International Journal of Computer Mathematics
Vol. 87, No. 14, November 2010, 3175–3185

Listing all the minimum spanning trees in an undirected graph

Takeo Yamada*, Seiji Kataoka and Kohtaro Watanabe

Department of Computer Science, National Defense Academy, Yokosuka, Kanagawa 239-8686, Japan

(*Received 05 December 2008; revised version received 06 April 2009; second revision received 15 July 2009; accepted 06 September 2009*)

Efficient polynomial time algorithms are well known for the minimum spanning tree problem. However, given an undirected graph with integer edge weights, minimum spanning trees may not be unique. In this article, we present an algorithm that lists all the minimum spanning trees included in the graph. The computational complexity of the algorithm is $\mathrm{O}(N(mn + n^2 \log n))$ in time and $\mathrm{O}(m)$ in space, where n, m and N stand for the number of nodes, edges and minimum spanning trees, respectively. Next, we explore some properties of cut-sets, and based on these we construct an improved algorithm, which runs in $\mathrm{O}(Nm \log n)$ time and $\mathrm{O}(m)$ space. These algorithms are implemented in C language, and some numerical experiments are conducted for planar as well as complete graphs with random edge weights.

Keywords: minimum spanning tree; enumeration algorithm

2000 AMS Classifications: 05C05; 05C30; 68W05

1. Introduction

Let $G = (V, E)$ be an undirected graph with vertex set $V = \{v_1, \ldots, v_n\}$ and edge set $E = \{e_1, \ldots, e_m\} \subseteq V \times V$. Each edge $e \in E$ is associated with an integer *weight* $w(e) > 0$. We assume that G is *connected* and *simple* (i.e. there exist neither self-loops nor multiple edges). A *forest* is an acyclic subgraph of G, and a *tree* is a connected forest. For a tree T, its weight $w(T)$ is defined as the sum of the weights of constituent edges. A tree is a *spanning tree* if it covers all the nodes of G, and a *minimum spanning tree* is a spanning tree with minimum weight. Throughout the paper, this weight is denoted as $z^\star$, and by MST we denote an arbitrary spanning tree of this weight. (Later we consider spanning trees in some limited framework, where we might have minimum spanning trees with weights larger than $z^\star$. Such a tree is not an MST in our terminology.) Efficient algorithms are well known to find an MST in an undirected graph [1,6,10]. However, there may be more than one MST, and we are concerned with the following problem.

P : List all the MSTs in graph G.

Related to this problem are the algorithms to list all the spanning trees [5,9,11], do the same in non-decreasing order of cost [3,13] and find K-shortest spanning trees in a graph [2,8]. Indeed, P may be solved by finding all the spanning trees, or more preferably by applying a K-shortest

*Corresponding author. Email: yamada@nda.ac.jp

ISSN 0020-7160 print/ISSN 1029-0265 online
© 2010 Taylor & Francis
DOI: 10.1080/00207160903329699
http://www.informaworld.com

spanning tree algorithm with sufficiently large K and truncating its execution as soon as we have a spanning tree of weight larger than $z^\star$. Unfortunately, the latter requires $O(K)$ memory space which is infeasible for problems with many MSTs. In this article, we give an algorithm, and its refinement, that lists all, and only, MSTs. These algorithms require $O(m)$ space.

2. A prototype enumeration algorithm

Let $T = \{e^1, \ldots, e^{n-1}\}$ be an arbitrary MST in G, which can be obtained by any standard MST algorithms. Then, following the general scheme for solving enumeration problems in combinatorial optimization framework [4,7], we make use of this MST to divide P into the following set of mutually disjoint subproblems ($i = 1, \ldots, n-1$).

$P(\{e^1, \ldots, e^{i-1}\}, \{e^i\})$: List all the MSTs that contain $e^1, \ldots, e^{i-1}$, but do not contain e^i.

More generally, for a forest $F = \{e^1, \ldots, e^k\}$ in G and a set of edges $R \subseteq E$ that is disjoint with F (i.e. $F \cap R = \emptyset$), a spanning tree T is (F, R)-*admissible* if it contains all edges of F, but does not contain those of R. That is, T is (F, R)-admissible if and only if $F \subseteq T$ and $R \cap T = \emptyset$. Here we use T to represent a spanning tree, as well as the set of edges included in that tree. F and R are the sets of *fixed* and *restricted* edges, respectively, and we pose the following problem.

$P(F, R)$: List all the MSTs which are (F, R)-admissible.

Clearly, P is identical to $P(\emptyset, \emptyset)$. We note that an (F, R)-admissible spanning tree of the minimum weight can be easily obtained by slightly modifying the standard MST algorithms. Let An_MST(F, R) denote the algorithm to do this, that is it accepts F and R as inputs and returns an (F, R)-admissible spanning tree $T^\star(F, R)$ of the minimum weight $z^\star(F, R)$. Kruskal's or Prim's algorithm [6,10] can be easily tailored for this purpose. If no (F, R)-admissible spanning tree exists, we take the convention of writing $z^\star(F, R) = \infty$.

Then, if $z^\star(F, R) > z^\star$, no MST can be (F, R)-admissible, and hence we *terminate* the subproblem $P(F, R)$. Otherwise, we output $T^\star(F, R)$ as an MST, and make use of this tree to introduce a set of subproblems in the following way. Let $k := |F|$ and the tree be represented as $T^\star(F, R) = F \cup \{e^{k+1}, \ldots, e^{n-1}\}$. For $i = k+1, \ldots, n-1$, we define

$$F_i := F \cup \{e^{k+1}, \ldots, e^{i-1}\}, \quad R_i := R \cup \{e^i\}. \tag{1}$$

Here, in case $i = k+1$, we interpret $\{e^{k+1}, \ldots, e^{i-1}\} = \emptyset$ and $F_i = F$. Then, $P(F, R)$ is divided into a set of mutually disjoint subproblems $P(F_i, R_i)$ ($i = k+1, \ldots, n-1$), and the problem is solved if we solve all these subproblems. Thus, a prototype algorithm can be constructed in the *divide and conquer* paradigm [12] as follows.

```
ALGORITHM  All_MST(F,R)
Comment: $F$ is a forest in $G$, $R \subseteq E$ and $F \cap R = \emptyset$.

Step 1: Apply An_MST$(F, R)$ to find $T^\star(F, R)$ and $z^\star(F, R)$.
Step 2: If $z^\star(F, R) > z^\star$, return.
Step 3: Output $T^\star(F, R)$, and for $i = k+1, \ldots, n-1$ do
            Define $(F_i, R_i)$ by (1), and call All_MST$(F_i, R_i)$.
        endfor.
```

Example 2.1 Consider the graph G of Figure 1, where edge weights are shown in *italic*. We start from $P_0 := P(\emptyset, \emptyset)$, and obtain an MST with $z^\star = 8$ (Step 1). From P_0, 5 children are generated (Step 3) as shown in Figure 2, and in total we obtain 6 MSTs after examining 17 subproblems. Due to the recursive nature of the algorithm, the subproblems are visited and numbered in a *depth-first*

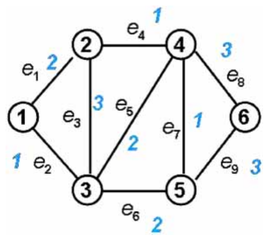


Figure 1. Graph for Example 1.

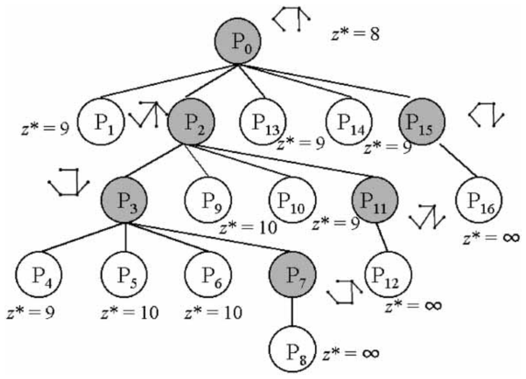


Figure 2. Behaviour of All_MST.

fashion. Here, the subproblems where MSTs are found are shaded with the corresponding MST shown at each shoulder. Details of the subproblems are summarized in Table 1. Here, in each row we show (F, R) that defines subproblem P_i, its *parent*, an (F, R)-admissible minimum spanning tree (if one exists) and the corresponding weight $z^\star(F, R)$ of that tree. Underlined edges represent the set of *newly fixed* (resp. *restricted*) edges at each subproblem, which is denoted as ΔF (resp.

