Efficient Algorithms for finding a Trunk on a Tree Network and its Applications

Yamin Li, Shietung Peng
Department of Computer Science
Hosei University
Tokyo 184-8584 Japan
{yamin;speng}@k.hosei.ac.jp

Wanming Chu
Department of Computer Hardware
University of Aizu
Aizu-Wakamatsu 965-8580 Japan
w-chu@u-aizu.ac.jp

Abstract

Given an edge-weighted tree T, a trunk is a path P in T which minimizes the sum of the distances of all vertices in T from P plus the weight of path P. In this paper, we give efficient algorithms for finding a trunk of T. The first algorithm is a sequential algorithm which runs in $O(n)$ time, where n is the number of vertices in T. The second algorithm is a parallel algorithm which runs in $O(\log n)$ time using $O(n/\log n)$ processors on EREW PRAM model. We also present an application of trunk for efficient multicast in wireless ad hoc networks.

1. Introduction

In network theory, optimally locating a service facility/infrastructure for communication in a network has long been of great interest for decades. Due to the variety of facilities/infrastructures and different criteria for optimality, abundant optimization problems in networks have been defined and studied. The one that was extensively studied in the literature is the "core-family", a path-shaped or tree-shaped facility which minimizes the sum of the distances from the facility to all vertices in the network [6, 7, 8, 9].

However, in some applications for wireless ad hoc networks, the criteria for optimization are different with that of the traditional core-family: Instead of merely minimizing the cumulative distances, the cumulative distances plus the weight of the facility, defined as the sum of the weights of all edges in the facility, should be minimized. The path that satisfies the above criteria of optimality in a tree network is called a *trunk* in this paper. The contributions of this paper are:

1. Show a new type of optimization problem in tree networks;
2. Give efficient algorithms, both sequential and parallel ones, for constructing a trunk in a tree network; and
3. Give an example application of the new optimization problem in wireless ad hoc networks.

The rest of this paper is organized into five sections. In Section 2, we give the necessary notation, definitions, and preliminary results for trunk. In Section 3 we give the definition of the rooted trunk and theoretical background for the design of efficient algorithms to construct a trunk. The sequential algorithm and the parallel algorithm are presented in Sections 4 and 5, respectively. We give an application and conclude this paper in Section 6.

2. A Trunk of A Tree

Let $T = (V, E)$ be a free tree. The size of T, $|T|$, is the number of vertices in V. Each edge $e \in E$ has an associated positive weight $w(e)$. If $w(e) = 1$ for every $e \in E$ then T is unweighted, otherwise weighted. Let $P = (V', E')$, $V' \subseteq V$ and $E' \subseteq E$, be a path in T. For any two vertices $u, v \in V$, let $P(u, v)$ be the unique path from u to v, and let the distance from u to v, $d(u, v) = \sum_{e \in E'} w(e)$. Let the degree of vertex $v \in V$, denoted as $\deg(v)$, be the number of vertices adjacent to v. A leaf of T is a vertex $l \in V$ with $\deg(l) = 1$. For any path P, let $w(P) = \sum_{e \in E'} w(e)$ be the weight of path P. When $P = P(u, v)$ we have $w(P) = d(u, v)$. For $v \in V$, we define the distance $d(v, P) = \min_{u \in V'} d(u, v)$, and the cumulative distance $\delta(P) = \sum_{v \in V} d(v, P)$. When P is a single vertex u, we have $\delta(u) = \sum_{v \in V} d(v, u)$.

A trunk is a path P in T which minimizes $\delta(P) + w(P)$. Notice that the core that was traditionally studied in the literature is defined as a path P that minimizes $\delta(P)$. For the tree in Figure 1, there is a unique core $C = a - b - c - d$ with $\delta(C) = 1$. However, there are total six trunks P_t: $b - c$, $b - c - d$, $b - c - e$, $a - b - c$, $a - b - c - d$, and $a - b - c - e$ with $\delta(P_t) + w(P_t) = 5$.

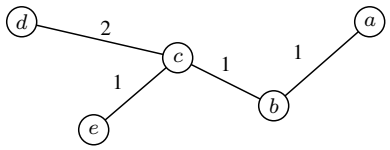
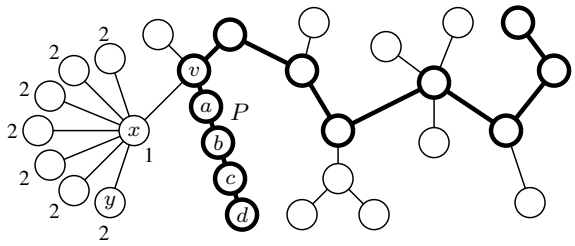
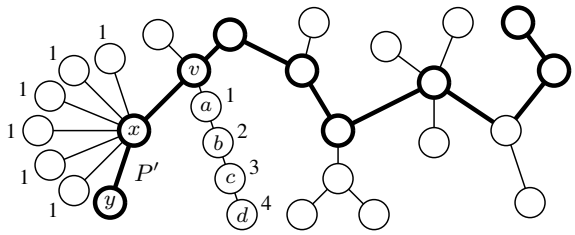


Figure 1. A simple example tree

The unweighted tree in Figure 2 shows that there exists a tree T in which a core is not a trunk and vice versa. For simplicity, we show only the calculation on the subtree as shown as in Figure 2, the contribution from the same (right) part had been exluded. Let path $P = v - a - b - c - d$ (Figure 2(a)) and path $P' = v - x - y$ (Figure 2(b)). Since $\delta(P) = 15$ and $\delta(P') = 16$, path P will be selected for core. However, since $\delta(P) + w(P) = 15 + 4 = 19$ and $\delta(P') + w(P') = 16 + 2 = 18$, P' will be selected for trunk.



