versão impressa ISSN 0101-7438 / versão online ISSN 1678-5142

AN ALGORITHM TO GENERATE ALL SPANNING TREES OF A GRAPH IN ORDER OF INCREASING COST

Kenneth Sörensen
University of Antwerp
Prinsstraat 13
B-2000 Antwerpen – Belgium
kenneth.sorensen@ua.ac.be

Gerrit K. Janssens *
Hasselt University
Agoralaan – Building D
B-3590 Diepenbeek – Belgium
gerrit.janssens@uhasselt.be

* *Corresponding author*/autor para quem as correspondências devem ser encaminhadas

Recebido em 08/2003; aceito em 04/2005 após 1 revisão
Received August 2003; accepted April 2005 after one revision

Abstract

A minimum spanning tree of an undirected graph can be easily obtained using classical algorithms by Prim or Kruskal. A number of algorithms have been proposed to enumerate all spanning trees of an undirected graph. Good time and space complexities are the major concerns of these algorithms. Most algorithms generate spanning trees using some fundamental cut or circuit. In the generation process, the cost of the tree is not taken into consideration. This paper presents an algorithm to generate spanning trees of a graph in order of increasing cost. By generating spanning trees in order of increasing cost, new opportunities appear. In this way, it is possible to determine the second smallest or, in general, the *k*-th smallest spanning tree. The smallest spanning tree satisfying some additional constraints can be found by checking at each generation whether these constraints are satisfied. Our algorithm is based on an algorithm by Murty (1967), which enumerates all solutions of an assignment problem in order of increasing cost. Both time and space complexities are discussed.

Keywords: weighted spanning trees; enumeration; computational complexity.

1. The Minimum Spanning Tree Problem

An *undirected graph G* is defined as a pair *(V,E)*, where *V* is a set of *vertices* and *E* is a set of *edges*. Each edge connects two vertices, i.e. $E = \{(u,v) \mid u,v \in V\}$. An undirected, *weighted* graph has a *weighting function* $w: E \rightarrow \Re$, which assigns a weight to each edge. The weight of an edge is often called its cost or its distance.

A *tree* is a subgraph of *G* that does not contain any circuits. As a result, there is exactly one path from each vertex in the tree to each other vertex in the tree. A *spanning tree* of a graph *G* is a tree containing all vertices of *G*. A *minimum spanning tree* (MST) of an undirected, weighted graph *G* is a spanning tree of which the sum of the edge weights (costs) is minimal.

There are several greedy algorithms for finding a minimal spanning tree *M* of a graph. The algorithms of Kruskal and Prim are well known.

Kruskal's algorithm. Repeat the following step until the set *M* has *n-1* edges (initially *M* is empty). Add to *M* the shortest edge that does not form a circuit with edges already in *M*.

Prim's algorithm. Repeat the following step until the set *M* has *n-1* edges (initially *M* is empty): Add to *M* the shortest edge between a vertex in *M* and a vertex not in *M* (initially pick any edge of shortest length).

Although both are greedy algorithms, they are different in the sense that Prim's algorithm grows a tree until it becomes the MST, whereas Kruskal's algorithm grows a forest of trees until this forest reduces to a single tree, the MST.

A spanning tree *s* can be represented by a set of *n-1* edges. An edge can be represented by an unordered couple of vertices.

$$s = \{(a_1, b_1,), ...(a_{n-1}, b_{n-1})\}$$

We define *A* as the set of all spanning trees of a graph *G*.

Several algorithms exist for generating all spanning trees of a graph (e.g. Gabow & Myers, 1978; Kapoor & Ramesh, 1995; Matsui, 1993; Minty, 1965; Shioura & Tamura, 1995; Read & Tarjan, 1975; Kapoor & Ramesh, 2000; Matsui, 1997). Good space and time complexities are the most important concerns of these algorithms. Most algorithms generate spanning trees using some fundamental cut or circuit, but none of them takes the cost of the tree into account while generating spanning trees. The algorithms, which generate all spanning trees without weights (Minty, 1965; Read & Tarjan, 1975), can be applied to our problem by sorting the trees according to an increasing weight after they have been generated. As the number of trees can be very large (especially for complete graphs) this option is excluded for practical purposes.