Table 1. Subproblems generated from All_MST.

Subproblems	Parent	F	R	MST	$z^\star(F, R)$
P_0	–	$\emptyset$	$\emptyset$	e_2, e_1, e_4, e_7, e_8	8
P_1	P_0	$\emptyset$	$\underline{e_2}$	–	9
P_2	P_0	$\underline{e_2}$	$\underline{e_1}$	$[e_2], e_5, e_4, e_7, e_8$	8
P_3	P_2	e_2	$e_1, \underline{e_5}$	$[e_2], e_6, e_7, e_4, e_9$	8
P_4	P_3	e_2	$e_1, e_5, \underline{e_6}$	–	9
P_5	P_3	$e_2, \underline{e_6}$	$e_1, e_5, \underline{e_7}$	–	10
P_6	P_3	$e_2, \underline{e_6}, \underline{e_7}$	$e_1, e_5, \underline{e_4}$	–	10
P_7	P_3	$e_2, \underline{e_6}, \underline{e_7}, \underline{e_4}$	$e_1, e_5, \underline{e_9}$	$[e_2, e_6, e_7, e_4], e_8$	8
P_8	P_7	$e_2, e_6, e_7, 4_4$	$e_1, e_5, e_9, \underline{e_8}$	–	∞
P_9	P_2	$e_2, \underline{e_5}$	$e_1, \underline{e_4}$	–	10
P_{10}	P_2	$e_2, \underline{e_5}, \underline{e_4}$	$e_1, \underline{e_7}$	–	9
P_{11}	P_2	$e_2, \underline{e_5}, \underline{e_4}, \underline{e_7}$	$e_1, \underline{e_8}$	$[e_2, e_5, e_4, e_7], e_9$	8
P_{12}	P_{11}	e_2, e_5, e_4, e_7	$e_1, e_8, \underline{e_9}$	–	∞
P_{13}	P_0	$\underline{e_2}, \underline{e_1}$	$\underline{e_4}$	–	9
P_{14}	P_0	$\underline{e_2}, \underline{e_1}, \underline{e_4}$	$\underline{e_7}$	–	9
P_{15}	P_0	$\underline{e_2}, \underline{e_1}, \underline{e_4}, \underline{e_7}$	$\underline{e_8}$	$[e_2, e_1, e_4, e_7], e_9$	8
P_{16}	P_{15}	e_2, e_1, e_4, e_7	$e_8, \underline{e_9}$	–	∞

ΔR) for later use. In the column 'MST', edges within braces are the fixed edges, and hyphen(-) shows that the subproblem is terminated because $z^\star(F, R) > z^\star$.

The computational complexity of the above algorithm can be evaluated as follows. Let N denote the number of MSTs included in G, and $T_{\mathrm{MST}}(n, m)$ be the time required to solve an MST problem for a graph with n nodes and m edges. Note that each subproblem, including an MST, produces at most $n - 1$ children. So the total number of subproblems generated in All_MST is at most $Nn - N + 1 \approx Nn$. In each subproblem, we solve an MST problem, which requires $O(T_{\mathrm{MST}}(n, m))$ time. Thus, the total time complexity is $\mathrm{O}(NnT_{\mathrm{MST}}(n, m))$.

To access the space complexity, consider the tree of subproblems as shown in Figure 2. We note that the maximum height of the tree is at most m, since at each branch at least one edge is either fixed or restricted. While solving $P(F, R)$, we only need to keep the differential information $(\Delta F', \Delta R')$ in memory for all ancestors $P(F', R')$ of $P(F, R)$. From these incremental information, we can reconstruct (F, R), and to keep these in memory $\mathrm{O}(m)$ space suffices. Thus, we have the following.

THEOREM 2.2 *The time and space complexities of All_MST are $O(NnT_{MST}(n, m))$ and $O(m)$, respectively.*

Remark 2.3 A simple implementation of Kruskal's algorithm [6] runs in $T_{\mathrm{MST}}(n, m) = \mathrm{O}(mn)$ time, but by using sophisticated data structure such as Fibonacci heap, this can be improved to $T_{\mathrm{MST}}(n, m) = \mathrm{O}(m + n \log n)$ [1]. Therefore, the time complexity of All_MST is

$$\mathrm{O}(N(mn + n^2 \log n)). \tag{2}$$

3. A cut-set-based algorithm

Let T be an MST of G, and e an arbitrary edge of T. Deleting e from T divides the tree into two connected components with vertexes sets, say V_1 and V_2. We introduce the *cut-set* induced by $e \in T$ as the set of edges with one endpoint in V_1 and the other in V_2. If we define this as $\mathrm{Cut}(e) := \{e' \in E \mid e' \in (V_1 \times V_2) \cup (V_2 \times V_1)\}$, the following Proposition is clear from the 'cut optimality condition' [1, Theorem 13.1] for MST.

PROPOSITION 3.1 *Let T be an MST and $e \in T$. Then, for an arbitrary edge $e' \in \mathrm{Cut}(e)$,*

$$w(e') \geq w(e). \tag{3}$$

That is, $e \in T$ is an edge of the minimum weight in $\mathrm{Cut}(e)$. For a pair of edges $e \in T$ and $e' \in \mathrm{Cut}(e)\backslash\{e\}$, $T \cup \{e'\}\backslash\{e\}$ defines another spanning tree, which is denoted as $T \cup e'\backslash e$ for simplicity. Let $G^{(e)}$ denote the graph obtained from G by deleting e. We then have the following.

THEOREM 3.2 *Let T be an MST, $e \in T$ and $e' \neq e$ be* second minimum *in weight in* $\mathrm{Cut}(e)$. *Then, $T' := T \cup e'\backslash e$ is a minimum spanning tree in $G^{(e)}$.*

Proof Suppose that T' is not a minimum spanning tree in $G^{(e)}$. Then, by the 'path optimality condition' [1, Theorem 13.3] there exists a pair of edges $c \notin T'$ and $d \in T'$ such that (1) $T'' := T' \cup c\backslash d$ is a spanning tree and (2) $w(c) < w(d)$. We note that $c \in \mathrm{Cut}(e)$, otherwise $\tilde{T} := T \cup c\backslash d$ is a spanning tree with $w(\tilde{T}) < z^\star$, contradicting that T is an MST. Thus, we have

$$w(e) \leq w(e') \leq w(c) < w(d). \tag{4}$$

Let the cycles included in $T^\star \cup \{e'\}$, $T^\star \cup \{c\}$ and $T' \cup \{c\}$ be C_0, C_1 and C_2, respectively. Since $C_2 = C_0 \oplus C_1$ (in Boolean sense) and $d \in C_2$, we have $d \in C_0$ or $d \in C_1$. Define

$$\hat{T} := \begin{cases} T^\star \cup e' \backslash d & \text{if } d \in C_0, \\ T^\star \cup c \backslash d & \text{if } d \in C_1. \end{cases} \tag{5}$$

Then, $\hat{T}$ is a spanning tree with $w(\hat{T}) < z^\star$, which is a contradiction. ■

The following Corollaries are easily proved.

Corollary 3.3 *Let T be an MST, $e \in T$ and $e' \neq e$ be second minimum in weight in* Cut(e). *If $w(e) = w(e')$, then $T \cup e' \backslash e$ is an MST; otherwise, no MST exists in $G^{(e)}$.*

Corollary 3.4 *If all the edges of G are of distinct weights, then MST is unique.*

Let T be an MST and $e \in T$. We call $\tilde{e} \in \text{Cut}(e)$ a *substitute* of e if $\tilde{e} \neq e$ and $w(\tilde{e}) = w(e)$. $S(e)$ stands for the set of all substitutes of e, that is,

$$S(e) := \{\tilde{e} \in \text{Cut}(e) | \tilde{e} \neq e, w(\tilde{e}) = w(e)\}. \tag{6}$$

On the basis of the above theorem, we can modify All_MST in the following way. At each subproblem, in addition to the pair of sets (F, R) representing the fixed and restricted edges, we assume that an (F, R)-admissible MST exists. Let this tree be T. In the following algorithm, in the subproblem $P(F, R)$, for each $e^i \in T \backslash F$, we look for a substitute $\tilde{e}^i \in S(e)$. If this is found, we obtain $T \cup \tilde{e}^i \backslash e^i$ as a new MST, and use this to generate a sub-subproblem. Then, the algorithm is described as follows.