(a) A core of tree



(b) A trunk of tree

Figure 2. A tree in which a core is not a trunk and vice versa

In general, the two end-vertices of a trunk may not necessarily be leaves. Let v_0 be an end-vertex of trunk P. For $A, B \subseteq V$, let $A/B = \{u | u \in A,\ u \notin B\}$. Let $N(v)$ be the set of vertices adjacent to v in T. By Lemma 1, the vertices in $N(v_0)/P$ must be leaves of T, and the extension of P to any leaf in $N(v)$ must be a trunk. Therefore, without loss of generality, we assume that the end-vertices of a trunk are leaves.

Lemma 1 Let $P(v, w)$ be a path in subtree $ST(v, u)$. We have $\delta(P(v, w)) \geq \delta(P(u, w)) + w((v, u))$ and $\delta(P(v, w)) = \delta(P(u, w)) + w((v, u))$ if and only if u is a leaf.

Proof: From the definition of $\delta(P)$, we have $\delta(P(v, w)) = \delta(P(u, w)) + w((v, u)) \times |ST(u, v)|$. $|ST(u, v)| = 1$ if and only if u is a leaf. Therefore, the lemma is true. ❑

For each $v \in V(T)$, there are $|N(v)|$ subtrees attached to v through edges $(v, u) \in E$, where $u \in N(v)$. Let $ST(u, v)$ be the subtree of T attached to v through the edge (v, u). Notice that $ST(v, u)$ is the subtree of T attached to u through the edge (u, v) (see Figure 3).

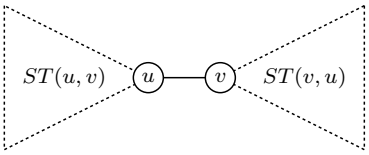


Figure 3. $ST(v, u)$ **and** $ST(u, v)$ **in** T

3. Rooted Trunk and Theory of Trunk

For efficiently constructing a trunk, we orient tree T into a rooted tree T_r with root r. For any vertex $v \in T_r$, we denote the parent of v as $p(v)$, the subtree rooted at v as T_v, and the number of vertices in T_v as $|T_v|$. Let a rooted trunk $P(r, l_0)$ be a path from root r to leaf l_0 which minimizes $\delta(P(r, l)) + w(P(r, l))$ among all paths from r to leaf l in T_r. We show that the problem of constructing a trunk in T can be reduced to the problem of constructing a rooted trunk in a rooted tree T_r. The following lemmas form the theoretical background for the reduction.

Lemma 2 Let rooted tree T_r be an orientation of T and $P(r, l_0)$ a rooted trunk in T_r. Then $P(r, l_0) \cap P(l_1, l_2) \neq \emptyset$ for any trunk $P(l_1, l_2)$ in T.

Proof: Assume that $P(r, l_0) \cap P(l_1, l_2) = \emptyset$ for a trunk $P(l_1, l_2)$. Let i be the closest vertex in $P(r, l_0)$ to $P(l_1, l_2)$ and j the closest vertex in $P(l_1, l_2)$ to $P(r, l_0)$ (see Figure 4). Let path $C = P(l_0, i) \cup P(i, j) \cup P(j, l_2)$. Since $P(r, l_0)$ is a rooted trunk, $\delta(P(l_0, i)) + w(P(l_0, i))) \leq \delta(P(l_1, i)) + w(P(l_1, i))$. Since i is not a leaf, by Lemma 1, we have $\delta(P(l_1, i)) + w(P(l_1, i)) < \delta((P(l_1, j)) + w(P(l_1, j))$. Similar, we have $\delta(P(l_0, j)) + w(P(l_0, j)) < \delta((P(l_0, i)) + w(P(l_0, i))$. From these equations, we get $\delta(P(l_0, j)) + w(P(l_0, j)) < \delta((P(l_1, j)) + w(P(l_1, j))$. This implies $\delta(C) + w(C) < \delta(P(l_1, l_2)) + w(P(l_1, l_2))$, a

contradiction to the fact that $P(l_1, l_2))$ is a trunk. Therefore, the lemma must be true. ❑

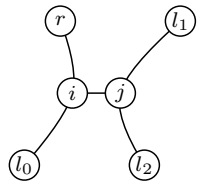


Figure 4. $P(r, l_0)$ **is a rooted trunk in** T_r **and** $P(l_1, l_2)$ **is a trunk**

Theorem 1 Let rooted tree T_r be an orientation of T and $P(r, l_0)$ a rooted trunk in T_r. Then a rooted trunk in rooted tree T_{l_0}, a new orientation of T, is a trunk in T.

