The swap edges of a multiple-sources routing tree

Bang Ye Wu* Chih-Yuan Hsiao† Kun-Mao Chao‡

Abstract

Let T be a spanning tree of a graph G and $S \subset V(G)$ be a set of sources. The routing cost of T is the total distance from all sources to all vertices. For an edge e of T, the swap edge of e is the edge f minimizing the routing cost of the tree formed by replacing e with f. Given an undirected graph G and a spanning tree T of G, we investigate the problem of finding the swap edge for every tree edge. In this paper, we propose an $O(m \log n + n^2)$-time algorithm for the case of two sources and an $O(mn)$-time algorithm for the case of more than two sources, where m and n are the numbers of edges and vertices of G, respectively.

Key words: algorithm, graph, swap edge, spanning tree, optimization problem.

1 Introduction

A routing tree of a graph is a spanning tree, along which the messages can be routed from sources to destinations. Let $G = (V, E, w)$ be an undirected graph with nonnegative edge length function w. All vertices are destinations, whereas some of them are sources. The routing cost of a spanning tree is defined by the total distance from sources to all vertices. For the sake of efficiency, the routing cost should be made as small as

*bangye@mail.stu.edu.tw, Department of Computer Science and Information Engineering, Shu-Te University, YenChau, Kaohsiung, Taiwan 824, R.O.C.

†skiky@mail2000.com.tw, Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan 106, R.O.C.

‡corresponding author, kmchao@csie.ntu.edu.tw, Department of Computer Science and Information Engineering / Graduate Institute of Networking and Multimedia, National Taiwan University, Taipei, Taiwan 106, R.O.C.

possible. Given a graph and a set of p sources, the problem of finding the minimum routing cost spanning tree (MRCT) is NP-hard for any constant $p > 1$ [9]. When $p = 1$, i.e., there is only one source, it is reduced to the well-known single-source shortest-paths tree problem and can be solved in polynomial time. For the p-sources MRCT problem, several approximation algorithms have been reported, including a polynomial time approximation scheme (PTAS) for the case that all vertices are sources [12], a PTAS for the case of two sources [8], and a 2-approximation algorithm for an arbitrary number of sources [11]. A survey of the MRCT and some other related problems can be found in [10].

The survivability of a network is the ability to recover the communication if a link of the network fails [3]. A tree has the benefits of low cost and simple routing algorithms, but it suffers from weak survivability. Therefore, how to recover a tree from failures is an important issue. Any edge of a spanning tree corresponds to a bipartition of the vertex set. To recover a spanning tree from the failure of an edge, we need to find another edge crossing the bipartition so as to minimize the routing cost. For an edge e of T, the swap edge of e is the edge f minimizing the routing cost of the tree formed by replacing e with f.

Let m and n be the numbers of edges and vertices of the input graph, respectively. In [7], the authors proposed an $O(n^2)$-time algorithm for finding the swap edge for every edge of a single-source shortest-paths tree. More recently, an $O(m \log^2 n)$-time algorithm for the same problem was reported in [1], which improves the previous one for sufficiently sparse graphs. In this paper, we investigate the problem of finding the swap edge for every tree edge when the number of sources is more than one. We present an $O(m \log n + n^2)$-time algorithm for the case of two sources and an $O(mn)$-time algorithm for the case of more than two sources.

The rest of the paper is organized as follows. We first formally define the notations and the problem in Section 2. In Section 3, we propose an $O(m \log n + n^2)$-time algorithm for the two-sources case. Then we propose an $O(mn)$-time algorithm for the multiple-sources case in Section 4. Finally, concluding remarks are given in Section 5.

2 Preliminaries

By $G = (V, E, w)$, we denote a graph G with vertex set V, edge set E, and edge weight (length) function w. In this paper, we consider only undirected graphs with nonnegative edge lengths. For any graph G, the vertex set and edge set are denoted by $V(G)$ and $E(G)$ respectively. Let T be a spanning tree of G. For any $e \in E(T)$, the removal of e cuts T into two subtrees, of which the vertex sets form a bipartition (V_1, V_2) of $V(G)$. By $\mathcal{C}(e)$, we denote the cut set of the bipartition, i.e., the set of edges with one endpoint in V_1 and the other endpoint in V_2. For convenience, we assume that $e \notin \mathcal{C}(e)$. It is clear that if $\mathcal{C}(e) \neq \emptyset$, then the two subtrees can be connected to form another spanning tree of G by inserting any edge in the cut set. We use $T_{e/f}$ to denote the new spanning tree obtained by replacing an edge e with another edge $f \in \mathcal{C}(e)$.

For $u, v \in V(G)$, the distance between u and v in G is denoted by $d_G(u, v)$, which is the total length of the edges in a shortest path between the two vertices. For $U \subset V(G)$, let $D_G(v, U) = \sum_{u \in U} d_G(v, u)$ denote the total distance from v to all vertices in U. For a subgraph H, we also use $D_G(v, H)$ to denote $D_G(v, V(H))$. A source (vertex) is a distinguished vertex in $V(G)$. For a given set S of source vertices, the routing cost of a spanning tree T is denoted by $c(T, S)$, which is the total distance from sources to all vertices, i.e., $c(T, S) = \sum_{s \in S} D_T(s, T)$. When there is no confusion, we also simply use $c(T)$ for the routing cost. For an edge $e \in E(T)$, the *swap cost* is defined by $\chi(T, e) = \min_{f \in \mathcal{C}(e)} c(T_{e/f}, S)$, which is the minimum routing cost of any spanning tree obtained by replacing e with another edge. The *swap edge problem* is formally defined

as follows:

PROBLEM: Swap Edges for a Routing Tree (SERT)

INSTANCE: A graph $G = (V, E, w)$, a spanning tree T of G, and a set $S \subset V$ of sources.