2. Generating Spanning Trees in Order of Increasing Cost

In the following we will assume that $c(s_i)$ is the cost assigned to spanning tree s_i and *i* is the rank of s_i when all spanning trees are ranked in order of increasing cost. We thus adopt the convention that $c(s_i) \leq c(s_j)$ if $i < j$. The sequence s_1, s_2, ... is a ranking of spanning trees in order of increasing cost.

2.1 Terminology

2.1.1 Partition

A partition P is defined to be a non-empty subset of the set of all spanning trees A of a graph G, that has the following form

$$P = \{ s : (i_1, j_1) \in s; ...; (i_r, j_r) \in s; (m_1, p_1) \notin s; ..., (m_l, p_l) \notin s \}$$

In other words, P is the set of spanning trees containing all of the edges $(i_1, j_1), \ldots, (i_r, j_r)$ (called *included* edges), and not containing any of the edges $(m_1, p_1), \ldots, (m_l, p_l)$ (called *excluded* edges). Edges of G that are neither included nor excluded edges of the partition, are called *open.*

For convenience, we indicate the partition P as

$$P = \left\{ (i_1, j_1), ..., (i_r, j_r); \left(\overline{m_1, p_1}\right), ..., \left(\overline{m_l, p_l}\right) \right\}.$$

The bar above edges $(m_1, p_1), \ldots, (m_l, p_l)$ indicates that they are excluded edges. Because of the excluded edges, some partitions may not contain any spanning trees. This is the case when the graph G from which the excluded edges of the partition are removed, is disconnected. Partitions that do not contain any spanning trees are called *empty* partitions.

It should be remarked that A, the set of all spanning trees, is also a partition, but a special one that has no included or excluded edges (i.e. all edges are open).

2.1.2 A minimum spanning tree in partition *P*

An MST in P is defined as a spanning tree of minimal cost that is an element of P and thus contains all included edges and none of the excluded edges of P. Since every spanning tree in partition P contains the edges $(i_1, j_1), \ldots, (i_r, j_r)$, a minimum spanning tree that is an element of this partition can be found by searching $n\text{-}r\text{-}1$ open edges of the partition. To ensure that all required edges are included into a minimum spanning tree of the partition, they can be added before all remaining edges. To ensure that excluded edges are not in an MST, they can be temporarily assigned infinite cost.

The way in which partitions are formed ensures that the set of included edges does not contain any circuits. Kruskal's algorithm can start from this partial spanning tree and continue to add edges to it.

Because the set of included edges is not necessarily a tree, Prim's algorithm has to be modified in the following way. Add to M the shortest edge between a vertex in M and another vertex, which does not form a circuit with edges already in M. This modified algorithm allows for edges to connect two disconnected parts of the spanning tree, but prevents from forming circuits in M.

A minimum spanning tree in partition P is indicated as $s(P)$. Its cost by $c[s(P)]$.

2.1.3 Partitioning *P* by its minimum spanning tree

The idea of partitioning is at the heart of the algorithm proposed in this paper. Given an MST of a partition, this partition can be split into a set of resulting partitions in such a way that the following statements hold:

- the intersection of any two of the resulting partitions is the empty set,
- the MST of the original partition is not an element of any of the resulting partitions,
- the union of the resulting partitions is equal to the original partition, minus the MST of the original partition.

