+E)$ time and $O(V+E)$ space, where N is the number of all spanning trees. If we enumerate all spanning trees by outputting all edges of each spanning tree, their algorithm is optimal in terms of time and space complexities.

Recently, several algorithms [5, 6, 9] which use another technique have been developed. These algorithms find a new spanning tree by exchanging one pair of edges, instead of

[*]Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, 2-12-1 Oh-okayama, Meguro-ku, Tokyo 152, Japan. `shioura@is.titech.ac.jp`

[†]Department of Computer Science and Information Mathematics, The University of Electro-Communications, 1-5-1 Chofugaoka, Chofu-shi, Tokyo 182, Japan. `tamura@im.uec.ac.jp`

[‡]Department of Systems Science, Tokyo Institute of Technology, 2-12-1 Oh-okayama, Meguro-ku, Tokyo 152, Japan. `uno@is.titech.ac.jp`

backtracking. Furthermore, if we enumerate all spanning trees by outputting only relative changes of edges between spanning trees, we can compress the size of output to $\Theta(N+V)$, and hence, total time complexity may be reduced. In fact, Kapoor and Ramesh [5] proposed an $O(N+V+E)$ time and $O(VE)$ space algorithm by adopting such a 'compact' output, which is optimal in the sense of time complexity. On the other hand, Matsui [6] developed an $O(NV+V+E)$ time and $O(V+E)$ space algorithm for enumerating all spanning trees explicitly, by applying the reverse search scheme [3]. Reverse search is a scheme for general enumeration problems (see [1], [2]). Shioura and Tamura [9] also developed an algorithm generating a compact output with the same time and space complexities as the Kapoor-Ramesh algorithm, by using the reverse search technique. The Kapoor-Ramesh algorithm, and the Shioura-Tamura algorithm, however, are not efficient in terms of space complexity, because they take $O(VE)$ space.

The main aim of this paper is to obtain an algorithm which generates a compact output, and is optimal in the sense of both time and space complexities, by refining the Shioura-Tamura algorithm. When the process goes to a lower level node of the computation tree of the original algorithm, some edge set can be efficiently divided without requiring extra information. However, in order to efficiently restore such an edge set when the process goes back to the higher level node, the algorithm requires extra $O(E)$ information. Since the depth of the computation tree is $V-1$, it takes $O(VE)$ space. We propose a useful property for efficiently restoring the edge set and a technique for restoring it which uses extra $O(V)$ space in all, while time complexity remains $O(N+V+E)$.

In Section 2, we explain the technique for enumeration of spanning trees and compact outputs. In Section 3, we define a nice child-parent relation between spanning trees and propose a naive algorithm. In Section 4, we show some properties which are useful for efficient manipulation of data structures in our implementation. Our implementation is presented in Section 5, and the time and space complexities are analyzed.

2 Compact output.

Let G be an undirected graph (not necessary simple) with V vertices $\{v_1, \cdots, v_V\}$ and E edges $\{e_1, \cdots, e_E\}$. We define two types of edge-sets which are necessary for our algorithm, so-called fundamental cuts and fundamental cycles. Let T be a spanning tree of G. Throughout this paper, we represent a spanning tree by its edge-set of size $V-1$. For any edge $f \in T$, deletion of f from T yields two connected components. The *fundamental cut* associated with T and f is defined as the set of edges connecting these components, and is denoted by $Cut(T \backslash f)$. Likewise, we define the *fundamental cycle* associated with T and $g \notin T$, as the set of edges contained in the unique cycle of $T \cup g$. We will denote it as $Cyc(T \cup g)$. From definition, $T \backslash f \cup g$ is a spanning tree for any $f \in T$ and any $g \in Cut(T \backslash f)$, Similarly, for any $g \notin T$ and any $f \in Cyc(T \cup g)$, $T \cup g \backslash f$ is also a spanning tree. These properties are useful for enumerating spanning trees, because by using fundamental cuts or cycles we can construct a different spanning tree from a given one by exchanging exactly one edge.

Given a graph G, let $\mathcal{S}(G)=(\mathcal{T}, \mathcal{A})$ be the graph 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 cycle. For example, the graph $\mathcal{S}(G_1)$ of the left one G_1 is shown in Figure 1.

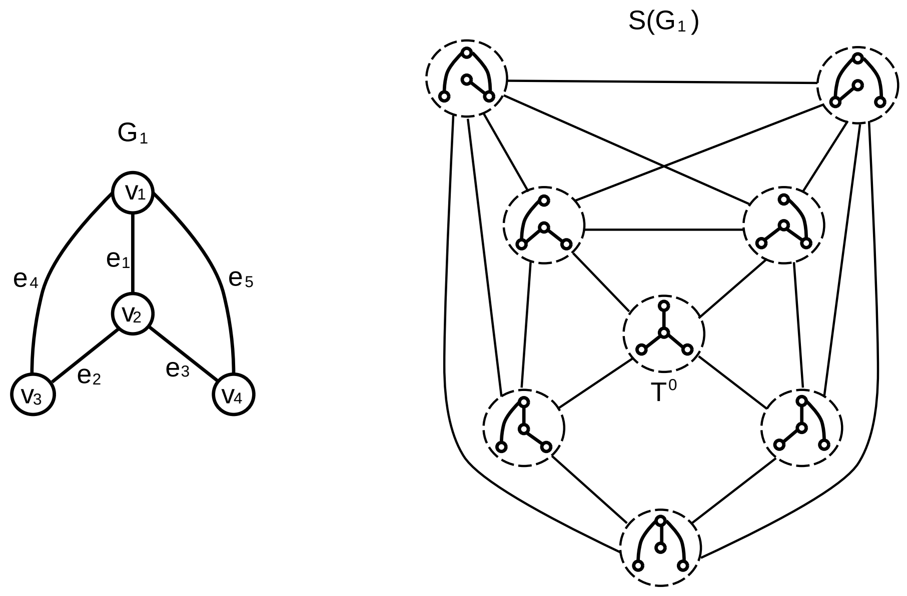

Figure 1: graph G_1 and graph $\mathcal{S}(G_1)$

Our algorithm finds all spanning trees of G by implicitly traversing some spanning tree $\mathcal{D}$ of $\mathcal{S}(G)$. In order to output all $(V-1)$ edges of each spanning tree, $\Theta(|\mathcal{T}|\cdot V) = \Theta(N\cdot V)$ time is required. However, if we output all edges of the first spanning tree, and then only the sequence of exchanged edge-pairs of G obtained by traversing $\mathcal{D}$, we need only $\Theta(|\mathcal{T}| + V) = \Theta(N+V)$ time, 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.