GOAL: For each $e \in E(T)$, find $\chi(T, e)$.

In case that $\mathcal{C}(e)$ is empty for some e, we say that there is no swap edge for e and the cost $\chi(T, e) = \infty$. Since the swap edges will be implicitly determined when minimizing the swap costs, we shall focus on the swap costs only. In the remaining paragraphs, we shall assume that G, T, and S are the input data of the SERT problem.

3 Two-sources case

In this section, we discuss the two-sources case of the SERT problem. Let $S = \{s_1, s_2\} \subset V$ be the set of two specified source vertices and $P = (s_1 = r_0, r_1, r_2, \ldots, r_k = s_2)$ be the path between the two sources in T. The routing cost of T in this case is defined by

$$c(T, S) = \sum_{v \in V} D_T(v, S) = \sum_{v \in V} \left(d_T(v, s_1) + d_T(s_2, v) \right).$$

Removing all the edges of P from T divides T into $k + 1$ subtrees. Let T_i denote the subtree containing r_i and $V_i = V(T_i)$ for $0 \leq i \leq k$. Our method for computing the swap costs for all tree edges is divided into two parts: $e \notin E(P)$ and $e \in E(P)$.

3.1 Edges not in the path

To compute the swap costs of edges in T_i, we treat all vertices not in V_i as a super node, and reduce the problem to the single-source case, as shown in Figure 1. Construct an auxiliary graph $G_i = (V_i', E_i', w_i')$ by shrinking all vertices not in V_i to a created vertex s as follows.

- $V_i' = V_i \cup \{s\}$, in which $s \notin V$ is a created node.

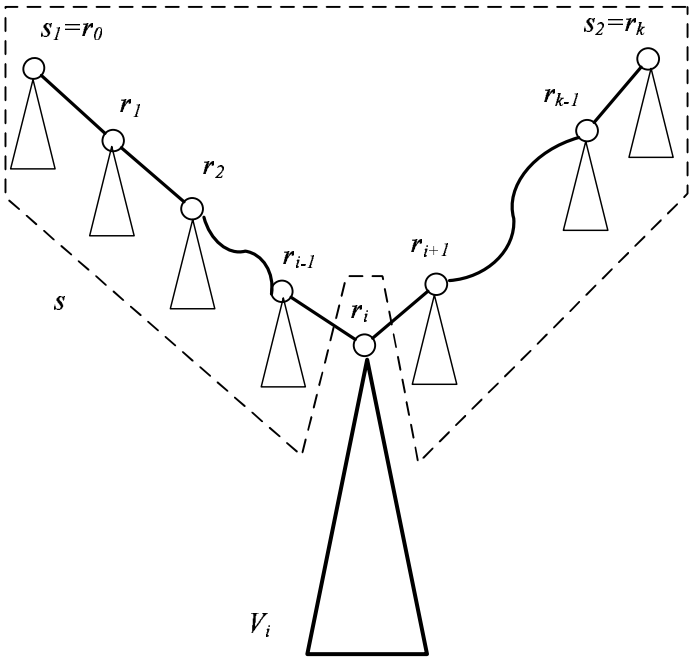

Figure 1: The super node s for reduction to single-source SERT problem.

- The edge set E_i' consists of two parts: The first part contains all the edges of which both endpoints are in V_i, and the other part contains the edges (s, v) if there exists an edge (v, u) crossing the cut $(V_i, V - V_i)$ for any $v \in V_i$.
- For each edge (u, v) such that $u, v \in V_i$, set $w_i'(u, v) = 2w(u, v)$.
- $w_i'(s, r_i) = D_T(r_i, S)$ and for each edge (v, s), $v \in V_i - \{r_i\}$, set $w_i'(v, s) = \min_{u \notin V_i}\{2w(u, v) + D_T(u, S)\}$.

Let T' be the spanning tree of G_i, which is obtained by inserting edge (s, r_i) into T_i, and let s be the source of T'.

Lemma 1: For any vertex $v \in V_i$, $D_T(v, S) = d_{T'}(v, s)$.

Proof:

$$\begin{aligned} d_{T'}(v, s) &= d_{T'}(v, r_i) + w_i'(r_i, s) \\ &= 2d_T(v, r_i) + D_T(r_i, S) = d_T(v, s_1) + d_T(v, s_2) \end{aligned}$$

□

Similarly, the following result can be easily shown by the definitions, and the proof is omitted.

Lemma 2: The swap costs of all edges in T_i can be computed by solving the single-source SERT problem with input (G_i, T', s).

Lemma 3: The swap costs for all edges not in P can be found in $O(n^2)$ time.