More formally, we can express this as follows. Let a minimum spanning tree in P be

$$s(P) = \{(i_1, j_1), ..., (i_r, j_r), (t_1, v_1), ..., (t_{n-r-1}, v_{n-r-1})\}$$

where $(t_1, v_1), \ldots, (t_{n-r-1}, v_{n-r-1})$ are all different from $(m_1, p_1), \ldots, (m_l, p_l)$. Then P can be expressed as the union of the singleton set $\{s(P)\}$ and the partitions $P_1, \ldots, P_{n-r-1}$, which are mutually disjoint, where

$$P_1 = \{(i_1, j_1), ..., (i_r, j_r), (\overline{m_1, p_1}), ..., (\overline{m_l, p_l}), (\overline{t_1, v_1})\}$$
$$P_2 = \{(i_1, j_1), ..., (i_r, j_r), (t_1, v_1), (\overline{m_1, p_1}), ..., (\overline{m_l, p_l}), (\overline{t_2, v_2})\}$$
$$P_3 = \{(i_1, j_1), ..., (i_r, j_r), (t_1, v_1), (t_2, v_2), (\overline{m_1, p_1}), ..., (\overline{m_l, p_l}), (\overline{t_3, v_3})\}$$
$$...$$
$$P_{n-r-1} = \{(i_1, j_1), ..., (i_r, j_r), (t_1, v_1), ..., (t_{n-r-2}, v_{n-r-2}), (\overline{m_1, p_1}), ..., (\overline{m_l, p_l}), (\overline{t_{n-r-1}, v_{n-r-1}})\}$$

It can be shown that the partitions $P_1, \ldots, P_{n-r-1}$ are mutually disjoint by remarking that any spanning tree in P either contains (t_1, v_1) or does not (in which case it is an element of P_1). If it does, it either contains (t_2, v_2) or does not (in which case it is an element of P_2). Continuing like this and remarking that the only spanning tree that contains the edges $(i_1, j_1), \ldots, (i_r, j_r), (t_1, v_1), \ldots, (t_{n-r-1}, v_{n-r-1})$ is $s(P)$, we find that

$$P = \{s(P)\} \cup \bigcup_{i=1}^{n-r-1} P_i$$

Every spanning tree in partitions P_1 to P_{n-r-1} contains $(i_1, j_1), \ldots, (i_r, j_r)$ and every spanning tree does not contain $(m_1, p_1), \ldots, (m_l, p_l)$.

2.1.4 A list at stage *k*

Stage k in the enumeration process refers to the stage in which $s_1, ..., s_k$ are determined. At this stage, a list contains a set of partitions $M_1, \ldots, M_e$ with the properties that

- $M_1, \ldots, M_e$ are mutually disjoint,
- none of the partitions in the list contains any of the spanning trees already generated (s_u, $u = 1, ..., k$),
- the union of all partitions in the list is the set of all spanning trees not yet generated.

From these properties, it holds that

$$A = \bigcup_{u=1}^{k} \{s_u\} \cup \bigcup_{v=1}^{e} M_v .$$

From the definition of a list for stage k, it is clear that the k-th smallest spanning tree s_{k+1} is equal to $s(M_d)$ where M_d is any partition in the list for which $c[s(M_d)] = \min_{i=1..e} \{c[s(M_i)]\}$.

2.2 Algorithm for ranking spanning trees in order of increasing cost

Given a graph G containing n vertices, the algorithm proceeds in stages. At stage k, the k-th smallest spanning tree is generated.

2.2.1 Stage 1

Set the list for stage 0 equal to the partition A. Find an MST of A (or of G). Let it be

$$s_1 = \{(i_1, j_1),...,(i_{n-1}, j_{n-1})\}.$$

Partition A by its MST, creating the partitions $M_1, \ldots, M_{n-1}$, defined as

$$M_1 = \left\{\left(\overline{i_1, j_1}\right)\right\}$$
$$M_2 = \left\{(i_1, j_1), \left(\overline{i_2, j_2}\right)\right\},$$
$$M_3 = \left\{(i_1, j_1), (i_2, j_2), \left(\overline{i_3, j_3}\right)\right\}$$
$$\ldots$$
$$M_{n-1} = \left\{(i_1, j_1), \ldots (i_{n-2}, j_{n-2}), \left(\overline{i_{n-1}, j_{n-1}}\right)\right\}$$

Then $\{M_1,...,M_{n-1}\}$ forms a list for stage 1. Empty partitions (that do not contain any spanning trees) may be removed from the list.