3 Basic ideas and naive algorithm.

In this section we explain the basic ideas and the naive algorithm.

We define the total orders over the vertex-set $\{v_1, \cdots, v_V\}$ and the edge-set $\{e_1, \cdots, e_E\}$ of G by their indices as $v_1 < v_2 < \cdots < v_V$ and $e_1 < e_2 < \cdots < e_E$. Especially, we call the smallest vertex v_1 the *root*. For each edge e, we call the smaller incident vertex the *tail*, denoted by $\partial^+ e$ and call the larger one the *head*, denoted by $\partial^- e$. 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.

In our algorithm, we make several assumptions for the vertex-set and the edge-set of G.

Assumption (1). *T^0 is a depth-first spanning tree of G.*

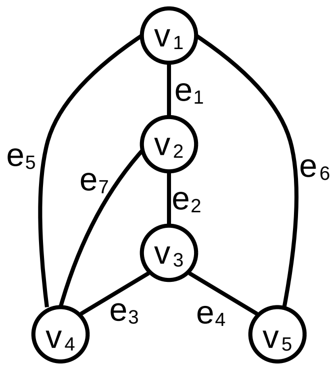

Figure 2: graph G_2

Assumption (2). $T^0 = \{e_1, \cdots, e_{V-1}\}$.

Assumption (3). *Any edge in T^0 is smaller than its proper descendants.*

Assumption (4). *Each vertex v is smaller than its proper descendants relative to T^0.*

Assumption (5). *For any two edges $e, f \notin T^0$, if $e < f$ then $\partial^+ e \leq \partial^+ f$.*

Vertices and edges of graph G_2 in Figure 2 satisfy these assumptions. In fact, one can find T^0 and sort vertices and edges of G in $O(V+E)$ time so that G satisfies the above assumptions by applying Tarjan's depth-first search [10]. We note that assumptions (1), (2), and (3) are sufficient for the correctness of our algorithm. We, however, need further assumptions (4) and (5) for an efficient implementation.

For any nonempty subset S of $\{e_1, \cdots, e_E\}$, $\mathrm{Min}(S)$ denotes the smallest edge in S. For convenience, we assume that $\mathrm{Min}(\emptyset) = e_V$.

Lemma 3.1. [9] *Under assumptions* (1) *and* (3)*, for any spanning tree $T^c \neq T^0$, if $f = \mathrm{Min}(T^0 \setminus T^c)$ then $Cyc(T^c \cup f) \cap Cut(T^0 \setminus f) \setminus f$ contains exactly one edge.*

Proof. The set $T^0 \setminus f$ has exactly two components, one containing $\partial^- f$ and the other $\partial^+ f$. Therefore the unique path $Cyc(T^c \cup f) \setminus f$ from $\partial^- f$ to $\partial^+ f$ in T^c, contains at least one edge in $Cut(T^0 \setminus f)$. Hence $Cyc(T^c \cup f) \cap Cut(T^0 \setminus f) \setminus f \neq \emptyset$.

Since T^0 is a depth-first spanning tree, we may assume the head of any edge is a descendant of its tail relative to T^0, without loss of generality. Let e be the first edge from $\partial^- f$ on the path such that $e \in Cut(T^0 \setminus f)$. Then the head $\partial^- e$ is a descendant of $\partial^- f$ relative to T^0, and the tail $\partial^+ e$ is an ancestor of $\partial^+ f$. From assumption (3) and the minimality of f, $\partial^+ e$ and $\partial^+ f$ are connected in $T^c \cap T^0$. Thus, there is no edge contained in $Cut(T^0 \setminus f)$ between $\partial^+ e$ and $\partial^+ f$ in the path $Cyc(T^c \cup f) \setminus f$. Hence e is the only edge in $Cyc(T^c \cup f) \setminus f$ and $Cut(T^0 \setminus f)$. ∎

Consider the graph G_2 of Figure 2. Here let $T^0 = \{e_1, e_2, e_3, e_4\}$ and $T^c = \{e_4, e_5, e_6, e_7\}$. In graph G_2,

$$f = \mathrm{Min}\{e_1, e_2, e_3\} = e_1,$$
$$Cyc(T^c \cup f) = \{e_1, e_5, e_7\}$$
$$Cut(T^0 \setminus f) = \{e_1, e_5, e_6\}$$

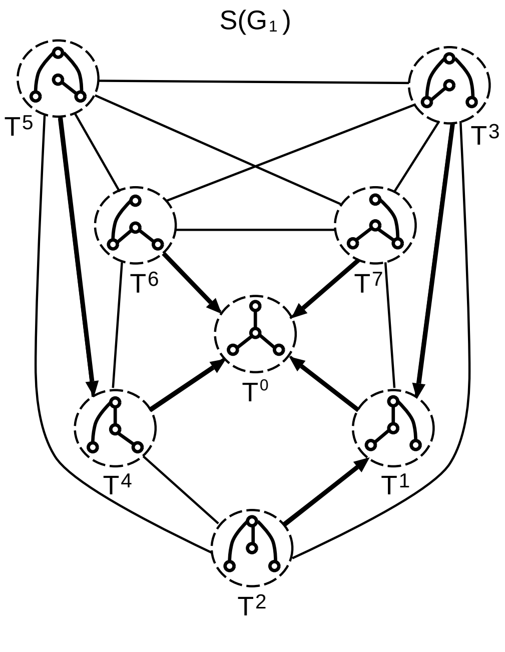

Figure 3: child-parent relations in $\mathcal{S}(G_1)$

Therefore, $Cyc(T^c \cup f) \cap Cut(T^0 \backslash f) \setminus f = \{e_5\}$.