Proof: To compute the swap costs, for each V_i, we construct the auxiliary graph and then solve the single-source SERT problem. First we compute $d_T(v, S)$ for every $v \in V$ by traversing T twice: one for computing $d_T(v, s_1)$ and the other for $d_T(v, s_2)$. Also the set V_i for each i can be found by traversing the tree in linear time. To construct G_i, we traverse T_i in a depth first search order. By storing the graph in an adjacent list, all the auxiliary graphs can be constructed in $O(n + m)$ time since each edge is checked twice. Since the single-source SERT problem for G_i can be solved in $O(|V(G_i)|^2)$ time [7], the total time complexity is $O(n^2)$. □

3.2 Edges in the path

In the following, we focus on how to find out the swap costs for the edges in $P = (s_1 = r_0, r_1, r_2, \ldots, r_k = s_2)$. Let $e_i = (r_i, r_{i+1})$ and $n_i = |V_i|$. Let Y_i and $\bar{Y}_i$ be the two subtrees obtained by removing the edge e_i from T, in which $s_1 \in V(Y_i)$. Also let $N_i = |V(Y_i)|$. Note that $V(Y_i) = \bigcup_{j \le i} V_j$ and $N_i = \sum_{j \le i} n_j$. For any $f \in \mathcal{C}(e_i)$ and $H = T_{e/f}$, we have

$$c(H) = (D_H(s_1, Y_i) + D_H(s_2, \bar{Y}_i)) + (D_H(s_2, Y_i) + D_H(s_1, \bar{Y}_i)).$$

Obviously, $D_H(s_1, Y_i) = D_T(s_1, Y_i)$ and $D_H(s_2, \bar{Y}_i) = D_T(s_2, \bar{Y}_i)$ since they are not affected by the removal of e_i. Let $f = (x, y)$, as illustrated in Figure 2. By defini-

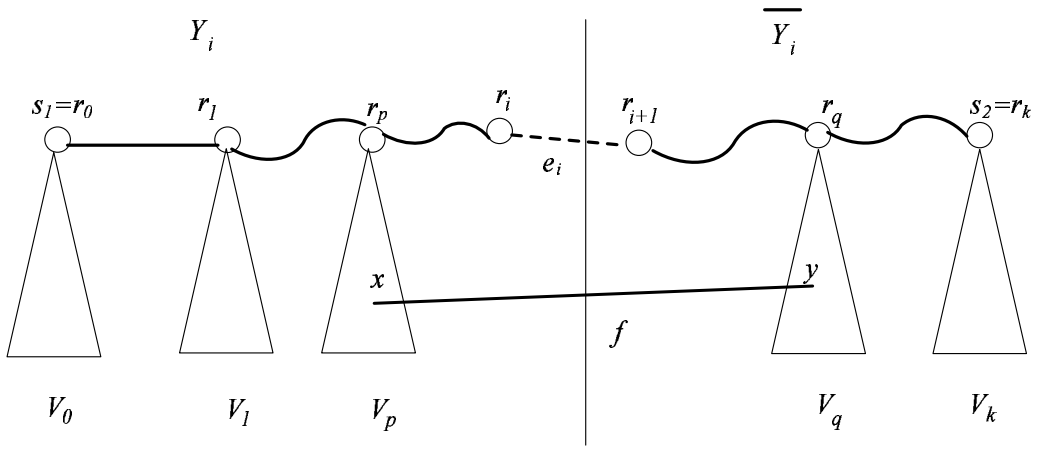

Figure 2: A swap edge for an edge (r_i, r_{i+1}) in the path between the two sources.

tion, $D_H(s_2, Y_i) = D_T(x, Y_i) + N_i(w(f) + d_T(y, s_2))$ and $D_H(s_1, \bar{Y}_i) = D_T(y, \bar{Y}_i) + (n - N_i)(w(f) + d_T(x, s_1))$. Let

$$\begin{aligned} \delta(f,i) &= D_H(s_2, Y_i) + D_H(s_1, \bar{Y}_i) \\ &= D_T(x, Y_i) + D_T(y, \bar{Y}_i) + nw(f) + N_i d_T(y, s_2) + (n - N_i) d_T(x, s_1), \quad (1) \end{aligned}$$

and our goal is to compute

$$\chi(T, e_i) = (D_T(s_1, Y_i) + D_T(s_2, \bar{Y}_i)) + \min_{f \in \mathcal{C}(e_i)} \delta(f, i),$$

for every $0 \leq i < k$. By a naïve method, $\delta(f, i)$ can be computed in $O(n)$ time, which implies that the swap cost $\chi(T, e_i)$ can be computed in $O(mn)$ time. Since the number of edges in P may be up to $\Omega(n)$, it takes totally $O(mn^2)$ to compute all the swap costs. In the following, we present an incremental method and show that the two-sources SERT problem can be solved in $O(m \log n + n^2)$ time.