Algorithm *All_MST*$_1(F, R, T)$
Comment: T is an (F, R)-admissible MST, which is written as $T = F \cup \{e^{k+1}, \dots, e^{n-1}\}$, with $F = \{e^1, \dots, e^k\}$.

```
Step 1: For $i = k+1, \dots, n-1$, do the following:
          • Find the cut-set Cut$(e^i)$.
          • Find, if one exists, a substitute $\tilde{e}^i \in S(e^i)$.
Step 2: For $i = k+1, \dots, n-1$, if $\tilde{e}_i$ exists do the following:
          • Set $T_i := T \cup \tilde{e}^i \setminus e^i$ and output $T_i$. {Comment: A new MST is found}
          • Set $F_i := F \cup \{e^{k+1}, \dots, e^{i-1}\}$ and $R_i := R \cup \{e^i\}$.
          • Call All_MST$_1(F_i, R_i, T_i)$ recursively.
```

Example 3.5 For the graph of Figure 1, Figure 3 shows the behaviour of All_MST$_1$, where details of the generated subproblems are summarized in Table 2. In addition to the information given in Table 1, here is a substitute for the newly restricted edges (ΔR) in each subproblem. In this algorithm, each subproblem generated includes an (F, R)-admissible MST as shown in Figure 3, and newly fixed edges (ΔF) are underlined in Table 2.

For the computational complexity of this algorithm, we have the following.

Theorem 3.6 *The running time of All_MST$_1$ is $O(Nmn)$.*

Proof Note that Cut(e^i) can be found in O(m) time. At each subproblem, this is repeated at most n times. ■

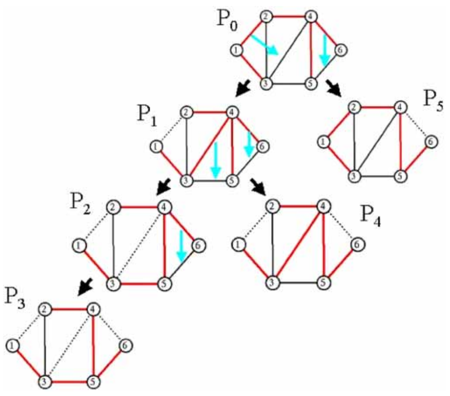


Figure 3. Behaviours of All_MST$_1$.

Table 2. Subproblems generated for the graph of Figure 1.

Subproblems	Parent	F	R	MST	Substitute
P_0	–	$\emptyset$	$\emptyset$	e_2, e_1, e_4, e_7, e_8	–
P_1	P_0	$\underline{e_2}$	$\underline{e_1}$	$[e_2], e_5, e_4, e_7, e_8$	e_5
P_2	P_1	e_2	$e_1, \underline{e_5}$	$[e_2], e_6, e_4, e_7, e_8$	e_6
P_3	P_2	$e_2, \underline{e_6}, \underline{e_4}, \underline{e_7}$	$e_1, e_5, \underline{e_8}$	$[e_2, e_6, e_4, e_7], e_9$	e_9
P_4	P_1	$e_2, \underline{e_5}, \underline{e_4}, \underline{e_7}$	$e_1, \underline{e_8}$	$[e_2, e_5, e_4, e_7], e_9$	e_9
P_5	P_0	$\underline{e_2}, \underline{e_1}, \underline{e_4}, \underline{e_7}$	$\underline{e_8}$	$[e_2, e_1, e_4, e_7], e_9$	e_9

4. A further improvement

In the previous section, it took O(m) time, in the worst case, to find a substitute $\tilde{e}^i$ for each $e^i \in T$. This computation was carried out for each e^i from scratch, and the computation of all substitutes took O(mn) time. In this section, we present an algorithm that finds $\tilde{e}^i$ one-by-one, from the reduced set of possible cut-set edges for e^i, and after completing step i this information is carried on to the next step. We try to make the size of this set as small as possible, and as a result, in total we obtain the set of all substitutes $\{\tilde{e}^i \mid e^i \in T \backslash F\}$ in O$(m \log n)$ time, instead of O(mn) in All_MST$_1$.

To accomplish this, at each subproblem $P(F, R)$ with an (F, R)-admissible MST T, we renumber vertexes in *postorder* fashion [12] as we traverse T from an arbitrary 'root' vertex. Let the vertexes thus renumbered be $\{v^i | i = 1, \ldots, n\}$. Then, T is a tree rooted at v^n, and by e^i we denote the tree edge connecting v^i to its parent vertex. (See Figure 4, where T is shown in bold). Associated with v^i is an interval $[\underline{\sigma}_i, \overline{\sigma}_i]$, which represents the set of descendants of this vertex, that is,

$$j \in [\underline{\sigma}_i, \overline{\sigma}_i] \Leftrightarrow v^j \text{ is a descendant of } v^i \text{ in tree } T \text{ rooted at } v^n. \qquad (7)$$

Let $E^i := \{(v^i, v^j) \in E | (v^i, v^j) \notin T\}$ be a set of non-tree edges incident on node v^i, and Q be a set of elements of the form $(w, v, v') \in Z \times V \times V$. Here, $(w, v, v') \in Q$ means that the edge $e = (v, v') \in E$ has weight $w(e) = w$, and is a *candidate* of a cut-set edge at this, and subsequent, stage. We call Q the set of *quasi-cuts*, and assume that it is *lexicographically* ordered with respect to w and v. For $e^i \in T \backslash F$, we make use of Q to find its substitute $\tilde{e}^i$ and update it for the next i in the following way.

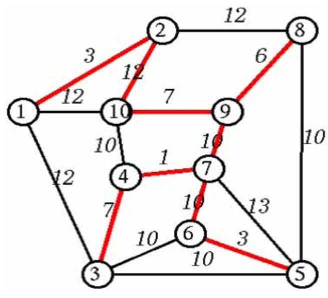


Figure 4. Graph for Example 3.

Algorithm *Substitute*(F, R, T)

```
Step 1: Set $Q := \emptyset$.
Step 2: For $i = 1$ to $n - 1$ do the following.
        (1) For $\forall e = (v^i, v^j) \in E^i \setminus R$ do either of the following.
            a) If $j < \underline{\sigma}_i$, {Comment: Reverse the direction.}
               • If $(w(e), j, i) \in Q$, delete it from $Q$.
               • Insert $(w(e), i, j)$ into $Q$.
            (b) If $j \in [\underline{\sigma}_i, \overline{\sigma}_i]$,
               • If $(w(e), j, i) \in Q$, delete it from $Q$.
            (c) If $j > \overline{\sigma}_i$,
               • Insert $(w(e), i, j)$ into $Q$.
        (2) If $e^i \notin F$ do the following.
            (a) Find $(w, i', j') \in Q$ such that $w = w(e^i)$ and $i' \in [\underline{\sigma}_i, \overline{\sigma}_i]$
            (b) If such an $(w, i', j')$ is found with $j' \in [\underline{\sigma}_i, \overline{\sigma}_i]$,
               • Delete $(w, i', j')$ from $Q$, and go to (2)-(a).
            (c) If such an $(w, i', j')$ is found with $j' \notin [\underline{\sigma}_i, \overline{\sigma}_i]$,
               • Set $\tilde{e}^i := (v^{i'}, v^{j'})$. {Comment: Substitute for $e^i$ found.}
```

In Step 2-(1), we update Q by including the edges of $E^i \setminus R$ as a candidate of cut-set edges for e^i and subsequent tree edges. Specifically, in Step 2-(1)-(c) edge (v^i, v^j) is included in Q if this is an edge from v^i to v^j with $j > i$. On the other hand, edges from e^i to its descendants are deleted [Step 2-(1)-(b)], since from now on these can no longer be a cut-set edge. If we have an edge coming into v^i from a previously found vertex v^j, the direction is reversed [Step 2-(1)-(a)].