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 $Cyc(T^c \cup f) \cap Cut(T^0 \backslash f) \setminus f$. Clearly, $T^p = T^c \cup f \backslash g$ is a spanning tree. We call T^p the *parent* of T^c and T^c a *child* of T^p. Lemma 3.1 guarantees that each spanning tree other than T^0 has a unique parent. Since $|T^p \cap T^0| = |T^c \cap T^0| + 1$ holds, T^0 is the ancestor of all spanning trees. For the graph G_1 in Figure 1, all child-parent pairs are shown by the arrows in Figure 3. Each arrow goes from a child to its parent. We can see that all arrows construct a spanning tree of $\mathcal{S}(G_1)$ rooted at T^0.

Let $\mathcal{D}$ be the spanning tree of $\mathcal{S}(G)$ consisting 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. [9] *Let T^p be an arbitrary spanning tree of G, and let f, g be two distinct edges. Under assumptions* (1), (2), *and* (3), *$T^c = T^p \backslash f \cup g$ is a child of T^p if and only if f and g satisfy the following conditions:*

$$f < \mathrm{Min}(T^0 \setminus T^p) \quad \textit{and} \quad g \in Cut(T^p \backslash f) \cap Cut(T^0 \backslash f) \setminus f. \tag{3.1}$$

Proof. Under assumptions (1) and (3), T^c is a child of T^p if and only if the following conditions hold:

$$T^c \text{ is a spanning tree different from } T^0, \tag{3.2}$$

$$f' = \mathrm{Min}(T^0 \setminus T^c) \text{ and } g' \in Cyc(T^c \cup f') \cap Cut(T^0 \backslash f') \setminus f', \tag{3.3}$$

$$T^p = T^c \cup f' \backslash g'. \tag{3.4}$$

We first show that $f = f'$ and $g = g'$. From (3.2), (3.3) and (3.4), T^c and T^p are different spanning trees. Assume on the contrary that $f \notin T^p$, then $T^p \setminus f = T^p$. Since T^c is a spanning tree and $f \neq g$, we have $g \in T^p$ and $T^c = T^p \backslash f \cup g = T^p$, which is a contradiction. Thus, $f \in T^p$ and $g \notin T^p$. From (3.4), $T^p = \{T^p \backslash f \cup g\} \cup f' \backslash g'$, and hence $f = f'$ and $g = g'$ must hold.

Conditions (3.2), (3.3), and (3.4) imply

$$f \in T^p \cap T^0 \text{ and } g \notin T^p \cup T^0. \tag{3.5}$$

On the other hand, under assumption (2), (3.1) implies (3.5). Moreover, (3.1) and (3.5) imply (3.2) and (3.4). All we have to do is to show that (3.1) and (3.3) are equivalent under conditions (3.2), (3.4), and (3.5).

From definition of T^c and (3.5), $T^0 \setminus T^c = T^0 \setminus (T^p \backslash f \cup g) = (T^0 \setminus T^p) \cup \{f\}$. Hence $\mathrm{Min}(T^0 \setminus T^c) = \mathrm{Min}(\mathrm{Min}(T^0 \setminus T^p) \cup \{f\})$. This implies that $f = \mathrm{Min}(T^0 \setminus T^c)$ if and only if $f < \mathrm{Min}(T^0 \setminus T^p)$. Since T^p and $T^c = T^p \backslash f \cup g$ are distinct, $g \in Cyc(T^c \cup f)$ is equivalent to $g \in Cut(T^p \backslash f)$. Therefore, the second condition of (3.1) is equivalent to the second condition of (3.3). ∎

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

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

Therefore, T^1 has only the two children $T^1 \backslash e_2 \cup e_4$ and $T^1 \backslash e_1 \cup e_4$.

In the rest of paper we shortly write $Cut(T^p \backslash e_j) \cap Cut(T^0 \backslash e_j) \setminus e_j$ as $Entr(T^p, e_j)$ on grounds that any edge in $Cut(T^p \backslash e_j) \cap Cut(T^0 \backslash e_j) \setminus e_j$ can be 'entered' into T^p in place of e_j. From the above consideration, we can construct the algorithm 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,
        · find a depth-first spanning tree $T^0$ of $G$,
        · sort vertices and edges to satisfy assumptions (2), (3), (4), and (5);
    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  $k \leq 0$  then return ;
    for  each $g \in Entr(T^p, e_k)$  do begin  {output all children of $T^p$ not containing $e_k$}
        $T^c := T^p \backslash e_k \cup g$ ;
```

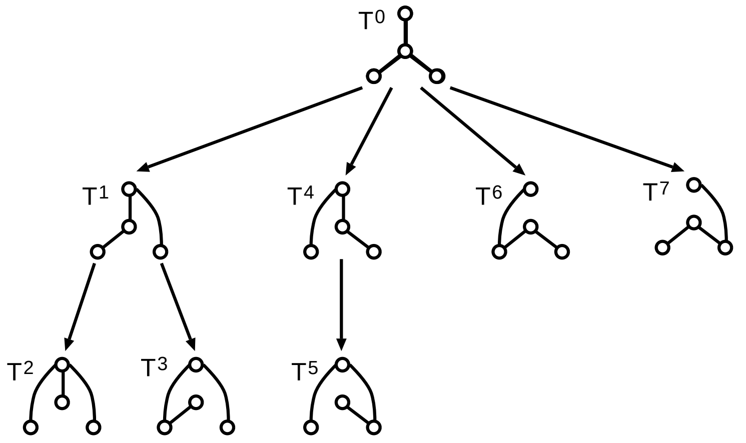

Figure 4: enumeration tree of spanning trees in G_1