Proof: Let $P(l_0, l'_0)$ be a rooted trunk in T_{l_0}. Assume that $P(l_1, l_2)$ is a trunk in T. From Lemma 2, $P(l_0, l'_0) \cap P(l_1, l_2) \neq \emptyset$. Let $P(i, j) = P(l_0, l'_0) \cap P(l_1, l_2)$, where i is the vertex in $P(i, j)$ closest to vertices l_0 and l_1 (see figure 5). Since $P(r, l_0)$ is a rooted trunk, we have $\delta(P(l_0, i)) + w(P(l_0, i)) \leq \delta(P(l_1, i)) + w(P(l_1, i))$. Similarly, Since $P(l_0, l'_0)$ is a rooted trunk, we have $\delta(P(l'_0, j)) + w(P(l'_0, j)) \leq \delta(P(l_2, j)) + w(P(l_2, j))$. Therefore, we get $\delta(P(l_0, l'_0)) + w(P(l_0, l'_0)) \leq \delta(P(l_1, l_2)) + w(P(l_1, l_2))$. We conclude that $P(l_0, l'_0)$ is a trunk in T. ❑

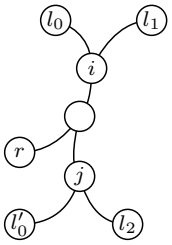


Figure 5. $P(i, j) = P(l_0, l'_0) \cap P(l_1, l_2)$

From Theorem 1, the problem of constructing a trunk in T can be solved as follows:

1. Orient tree T into a rooted tree T_r with an arbitrary vertex r;
2. Construct a rooted trunk $P(r, l_0)$ in T_r;
3. Re-orient T into T_{l_0};
4. Construct a rooted trunk in T_{l_0}.

In Figure 6, we first show an example tree T with an arbitrarily selected vertex r in Figure 6(a). Then, in Figure 6(b), we show the rooted tree T_r and a rooted trunk $P(r, l_0)$ in T_r. We have $\delta(P(r, l_0)) + w(P(r, l_0)) = 26 + 5 = 31$. Finally in Figure 6(c), we show the rooted tree T_{l_0} and a rooted trunk $P(l'_0, l_0)$ in T_{l_0}. The path $P(l_0, l'_0)$ is a trunk in T, and we have $\delta(P(l_0, l'_0)) + w(P(l_0, l'_0)) = 19 + 7 = 26$.

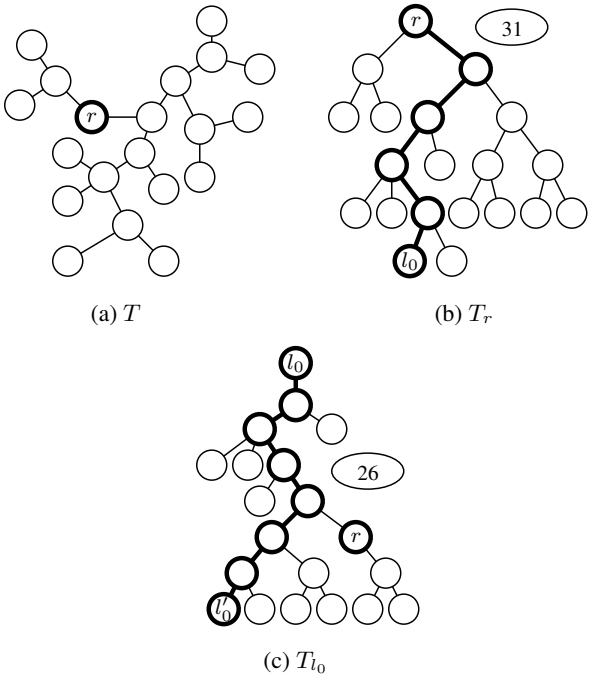


Figure 6. (a) An example tree; (b) a rooted trunk in T_r**; (c) a rooted trunk in** T_{l_0} **which is a trunk in** T

It was well known that orientation of a tree can be done optimally both sequentially and in parallel. The key for an efficient algorithm to construct a trunk lies on efficiently constructing a rooted trunk. We present two optimal algorithms, sequentially and in parallel, for constructing a rooted trunk in a rooted tree T_r in the next two sections, respectively. Based on these two algorithms, the problem of constructing a trunk in tree T can be solved optimally both sequentially and in parallel.

4. Sequential algorithm for finding a rooted trunk

In order to find $\min_{l \text{ is a leaf in } T_r}\{\delta(P(r, l)) + w(P(r, l))\}$, the value of $|T_v|$ for every node v and $\delta(l)$ for every leaf l in T_r should be computed in advance. Computing $|T_v|$ for every node v in $O(n)$ time is trivial. We show in the

following lemma, Lemma 3, that $\delta(l)$, for every leaf l in T_r can be computed in $O(n)$ time.

Lemma 3 Given a weighted tree T, $\delta(v)$ for $v \in V(T)$ can be computed in $O(n)$ time.