2.2.2 Stage *k*

Given a list for stage *k-1* consisting of t partitions $L_1, \ldots, L_t$, we calculate the minimum spanning tree $s(L_1), \ldots, s(L_t)$ for each partition in the list and the cost $c[s(L_1)], \ldots, c[s(L_t)]$ of each of these spanning trees.

Then, the k-th smallest spanning tree is the spanning tree with the lowest cost:

$$s_k = \left\{ s(L_i) \mid c\left[s(L_i)\right] = \min_{j=1..t} c\left[s(L_j)\right] \right\}.$$

L_i is the partition that contains the smallest spanning tree of all spanning trees not yet generated. A list for stage k is formed by deleting L_I from the list for stage *k-1* and replacing it with the partitions formed by partioning L_i by $s(L_i)$. Empty partitions are removed from the list. Ties are solved by picking one partition at random and by leaving the others in the list.

2.3 Example

The algorithm is illustrated for ranking all spanning trees in order of increasing cost by means of an example. Consider graph *G*, consisting of five vertices *A, B, C, D, E*. Any spanning tree of G consists of four edges.

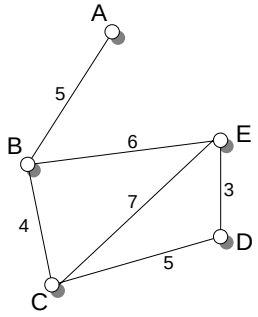

Figure 1 – Example graph G.

The first step in ranking all spanning trees in order of increasing cost is to determine the minimum spanning tree in the partition *A*. The minimum spanning tree of *G* equals $s_1=\{(A, B), (B, C), (C, D), (D, E)\}$ and $c[s_1] = 17$.

Now, *G* is partitioned by s_1, obtaining four partitions, $P_1, \ldots, P_4$, forming a list for stage 1:

$$P_1 = \left\{\left(\overline{A,B}\right)\right\}$$
$$P_2 = \left\{(A,B),\left(\overline{B,C}\right)\right\}$$
$$P_3 = \left\{(A,B),(B,C),\left(\overline{C,D}\right)\right\}$$
$$P_4 = \left\{(A,B),(B,C),(C,D),\left(\overline{D,E}\right)\right\}$$

Graphically, the partitions can be represented as in Figure 2 (a dotted line depicts an excluded edge, a bold line an included edge).

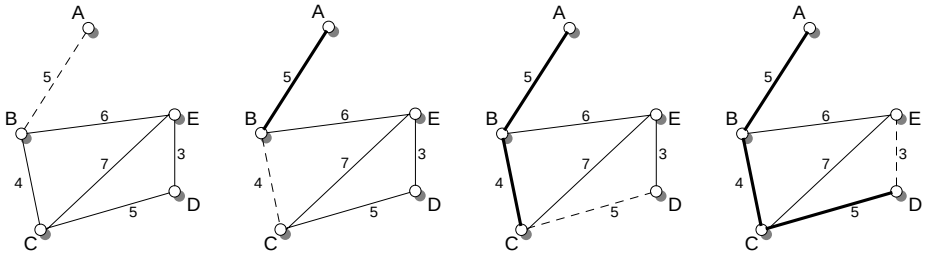

Figure 2 – Partitions $P_1, \ldots, P_4$.

