K-Trunk and Efficient Algorithms for Finding a K-Trunk on a Tree Network

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 k-trunk is a subtree T_k with k leaves in T which minimizes the sum of the distances of all vertices in T from T_k plus the weight of T_k. In this paper, we first give motivation for using a k-trunk in tree networks. Then we develop efficient algorithms for finding a k-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.

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" [6, 9, 7, 8]; a path-shaped or tree-shaped facility which minimizes the sum of the distances from the facility to all vertices in the network.

However, in some applications for mobile wireless ad hoc networks, the criteria for optimization are different from the traditional minimization criteria: 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 subtree with exactly k leaves that satisfies the above criteria of optimality in a tree network is called a *k-trunk* in this paper. We will show an example application of k-trunk for efficient multicast in mobile wireless ad hoc networks in the last section of the paper.

The contributions of this paper are:

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

The rest of this paper is organized into five sections. In Section 2, we give the necessary notation, definitions, and preliminary results for k-trunk. In Section 3, we give theoretical background and basic algorithms for constructing a trunk (2-trunk) and k-trunk ($k \geq 2$) in general. The basic algorithms in Section 3 are the corner-stones for efficiently constructing a trunk and a k-trunk. The sequential and the parallel algorithms for finding a rooted trunk as well as partitioning of a rooted tree are presented in Sections 4 and 5, respectively. We give an application and conclude this paper in the last section.

2. Trunk and k-trunk

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 $T' = (V', E')$, $V' \subseteq V$ and $E' \subseteq E$, be a subtree in T. Let the weight of T', $w(T') = \sum_{e \in E'} w(e)$. 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, $d(u, v) = \sum_{e \in P(u,v)} w(e)$. 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 $\deg(l) = 1$. When $P = P(u, v)$, we have $w(P) = d(u, v)$. For $v \in V$, we define the distance $d(v, T') = \min_{u \in V'} d(u, v)$, and the cumulative distance $\delta(T') = \sum_{v \in V} d(v, T')$. When T' is a single vertex u, we have $\delta(u) = \sum_{v \in V} d(v, u)$.

A k-trunk is a subtree T_k with k leaves in T which minimizes $\delta(T_k)+w(T_k)$. Notice that the k-core in [9] is defined as a subtree T_k with k leaves that minimizes $\delta(T_k)$ only. In Figure 1, a 4-trunk of an example tree is shown in bold lines.

A subtree with two leaves is just a path. we call the path presented by a 2-trunk T_2 *trunk*. A trunk has simple struc-

ture, and the theory on trunk lays the ground for that of k-trunk.

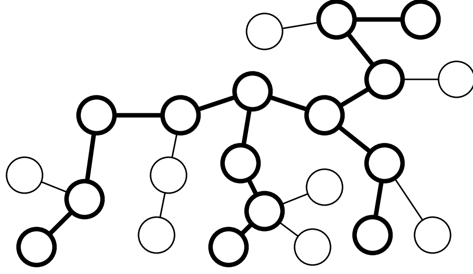


Figure 1. An example tree and a 4-trunk

Let $N(v)$ be the set of vertices adjacent to v in T. 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 2).

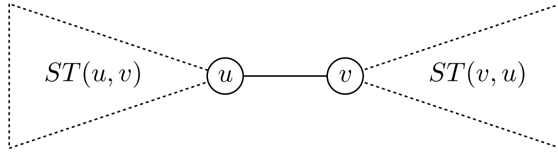


Figure 2. $ST(v,u)$ and $ST(u,v)$ in T

In general, the leaves of a k-trunk may not necessarily be leaves of tree T. For $A, B \subseteq V$, let $A/B = \{u | u \in A,\ u \notin B\}$. Let v_0 be a leaf of trunk P. 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. Similar argument holds for k-trunk. Therefore, without loss of generality, we assume that the leaves of a k-trunk are leaves of tree T.

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. ❏

3. Rooted trunk

For efficiently constructing a trunk, we orient the 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 3). 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. ❏

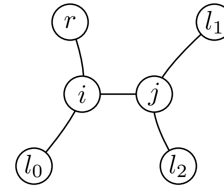


Figure 3. $P(r,l_0)$ is a rooted trunk in T_r and $P(l_1,l_2)$ is a trunk

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 4). 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. ❏

From Theorem 1, the problem of constructing a trunk in T can be solved by Algorithm 1.

Theorem 2 Given a weighted tree T, Algorithm 1 finds a trunk in T.