Next, in Step 2-(2) we look for a substitute $\tilde{e}^i$ of e^i included in Q. If we have an edge with the weight identical to $w(e^i)$ and emerging from a vertex within the interval $[\underline{\sigma}_i, \overline{\sigma}_i]$ [Step 2-(2)-(c)], this is a substitute for e^i, and thus we are done. If this is an edge from the above interval to the same interval, we delete this edge from Q, and repeat Step 2-(2) again.

Example 4.1 Consider the graph of Figure 4 with an MST T shown in thick lines. Nodes and edges are numbered in the postorder fashion as traversed along T from node v^{10}. Table 3 shows $w(e^i)$, $[\underline{\sigma}_i, \overline{\sigma}_i]$ and Q at each stage. The elements superscribed with $\circ$, $\times$ and ! show, respectively, the newly found, deleted and substitute for e^i. The elements associated with † shows that the edge direction is reversed here.

Let All_MST$_2$ be the algorithm obtained from All_MST$_1$ by replacing Step 1 as follows.
Step 1†: Execute Substitute(F, R, T).

Table 3. Behaviour of *Substitute* for the graph of Figure 4.

i	$w(e^i)$	$[\underline{\sigma}_i, \overline{\sigma}_i]$	Q
1	3	[1,1]	(12:1, 3)$^{\circ}$ (12:1, 10)$^{\circ}$
2	12	[1,2]	(12:1, 3)$^{!}$ (12:1,10) (12:2, 8)$^{\circ}$
3	7	[3,3]	(10:3, 5)$^{\circ}$ (10:3, 6)$^{\circ}$ (12:1, 10) (12:2, 8) (12:3, 1)†
4	1	[3,4]	(10:3, 5) (10:3, 6) (10:4,10)$^{\circ}$ (12:1,10) (12:2, 8) (12:3, 1)
5	3	[5,5]	(10:3, 6) (10:4,10) (10:5, 3)† (10:5, 8)$^{\circ}$ (12:1,10) (12:2, 8) (12:3, 1) (13:5, 7)$^{\circ}$
6	10	[5,6]	(10:4, 10) (10:5, 3)$^{!}$ (10:5, 8) (10:6,3)† (12:1, 10) (12:2, 8) (12:3, 1) (13:5, 7)
7	10	[3,7]	(10:4, 10)$^{!}$ (10:5, 3)$^{\times}$ (10:5, 8) (10:6, 3)$^{\times}$ (12:1, 10) (12:2, 8) (12:3, 1)
8	6	[8,8]	(10:4,10) (10:8, 5)† (12:1,10) (12:3, 1) (12:8, 2)†
9	7	[3,9]	(10:4,10) (10:8, 5) (12:1, 10) (12:3, 1) (12:8, 2)

Computational complexity of All_MST$_2$ can be evaluated as follows. First, note that Q includes at most m elements, and for each non-tree edges at most two *Insert*s and two *Delete*s are executed. For each tree edge at most one *Find* is executed. Since Q is an ordered set, each of *Insert*, *Delete* and *Find* can be done in $\mathrm{O}(\log m)$ time, provided that Q is organized as an appropriate binary tree (such as the Adelson-Velskii and Landis tree (AVL-tree) [12]). Thus, in total *Substitutes* can be done in $\mathrm{O}(m \log m) = \mathrm{O}(m \log n)$ time. All other computations, such as traversing G along T, renumbering V in postorder fashion and finding intervals $[\underline{\sigma}_i, \overline{\sigma}_i]$, can be accomplished in $\mathrm{O}(m)$ time. Thus, we have the following.

THEOREM 4.2 *The running time of All_MST$_2$ is $O(Nm \log n)$.*

5. Numerical experiments

We have implemented All_MST and All_MST$_1$ in C language and conducted a series of numerical experiments on an IBM RS/6000 44P MODEL 270 workstation (375 MHz). Throughout the experiments, All_MST and All_MST$_1$ found the same number of MSTs.

5.1 *Complete graphs with constant edge weights*

First, we consider the complete graph K_n with edge weights fixed at $w(e) \equiv 1$ for all $e \in E$. In this case, all the spanning trees are MSTs, and the total number of spanning trees can be shown [9] to be

$$N = n^{n-2}. \tag{8}$$

Table 4. Result of experiments (complete graphs, constant edge weights).

		All_MST		All_MST$_1$
Graph	N	#sub	CPU	CPU
K_3	3	6	0.00	0.00
K_4	16	29	0.00	0.00
K_5	125	212	0.00	0.00
K_6	1296	2117	0.02	0.01
K_7	16,807	26,830	0.22	0.08
K_8	262,144	412,015	3.60	1.35
K_9	4,782,969	7,433,032	60.74	26.99
K_{10}	100,000,000	154,076,201	1694.27	616.10

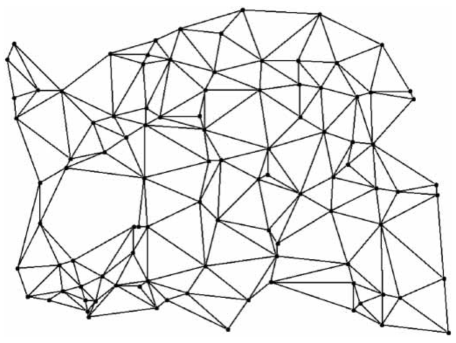


Figure 5. Planar graph $P_{100\times 260}$.

Table 4 gives the result of experiments for K_n. The number of spanning trees (N), the number of generated subproblems (#sub) and CPU time in seconds are shown. Both All_MST and All_MST$_1$ compute N_n correctly for $n \leq 10$, and All_MST$_1$ is approximately 2.5 times faster than All_MST.

5.2 *Planar graphs with random edge weights*

Let $P_{n\times m}$ be a planar graph with n nodes and m edges (See Figure 5 for $P_{100\times 260}$), and the edge weights are both randomly and uniformly distributed over $[1, 10^L]$ $(L = 2, 3)$.

Table 5 summarizes the result of the experiments for these graphs. Each row is the average of 10 independent runs. In this case, All_MST$_1$ runs 30 – 60 times faster than All_MST. The number of MSTs is much smaller in the case where edge weights are distributed over [1, 1000] $(L = 3)$ than in the case $L = 2$. This is explained as follows. In the latter case, the chance of having edges of identical weights are much larger than in the former case.

5.3 *Complete graphs with random edge weights*

Table 6 gives the result for complete graphs with random edge weights uniformly distributed over $[1, 10^L]$ $(L = 2, 3)$. Again each row is the average of 10 independent runs. In this case, All_MST$_1$ is faster than All_MST, but only 2–4 times, as opposed to 50–70 times in the case of planar graphs. This may be explained by considering the ratio of the number of subproblems generated in All_MST over the total number of MSTs (i.e. #sub/N in Tables 5 and 6). For planar graphs this ratio is approximately 20–50 in the case of $L = 2$ and 100–500 for $L = 3$. Similarly, for complete graphs the ratio is 3–10 $(L = 2)$ and 10–40 $(L = 3)$, respectively. Thus, the ratio is larger for planar graphs than for complete graphs, meaning that All_MST generates more

Table 5. Result of experiments (planar graphs, random edge weights).

				All_MST		All_MST$_1$
L	Graph	$z^\star$	N	# sub	CPU	CPU
2	$P_{20\times 46}$	459.1	1.2	20.6	0.00	0.00
	$P_{50\times 127}$	1110.9	1.2	53.2	0.01	0.00
	$P_{100\times 260}$	2224.7	2.2	119.3	0.04	0.00
	$P_{200\times 560}$	4211.6	18.0	1108.3	1.34	0.06
	$P_{400\times 1120}$	8367.9	147.0	6995.3	29.68	0.85
	$P_{600\times 1680}$	12,615.5	1921.6	65,154.7	587.30	14.16
	$P_{800\times 2240}$	17,065.8	15,592.8	603,647.5	5604.70	185.15
3	$P_{20\times 46}$	4080.5	1.0	20.0	0.00	0.00
	$P_{50\times 127}$	10,535.3	1.1	52.9	0.01	0.00
	$P_{100\times 260}$	21,479.5	1.1	107.4	0.04	0.00
	$P_{200\times 560}$	40,590.2	1.6	239.0	0.30	0.01
	$P_{400\times 1120}$	81,766.4	1.9	518.8	2.24	0.07
	$P_{600\times 1680}$	122,550.5	3.1	1116.5	10.46	0.23
	$P_{800\times 2240}$	162,966.2	2.0	1169.3	20.83	0.36
	$P_{1000\times 2800}$	204,548.9	3.2	1673.4	43.28	0.67

Table 6. Result of experiments (complete graphs, random edge weights).