The next step is to calculate a minimum spanning tree in each partition in the list. Since P_1 is not connected, it does not have a minimum spanning tree. The minimum spanning trees of nodes P_2 to P_4 are

$$s(P_2) = \{(A,B),(B,E),(E,D),(D,C)\}$$
$$s(P_3) = \{(A,B),(B,C),(B,E),(E,D)\} \ .$$
$$s(P_4) = \{(A,B),(B,C),(C,D),(B,E)\}$$

Their respective costs are

$$c[s(P_2)] = 19, \quad c[s(P_3)] = 18, \quad c[s(P_4)] = 20 .$$

Since P_3 has the minimum spanning tree with lowest cost:

$$s_2 = s(P_3) = \{(A,B),(B,C),(B,E),(E,D)\} .$$

By partitioning P_3 by its minimum spanning tree $s(P_3)$, we obtain partitions P_{31} and P_{32}.

$$P_{31} = \{(A,B),(B,C),(\overline{C,D}),(\overline{B,E})\}$$
$$P_{32} = \{(A,B),(B,C),(B,E),(\overline{C,D}),(\overline{E,D})\} .$$

Graphically, the partition is represented in Figure 3.

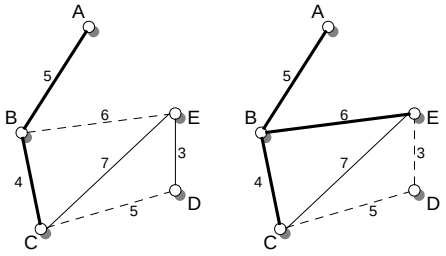

Figure 3 – Partitions P_{31} and P_{32} .

A list for stage 2 consists of $\{P_2, P_{31}, P_4\}$. Since P_{32} is not connected, it is removed from the list. The minimum spanning tree for node P_{31} is

$$s(P_{31}) = \{(A,B),(B,C),(C,E),(E,D)\}$$

with cost

$$c[s(P_{31})] = 19 .$$

The list for stage 2 contains two partitions that have a minimum spanning tree with minimal cost (P_2 and P_{31}). Ties like this one are solved by picking any of both partitions for partitioning in the next stage.

Continuing in the same way, eight spanning trees are obtained with costs ranging from 17 to 23.

3. Implementation of the Algorithm on a Computer

To implement the algorithm on a computer, the nodes in the list for the current stage need to be stored in memory. A partition can be represented by its included and its excluded edges. The given graph can be represented by three arrays, representing the head and tail of each edge and the weight of the edge respectively. A partition can be represented in two ways.

The first is to indicate the head and tail of the included and excluded edges. The second is to indicate for each edge in the graph whether it is included, excluded or open. The list of partitions can be efficiently implemented using a linked list.

A possible structure for the program generating all spanning trees in order of increasing cost, is:

ALGORITHM 1: GENERATE SPANNING TREES IN ORDER OF INCREASING COST

Input: Graph $G(V,E)$ and weight function w

Output: Output_File (all spanning trees of G, sorted in order of increasing cost)

List = $\{A\}$

Calculate_MST (A)

while MST $\neq \varnothing$ **do**

Get partition $P_s \in$ List that contains the smallest spanning tree

Write MST of P_s to Output_File

Remove P_s from List

Partition(P_s).

The partitioning procedure adds partitions to the list after checking whether they are connected and calculating their minimum spanning tree. The main disadvantage of this approach is that we either have to keep a minimum spanning tree of the partition in the list (wasting memory) or calculate it again when the partition is retrieved from the list (wasting time). The main advantage is that we can keep a sorted list of partitions instead of an unsorted one and that retrieval of the smallest partition becomes easy. A possible program structure for the partitioning procedure is:

PROCEDURE PARTITION (P)

$P_1 = P_2 = P$;

for each edge i in P **do**

if *i not included in P and not excluded from P* **then**

make i excluded from P_1;

make i included in P_2;

Calculate_MST (P_1);

if Connected (P_1) **then**

add P_1 to List;

$P_1 = P_2$;

4. Storage Requirements (Space and Time Complexities)

Let $|E|$ be the number of edges, $|V|$ the number of vertices and N the number of spanning trees of a given graph G. Many algorithms for generating all spanning trees obtain good time complexity by outputting spanning trees in a certain order so that a short notation can be used. Spanning trees can e.g. be generated by exchange of one edge from the previous

spanning tree in the generation process. In this way, a short notation format can be developed where the first spanning tree is written as output and the rest is restricted to the exchanged pair of edges.