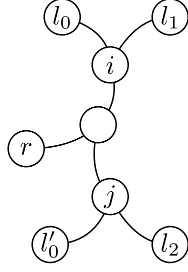


Figure 4. $P(i,j) = P(l_0, l'_0) \cap P(l_1, l_2)$

Algorithm 1 (Find_trunk(T))
Input: tree T
Output: a trunk P_{l_0,l_1}
begin

1. orient tree T into a rooted tree T_r with an arbitrary node r;
2. compute $\delta(l)$ for all leaf $l \in T_r$;
3. find a rooted trunk $P(r, l_0)$ in T_r;
4. re-orient T into a rooted tree T_{l_0};
5. find q rooted trunk $P(l_0, l_1)$ in T_{l_0};

return $P(l_0, l_1)$
end

Proof: Follow directly from Theorem 1. ❑

In Figure 5, we first show an example tree T with an arbitrarily selected vertex r in Figure 5(a). Then, in Figure 5(b), we show the rooted tree T_r and a rooted trunk $P(r, l_0)$ in T_r. We have $\delta(P(r, l_0)) + w(P(r, l_0)) = 33 + 5 = 38$. Finally in Figure 5(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 $\delta(P(l_0, l'_0)) + w(P(l_0, l'_0)) = 23 + 8 = 31$.

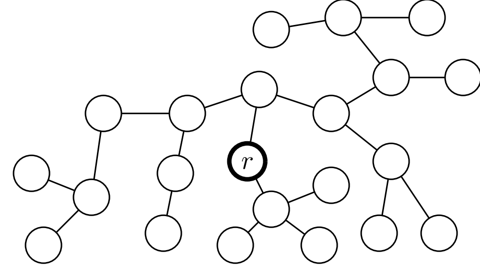
Next, we consider the problem of constructing a k-trunk ($k \geq 2$) in T. We first show some lemmas blow.

Lemma 3 Let T_k be a k-trunk and $P(l_1, l_2)$ is a trunk. Then $T_k \cap P(l_1, l_2) \neq \emptyset$.

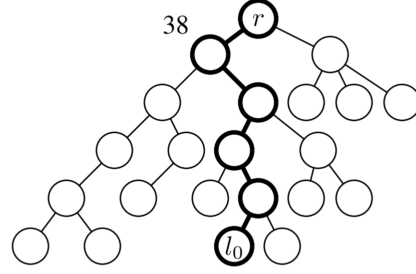
Proof: The proof of this lemma is similar to that of Lemma 2 and is omitted. ❑

Lemma 4 Let T_k be a k-trunk in tree T and $k > 2$. Then T_k must contain a trunk.

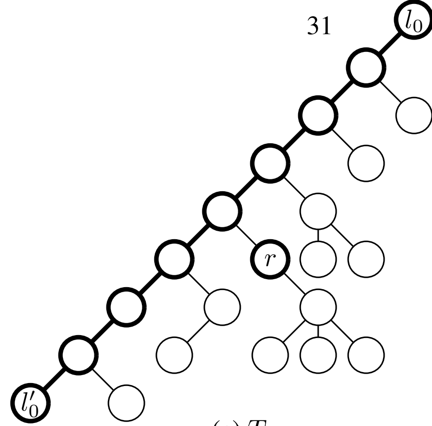
Proof: Assume that $P(l_1, l_2)$ is a trunk in T. From Lemma 3, $T_k \cap P(l_1, l_2) \neq \emptyset$. First, consider the case that $T_k \cap P(l_1, l_2) = \{u\}$, where u is a node in T. We claim that, for all the $k+2$ paths $P_i = P(l_i, u)$, where $l_i, 3 \leq i \leq k+2, \in T_k$, $\delta(P_i) + w(P_i)$ are the same. If for some $j > 2$, $\delta(P_j) + w(P_j) > \delta(P_1) + w(P_1)$ then we construct a subtree T'_k with k-leaves by replacing P_j with P_1. The fact $\delta(T'_k) + w(T'_k) < \delta(T_k) + w(T_k)$ contradicts to the definition that T_k is a k-trunk. If for some $j > 2$, $\delta(P_j) + w(P_j) < \delta(P_1) + w(P_1)$ then the fact