				All_MST		All_MST$_1$
L	Graph	$z^\star$	N	# sub	CPU	CPU
2	K_{20}	120.9	3.6	38.2	0.00	0.00
	K_{40}	141.1	11.0	112.8	0.02	0.01
	K_{60}	152.5	1669.2	8840.1	3.15	1.57
	K_{80}	165.9	1,494,553.0	4,647,056.0	3236.55	1511.29
3	K_{20}	1114.2	1.1	20.2	0.00	0.00
	K_{40}	1224.0	2.5	59.9	0.01	0.01
	K_{60}	1260.5	4.1	101.8	0.04	0.02
	K_{80}	1253.0	3.1	136.9	0.10	0.05
	K_{100}	1264.1	9.3	227.4	0.26	0.12
	K_{120}	1251.6	14.3	361.4	0.59	0.28
	K_{140}	1281.3	123.2	1768.9	3.78	1.88
	K_{160}	1274.3	337.4	7966.7	24.49	11.46
	K_{180}	1288.8	3980.0	63,892.2	287.48	136.65
	K_{200}	1296.4	7434.4	145,946.9	871.92	427.49

'redundant' subproblems in planar graphs than in complete graphs. Since All_MST$_1$ does not produce such an redundant subproblem, All_MST$_1$ is more superior to All_MST in planar graphs than in complete graphs.

6. Conclusion

We have developed an algorithm to list all the minimum spanning trees in an undirected graph, and explored some properties of cut-sets. Furthermore, based on these, we have constructed an improved algorithm, which runs in $O(Nm \log n)$ time and $O(m)$ space. We have not implemented All_MST$_2$, but this is expected to be better than All_MST or All_MST$_1$ for larger problems. Numerical experiment of this part is left for our future work.

Acknowledgements

The authors are grateful to two anonymous referees and and a Editor for their constructive criticisms and suggestions. Numerous comments from Professor Kazuo Ouchi were substantial in improving the quality of English in this paper, for which we express deep appreciation.

References

[1] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice-Hall, Englewood Cliffs, NJ, 1993.
[2] P.M. Camerini, L. Fratta, and F. Maffioli, *Efficient methods for ranking trees*, Proceedings of the 3rd International Symposium on Network Theory, Split, Yugoslavia, 1975, pp. 1–10.
[3] H.N. Gabow, *Two algorithms for generating weighted spanning trees in order*, SIAM J. Comput. 6 (1977), pp. 139–150.
[4] H.W. Hamacher and M. Queyranne, *K-best solutions to combinatorial optimization problems*, Ann. Oper. Res. 4 (1985), pp. 123–143.
[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] J.B. Kruskal, *On the shortest spanning subtree of a graph and the traveling salesman problem*, Proc. Am. Math. Soc. 7 (1956), pp. 8–50.
[7] E.L. Lawler, *A procedure for computing the K best solutions to discrete optimization problems and its application to the shortest path problem*, Manag. Sci. 18 (1972), pp. 401–405.
[8] T. Matsui, *A flexible algorithm for generating all the spanning trees in undirected graphs*, Algorithmica, 18 (1997), pp. 530–543.
[9] G.J. Minty, *A simple algorithm for listing all the trees of a graph*, IEEE Trans. Circuit Theory CT-12 (1965), p. 120.
[10] R.C. Prim, *Shortest connection networks and some generalizations*, Bell Syst. Tech. J. 36 (1957), pp. 1389–1401.
[11] R.C. Read and R.E. Tarjan, *Bounds on backtrack algorithms for listing cycles, paths, and spanning trees*, Networks 5 (1975), pp. 237–252.
[12] R. Sedgewick, *Algorithms in C*, 3rd ed., Addison-Wesley, Reading, MA, 1998.
[13] K. Sörensen and G.K. Janssens, *An algorithm to generate all spanning trees of a graph in order of increasing cost*, Pesqui. Oper., 25 (2005), pp. 219–229.

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.

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

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 $i \langle j$. The sequence $s_1, s_2, \ldots$ 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

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

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

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

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-r-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

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

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 $c\left[s(M_d)\right] = \min\limits_{i=1..e} \left\{ c\left[s(M_i)\right] \right\}$.

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

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

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

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

Then $\{M_1, \ldots, M_{n-1}\}$ 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.

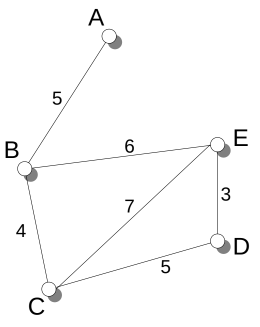


Figure 1 – Example graph G.

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:

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

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

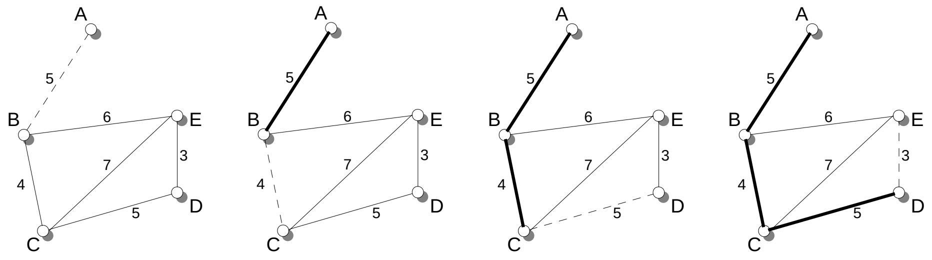


Figure 2 – Partitions $P_1, \ldots, P_4$.

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

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

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(P₃)*, we obtain partitions P_{31} and P_{32}.

$$\begin{aligned} P_{31} &= \left\{(A,B),(B,C),\left(\overline{C,D}\right),\left(\overline{B,E}\right)\right\} \\ P_{32} &= \left\{(A,B),(B,C),(B,E),\left(\overline{C,D}\right),\left(\overline{E,D}\right)\right\} \end{aligned} .$$

Graphically, the partition is represented in Figure 3.

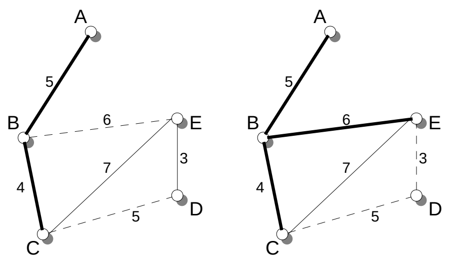


Figure 3 – Partitions P_{31} and P_{32} .

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 $\neq \emptyset$ 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.

Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in DISCRETE MATHEMATICS

Electronic Notes in Discrete Mathematics 53 (2016) 343–354

www.elsevier.com/locate/endm

An Algorithm for k^{th} Minimum Spanning Tree

Amal P M[1] Ajish Kumar K S[2]

Department of Computer Science & Engineering
College of Engineering
Trivandrum, India

Abstract

Minimum Spanning Tree problem is a well-known problem in graph theory. There are many polynomial time algorithms, for finding minimum spanning tree of an undirected connected graph. The k^{th} minimum spanning tree problem is a variation of minimum spanning tree problem that finds k^{th} minimum spanning tree of an undirected connected graph, for some integer k. An algorithm of time complexity $O(k|E||V|log\ k)$ and space complexity $O(k.|V| + |E|)$ for finding k^{th} minimum spanning tree is proposed in this paper. This algorithm is used to solve degree constrained minimum spanning tree problem, an NP-complete problem, with exponential time complexity.