Since there is no such order obtained by our algorithm, $O(N.|V|)$ space is needed to generate all spanning trees. Because all nodes in the list are mutually exclusive, the number of spanning trees puts an upper limit on the number of partitions in the list. Since the list of partitions is never larger than the number of spanning trees, it contains a maximum of N partitions. A partition can be represented by the status of each of its edges (*open*, *included*, or *excluded*). Therefore, the size of each node is $O(|E|)$. The space complexity of the partition list therefore is $O(N.|E|)$. Simulations however show that, in most cases, only a small fraction of the space is needed at any moment.

The time complexity of the algorithm can be calculated using the time complexity of the algorithms for generating spanning trees. The generation of a spanning tree using Kruskal's algorithm is $O(|E|log\ |E|)$. The time complexity of generating the spanning tree from a partition instead of a graph using this algorithm is obviously the same. To determine the time complexity of the algorithm, we investigate it in detail.

In the following paragraphs we assume that the partition list is always kept sorted. In that way, retrieving an item from the list can be done in constant time. Inserting an item into the list requires $O(N)$ operations, since the maximum length of the list is equal to the maximal number of partitions N. Input and output actions are disregarded.

Most steps in the algorithm can be executed in constant time. Checking whether a partition is empty or not (*if* Connected()) can be done in constant time because this is information is available from the minimum spanning tree algorithm. The main loop in the algorithm is executed exactly N times and therefore, the procedure PARTITION is executed N times. As indicated before, *Calculate_MST* is $O(|E|log\ |E|)$ and *Add to List* is $O(N)$. The algorithm has time complexity $O(N.|E|log\ .|E| + N^2)$.

Both time and space complexities of our algorithm are worse than those of other algorithms. Algorithms by Gabow & Meyers (1978), Matsui (1993) and Shioura & Tamura (1995) are able to generate all spanning trees of a graph in $O(|V|.|E|)$ space and $O(N.|V| + |V| + |E|)$ time. As mentioned however, the goal of our algorithm is not to generate all spanning trees, but to stop generating spanning trees when a spanning tree has been found that satisfies some additional constraints. In general, this will require the generation of only a small portion of the total number of spanning trees.

5. Applications

Potential applications mainly are to be found in the class of minimum spanning tree problems with additional constraints. A general algorithm for these applications, using the algorithm in this paper, is to generate spanning trees in order of increasing cost and check at each generation whether the additional constraints are satisfied. It is easy to see that the first spanning tree to be found that satisfies the additional constraints is a minimum spanning tree that satisfies the constraints.

Murty's algorithm for ranking assignments in order of increasing cost has been used in a similar fashion to generate an optimal solution to the travelling salesman problem

(Panayiotopoulos, 1982). If a given travelling salesman problem is described as an assignment problem, then the first assignment that also is a tour, is the optimal tour.

Some potential example applications:

- The capacitated minimum spanning tree at a given root partition, that has a cardinality constraint on the size of the subtrees off of a given root node partition. See e.g. Hall & Magnanti (1992) and Papadimitriou (1978).
- The degree-constrained minimum spanning tree, which has an upper limit on the degree of every vertex (or of a specified vertex *r*). See e.g. Gabow (1978).
- The hop-constrained minimum spanning tree, imposing that the number of edges between the root and any leaf of the tree is limited to a specified integer number. A well-known special case of this application is the 2-hop spanning tree, which is worked out in detail by Dahl (1998).

The main advantage of the proposed algorithm is its versatility. In theory, any minimum spanning tree problem with additional constraints can be solved using the proposed method.