Proof: First, we orient T into a rooted tree T_r in $O(n)$ time. $\delta(r)$ can be computed in $O(n)$ time through a post-order traversal of T_r by the formula $\delta(r) = \sum_{v \in V(T_r)} d(v,r) = \sum_{v \in V(T_r)/\{r\}} w((v,p(v))) \times |T_v|$. Then, $\delta(v)$ for all $v \neq r$ can be computed in $O(n)$ time through a pre-order traversal of T_r by the formula $\delta(v) = \delta(p(v)) - w((v,p(v))) \times (|T_v| - (n - |T_v|)) = \delta(p(v)) - w((v,p(v))) \times (2|T_v| - n)$. ❑

The sequential algorithm for finding a rooted trunk of a rooted tree T_r works from bottom up: once the rooted trunks of all subtrees T_w, where node w is a child of node v, have been found, the rooted trunk of subtree T_v can be computed. Computing a rooted trunk of subtree T_v is done as follows: Let $v_1, \ldots, v_k$ be the children of v in T_r. Let $P(l_i, v_i)$ be the rooted trunk in subtree T_{v_i}. From the formula $\delta(P(l_i,v)) + w(P(l_i,v)) = \delta(P(l_i,v_i)) + w(P(l_i,v_i)) - w((v,v_i)) \times (|T_{v_i}| - 1)$, the algorithm computes $\delta(P(l_i,v)) + w(P(l_i,v))$ and finds the minimum for all $i, 1 \leq i \leq k$. The path that reaches this minimum for v is a rooted trunk in subtree T_v. The algorithm is described formally in Algorithm 1. For finding a rooted trunk in T_r, we simply call Find_rooted_trunk(T_r).

Algorithm 1 (Find_rooted_trunk(T_v))
Input: rooted tree T_v
Output: leaf l_v and $\delta(P(l_v,v)) + w(P(l_v,v))$
/* $P(l_v,v)$ is a rooted trunk in T_v. */
begin
if v is a leaf **then** return $((v,\delta(v))$
else /* assume $v_i, 1 \leq i \leq k$ are the children of v. */
for $i \leftarrow 1$ to k **do**
$(l_i, value_i) \leftarrow$ Find_rooted_trunk(T_{v_i});
endfor
$min \leftarrow \min_{1 \leq i \leq k}\{value_i - w((v,v_i)) \times (|T_{v_i}| - 1)\}$;
$min\_leaf \leftarrow l_j$; /*$value_j - w((v,v_j)) \times (|T_{v_j}| - 1) = min$*/
return(min_leaf, min);
endif
end

Figure 7 and Figure 8 show how the algorithm works for the example tree T_r in Figure 6(b) and T_{l_0} in Figure 6(c), respectively.

Theorem 2 A rooted trunk in a rooted tree T_r can be found in $O(n)$ time.

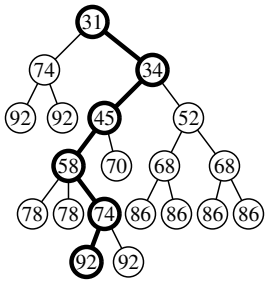


Figure 7. A working example 1 for Algorithm 1

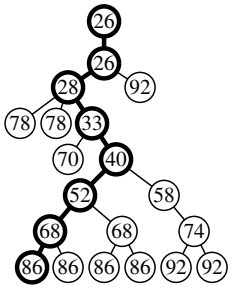


Figure 8. A working example 2 for Algorithm 1

Proof: From the definition of rooted trunk and the formula $\delta(P(l,p(v))) + w(P(l,p(v))) = \delta(P(l,v)) + w(P(l,v)) - w((p(v),v)) \times (|T_v| - 1)$, it is easy to see that Algorithm 1 finds a rooted trunk in T_r. The algorithm performs a post-order traversal of T_r with $O(1)$ computations per step. Therefore, rooted trunk in T_r can be found in $O(n)$ time. ❑

Corollary 1 A trunk in a tree T can be found in $O(n)$ time.

Proof: The rooted T_r can be obtained from an orientation of tree T in $O(n)$ time. From Theorem 2, finding a rooted trunk in a rooted tree takes $O(n)$ time. A trunk can be found by performing twice the process of a tree orientation followed by finding a rooted trunk in the rooted tree. Therefore, the corollary is true. ❑

5. Parallel algorithm for finding a rooted trunk

The parallel computation model used in this paper is EREW PRAM. A PRAM consists of a collection of autonomous processors, each having access to a common memory. At each step, every processor performs the same instruction, with a number of processors masked out. In the

EREW PRAM model, a memory location cannot be simultaneously accessed by more than one process. Two techniques of parallel computation for trees, parallel Euler-tour and tree contraction that are introduced briefly below, will be used in the proposed parallel algorithm.

Given a tree T, the Euler path of T is a linear list of $2n-2$ directed edges, where $n = |V(T)|$. We describe Euler tour briefly as follows: For each vertex v in $V(T)$, we set next$(u_i \rightarrow v) = v \rightarrow u_{(i+1) \bmod k}$ for $0 \leq i \leq k-1$, where $k = \deg(v)$ and $u_i, 0 \leq i \leq k-1$, are the neighbors of v in T (see Figure 9). For tree orientation with root r, we set next$(u_{\deg(r)-1} \rightarrow r) = end_of_list$. By applying the optimal list ranking algorithm [2], each edge in T can be oriented away from the root r by assigning it the direction of one of its two directed edges in the Euler path which has larger rank. We conclude that tree T can be oriented into a rooted tree T_r in $O(\log n)$ time using $O(n/\log n)$ processors on an EREW PRAM.