Suppose that $x \in V_p$ and $y \in V_q$, $p < q$. Observe that, for $p < i < q$,

$$\begin{aligned} & \delta(f,i) - \delta(f,i-1) \\ = \; & \big(D_T(x, Y_i) + D_T(y, \bar{Y}_i) + nw(f) + N_i d_T(y, s_2) + (n - N_i) d_T(x, s_1)\big) \\ & - \big(D_T(x, Y_{i-1}) + D_T(y, \bar{Y}_{i-1}) + nw(f) + N_{i-1} d_T(y, s_2) + (n - N_{i-1}) d_T(x, s_1)\big) \end{aligned}$$

$$
\begin{aligned}
&= (D_T(x, Y_i) - D_T(x, Y_{i-1})) + (D_T(y, \bar{Y}_i) - D_T(y, \bar{Y}_{i-1})) \\
&\quad +(N_i - N_{i-1}) d_T(y, s_2) + ((n - N_i) - (n - N_{i-1})) d_T(x, s_1) \\
&= D_T(x, V_i) - D_T(y, V_i) + n_i d_T(y, s_2) - n_i d_T(x, s_1).
\end{aligned}
$$

Since $D_T(x, V_i) - D_T(y, V_i) = n_i(d_T(r_i, x) - d_T(r_i, y))$, we have

$$
\begin{aligned}
&\delta(f, i) - \delta(f, i-1) \\
&= n_i(d_T(r_i, x) - d_T(r_i, y) + d_T(y, s_2) - d_T(x, s_1)) \\
&= n_i \left(d_T(r_i, x) - d_T(x, s_1) + d_T(y, s_2) - d_T(r_i, y)\right) \\
&= n_i \left(d_T(r_i, r_p) - d_T(r_p, s_1) + d_T(r_q, s_2) - d_T(r_i, r_q)\right). \qquad (2)
\end{aligned}
$$

It implies that $\delta(f, i)$ can be computed from $\delta(f, i-1)$ in constant time if all the required distances are available after a preprocessing stage. Consequently all the swap edges can be found in $O(mn)$ time by computing $\delta(f, i)$ for each f and each i. In the following, we propose an efficient method to find the edge minimizing $\delta(f, i)$ without calculating $\delta(f, i)$ for all the edges crossing the cut.

For any edge (x, y) in which $x \in V_p$ and $y \in V_q$, we say that x is the left endpoint and y is the right endpoint if $p < q$. In the following, we shall assume that x is in V_p, $p < k$. Let E_x denote the set of all edges with left endpoint x. Let $\widehat{f}(i) = \delta(f, i)$. Our approach is to compute a function F_x defined by

$$
F_x(i) = \min_{f \in E_x} \widehat{f}(i).
$$

Once the function F_x for each vertex x is determined, the minimized $\delta(f, i)$ can be found by taking the minimum of $F_x(i)$ for all x. Since F_x is defined by the minimum of a set of functions, it can be regarded as a piecewise function. To determine F_x, it is required to find the consecutive intervals and to determine which of the functions $\widehat{f}$ is the minimum for each interval.

For two discrete functions $\widehat{f_1}$ and $\widehat{f_2}$, we say the two functions intersect in the interval $[l_1, l_2]$ if $\widehat{f_1}(l_1) < \widehat{f_2}(l_1)$ and $\widehat{f_1}(l_2) > \widehat{f_2}(l_2)$, or vice versa. If $\widehat{f_1}(l_1) < \widehat{f_2}(l_1)$ and $[l_1, l_2]$ is an intersection interval, the insertion point is the minimum i such that $\widehat{f_1}(i) > \widehat{f_2}(i)$. Define $right(f) = q$ if the right endpoint of f is in V_q. We shall show some properties which are crucial for our algorithm.

Lemma 4: For two edges f_1 and f_2 in E_x, if $right(f_1) = right(f_2)$, then $\widehat{f_1}$ and $\widehat{f_2}$ never intersect. Furthermore, by an $O(n^2)$-time preprocessing, we can determine which of the two functions is the smaller in constant time.

Proof: Let $f_1 = (x, y_1)$, $f_2 = (x, y_2)$, and $right(f_1) = q$. By (2), $\widehat{f_1}(i) - \widehat{f_1}(i-1) = \widehat{f_2}(i) - \widehat{f_2}(i-1)$ for any $p < i < q$. That is, the difference between $\widehat{f_1}$ and $\widehat{f_2}$ is fixed in the domain, thus they never intersect.

By (1),

$$\begin{aligned} \widehat{f_1}(i) - \widehat{f_2}(i) &= N_i(d_T(y_1, s_2) - d_T(y_2, s_2)) + n(w(x, y_1) - w(x, y_2)) \\ &\quad + D_T(y_1, \bar{Y_i}) - D_T(y_2, \bar{Y_i}), \end{aligned}$$

and

$$\begin{aligned} & D_T(y_1, \bar{Y_i}) - D_T(y_2, \bar{Y_i}) \\ = \; & (n - N_i - n_q)(d_T(r_q, y_1) - d_T(r_q, y_2)) + D_T(y_1, V_q) - D_T(y_2, V_q). \end{aligned}$$

In a preprocessing stage, we compute the following:

- $d_T(u, v)$ for any $u, v \in V$.
- For each vertex v, find i such that $v \in V_i$.
- $n_i = |V_i|$ and $N_i = |V(Y_i)|$ for each i.
- $D_T(v, V_i)$ for each i and each $v \in V_i$.

The preprocessing can be done in $O(n^2)$ time. After the preprocessing stage, we can compute $\widehat{f}_1(i) - \widehat{f}_2(i)$ in constant time and determine the smaller one. □

Lemma 5: If $f_1, f_2 \in E_x$ and $right(f_2) < right(f_1)$, the difference $(\widehat{f}_2(i) - \widehat{f}_1(i))$ is monotonically increasing as i increases for any $p \leq i < right(f_2)$.