The disadvantage of the proposed algorithm is that it cannot guarantee fast results. It is theoretically possible that the smallest spanning tree that satisfies the additional constraints is the largest spanning tree of the graph. In this case, according to a theorem by Cayley, the algorithm may need to generate up to $|V|^{|V|-2}$ trees (depending on the number of edges) before the required spanning tree is found, which is, of course, not acceptable.

However, in many cases it is not unreasonable to assume that the smallest spanning tree that satisfies additional constraints is not much larger than the minimum spanning tree of the graph. In these cases, the algorithm is able to quickly produce the required spanning tree.

6. Generating Spanning Trees in Order of Decreasing Cost

Until now, we have only discussed the case in which the smallest spanning tree satisfying additional constraints was sought. In some cases, we may want to find the *largest* spanning tree satisfying additional constraints. It is clear that the algorithm can be easily adapted to be able to do just this function. Both Kruskal's and Prim's algorithm can be easily changed to look for the maximum spanning tree instead of the minimum spanning tree. Likewise, the algorithm for generating spanning trees in order of increasing cost can easily be transformed into an algorithm for generating spanning trees in order of decreasing cost.

7. Conclusion

In this paper, an algorithm has been developed for ranking all spanning trees of a given graph in order of increasing cost. The algorithm is based on an algorithm, developed by Murty, for ranking assignments of an assignment problem in order of increasing cost.

Some guidelines were given to implement the algorithm on a computer and the space and time complexities of the algorithm were discussed briefly.

Finally, some potential applications of the algorithm were given. All potential applications can be categorized as minimum spanning tree problems with additional constraints.

References

(1) Dahl, G. (1998). The 2-hop spanning tree problem. *Operations Research Letters*, **23**, 21-26.

(2) Diestel, R. (1996). *Graph Theory*. Springer, New York, xiv + 266 pp.

(3) Gabow, H.N. (1977). Two algorithms for generating weighted spanning trees in order. *SIAM Journal on Computing*, **6**(1), 139-150.

(4) Gabow, H.N. (1978). A good algorithm for smallest spanning trees with a degree constraint. *Networks*, **8**, 201-208.

(5) Gabow, H.N. & Myers, E.W. (1978). Finding all spanning trees of directed and undirected graphs. *SIAM Journal on Computing*, **7**, 280-287.

(6) Hall, L. & Magnanti, T. (1992). A polyhedral intersection theorem for capacitated trees. *Mathematics of Operations Research*, **17**, 398-410.

(7) Kapoor, S. & Ramesh, H. (1995). Algorithms for enumerating all spanning trees of undirected and weighted graphs. *SIAM Journal on Computing*, **24**, 247-265.

(8) Kapoor, S. & Ramesh, H. (1997). An algorithm for enumerating all spanning trees of a directed graph. *Algorithmica*, **27**(2), 120-130.

(9) Matsui, T. (1993). 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.

(10) Matsui, T. (1997). A flexible algorithm for generating all the spanning trees in undirected graphs. *Algorithmica*, **18**(4), 530-543.

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

(12) Murty, K.G. (1986). An algorithm for ranking all the assignments in order of increasing cost. *Operations Research*, **16**, 682-687.

(13) Panayiotopoulos, J-C. (1982). Probabilistic analysis of solving the assignment problem for the travelling salesman problem. *European Journal of Operational Research*, **9**, 77-82.

(14) Papadimitriou, C. (1978). The complexity of the capacitated tree problem. *Networks*, **8**, 219-234.

(15) Read, R.C. & Tarjan, R.E. (1975). Bounds on backtrack algorithms for listing cycles, paths and spanning trees. *Networks*, **5**(3), 237-252.

(16) Shioura, A. & Tamura, A. (1995). Efficiently scanning all spanning trees of an undirected graph. *Journal of the Operations Research Society of Japan*, **38**(3), 331-344.