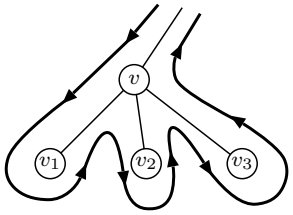


Figure 9. An Euler tour

The tree contraction is a parallel technique on a rooted tree T_r which reduces T_r in parallel to its root by a sequence of vertex removals. The tree contraction algorithm of Abrahamson et al. [1], which will be used in our algorithm, is described briefly below. First, the rooted tree T_r should be presented as a binary tree through the standard transformation in which a node v with $k > 2$ children is presented as a binary subtree of hight $k-1$ with $k-2$ dummy nodes of v (see Figure 10).

Figure 10. A binary tree presentation of a tree

A tree contraction sequence of length s is defined as a set of binary trees $\{BT_i | 1 \leq i \leq s\}$ such that BT_i is obtained from BT_{i-1} by one of the following two operations:

1. $prune(v)$: leaf v in BT_{i-1} is removed;
2. $bypass(v)$: a non-root node v in BT_{i-1} with only one child is removed and the parent of v becomes the parent of the unique child of v.

It was showed in [1] that every binary tree has an optimal contraction sequence of length $O(\log n)$ and this sequence can be obtained in $O(\log n)$ time using $O(n/\log n)$ processors on an EREW PRAM. The following lemma shows that $\delta(v)$ for all $v \in V(T)$ can be computed efficiently in parallel.

Lemma 4 Given a rooted tree T_r, $|T_v|$ and $\delta(v)$ for all $v \in V(T)$ can be computed in $O(\log n)$ time using $O(n/\log n)$ processors on an EREW PRAM.

Proof: The number of nodes in subtree T_v rooted at v, $|T_v|$, for all $v \in V(T_r)$, can be computed in parallel using Euler-tour technique as follows: For the directed edges e oriented away from root r, set $f(e) = 0$; and for the directed edges e' oriented toward r, set $f(e') = 1$. Then, perform parallel prefix computation of f on the list of Euler tour and let the result be the function g. It is easy to see that $g(e'_0)$ equals to $|T_v|$, where $e'_0 = (v, u)$ is the unique directed edge starting from v that is oriented toward r. $\delta(v)$ for all $v \in V(T_r)$ can be computed in parallel using tree contraction and expansion techniques. First, we show that $\delta(r)$ can be computed in parallel using tree contraction as follows: For binary tree BT_1 in the sequence of tree contraction, set $f(v) = 1$ if v is a leaf; otherwise, $f(v) = 0$. While performing $prune(v)$, where v is a leaf in BT_i, we set $f(p(v)) = f(p(v)) + w((v, p(v)) \times |T_v|$. While performing $bypass(v)$, where v has a unique child u in T_i, we set $f(p(v)) = f(p(v)) + w((v, p(v)) \times |T_v|$ and $w((u, p(v)) = w((v, p(v)) + w((u, v))$, where $(u, p(v))$ is a new edge created by $bypass$ operation. It is easy to see from the definition of δ that $f(r) = \delta(r)$ while tree contraction is done. Finally, $\delta(v)$ for $v \neq r$ can be computed in parallel through the tree expansion technique which is the inverse process of tree contraction. The tree expansion expands r to T_r through inv_prune and inv_bypass operations. Initially, we set $g(r) = \delta(r)$. While performing $inv\_prune(v)$ to create a child u of v, we set $g(u) = g(v) + w((u, v)) \times (n - 2|T_u|)$ ($w((u, v))$ and $|T_u|$ are kept in v while performing $prune(u)$). Similarly, while performing $inv\_bypass(v)$, we set $g(u) = g(v) + w((u, v)) \times (n - 2|T_u|)$ and $w((u, z)) = w((v, z)) - w((u, v))$, where z is the unique child of v generated by $bypass(u)$ in tree contraction. From the formula $\delta(v) = \delta(p(v)) + w((v, p(v)) \times (n - 2|T_v|)$, it is easy to see that after tree expansion is done, $g(v) = \delta(v)$ for every $v \in T_r$. ❑

Next, we use the tree contraction technique on T_r to compute in parallel the value $\min\{\delta(P(r, l)) + w(P(r, l))\}$, where l is a leaf in T_r. For each node v in BT_i, a binary tree in the sequence of trees generated by the tree