Keywords: k^{th} minimum spanning tree

1 Introduction

Computation of a Minimum Spanning Tree (MST) is one of the fundamental problems in graph theory. Given a weighted undirected connected graph

[1] Email: amalamalpm@gmail.com

[2] Email: ajish@cet.ac.in

http://dx.doi.org/10.1016/j.endm.2016.05.029

G(V,E), the minimum spanning tree of G is a tree spanning G which has the minimum weight among all possible spanning trees.

k^{th} minimum spanning tree of a graph is a spanning tree with k^{th} minimum weight among all the possible spanning trees, for some input parameter k.

Spanning subgraph of a graph $G(V, E)$ is a graph $G'(V, E')$ where $V' = V$ and $E' \subseteq E$. Tree is connected acyclic graph. Spanning tree of a Graph G is spanning subgraph of G, which is a tree. There may exist one or more spanning tree for a connected graph. Weight of a graph is the total weight of edges. A minimum spanning tree of graph $G(V, E)$ is a spanning tree of G with weight less than or equal to the weight of every other spanning tree of G. The spanning tree with 2^{nd} minimum weight is called 2^{nd} minimum spanning tree. Thus for a k^{th} minimum spanning tree there exists at least $k-1$ spanning trees with weight less than or equal to weight of that spanning tree.

Algorithms for k^{th} minimum spanning tree can be used to find solutions for constrained spanning tree problems like degree constrained spanning tree, Steiner tree etc., which are known NP-Hard problems. Degree constrained spanning tree of an undirected connected graph is a spanning tree with minimum weight, whose vertices have degree at most Δ, where Δ is the degree constraint.

Here we are proposing a new algorithm for k^{th} minimum spanning tree of an undirected connected graph. The rest of the paper organized as follows. Section 2 discusses some of the previous algorithms and studies related to minimum spanning trees, k^{th} minimum spanning tree and degree constrained minimum spanning tree. The proposed algorithm to find k^{th} minimum spanning tree is explained in the section 3. An application of k^{th} minimum spanning tree to find degree constrained minimum spanning treeis explained in section 4. Section 5 presents a summary and the conclusions.

2 Literature Review

There are Several polynomial time algorithms for the Minimum Spanning Tree problem using greedy strategy. In 1926 Boruvka proposed an algorithm for minimum spanning tree problem. The time complexity of this algorithm is $O(|E|\ log\ |V|)$.

Prim's algorithm[8] is discovered in 1957. It builds a partial spanning tree starting from an arbitrary vertex and in each iteration it adds a vertex to this partial spanning tree which is nearest to it. This repeats until all the vertices are added to the tree. The running time of Prim's algorithm when using binary heap is $O(|E|\ log\ |V|)$.

In 1956 Joseph Kruskal developed an algorithm [9] for MST Problem, which selects edges of graph in non-decreasing order of weight. It repeatedly adds next nearest edge that doesn't forms a cycle until $|V|-1$ edges are added.

2.1 Degree Constrained Minimal Spanning Tree

The Degree Constrained Minimal Spanning Tree (DCMST) problem is a variation of minimum spanning tree problem with an additional constraint over the degree of each vertex in the spanning tree. In 1979 Garey and Johnson[5] proved that in general the problem of finding degree constraint minimum spanning tree (DCMST) is NP-complete by reducing it to Hamiltonian path problem. There exist several methods to find approximate solutions of this NP-complete problem. Gabow [6] developed a branch exchange algorithm to find an approximate solution. Gavish [7] used a branch and bound heuristic which solves the problem in a graph up to 200 nodes.

2.2 k^{th} minimum spanning tree

In 1975 Harold N Gabow[1] developed an algorithm for k smallest spanning trees of a connected graph. k may be known in advance or specified as the trees are generated. The algorithm requires time $O(k|E|\alpha(|E|,|V|)+|E|log|E|)$ and space $O(k+|E|)$. Here $|V|$ is the number of vertices, $|E|$ is the number of edges, and α is Tarjans inverse of Ackermanns function which is very slow growing function.

N. Katoh, T. Ibaraki and H. Mines [2] presented an algorithm for finding k minimum spanning trees in an undirected graph with time $O(k|E| + min(|V|^2, |E|\ log\ log\ |V|))$ and space $O(k+|E|)$. It has less time complexity compared to Gabow's algorithm.

David Eppstein[3] improved the algorithms for generate k smallest spanning trees in a general graph and planar graph. The algorithm for general graphs takes time $O(|E|log\ \beta(|E|,|V|)+k^2)$. And the time required to find planar graphs improved to $O(|V|+k^2)$.

A parallel algorithm proposed in [4] for find k spanning trees from an undirected connected graph $G(V,E)$, which uses $O(|V|^2/log\ |V|)$ processors on a CREW PRAM, with time complexity of $O(T(|V|)+k\ log|V|)$. A parallel algorithm for the k-MST problem on CREW PRAM is presented in this by modifying KIM algorithm[2].

3 Algorithm for k^{th} minimum spanning tree

3.1 Basic idea

Let $G(V, E)$ be the given graph, where V is the set of vertices and E is set of edges.

- Find the minimum spanning tree T_1, using Prim's algorithm. Let W_1 be its weight. The running time of Prim's algorithm is $O(|E|\ log\ |V|)$ and it does not sort the edges for finding minimum spanning tree. So it is a better choice for dense graph than Kruskal's method.
- Then the next minimum spanning tree is constructed by a process called 'minimum edge exchange'. Here an edge $(u, v) \in E$, which not in $T_1$is added to T_1.
  - · For every spanning tree there will be one path $uv_0v_1...v_lv$ exists between every pair of vertices. So when adding edge (u, v), there creates a cycle $uv_0v_1...v_lvu$. An example in figure 1

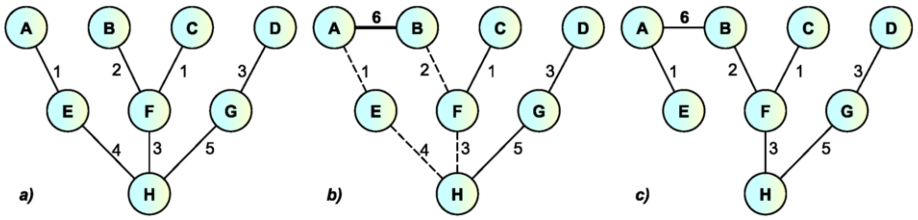


Fig. 1. Steps of edge exchange **a)** A spanning tree **b)** Adding edge (A,B) **c)** New spanning tree after removing edge (H,E)

  - · Remove the edge v_iv_j with largest weight smaller than(consider the equal weighted edges too for the first iteration) weight of (u, v) in the cycle $uv_0v_1...v_ivu$.
  - · This creates a new spanning tree T_1' that has weight W_1' more than or equal to W_1
- Repeat this edge exchange process with all other edges that are not in T_1
- Find the 2^{nd} minimum spanning tree T_2 from this set of newly generated spanning trees, which have next minimum weight.
- For finding T_3 we have to do edge exchange process in T_1 and T_2.
- Repeat this steps until we get the k^{th} minimum spanning tree T_k.

Algorithm 1 k_minimum_spanning_trees (G, k)

```
Input: Graph G(V,E) and k
Output: An array(K_MSTS) of length k, which contain k minimum spanning
  trees.
  $k^{th}$ element of array will be $k^{th}$ minimum spanning tree.
  Assumptions:
  Prims(G) return minimum spanning tree of G
  List_of_trees is a data structure which stores a list of trees
  Select_min() select a tree with minimum weight from the list
  Add(t) adds tree t into the list, only if tree t is not in the list. If the list
  contain more than k trees then it removes the tree which have largest weight
  among them.
  Remove(t) removes tree t from the list.

  MST=Prims(G) // Finding the min spanning tree of G
  List_of_trees.Add(MST)
  MST=List_of_trees.select_min() // Select tree with min weight from list
  K_MSTS[1]=MST
  for $i$=2 to $k$ do
    List_of_trees.Remove(MST)
    for each edge E which is not in MST do
      Add E to MST // so there will generate a new cycle
      if i=2 then
        // For first MST we have to consider the equal edges too
        Select edges E' from the cycle which have max weight and weight
          must be less than or equal to E
      else
        Select edges E' from the cycle which have max weight and weight
          must be less than E
      end if
      for each edge in E' do
        MST'=Remove E' from MST
        List_of_trees.Add(MST')
      end for
    end for
    MST=List_of_trees.select_min()
    K_MSTS[i]=MST
  end for
  return K_MSTS
```

3.2 Explanation

Here we are using mainly two data structures.