Proof: Let $right(f_2) = q$ and $right(f_1) = j$. For any $p < i < q$,

$$\begin{aligned}
& (\widehat{f}_2(i) - \widehat{f}_1(i)) - (\widehat{f}_2(i-1) - \widehat{f}_1(i-1)) \\
= \; & (\widehat{f}_2(i) - \widehat{f}_2(i-1)) - (\widehat{f}_1(i) - \widehat{f}_1(i-1)) \\
= \; & n_i \left(d_T(r_i, r_p) - d_T(r_p, s_1) + d_T(r_q, s_2) - d_T(r_i, r_q)\right) \\
& -n_i \left(d_T(r_i, r_p) - d_T(r_p, s_1) + d_T(r_j, s_2) - d_T(r_i, r_j)\right) \\
= \; & n_i \left((d_T(r_q, s_2) - d_T(r_j, s_2)) + (d_T(r_i, r_j) - d_T(r_i, r_q))\right) \\
= \; & n_i \left(d_T(r_q, r_j) + d_T(r_q, r_j)\right) \\
= \; & 2n_i d_T(r_q, r_j) \geq 0.
\end{aligned}$$

It follows that $(\widehat{f}_2(i) - \widehat{f}_1(i)) \geq (\widehat{f}_2(i-1) - \widehat{f}_1(i-1))$. That is, the difference is monotonically increasing as i increases. □

By Lemma 4, for each $q > p$, at most one edge with right endpoint in V_q needs to be considered, and all the other edges in E_x can be eliminated. Let $E'_x = \{f_i | 1 \leq i \leq h\}$ be the set of edges that should be taken into consideration, in which $right(f_i) > right(f_{i+1})$ for each i. Starting at $\widehat{F}_1 = \widehat{f}_1$, our algorithm iteratively computes $\widehat{F}_i = \min\{\widehat{F}_{i-1}, \widehat{f}_i\}$ for i from 2 to h, and by definition $F_x = \widehat{F}_h$.

Let $p = \alpha_0 < \alpha_1 < \ldots < \alpha_t \leq k$. We denote the piecewise function $\widehat{F}_i$ by $\bigcup_{j=1}^{t}(\alpha_j, \widehat{f}_{g(j)})$, in which $g(j) \leq i$ and $\widehat{F}_i(\lambda) = \widehat{f}_{g(j)}(\lambda)$ for each j and any $\lambda \in [\alpha_{j-1} + 1..\alpha_j]$. We say that $[\alpha_{j-1} + 1..\alpha_j]$ is the j-th interval of $\widehat{F}_i$.

Lemma 6: For any i, the function $\widehat{f_i}$ intersects $\widehat{F}_{i-1}$ at most once. If they intersect, $\widehat{f_i}$ must appear at the first interval of $\widehat{F}_i$.

Proof: By definition, $\widehat{F}_{i-1} = \min_{j<i} \widehat{f_j}$, and $right(f_i) < right(f_j)$ for any $j < i$. By Lemma 5, the difference $(\widehat{f_i}(\lambda) - \widehat{f_j}(\lambda))$ is monotonically increasing as λ increases. That is, if $\widehat{f_i}(\lambda) > \widehat{F}_{i-1}(\lambda)$ for some λ, we can conclude that $\widehat{f_i}(\lambda') > \widehat{F}_{i-1}(\lambda')$ for any $\lambda' > \lambda$, and the result follows. □

Based on Lemma 6, the merging algorithm for computing $\widehat{F}_i$ is designed as follows. For convenience, let $\widehat{f}(\lambda) = \infty$ for any $\lambda \geq right(f)$.

Algorithm Merge($\widehat{F}_{i-1}, \widehat{f_i}$)

Input: An edge $f_i = (x, y)$ and a piecewise function

$\widehat{F}_{i-1} = \bigcup_{j=1}^{t} (\alpha_j, \widehat{f}_{g(j)})$ stored in a link list L.

Output: $\widehat{F}_i$ stored in L.

1: **if** $\widehat{F}_{i-1}(p) \leq f_i(p)$

2: **return**; /* no intersection */

3: Scan L from the head so as to find the smallest α_j such that

$\widehat{F}_{i-1}(\alpha_j) < f_i(\alpha_j)$;

4: Use binary search to find the intersection point λ in $[\alpha_{j-1}, \alpha_j]$;

5: Remove all intervals before the j-th interval from L;

6: Break the j-th interval by inserting $(\lambda - 1, \widehat{f_i})$ at the head of L;

7: **return** L.

The algorithm for computing the swap costs for all edges in the path is as follows.

Algorithm Two_SERT_P

Input: A graph $G = (V, E, w)$, a spanning tree T and two source vertices s_1 and s_2.