```
        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 $T^p$ not containing $e_{k-1}$}
end .
```

In this algorithm, procedure find-children( ) finds all children of each spanning tree. When it is called with two arguments T^p and k, it finds all children of T^p not containing an edge e_k. Whenever it finds such a child T^c, it recursively calls itself again for finding all children of T^c. In this stage, arguments are set to T^c and $k-1$, because if $k > 1$ then e_{k-1} becomes the largest edge less than $\mathrm{Min}(T^0 \setminus T^c)$. If all children of T^p not containing e_k have been found, it recursively calls itself again for finding all children of T^p not containing e_{k-1}. In this case arguments are T^p and $k-1$. Initially, algorithm all-spanning-trees(G) calls find-children( ) with arguments T^0 and $V-1$, and all spanning trees of G are found. Figure 4 shows the enumeration tree of spanning trees in graph G_1.

Theorem 3.3. [9] *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. Since T^0 is the ancestor of all spanning trees, the algorithm outputs each spanning tree exactly once. ∎

4 Manipulating data structures.

In our algorithm, we define each state when we find all children of T^p not containing e_k by a pair (T^p, k). When we call procedure find-children(T^p, k), the current state becomes (T^p, k), and if we find a child T^c of T^p not containing e_k, the state moves to $(T^c, k-1)$. After all children of T^p not containing e_k have been found, the state moves to $(T^p, k-1)$. At the state (T^p, k), the entering edge-set $Entr(T^p, e_k)$ is required to output all children

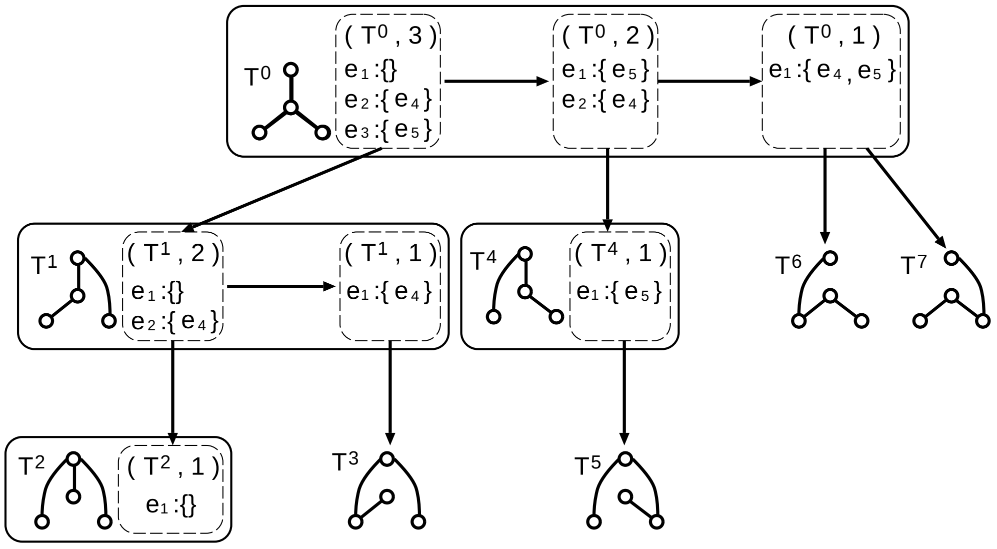

Figure 5: movement of the state and $Can(*;*,*)$

of T^p not containing e_k. After the state moves to $(T^c, k{-}1)$ (or $(T^p, k{-}1)$), the necessity of the entering edge-set $Entr(T^c, e_{k-1})$ (or $Entr(T^p, e_{k-1})$) occurs for the first time. The key point is finding an entering edge-set $Entr(T^c, e_{k-1})$ (or $Entr(T^p, e_{k-1})$) efficiently. For constructing an entering edge-set efficiently, our implementation maintains edge-sets $Can(e_j; T^p, k)$ for $j = 1, \cdots, k$ defined below. Let T^p be a spanning tree and k be a positive integer with $e_k < \mathrm{Min}(T^0 \setminus T^p)$. For each 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 the state (T^p, k). We can find $Entr(T^p, e_k)$ very easily by maintaining $Can(e_j; T^p, k)$ for $j = 1, \cdots, k$, because $Can(e_k; T^p, k) = Entr(T^p, e_k)$ from definition (4.1). When we find a child T^c of T^p, we update $Can(e_j; T^p, k)$ for $j = 1, \cdots, k$ to $Can(e_j; T^c, k{-}1)$ for $j = 1, \cdots, k{-}1$. On the other hand, after we have found all children of T^p not containing e_{k-1}, we construct $Can(e_j; T^p, k{-}1)$ for $j = 1, \cdots, k{-}1$ from $Can(e_j; T^p, k)$ for $j = 1, \cdots, k$. Efficiency of our implementation depends on how to maintain $Can(*;*,*)$ efficiently.

Figure 5 shows states and edge-sets $Can(*;*,*)$ during enumerating all spanning trees of G_1 in Figure 1. For example, at the initial state $(T^0, 3)$,

$$\begin{aligned} Can(e_1; T^0, 3) &= \emptyset, \\ Can(e_2; T^0, 3) &= \{e_4\}, \\ Can(e_3; T^0, 3) &= \{e_5\}. \end{aligned}$$

At the succeeding states $(T^1, 2)$ and $(T^0, 2)$,

$$\begin{aligned} Can(e_1; T^1, 2) &= \emptyset, \\ Can(e_2; T^1, 2) &= \{e_4\}, \end{aligned}$$

and

$$\begin{aligned} Can(e_1;T^0,2) &= \{e_5\}, \\ Can(e_2;T^0,2) &= \{e_4\}. \end{aligned}$$

Here we consider how to maintain such edge-sets. First we show that the initial edge-sets $Can(e_j;T^0,V-1)$ for $j=1,\cdots,V-1$ can be found easily.

Lemma 4.1. [9] *Under assumptions* (1), (2), (3), *and* (4),

$$Can(e_j;T^0,V-1) = \{e \mid e \notin T^0,\ \partial^+e \le \partial^+e_j \text{ and } \partial^-e = \partial^-e_j\} \qquad (j=1,\cdots,V-1). \quad (4.2)$$

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

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

Under assumptions (1) and (4), an edge $e \notin T^0$ belongs to $Cut(T^0\backslash e_j)$ if and only if ∂^-e is a descendant of ∂^-e_j and ∂^+e is an ancestor of ∂^+e_j relative to T^0. In addition, under assumption (3), for $e \notin T^0$, e_j is the largest edge with $e \in Cut(T^0\backslash e_j)$ if and only if $\partial^-e = \partial^-e_j$ and $\partial^+e \le \partial^+e_j$. ∎

From the lemma, we can find $Can(e_j;T^0,V-1)$ for $j=1,\cdots,V-1$ in $O(V+E)$ time by applying a depth-first search.

Lemma 4.2. *For any spanning tree T^p and any positive integer k with $e_k < \mathrm{Min}(T^0 \setminus T^p)$, let g be an arbitrary edge in $Entr(T^p,e_k) \cup \{e_k\}$. Under assumptions* (1), (2), (3), *and* (4), *the following relation holds for a spanning tree $T = T^p\backslash e_k \cup g$ and an edge e_j with $j<k$:*

$$Entr(T,e_j) = \begin{cases} Entr(T^p,e_j) & \text{if } e_j \in A, \\ Entr(T^p,e_j) \setminus Entr(T^p,e_k) & \text{otherwise,} \end{cases} \quad (4.3)$$

where A is the set of ancestors of the edge e_t in T^0 with $\partial^-e_t = \partial^+g$ if it exists; otherwise $A = \emptyset$.

Proof. We note that if $g \in Entr(T^p,e_k)$ then T is a child of T^p, and that if $g = e_k$ then $T = T^p$.

Each descendant of ∂^-e_k relative to T^p is a descendant of ∂^-g relative to T, and vice versa. Therefore, for any $e_j \in A$, $Entr(T,e_j) = Entr(T^p,e_j)$. If $e_j \notin A$ is an ancestor of e_k then $Entr(T,e_j) \subsetneq Entr(T^p,e_j)$. More precisely, for any edge $e \in Entr(T^p,e_j)$ such that ∂^-e is a descendant of ∂^-e_k relative to T^p, e does not belong to $Entr(T,e_j)$, and the other edges obviously belong to $Entr(T,e_j)$. That is $Entr(T,e_j) = Entr(T^p,e_j) \setminus Entr(T^p,e_k)$. If e_j is not an ancestor of e_k, $Entr(T,e_j) = Entr(T^p,e_j) = Entr(T^p,e_j) \setminus Entr(T^p,e_k)$ holds, because $Entr(T^p,e_j) \cap Entr(T^p,e_k) = \emptyset$. ∎

Lemma 4.3. [9] *Let* T^p *be a spanning tree and let* k *be a positive integer with* $e_k < \mathrm{Min}(T^0 \setminus T^p)$. *Under assumptions* (1), (2), (3), *and* (4), *for any edge* $g \in Can(e_k; T^p, k) \cup \{e_k\}$ *and for a spanning tree* $T = T^p \backslash e_k \cup g$, *the following relation holds* :

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

Proof. From the assumptions, for two edges e and f with $e, f < \mathrm{Min}(T^0 \setminus T^p)$, e is an ancestor of f relative to T^0 if and only if e is an ancestor of f relative to T^p, so we will omit the phrase 'relative to T^0 (or T^p)' for such edges. Let e_t be the edge with $\partial^- e_t = \partial^+ g$ if it exists, and let A be the set of edges in T^0 which are ancestors of e_t if e_t exists; otherwise $A = \emptyset$. We prove (4.4) by using the relation (4.3).
Case (1) : If $e_j \notin A$ then

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

Case (2) : If $e_j \in A$ then

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

If $e_j = e_t$ then there is no edge e_h with $j < h < k$ and $e_h \in A$. Therefore,

$$\begin{aligned} Can(e_j; T, k-1) &= Can(e_j; T^p, k) \bigcup [Entr(T^p, e_j) \cap Entr(T^p, e_k)] \\ &= Can(e_j; T^p, k) \bigcup \left[Can(e_k; T^p, k) \cap \{e | \partial^+ e < \partial^- e_t\} \right]. \end{aligned}$$

If e_j is a proper ancestor of e_t then $Entr(T^p, e_j) \cap Entr(T^p, e_k) \subseteq Entr(T^p, e_t)$, and e_t satisfies $j < t < k$ and $e_t \in A$. Hence $Can(e_j; T, k-1) = Can(e_j; T^p, k)$. ∎

Lemma 4.3 guarantees that at most one of sets $Can(*; T^p, k)$ is updated when we want to find all children of T^c or all children of T^p containing e_k. In Figure 5, when the state moves from $(T^0, 3)$ to $(T^0, 2)$, e_1 is the edge such that $\partial^- e_1 = \partial^+ e_3$ and the following equations hold:

$$\begin{aligned} Can(e_2; T^0, 2) &= Can(e_2; T^0, 3) \;=\; \{e_4\} \\ Can(e_1; T^0, 2) &= Can(e_1; T^0, 3) \cup \left[Can(e_3; T^0, 3) \cap \{e | \partial^+ e < \partial^+ e_3\} \right] \\ &= \emptyset \cup \left[\{e_5\} \cap \{e | \partial^+ e < v_2\} \right] \;=\; \{e_5\} \end{aligned}$$

On the other hand, when the state moves from $(T^0, 3)$ to $(T^1, 2)$, no candidate edge-set is updated because there is no edge with $\partial^- e_t = \partial^+ e_5$:

$$\begin{aligned} Can(e_2; T^1, 2) &= Can(e_2; T^0, 3) = \{e_4\} \\ Can(e_1; T^1, 2) &= Can(e_1; T^0, 3) = \emptyset \end{aligned}$$

In our implementation, we use global variables `candi`$(*)$ and `leave`. At the state (T^p, k), variable `candi`(e_j) $(j{=}1, \cdots, k)$ represents the edge-set $Can(e_j; T^p, k)$ and variable `leave` represents the edge-set $\{e_j \mid j \leq k \text{ and } Can(e_j; T^p, k) \neq \emptyset\}$. We can check in constant time whether the current spanning tree has children or not by checking to see if `leave` $\neq \emptyset$. Suppose that each edge-set is represented as an ascending ordered list realized by a doubly linked list. We also use a data structure for a given graph G so that two incidence vertices of any edge are found in constant time, and a data structure for the initial spanning tree T^0 so that for any vertex v other than the root, the unique edge e with $\partial^- e = v$ is found in constant time. Recall that graph G satisfies:

Assumption (5). *For any two edges* $e, f \notin T^0$, *if* $e < f$ *then* $\partial^+ e \leq \partial^+ f$.

From this assumption, one can find the edge-set $Can(e_k; T^p, k) \cap \{e|\partial^+ e{<}\partial^+ g\}$ by searching the ordered list `candi`(e_k) from the beginning. Thus we can complete this in time proportional to the size of this edge-set. Merging two edge-sets can be executed in time proportional to the sum of the size of two edge-sets. Therefore, it takes $O(|Can(e_t; T^p, k)| + |Can(e_k; T^p, k) \cap \{e|\partial^+ e{<}\partial^+ g\}|)$ time for updating edge-sets `candi`$(*)$ when the current state (T^p, k) goes to a succeeding state $(T, k{-}1)$. If `candi`(e_t) changes from empty to nonempty then we must insert an edge e_t into `leave`. Since `leave` is an ascending ordered list, we can complete it in $O(|\{e{\in}\texttt{leave}|e{<}e_t\}|) = O(|\{e_j|j < t \text{ and } Can(e_j; T^p, k) \neq \emptyset\}|)$ time.

On the other hand, when the state goes back from $(T, k{-}1)$ to (T^p, k), we must reconstruct $Can(*; T^p, k)$ from $Can(*; T, k{-}1)$. To do this, we must restore the edges $Can(e_k; T^p, k) \cap \{e|\partial^+ e{<}\partial^+ g\}$ from `candi`(e_t) to `candi`(e_k). In the Shioura-Tamura algorithm [9], such a restoration is efficiently executed by recording $Can(e_k; T^p, k) \cap \{e|\partial^+ e{<}\partial^+ g\}$ before state (T^p, k) goes to $(T, k{-}1)$. This idea, however, requires $O(VE)$ extra space since the depth of recursive calls of the algorithm is $O(V)$. In the rest of this section, we discuss our idea for reducing extra space.

Let $Head(e_j; T^p, k)$ denote the head-set of edges contained in $Can(e_j; T^p, k)$. Then

Lemma 4.4. *Under assumptions* (1), (2), (3) *and* (4), *all head-sets* $Head(e_j; T^p, k)$ *for* $j = 1, \cdots, k$ *are mutually disjoint at any state* (T^p, k).

Proof. From Lemma 4.1, $Head(e_j; T^0, V{-}1) = \{\partial^- e_j\}$ at the initial state $(T^0, V{-}1)$ if $Can(e_j; T^0, V{-}1)$ is nonempty. Thus the assertion is true at the initial state.

We assume that the lemma holds at the state (T^p, k) and prove that this holds at the next state $(T^p \backslash e_k \cup g, k{-}1)$, where $g \in Can(e_k; T^p, k) \cup \{e_k\}$. From Lemma 4.3, the following relation holds:

$$Head(e_j; T, k{-}1) = \begin{cases} Head(e_j; T^p, k) \cup HS & \text{if } \partial^- e_j = \partial^+ g, \\ Head(e_j; T^p, k) & \text{if } \partial^- e_j \neq \partial^+ g, \end{cases} \tag{4.5}$$

where HS is the head-set of all edges in $Can(e_k; T^p, k) \cap \{e|\partial^+ e{<}\partial^+ g\}$. Because $HS \subseteq Head(e_k; T^p, k)$ and each $Head(e_j; T^p, k)$ for $j = 1, \cdots, k{-}1$ does not intersect HS, all head-sets $Head(e_j; T^p, k{-}1)$ for $j = 1, \cdots, k{-}1$ are mutually disjoint. ∎

By Lemma 4.4, the head-set HS of edges in $Can(g;T^p,k)\cap\{e|\partial^+e<\partial^+g\}$ has no intersection with any head set $Head(e_j;T^p,k)$ $(j=1,\cdots,k-1)$. Hence, if we can find HS before restoring `candi`$(*)$, it is easy to pick up the edges $Can(e_k;T^p,k)\cap\{e|\partial^+e<\partial^+g\}=\{e\in Can(e_t;T,k-1)|\partial^-e\in HS\}$ from $Can(e_t;T,k-1)$.

In Figure 5, when the state goes back from $(T^0,1)$ to $(T^0,2)$, all edges in $Can(e_2;T^0,2)\cap\{e|\partial^+e<\partial^+e_2\}=\{e_4\}$ must be restored from `candi`$(e_1)=Can(e_1;T^0,1)=\{e_4,e_5\}$ to `candi`(e_2). The head-set of $Can(e_2;T^0,2)\cap\{e|\partial^+e<\partial^+e_2\}$ is equal to $\{v_3\}$. In this case, $e_4\in$ `candi`(e_1) is put back into `candi`(e_2) to reconstruct $Can(e_2;T^0,2)$.

Our implementation uses global variables `head`$(*)$ for representing each $Head(e_j;T^p,k)$ for $j=1,\cdots,k$ at state (T^p,k). Suppose that each head-set is represented by a (not necessarily ascending) doubly linked list. From Lemma 4.4, we require $O(V)$ space for manipulating these head-sets.

Now we describe two procedures for manipulating data structures `candi`$(*)$, `leave`, and `head`$(*)$ when the current state (T^p,k) goes to a succeeding state $(T,k-1)$ or $(T,k-1)$ goes back to (T^p,k), respectively. The procedure for the first case is shown below:

```
procedure update-data-structure($e_k$,$g$) ;
{ the current state $(T^p,k)$ goes to a succeeding state $(T,k-1)=(T^p\backslash e_k\cup g,k-1)$ }
begin
    $e_t$ := the edge in $T^0$ with $\partial^-e_t=\partial^+g$ if it exists, otherwise return ;
    move $\{e\in$ candi$(e_k)|\partial^+e<\partial^+g\}$ from candi$(e_k)$ to candi$(e_t)$ ;
    if candi$(e_t)$ changes from empty to nonempty then insert $e_t$ into leave ;
    $HS$ := the head set of the edges in $\{e\in$ candi$(e_k)|\partial^+e<\partial^+g\}$ ;
    for each maximal sublist of consecutive elements of $HS$ in head$(e_k)$ do begin
        record the first element of the sublist and its position in head$(e_k)$ on a stack ;
        delete the sublist from head$(e_k)$ ;
        add this to the end of head$(e_t)$ ;
    end ;
    record the position of the first element of $HS$ in head$(e_t)$ on a stack ;
end .
```

When the state changes from (T^p,k) to $(T,k-1)$, we must move the head-set HS of all edges in $Can(e_k;T^p,k)\cap\{e|\partial^+e<\partial^+g\}$ from `head`(e_k) to `head`(e_t). At this time, we do not move each element of HS one by one, but move each maximal sublist of consecutive elements of HS in `head`(e_k) to `head`(e_t) as Figure 6. Then extra space for recording positions of such maximal sublists is $O(V)$ in all because the number of maximal sublists is at most $|$`head`$(e_k)\setminus HS|+1$, and `head`$(e_k)\setminus HS$ is unchanged until the state comes back to (T^p,k). It is easy to manipulate `head`$(*)$ in the same time as `candi`$(*)$, because $|HS|\le|Can(e_k;T^p,k)\cap\{e|\partial^+e<\partial^+g\}|$. Here we omit details. Thus the time complexity of the procedure is $O(|Can(e_t;T^p,k)|+|Can(e_k;T^p,k)\cap\{e|\partial^+e<\partial^+g\}|+|\{e_j|j<t$ and $Can(e_j;T^p,k)\neq\emptyset\}|)$.

The second procedure restores data structures in the following way:

```
procedure restore-data-structure($e_k$,$g$) ;
```

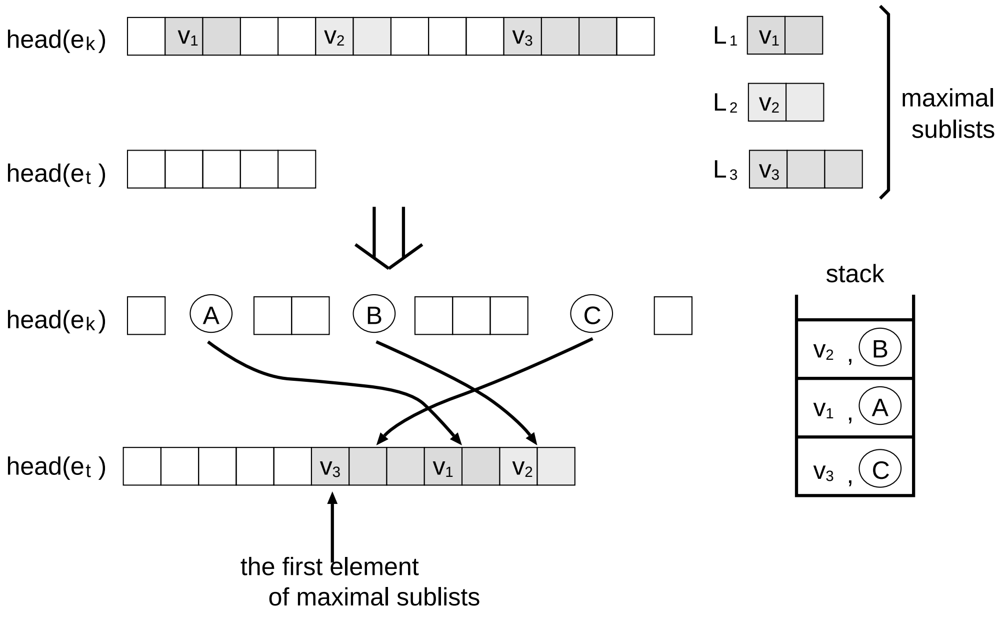

Figure 6: update of `head`$(*)$