- K_MSTS: which is an array to store k minimum spanning trees.
- List_of_trees: A list to store k trees which can be next minimum spanning tree.

In the first step of the algorithm finds the minimum spanning tree using Prims algorithm and store that in the List_of_trees. Before starting loop we initially put the minimum spanning tree to the K_MSTS. Each iteration of the loop removes the tree with minimum weight from List_of_trees. Then we make a new set of trees which can be next minimum spanning tree by adding one edge, which is not in current minimum spanning tree, to it. When adding that edge the tree becomes a graph with $|V|$ edges, which means that the new graph contains a new cycle. Select an edge which have cost less than the added edge (consider the equal weighted edges too for the first iteration). Remove that edge and add that spanning tree to the List_of_trees. If there is more than one edge with maximum weight, create separate spanning trees by removing each of them. Repeat this step for each edge which are not added to the current minimum spanning tree. After this take the spanning tree with minimum weight from List_of_trees, which is next minimum spanning tree. Add that tree as next spanning tree in K_MSTS. And make new set of trees using this spanning tree. Repeat this process until the k^{th} minimum spanning tree is obtained.

3.3 Correctness of the algorithm

- Initialization-
  - When i=2, the loop invariant holds
    Array K_MSTS contain minimum spanning tree T_1
    List List_of_trees contain all the trees, which are obtained minimum edge exchange.
  - When i=3, the loop invariant holds
    Array K_MSTS contain T_1 and T_2
    List List_of_trees contain all the trees, which are obtained by minimum edge exchange in T_1 and T_2
- Maintenance
  - In each iteration value of i incrementing by one and loop invariants holds
    Array K_MSTS contain i-1 minimum spanning trees and
    List List_of_trees contain candidates for next minimum spanning tree

- · Then we select the minimum weighted tree T_i from this list (which will contain all the spanning trees which have chance to next minimum spanning tree). We remove it from List_of_trees and add to the K_MSTS as the i^{th} minimum spanning tree.
- · Then update List_of_trees by adding new spanning trees generated from minimum edge exchange process on T_i.

- Termination
  - · The loop terminates when i=k+1.
  - · So in this case the loop invariant K_MSTS contain k minimum spanning trees.
  - · Last tree in K_MSTS is the k^{th} minimum spanning tree, which is the required output of our algorithm.

3.4 Complexity analysis

3.4.1 Time complexity analysis - Worst case

- Prims algorithm has complexity $O(ElogV)$.
- List_of_tree will have at most k trees. Because in any time we want to store only minimum k trees. So if we use ordinary lists add, remove, select_min have complexity $O(k)$.
- To find a cycle from a graph which have only one cycle has complexity $O(V)$. (if we use depth first traversal)
- Finding edge E' for edge exchange is $O(V)$
- Number of E' in one cycle will be at most $|V| - 1$, because there is only $|V| - 1$ edges.

  $T(G(V, E), k) = |E|log|V| + k(k + (|E|)(|V| + |V|k) + k)$
  $= O(|E|log|V| + k^2 + k^2|E||V|)$
  $= O(k^2|E||V|)$.

  If we use AVL tree for storing List_of_tree, then time complexity of add, remove and search will reduce to $O(log\ k)$. Then time complexity of algorithm will reduce to
  $T(G(V, E), k) = |E|log|V| + k(log\ k + (|E|)(|V| + |V|log\ k) + log\ k)$
  $= O(|E|log|V| + k\ log\ k + k|E||V|log\ k)$
  $= O(k|E||V|log\ k)$.

3.4.2 Space complexity analysis

Algorithm k_minimum_spanning_trees (G, k) uses two additional data structures List_of_trees and K_MSTS to store spanning trees. For each tree we use space $O(|V|)$. Space needed for K_MSTS is $O(k.|V|)$. Space needed for List_of_trees is also $O(k.|V|)$ because these are storing maximum k spanning trees in anytime. If List_of_trees containing k trees and trying to add a new spanning tree to List_of_trees results in removal of the largest tree from that $k+1$ spanning trees, k trees from list and one new tree. But to store input graph we need $O(|E|)$ extra space. So the space complexity of algorithm is $O(k.|V|) + O(k.|V|) + O(|E|) = O(k.|V| + |E|)$.

3.5 Comparison with previous algorithms

Harold N Gabow's algorithm [1] requires time $O(k|E|\alpha(|E|,|V|) + |E|log|E|)$ and space $O(k + |E|)$. KIM algorithm [2] slightly faster than Gabows algorithm, and its time complexity is $O(k|E| + min(|V|^2, |E|\ log\ log\ |V|))$ and space is of the same order. David Eppstein's algorithm[3] for general graphs takes time $O(|E|log\ \beta(|E|,|V|) + k^2)$. Our proposed algorithm has time complexity $O(k|E||V|log\ k)$ which is higher than previous algorithms, and space complexity is $O(k.|V| + |E|)$.

3.6 Working of the algorithm with an example

Given a graph G(V,E) with 7 vertices and 8 edges.
$V = \{A, B, C, D, E, F, G\}$
$E = \{AB, BC, BD, CD, DE, DF, EF, FG\}$

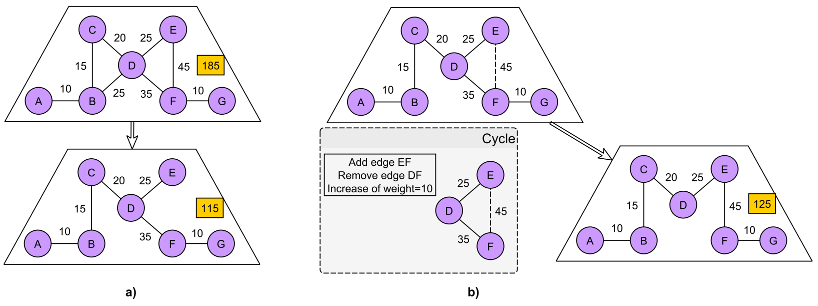


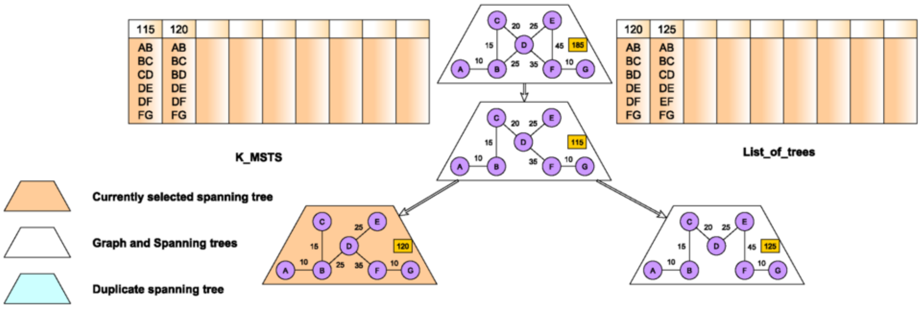
Fig. 2. Adding edge EF to minimum spanning tree

If k is 9, we have to find 9th minimum spanning tree. First find out the minimum spanning tree using Prim's algorithm.

In this example minimum spanning tree has weight 115. Now we are creating next spanning trees from this minimum spanning tree. In this spanning tree only two edges are missing $\{BD, EF\}$. First we add edge EF to the spanning tree. It will create a cycle DEFD, as in figure 2.

We select an edge with highest weight from that cycle which have weight less than newly added edge EF. So we select edge DF. Because it have weight 35 and edge EF have weight 45. We remove this selected edge DF. So we get another spanning tree. It adds weight of 10 to minimum spanning tree. Add this new spanning tree to the list.

Similarly make new spanning tree by adding other edge BD. when adding BD it will generate a new cycle BCDB. We select edge CD to remove. So this edge exchange will add weight 5 to the graph.