Assume that G is stored in an adjacency list;

Output: $\chi(T, e)$ for each e in the path between the two sources;

1: Compute the following:

1.1: the path $P = (r_0, r_1, \ldots, r_k)$ between s_1 and s_2;

1.2: $d_T(u, v)$ for any $u, v \in V$;

1.3: **for** each vertex v **do** find i such that $v \in V_i$;

1.4: $n_i = |V_i|$ and $N_i = |V(Y_i)|$ for each i;

1.5: $D_T(v, V_i)$ for each i and each $v \in V_i$;

1.6: $D_T(v, Y_i)$ and $D_T(v, \bar{Y_i})$ for each vertex v and each i;

2: **for** each vertex x **do**

3: Find E_x by scanning the adjacency list;

4: Sort the edges f in E_x such that $right(f)$ are in a non-increasing order;

5: Construct $E'_x = \{f_i | 1 \leq i \leq h\}$ by eliminating the useless edges

as described in Lemma 4;

6: $\widehat{F}_1 \leftarrow \widehat{f}_1$; and create a link list L with one element $(right(f_1), \widehat{f}_1)$;

7: **for** $i \leftarrow 2$ **to** h **do**

$\widehat{F}_i \leftarrow \text{Merge}(\widehat{F}_{i-1}, \widehat{f}_i)$;

8: Compute $F_x(i)$ for each e_i in P;

9: **for** each e_i in P

Find $\chi(T, e_i) = \min_x F_x(i)$;

Lemma 7: In algorithm **Two_SERT_P**, Step 1 can be done in $O(n^2)$ time.

Proof: It is obvious that Step 1.1 can be done in linear time. As shown in Lemma 4,

Steps 1.2–1.5 takes $O(n^2)$ time. Now let's calculate the time complexity of Step 1.6 as follows. Observe that $D_T(v, Y_0) = n_0 d_T(v, r_0) + D_T(r_0, V_0)$ and that $D_T(v, Y_i) = D_T(v, Y_{i-1}) + n_i d_T(v, r_i) + D_T(r_i, V_i)$ for $i > 0$. Since all the required values are available before this step, $D_T(v, Y_i)$ for all i can be done in $O(n)$ time. Similarly, the distances $D_T(v, \bar{Y}_i)$ for all i can also be found by an incremental method. Therefore, the total time complexity for Step 1 is $O(n^2)$. □

The next result is straightforward by (1).

Lemma 8: After Step 1, the value $\delta(f, i)$ for any edge f and any i can be computed in constant time.

Lemma 9: The time complexity of algorithm **Two_SERT_P** is $O(m \log n + n^2)$ time.

Proof: First, by Lemma 7, Step 1 can be completed in $O(n^2)$ time. Let's analyze the time for the loop in Steps 2–8. Step 3 takes the linear time $O(|E_x|)$, and the sorting at Step 4 can be completed in $O(|E_x| \log |E_x|)$ time. By Lemma 4, Step 5 can be also done in $O(|E_x|)$ time. Step 6 takes only constant time. By Lemma 8, Step 8 takes $O(n)$ time.

To analyze the time complexity of Step 7, we consider the **Merge** algorithm first. In **Merge**, we check the intervals one by one until we find the intersecting interval. Once the interval is found, a binary search is employed to find the intersecting point. Obviously, the binary search needs at most $O(\log n)$ comparisons between the values of two functions. Since, by Lemma 8, the function values can be found in constant time, the binary search takes $O(\log n)$ time. To show the time complexity of Steps 3 and 5 in **Merge**, we use an amortized analysis. Let L_i denote the number of intervals of $\widehat{F}_i$ and β_i denote the number of intervals checked while merging $\widehat{f}_i$. We have that

$$L_i = L_{i-1} - \beta_i + 2.$$

Summing up for i from 2 to h, we obtain

$$L_h = L_1 - \sum_i \beta_i + 2(h-1).$$

Since $L_h \geq 1$ and $L_1 = 1$,

$$\sum_i \beta_i = L_1 + 2(h-1) - L_h \leq 2h - 2.$$

That is, the total number of checked intervals is $O(|E'_x|)$. Since checking for an interval takes only constant time by Lemma 8, the total time complexity of Step 7 in Algorithm **Two_SERT_P** is $O(|E'_x| \log n)$.

In summary, the time complexity of the whole loop for each x is $O(|E_x| \log |E_x|)$. Since $\sum_{x \in V} |E_x| = m$, summing up for all vertices, the total time complexity of the loop is $O(m \log n)$. Finally, since Step 9 can be easily done in $O(n^2)$ time, we conclude that the total time complexity of the algorithm is $O(m \log n + n^2)$. □

The next theorem summarizes the result in this section, which directly follows Lemmas 3 and 9.

Theorem 10: The two-sources SERT problem can be solved in $O(m \log n + n^2)$ time.

4 Multiple-sources case

In this section, we consider the case where there are more than two sources. By definition, the routing cost of a tree T with source set S can be calculated by $c(T, S) = \sum_{s \in S} D_T(s, V(T))$. Since there is no limit on the number of sources, a direct method for computing the routing cost of a tree will take $O(n^2)$ time even when the all-to-all distances are already known. For each edge e in $E(T)$, if we compute $c(T_{e/f}, S)$ for every f in the cut set, the time complexity for finding the swap cost $\chi(T, e)$ will be up to

$O(mn^2)$, and consequently it takes $O(mn^3)$ to find the swap costs for all edges in T. We shall present an $O(mn)$ time algorithm for the SERT problem with multiple sources.

First we introduce a new formula for computing the routing cost of a tree. Let $e = (u_1, u_2) \in E(T)$. As mentioned previously, removing e from T will result in two subtrees. Let T_1 and T_2 be the two subtrees, in which $u_1 \in V(T_1)$. Let $V_i = V(T_i)$ and $E_i = E(T_i)$ for $i = 1, 2$. Let S_1 and S_2 denote the sets of sources in V_1 and V_2, respectively.