```
{ the state $(T^p \backslash e_k \cup g, k-1)$ goes back to $(T^p, k)$ }
**begin**
    $e_t$ := the edge in $T^0$ with $\partial^- e_t = \partial^+ g$ if it exists, otherwise return ;
    find $HS$ by the record of the position of its first element in head($e_t$) ;
    delete $HS$ from head($e_t$) ;
    move $\{e \in$ candi($e_t$)$| \partial^- e \in HS\}$ from candi($e_t$) to the beginning of candi($e_k$) ;
    **if** candi($e_t$) changes from nonempty to empty **then** delete $e_t$ from leave ;
    move each sublist in $HS$ to the correct place in head($e_k$) by using records on a stack ;
**end** .
```

Since we recorded the first element of head vertices which were added to `head`(e_t), we can find HS in constant time. For each edge in `candi`(e_t), we can check in constant time whether it is in HS, by marking all elements of HS in advance. Hence we can restore `candi`$(*)$ in $O(|Can(e_t; T, k-1)|) = O(|Can(e_t; T^p, k)| + |\{e \in Can(e_k; T^p, k) | \partial^+ e < \partial^+ g\}|)$ time. Deletion of an edge from `leave` is completed in constant time. The head-set HS is returned from `head`(e_t) to `head`(e_k) in time proportional to the number of maximal sublists by the information of the places in `head`(e_k). Therefore, procedure restore-data-structure( ) takes $O(|Can(e_t; T^p, k)| + |\{e \in Can(e_k; T^p, k) | \partial^+ e < \partial^+ g\}|)$ time.

5 An optimal implementation and its analysis.

Finally we describe our efficient implementation and analyze its time and space complexities. Our implementation 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
    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) and (5),
        · for each $e_j \in T^0$, $\texttt{candi}(e_j) := \{e | e \notin T^0, \partial^+ e \le \partial^+ e_j$ and $\partial^- e = \partial^- e_j\}$,
        · for each $e_j \in T^0$, $\texttt{head}(e_j) := \{\partial^- e_j\}$,
        · $\texttt{leave} := \{e_j \in T^0 | \texttt{candi}(e_j) \ne \emptyset\}$ ;
    output(“$e_1, e_2, \cdots, e_{V-1}, tree,$”) ;                {output $T^0$}
    find-children( ) ; {of $T^0$}
end .

procedure find-children( ) ;    {$T^p$:current spanning tree}
begin
    if $\texttt{leave} = \emptyset$ then return ;
    $Q := \emptyset$ ;
    $e_k :=$ the last entry of $\texttt{leave}$;
    delete $e_k$ from $\texttt{leave}$;
    while $\texttt{candi}(e_k) \ne \emptyset$ 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 $T^c := T^p \setminus e_k \cup g$}
        update-data-structure($e_k$,$g$) ;
        find-children( ) ;                           {find children of $T^c$}
        restore-data-structure($e_k$,$g$) ;
        output(“$-g, +e_k,$”) ;                      {reconstruct $T^p := T^c \cup e_k \setminus g$}
    end ;
    move all entries of $Q$ to $\texttt{candi}(e_k)$ ;
    update-data-structure($e_k$,$e_k$) ;
    find-children( ) ;                               {find children of $T^p$ containing $e_k$}
    restore-data-structure($e_k$,$e_k$) ;
    add $e_k$ to the end of $\texttt{leave}$;
end .
```

Now we discuss the time complexity of our implementation. The next lemma is useful for analyzing the time complexity.