Algorithm 2 (Parallel_rooted_trunk(T_r))
Input: rooted tree T_r
Output: $\min\{\delta(P(r,l)) + w(P(r,l))\}$, where l is a leaf in T_r
begin
In parallel, compute $\delta(l)$ for all leaf $l \in T_r$;
Transfer T_r into a binary tree presentation;
for each node $v \in T_r$ **do**
if v is a dummy node **then** $w(v, p(v)) = 0$;
if v is a leaf **then** $f(v) = \delta(v)$ **else** $f(v) = \infty$;
$g(v) = 0$;
endfor
Perform tree contraction to generate a sequence of binary trees $\{BT_i | 1 \leq i \leq s\}$,
where BT_1 is a binary representation of T_r and $BT_s = \{r\}$;
/* BT_i is obtained from BT_{i-1} by $prune(v)$ or $bypass(v)$ operations. */
for each $prune(v)$ **do**
$f(p(v)) = \min\{f(p(v)), f(v) - w((v, p(v)) \times (n - |T_v| - 1)\} + g(v)$;
endfor
for each $bypass(v)$ **do**
$f(p(v)) = \min\{f(p(v)), f(v) - w((v, p(v)) \times (n - |T_v| - 1)\} + g(v)$;
$w((u, p(v)) = w((v, p(v)) + w((u, v))$; /* u is the unique child of v. */
$g(u) = g(u) + w(v, p(v)) \times (|T_v| - |T_u|)$;
endfor
return($f(r)$);
end

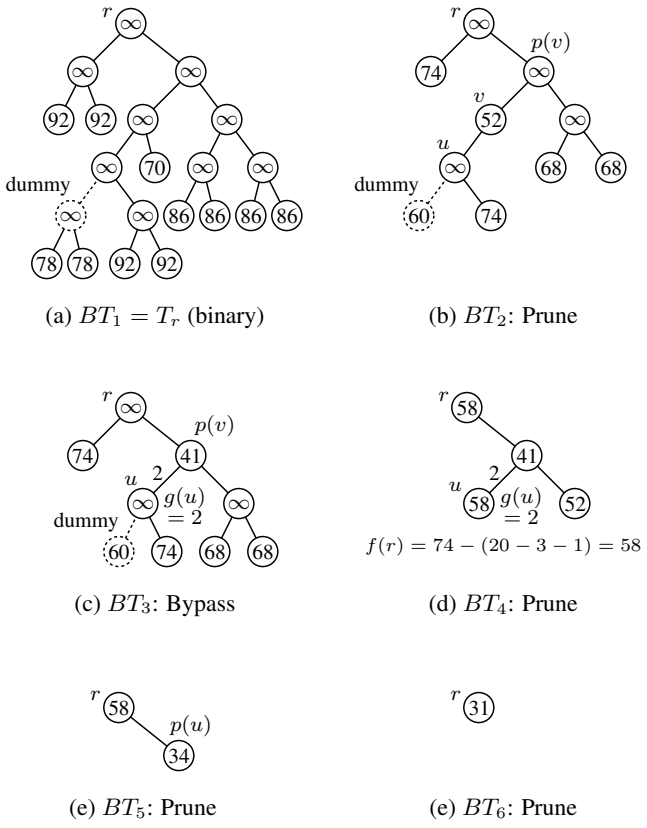
contraction, we compute two functions, $f(v)$ and $g(v)$. If v is a leaf in BT_i, the value of function $f(v)$ represents $\min\{\delta(P(v,l)) + w(P(v,l))\}$, where l is a leaf in T_v, the subtree of T_r rooted at v. Therefore, when tree contraction ends, $f(r)$ will be the answer we want. The function $g(v)$ is used to adjust the distance saving created by path extension from v to $p(v)$ where edge $(v, p(v))$ is an edge created by the $bypass$ operation. We will explain this effect in details later.

Initially, in BT_1, we set $f(v) = \delta(v)$ if v is a leaf, otherwise, $f(v) = \infty$; $g(v) = 0$ for all v. Then, for each $prune(v)$, we should update the value of $f(p(v))$ by the formula $f(p(v)) = \min\{f(p(v)), f(v) - w((v, p(v)) \times (n - |T_v| - 1)\}$; And, for each $bypass(v)$, we should update the value of $f(p(v))$ by the formula $f(p(v)) = \min\{f(p(v)), f(v) - w((v, p(v)) \times (n - |T_v| - 1)\}$ and give the new edge $(u, p(v))$ weight $w((v, p(v)) + w((u, v))$ so that, when perform $prune(u)$ the distance saving for the path extension from u to $p(v)$ calculated by the formula $w(u, p(v)) \times (n - |T_u| - 1)$ will be correct. However, the effect of the nodes in T_v on the distance saving due to the path extension from u to $p(v)$ is over-calculated in the formula (the distance saving due to the path extension for the nodes in T_v is $w(u, v)$, not $w(u, p(v))$). Therefore, a factor $g(u) = w(v, p(v)) \times (|T_v| - |T_u|)$ is needed to compensate this over-calculation. That is, the update formula for $f(p(v))$ should be $f(p(v)) = \min\{f(p(v)), f(v) - w((v, p(v)) \times (n - |T_v| - 1)\} + g(v)$. The algorithm is shown in Algorithm 2.

Figure 11 shows how the algorithm works for the example tree T_r in Figure 6(b). Notice that BT_3 comes from BT_2 by a $bypass(v)$. From the algorithm, we get $f(p(v)) = 52 - (20 - 8 - 1) = 41$, $w(u, p(v)) = 1 + 1 = 2$, and $g(u) = 8 - 6 = 2$. During the $prune(u)$ operation in BT_4, we get $f(p(u)) = \min\{41, 58 - 2 \times (20 - 6 - 1) + 2\} = 34$. Finally, since $BT_6 = \{r\}$ comes from BT_5 by a $prune$ operation, we get $f(r) = \min\{58, 34 - (20 - 16 - 1) + 0\} = 31$.