Apparently, there are $|S_1| \times |V_2|$ paths from a source in S_1 to a vertex in V_2. Similarly the number of paths from a source in S_2 to a vertex in V_1 is $|S_2| \times |V_1|$. By defining the routing load $l(T, e) = |S_1| \times |V_2| + |S_2| \times |V_1|$, we have the following formula.

$$c(T, S) = \sum_{e \in E(T)} l(T, e)w(e). \tag{3}$$

The routing cost of an edge e is defined by $l(T, e)w(e)$. The above property shows that the routing cost of T can be calculated by summing up the routing costs of all the tree edges. Based on (3), we can compute the routing cost of a tree in $O(n)$ time when the routing costs of all edges are available. All we need to know is the routing load of each edge. By rooting T at an arbitrary vertex and traversing the tree in post-order, we can easily compute the numbers of sources and the numbers of vertices in both sides of every edge. Thus, the routing loads of all edges can be obtained in $O(n)$ time.

To solve the multiple-sources SERT problem, our algorithm finds the swap cost of each edge individually. For each edge e, we calculate the routing cost of $T_{e/f}$ for each edge f crossing the cut. We shall show that, after a preprocessing stage, the routing cost of $T_{e/f}$ can be computed in constant time.

For any edge $f \in \mathcal{C}(e)$, by (3), we have

$$c(T_{e/f}, S) = \sum_{e' \in E_1} l(T_{e/f}, e')w(e') + l(T_{e/f}, f)w(f) + \sum_{e' \in E_2} l(T_{e/f}, e')w(e'), \tag{4}$$

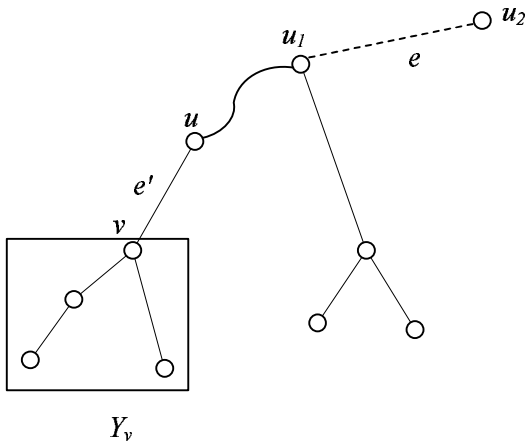

Figure 3: Y_v is the subtree of Y rooted at a vertex v.

in which $l(T_{e/f}, f) = l(T, e)$ since e and f are crossing the same cut.

To reduce the computation time, one difficulty is that the routing load of an edge p in a tree $T_{e/f}$ is not fixed but depends on f. The following simple observation is crucial for our algorithm. For each edge $e' \in E(T) - \{e\}$, there are only two possible values of $l(T_{e/f}, e')$ for all $f \in \mathcal{C}(e)$, depending on which side of e' the swap edge f is connected to.

In the following, we shall give a procedure for computing function $\gamma_1(e, v)$ for each $v \in V_1$, which is defined to be the total routing cost of all edges in E_1 subject to that the swap edge f is connected to v. By a similar procedure, we can compute function $\gamma_2(e, v)$, which is defined to be the total routing cost of all edges in E_2 subject to that the swap edge f is connected to v. Once the two functions are completed, the routing cost of $T_{e/f}$ for any f can be calculated in constant time.

Let Y be the tree obtained by rooting T_1 at u_1. By Y_v, we denote the subtree of Y rooted at a vertex v. For an edge $(u, v) \in E(Y)$, we assume that u is the parent of v in Y, as shown in Figure 3.

Let n_v and s_v denote the numbers of vertices and sources, respectively, in the subtree

Y_v. For any $e' = (u, v) \in E(Y)$, define

$$\bar{l}_1(e') = (s_v(n - n_v) + (|S| - s_v)n_v)w(e'), \tag{5}$$

which is the routing cost of e' if the swap edge is connected to any vertex not in $V(Y_v)$. Similarly, define

$$\bar{l}_2(e') = ((s_v + |S_2|)(|V_1| - n_v) + (|S_1| - s_v)(n_v + |V_2|))w(e'), \tag{6}$$

which is the routing cost of e' if the swap edge is connected to any vertex in $V(Y_v)$. Then, by definition, the value $\gamma_1(e, v)$ can be calculated by

$$\gamma_1(e, v) = \sum_{e' \notin P} \bar{l}_1(e') + \sum_{e' \in P} \bar{l}_2(e'),$$

in which P is the path between v and the root u_1. Clearly, for the root u_1,

$$\gamma_1(e, u_1) = \sum_{e' \in E(Y)} \bar{l}_1(e'), \tag{7}$$

and, for any child v of a vertex u,

$$\gamma_1(e, v) = \gamma_1(e, u) - \bar{l}_1(u, v) + \bar{l}_2(u, v). \tag{8}$$

By (7) and (8), we can compute $\gamma_1(e, v)$ for all vertices v in Y by traversing Y twice: a post-order traversal for computing $\gamma_1(e, u_1)$, and then a preorder traversal for computing $\gamma_1(e, v)$ from the value of its parent for every other vertex v. The whole algorithm is as follows.