Lemma 5.1. [9] *Suppose that* T *is a spanning tree and that* k *is a positive integer with* $e_k < \mathrm{Min}(T^0 \setminus T)$. *Under assumptions* (1), (2), (3), *and* (4), *for any edge* $g_j \in \{e_j\} \cup Can(e_j; T, k)$ $(j \le k)$, $T' = T \setminus \{e_1, \cdots, e_k\} \cup \{g_1, \cdots, g_k\}$ *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 4.3, $Can(e_{j-1}; T, j-1) \subseteq Can(e_{j-1}; T^j, j-1)$. Thus, $T^{j-1} = T^j \setminus e_{j-1} \cup g_{j-1}$ is a spanning tree. ∎

In algorithm all-spanning-tree( ), the time required other than calling find-children( ) is $O(V+E)$. At the state (T^p, k), $O(\#$ of children of T^p not containing $e_k)$ time is taken to execute procedure find-children( ) other than maintenance of data structures. Now we consider the time complexities of maintenance of data structures. From the discussion in Section 4, it takes $O(|Can(e_t;T^p,k)| + |Can(e_k;T^p,k) \cap \{e|\partial^+e{<}\partial^+g\}| + |\{e_j|j < t$ and $Can(e_j;T^p,k) \neq \emptyset\}|)$ to maintain data structures when the state changes between (T^p,k) and $(T^p \backslash e_k \cup g, k{-}1)$, where e_t is an edge with $\partial^-e_t{=}\partial^+g$. We consider the next two cases:

Case A: maintenance for finding children of T^c (i.e., $g \in Can(e_k;T^p,k)$)
Case B: maintenance for finding children of T^p containing e_k (i.e., $g = e_k$)

Note that Case A occurs exactly one time for each spanning tree T^c other than T^0, and that Case B occurs at most one time for each spanning tree T^p and for each edge $e_k \in \{e|e_1 \leq e < \mathrm{Min}(T^0 \setminus T^p)\}$. In Case A, $|Can(e_t;T^p,k)|+|Can(e_k;T^p,k)\cap\{e|\partial^+e{<}\partial^+g\}|$ is bounded by the number of children of T^c not containing e_t. Moreover, for each edge e_j with $j < t$ and $Can(e_j;T^p,k) \neq \emptyset$, there is a child of T^c not containing e_j. Therefore, the time complexity in Case A is $O(\#$ of children of $T^c)$. In Case B, $|Can(e_k;T^p,k)\cap\{e|\partial^+e{<}\partial^+e_k\}|$ is bounded by the number of children of T^p not containing e_k. From Lemma 5.1, T^p has at least $|\{e{\in}Can(e_k;T^p,k)|\partial^+e{<}\partial^+e_k\}| \times |Can(e_t;T^p,k)|$ grandchildren which contain neither e_k nor e_t. Similarly, $|\{e_j|j < t$ and $Can(e_j;T^p,k) \neq \emptyset\}|)$ is bounded by the number of grandchildren of T^p not containing e_k. Thus the time complexity in Case B is

$$O(\#\text{ of children of } T^p \text{ not containing } e_k) + O(\#\text{ of grandchildren of } T^p \text{ not containing } e_k).$$

We recall that procedure find-children( ) checks in constant time whether T^p has children. From the above discussion, the total required time of find-children( ) at the state (T^p,k) is

$$O(\#\text{ of children and grandchildren of } T^p \text{ not containing } e_k)$$

Thus, the total time complexity of our implementation is $O(N{+}V{+}E)$.

Finally we consider the space complexity. At any state, edge-sets `candi`(e_j) $(j = 1,\cdots,V{-}1)$ have no intersection with each other, and neither do head-sets `head`(e_j) $(j = 1,\cdots,V{-}1)$. Thus, we need $O(V{+}E)$ space for `candi` and $O(V)$ space for `head`. Obviously, the cardinality of `leave` is at most $V{-}1$. As we described in Section 4, the size of the stack recording positions maximal sublists of HS is $O(V)$ in all. The total size of local variables Q in find-children( ) is $O(E)$ because each edge is stored in one of global variables `candi`$(*)$ or local variables Q. Hence, the space complexity of our implementation is $O(V{+}E)$.

Theorem 5.2. *The time and space complexities of our implementation are $O(N{+}V{+}E)$ and $O(V{+}E)$, respectively.*

In this paper, we proposed an efficient algorithm for enumerating all spanning trees. This is optimal in sense of time and space complexities.

Acknowledgements

We are greatly indebted to Prof. Yoshiko T. Ikebe of Science University of Tokyo for her kind and valuable comments on this manuscript.

References

[1] D. Avis and K. Fukuda, *A basis enumeration algorithm for linear systems with geometric applications*, Applied Mathematics Letters, 4 (1991), pp. 39–42.

[2] D. Avis and K. Fukuda, *A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra*, Discrete and Computational Geometry, 8 (1992), pp. 295–313.

[3] D. Avis and K. Fukuda, *Reverse search for enumeration*, to appear in Discrete Applied Mathematics.

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

[5] S. Kapoor and H. Ramesh, *Algorithms for enumerating all spanning trees of undirected and weighted graphs*, SIAM J. Comput., 24 (1995), pp. 247–265.

[6] T. Matsui, *An algorithm for finding all the spanning trees in undirected graphs*, Research Report, Dept. of Mathematical Engineering and Information Physics, University of Tokyo, Tokyo, 1993.

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

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

[9] A. Shioura and A. Tamura *Efficiently scanning all spanning trees of an undirected graph*, to appear in J. Operations Research Society of Japan.

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