Theorem 3 Given a rooted tree T_r, $\min\{\delta(P(r,l)) + w(P(r,l))\}$, where l is any leaf in T_r, can be computed in $O(\log n)$ time using $O(n/\log n)$ processors on an EREW PRAM.

Proof: We apply tree contraction on tree T_r with $O(1)$ additional computations of functions f and g on $BT_i, 1 \leq i \leq s$, while performing $prune$ and $bypass$ operations. Initially, we set $g(v) = 0$ and $f(v) = \delta(v)$ if v is a leaf in BT_1; otherwise $f(v) = \infty$. If v is a dummy



(a) $BT_1 = T_r$ (binary) (b) BT_2: Prune

(c) BT_3: Bypass (d) BT_4: Prune

(e) BT_5: Prune (e) BT_6: Prune

Figure 11. A working example for Algorithm 2

node, since v and $p(v)$ are identical in the original tree, we should set $w(v, p(v)) = 0$. While performing $prune(v)$, where v is a leaf, we set $f(p(v)) = \min\{f(p(v)), f(v) - w((v, p(v)) \times (n - |T_v| - 1)\} + g(v)$, where the compensation factor $g(v)$ is introduced by the $bypass$ operation as explained before. The above formula for updating $f(p(v))$ comes from the fact $\delta(p(v)) + w(P(p(v), l) = \delta(P(v, l) + w(P(v, l)) - w((v, p(v)) \times (n - |T_v| - 1)$ for any path $P(v, l) \subset T_v$. While performing $bypass(v)$, where v has a unique child u in T_i, we set $f(p(v)) = \min\{f(p(v)), f(v) - w((v, p(v)) \times (n - |T_v| - 1)\} + g(v)$ and $w((u, p(v)) = w((v, p(v)) + w((u, v))$, where $(u, p(v))$ is a new edge created by $bypass$ operation. We also modify the compensation factor $g(v)$ by setting $g(v) = g(v) + w(v, p(v)) \times (|T_v| - |T_u|)$. Similar to the proof of Lemma 4, it is easy to verify that $f(r) = \min\{\delta(P(r, l)) + w(P(r, l))\}$ while tree contraction is done. ❑

Corollary 2 A rooted trunk in a rooted tree T_r can be found in $O(\log n)$ time using $O(n/\log n)$ processors on an EREW PRAM.

Proof: The leaf l_0 that achieves the minimum value in Theorem 3 can be obtained easily with $O(1)$ additional bookkeeping process per vertex while performing $prune$ and $bypass$ operations. ❑

Corollary 3 A trunk in a tree T can be found in $O(\log n)$ time using $O(n/\log n)$ processors on an EREW PRAM.

Proof: The orientation of tree T and finding a rooted trunk in a rooted tree take $O(\log n)$ time using $O(n/\log n)$ processors on an EREW PRAM. A trunk can be found by performing twice the process of a tree orientation followed by finding a rooted trunk in the rooted tree. Therefore, the corollary is true. ❑

6. An Application and Concluding remarks

The concept of trunk in tree networks has applications on efficient multicast for wireless ad hoc networks. Overlay multicast protocols [3, 4, 5] are used for efficient multicast at application layer. It constructs a virtual mesh spanning all member nodes of a multicast group and employs standard unicast routing to fulfill multicast functionality. A spanning tree T on the virtual mesh, an weighted tree in which the weight of an edge (u, v) is the number of hops from u to v in the original network, is commonly used by overlay multicast protocols for efficient multicast.

However, maintaining the spanning tree for mobile ad hoc networks is expensive. A path in the tree can be used to reduced the cost for maintenance due to its simple, linear structure. The total cost for multicast using a path P contains two parts: the cost for broadcasting inside P and the cost for unicasting from the nodes inside P to the nodes outside. Formally speaking, the cost of the first part is $\sum_{e \in P} w(e)$ and the cost of the second part is $\sum_{v \in V(T)} d(v, P)$. Therefore, the problem of finding a path with minimum communication cost is exactly the problem of finding a trunk in T. Figure 12 shows a trunk of a spanning tree on a virtual mesh of size 50 in an ad hoc wireless network of 100 nodes. The solid cycles in the figure represent the group members and the trunk is marked with thickest lines.

We had presented a new optimization problem, its solutions and efficient algorithms on theoretical models. We also showed a possible application on communication in ad hoc wireless networks. It is interesting to develop efficient algorithms for constructing a trunk in tree networks on the models that run asynchronously and use local information only.