Algorithm Multiple_SERT

Input: A graph $G = (V, E, w)$ and a spanning tree T and the source vertex set S.

Output: $\chi(T, e)$ for each $e \in E(T)$.

1: for each tree edge $e = (u_1, u_2)$ **do**

2: Find the two subtrees T_1 and T_2;

3: Construct Y by rooting T_1 at u_1;

4: Traverse Y in a post-order to find s_v and n_v for each v;

5: Compute $\bar{l}_1(e')$ for each $e' \in E(Y)$ by (5);

6: Compute $\bar{l}_2(e')$ for each $e' \in E(Y)$ by (6);

7: Compute $\gamma_1(e, u_1) = \sum_{e' \in E(Y)} \bar{l}_1(e')$ by traversing Y in a post-order;

8: Traverse Y in a preorder to find $\gamma_1(e, v)$ by (8) for each $v \in V_1 - \{u_1\}$;

9: Use the steps similar to Steps 3–8 to find $\gamma_2(e, v)$ for each $v \in V_2$;

10: **for** each $f = (x, y) \in \mathcal{C}(e)$ **do**

$$c(T_{e/f}) \leftarrow \gamma_1(e, x) + l(T, e)w(f) + \gamma_2(e, y)$$

11: Find $\chi(T, e) = \min_f c(T_{e/f})$.

Theorem 11: The multiple-sources SERT problem can be solved by **Multiple_SERT** in $O(mn)$ time.

Proof: The correctness follows directly from the above explanation. It remains to show the time complexity of **Multiple_SERT** is $O(mn)$. Clearly, Steps 2–9 can be done in $O(n)$ time. The time complexity of Steps 10 and 11 is $O(m)$. Since there are $n - 1$ edges in the tree, the total time complexity is $O(mn)$. □

5 Concluding Remarks

In this paper, we propose algorithms for finding the swap cost of every edge of a multiple-sources routing tree. Our algorithms take $O(m \log n + n^2)$ and $O(mn)$ time for the two-sources case and the multiple-sources case, respectively. It is easy to see that the swap edges have been implicitly found by the algorithms.

The algorithms developed in this paper can be easily extended to the case of weighted routing cost, in which we are given a nonnegative weight $r(v)$ for each vertex v, and the routing cost of v is defined by $r(v)D_T(v, S)$. Note that it includes the case that not all vertices are destinations by setting zero weight to vertices which are not destinations.

The swap edge problem we defined in this paper is to minimize the total, or equivalently the average, length of every path from a source to a destination. We are also interested in the swap edge problem with respect to other criteria [4, 7]. Another interesting problem about survivability is to find the most vital edge of a shortest path or a minimum spanning tree [2, 5, 6].

Acknowledgements

We thank the reviewers for their helpful comments. Bang Ye Wu was supported in part by NSC grants 93-2213-E-366-001 and 93-2213-E-366-014, Taiwan. Chih-Yuan Hsiao and Kun-Mao Chao were supported in part by NSC grants 92-2213-E-002-059 and 93-2213-E-002-029, Taiwan.

References

[1] A. Di Salvo and G. Proietti, Swapping a failing edge of a shortest paths tree by minimizing the average stretch factor, (SIROCCO'04), LNCS 3104, 99–110, 2004.

[2] B. Dixon, M. Rauch, and R.E. Tarjan, Verification and sensitivity analysis of minimum spanning trees in linear time, *SIAM Journal on Computing*, 21(6), 1184–1192, 1992.

[3] M. Grötschel, C.L. Monma, and M. Stoer, Design of survivable networks, *Handbook in OR and MS*, Vol.7, Elsevier, Amsterdam, 617–672, 1995.

[4] G. F. Italiano and R. Ramaswami, Maintaining spanning trees of small diameter, *Algorithmica*, 22(3), 275–304, 1998.

[5] K. Iwano and N. Katoh, Efficient algorithms for finding the most vital edge of a minimum spanning tree, *Information Processing Letters*, 48(5) , 211–213, 1993.

[6] E. Nardelli, G.Proietti, and P. Widmayer, A faster computation of the most vital edge of a shortest path, *Information Processing Letters*, 79(2), 81–85, 2001.

[7] E. Nardelli, G.Proietti, and P. Widmayer, Swapping a failing edge of a single source shortest paths tree is good and fast, *Algorithmica*, 35(1), 56–74, 2003.

[8] B.Y. Wu, A polynomial time approximation scheme for the two-source minimum routing cost spanning trees, *Journal of Algorithms*, 44, 359–378, 2002.

[9] B.Y. Wu, Approximation algorithms for the optimal p-source communication spanning tree, *Discrete Applied Mathematics*, 143, 31–42, 2004.

[10] B.Y. Wu and K.-M. Chao, *Spanning Trees and Optimization Problems*, Chapman & Hall / CRC Press, 2004.

[11] B.Y. Wu, K.-M. Chao, and C.Y. Tang, Approximation algorithms for some optimum communication spanning tree problems, *Discrete Applied Mathematics*, 102, 245–266, 2000.

[12] B.Y. Wu, G. Lancia, V. Bafna, K.-M. Chao, R. Ravi and C.Y. Tang, A polynomial time approximation scheme for minimum routing cost spanning trees, *SIAM Journal on Computing*, 29, 761–778, 1999.