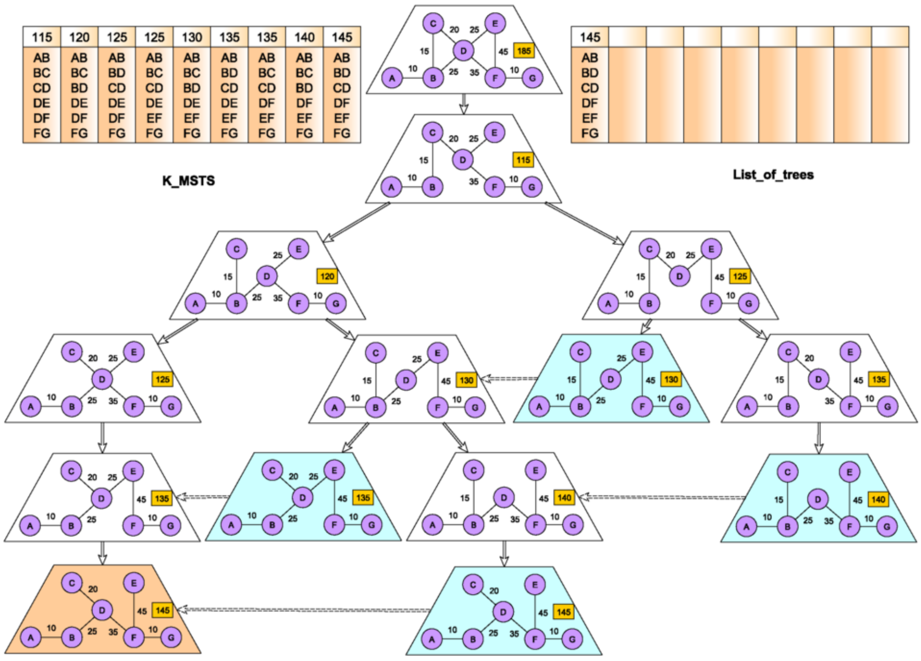
K_MSTS

115	120
AB	AB
BC	BC
CD	BD
DE	DE
DF	DF
FG	FG

List_of_trees

120	125
AB	AB
BC	BC
BD	CD
DE	DE
DF	EF
FG	FG

Fig. 3. 2nd minimum spanning tree



K_MSTS

115	120	125	125	130	135	135	140	145
AB	AB	AB	AB	AB	AB	AB	AB	AB
BC	BC	BD	BC	BC	BD	BC	BC	BD
CD	BD	CD	CD	BD	CD	CD	BD	CD
DE	DE	DE	DE	DE	DE	DF	DF	DF
DF	DF	DF	EF	EF	EF	EF	EF	EF
FG	FG	FG	FG	FG	FG	FG	FG	FG

List_of_trees

145
AB
BD
CD
DF
EF
FG

Fig. 4. 9th minimum spanning tree

We add this two new spanning trees to the list of spanning trees. When adding to the list it checks whether that tree is already added to the list or not. If it is already added we neglect that tree.

Select the spanning tree with minimum weight from the list. That should be the 2^{nd} minimum spanning tree. We make another set of spanning trees by using edge exchange of this spanning tree. Add this new set to the List of trees and remove the selected spanning tree from the list.

In each stage it select a spanning tree from list with minimum weight. Using edge exchange it adds a new set of spanning trees to the list. Continue this process until we get 9^{th} spanning tree. The different stages of algorithm is illustrated in figures 3 and 4.

4 An application of algorithm - Degree constrained minimum spanning tree

Degree constrained minimum spanning tree is the minimum weighted spanning tree of given undirected connected graph, where the degree of each vertex should be less than or equal to given value Δ. Degree constrained minimum spanning treeproblem is NP-complete because for a complete graph with n vertices, there exists at most n^{n-2} spanning trees and we have to find a minimum spanning tree among this, which satisfies the degree constraint.

We use algorithm k_minimum_spanning_trees (G, k) to find k^{th} minimum spanning tree of G. In this algorithm the input is an undirected connected graph G and maximum degree Δ. Output is a spanning tree with minimum weight whose vertices have degree at most Δ. Assumed that there exists a solution for the input graph G.

4.1 Explanation

To find degree constrained minimum spanning treewe generate the spanning trees in the order of weight, and for each tree we check whether it satisfies the degree constraint. Spanning trees are generated until Δ-degree constrained spanning tree is generated.

The working of this algorithm is illustrated in figure 5. The input graph is figure 5.a and value of $\Delta$is 2. Algorithm generate spanning tree in non-decreasing order of weight. In this case 4^{th} minimum spanning tree has maximum degree 2 and all other minimum spanning trees before it has maximum degree > 2. So 4^{th} minimum spanning tree in figure 5.e is the degree constrained minimum spanning treeof G with Δ=2.

Algorithm 2 Degree constrained minimum spanning treeusing k^{th} minimum spanning tree (G, Δ)

```
Input: Graph G and maximum degree $\Delta$.
Output: Degree constrained minimum spanning treeT
  Assumption:
  k_minimum_spanning_trees return $k^{th}$ minimum spanning tree.
  Max_Degree(T) return degree of vertex which have highest degree in T.

  i=1
  T=k_minimum_spanning_trees (G, i)
  while Max_Degree(T) $>\Delta$ do
    i=i+1
    T=k_minimum_spanning_trees (G, i)
  end while
  return T
```

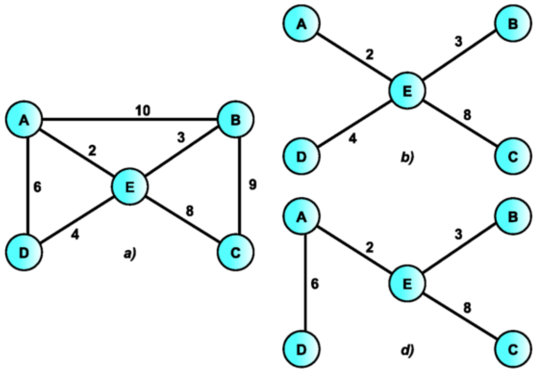


Fig. 5. **a)** Graph G **b)** 1st minimum spanning tree **c)** 2nd minimum spanning tree **d)** 3rd minimum spanning tree **e)** 4th minimum spanning tree which satisfies the constraint $\Delta <= 2$

In the worst case this algorithm has exponential time complexity because it runs until all the spanning trees of G are generated. This algorithm gives the exact solution to this NP-complete problem.

5 Conclusion

The proposed algorithm finds k^{th} minimum spanning tree in polynomial complexity $O(k^2|E||V|)$ in the worst case. When using AVL tree to store k trees its time complexity reduces to $O(k|E||V|log\ k)$ in worst case. This algorithm has a space complexity of $O(k.|V| + |E|)$. An algorithm to solve degree constraint minimum spanning tree using k^{th} minimum spanning tree is also presented here. But in worst case it would have exponential time complexity for finding exact solutions of degree constrained minimum spanning tree, which is an NP-complete problem.

References

[1] H.N. Gabow, Two Algorithms for Generating Weighted Spanning Trees in Order, *SIAM J. Comput.*, **6** (1977), 139–150.

[2] N. Katoh, T. Ibaraki, and H. Mine, An Algorithm for Finding k Minimum Spanning Trees, *SIAM J. Comput.*, **10** (1981), 247–255.

[3] David Eppstein, *Finding the k Smallest Spanning Trees*, Department of Information and Computer Science,University of California, Irvine, CA 92717

[4] Jun Ma, Kazuo Iwama and Qian-Ping Gu, *A Parallel Algorithm for k-Minimum Spanning Trees,IEEE 1997*

[5] M.R Garey and D.S.Johnson, *Computers and Intractability, A Guide to the Theory of NP-completeness*, Freeman, San Francisco, 1979.

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

[7] B. Gavish, Topological design of centralized computer networks- formulations and algorithms, *Networks*, **12** (1982), 355–377.

[8] R. C. Prim, Shortest connection networks and some generalizations, *Bell System Technical Journal*, **36** (1957), 1389-1401.

[9] J. B. Kruskal, On the shortest spanning subtree of a graph and the traveling salesman problem, *Proceedings of the American Mathematical Society*, **7** (1956), 48-50.