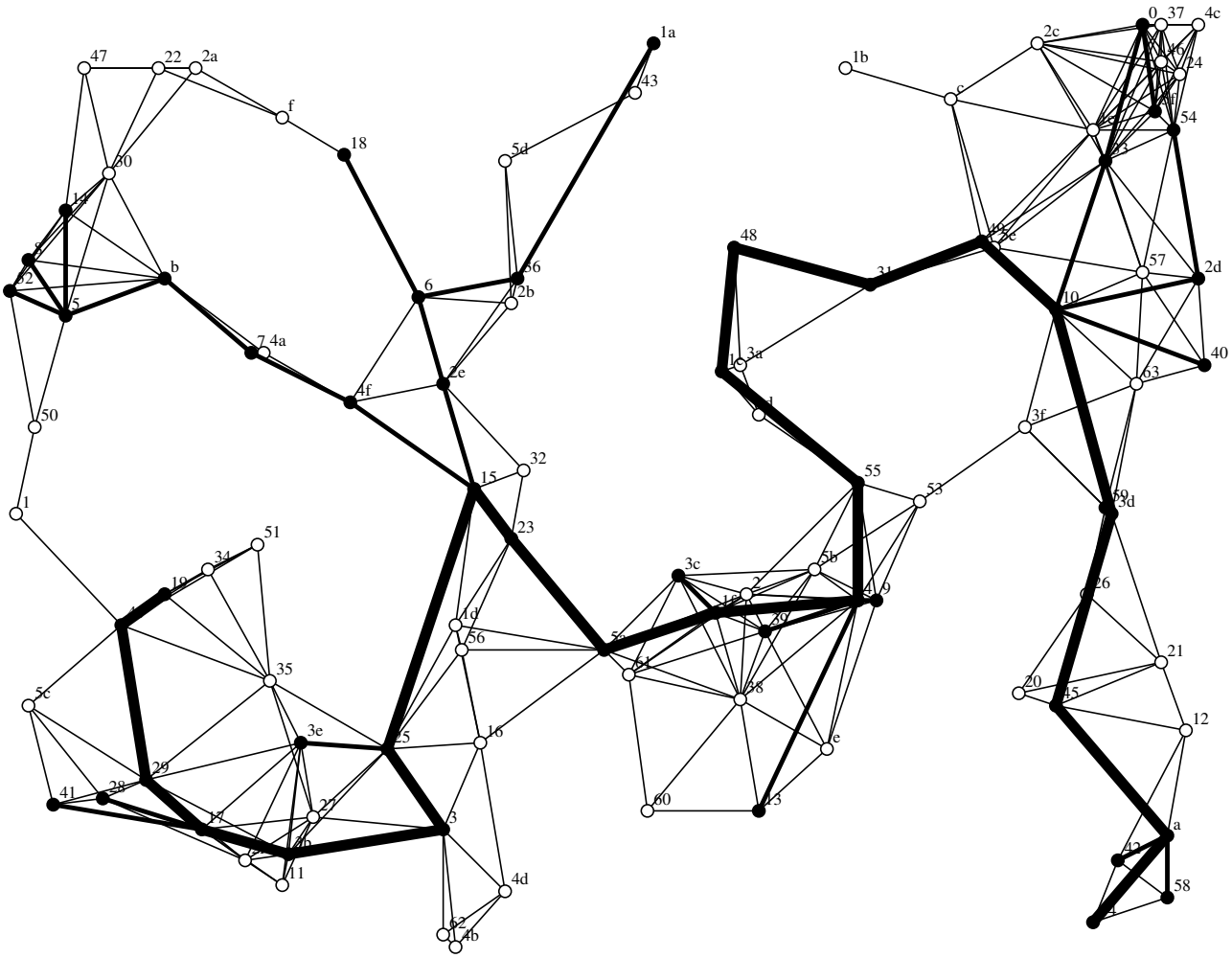


Figure 12. A trunk of a spanning tree on virtual mesh

References

[1] K. Abrahamson, N. Dadoun, D. G. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10(2):287–302, Jun. 1989.

[2] R. Cole and U. Vishkin. Approximate parallel scheduling. part i: The basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM Journal of Computing*, 17(1):128–142, 1988.

[3] C. Cordeiro, H. Gossain, and D. Agrawal. Multicast over wireless mobile ad hoc networks: Present and future directions. *IEEE Network*, 17(1):52–59, Jan. 2003.

[4] C. Gui and P. Mohapatra. Efficient overlay multicast for mobile ad hoc networks. In *Proceedings of IEEE Wireless Communications and Networking Conference (WCNC2003)*, Mar. 2003.

[5] J. Janotti, D. Gifford, K. Johnson, M. Kaashoek, and J. O'Toole. Overcast: Reliable multicasting with an overlay network. In *Proceedings of 4th Symp. Operating Systems Design and Implementation*, pages 197–212, Oct. 2000.

[6] C. A. Morgan and P. J. Slater. A linear algorithm for a core of a tree. *Journal of Algorithms*, 1(3):247–258, 1980.

[7] S. Peng and W. Lo. A simple optimal parallel algorithm for a core of a tree. *Journal of Parallel amd Distributed Computing*, 20(3):388–392, 1994.

[8] S. Peng and W. Lo. Efficient algorithms for finding a core with a specific length. *Journal of Algorithms*, 20(3):445–458, 1996.

[9] S. Peng, A. B. Stephens, and Y. Yesha. Algorithms for a core and k-tree core of a tree. *Journal of Algorithms*, 15(1):143–159, Jul. 1993.