(a) T



(b) T_r



(c) T_{l_0}

Figure 5. (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

$\delta(P(l_j, l_2)) + w(P(l_j, l_2) < \delta(P(l_1, l_2)) + w(P(l_1, l_2))$ contradicts to the definition that $P(l_1, l_2)$ is a trunk. Similar argument holds for P_2. Therefore, the claim holds. In this case, the union of any two edge-disjoint paths in k paths, $P_j, j > 2$, forms a trunk in T. Next, consider the case that T_k and $P(l_1, l_2)$ share a subpath $P(u, v)$. One of two nodes u and v must be nonleaf (otherwise, $P(l_1, l_2) \subset T_k$). Let u be a nonleaf and $P(u, l_1)$ is edge-disjoint with T_k. Then the similar argument as the first case shows that for all paths $P_i = P(u, l_i), 1 \leq i \leq k+2$, $\delta(P_i) + w(P_i)$ are the same. ❑

The problem of finding a k-trunk can be solved by the following algorithm (Algorithm 2).

Algorithm 2 (Find_k-trunk(T))
Input: tree T
Output: a k-trunk T_k
begin
1. orient tree T into a rooted tree T_r with an arbitrary node r;
2. compute $\delta(l)$ for all leaf $l \in T_r$;
3. find a rooted trunk $P(r, l_0)$ in T_r;
4. re-orient T into a rooted tree T_{l_0}; and
5. find a partition of T_{l_0} which is a set of edge-disjoint paths $\mathcal{P} = \{P_i\}$ such that $\cup P_i = T_{l_0}$ and $\delta(P_j) + w(P_j)$ is a minimum among all paths that are edge-disjoint with $\cup_{i=1}^{j-1} P_i$;
6. $T_k = \cup_{i=1}^{k-1} P_i$;

return T_k
end

Theorem 3 Algorithm 2 finds a k-trunk in T.

Proof: We show it by induction. For $k = 2$, by Theorem 1, the algorithm finds a trunk. Assume that it is held for $k-1$. That is, T_{k-1} is a $(k-1)$-trunk. Then, since P_k is selected such that $\delta(P_k) + w(P_k) \leq \delta(P') + w(P')$ for all P' that are edge-disjoint with T_{k-1}. Therefore, by induction, the algorithm generates a k-trunk T_k. ❏

It is well known that the orientation of a tree and finding the kth smallest number in a set of n numbers can be done in optimal time sequentially and in parallel. Given a rooted tree T_r, we show in the next two sections how to do the following operations in optimal time both sequentially and in parallel:

1. Compute $\delta(v)$ for all nodes $v \in T_r$;
2. Construct a rooted trunk in T_r;
3. Partition T_r into a set of edge-disjoint paths $\mathcal{P}$.

4. Sequential algorithms for rooted trunk and partition

Given a rooted tree T_r, we first show that computing $\delta(v)$ for $v \in V(T)$ can be done in $O(n)$ time.

Lemma 5 Given a rooted tree T_r, $\delta(v)$ for $v \in T_r$ can be computed in $O(n)$ time.

Proof: The algorithm includes two phases. In Phase 1, we compute $\delta(r)$ by a bottom up (post-order) traversal, and in Phase 2, $\delta(v)$ for $v \neq r$ are computed by a top down (pre-order) traversal. From the definition of δ, it is easy to see that $\delta(r)$ can be computed in $O(n)$ through a post-order traversal. Since for any node v, we have $\delta(v) = \delta(par(v)) + w(v, par(v)) \times (|T_v| - (n - |T_v|)) = \delta(par(v)) + w(v, par(v)) \times (2|T_v| - n)$, $\delta(v)$ for all $v \neq r$ can be computed in $O(n)$ time by a pre-order traversal of tree T_r. ❏

For each node $v \in T_r$, we construct a rooted trunk in subtree T_v recursively 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 $i, 1 \leq i \leq k$. This is done recursively for every node in T_r. The path that reaches this minimum for v is a rooted trunk in subtree T_v. The recursive algorithm is described formally in Algorithm 3.

Algorithm 3 (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 $v_i, 1 \leq i \leq s$ are the children of v. */
for $i \leftarrow 1$ to s **do**
$(l_i, value_i) \leftarrow$ Find_rooted_trunk(T_{v_i});
endfor
$value_v \leftarrow \min_{1 \leq i \leq s}\{value_{v_i} - w(v, v_i) \times (|T_{v_i}| - 1)\}$;
$l_v \leftarrow l_{v_j}$, where $value_{v_j} - w(v, v_j) \times (|T_{v_j}| - 1) = value_v$;
return($l_v, value_v$);
endif
end

The following figures, Figure 6 and Figure 7, show how the algorithm work for the example rooted trees in Figure 5(b) and Figure 5(c), respectively.

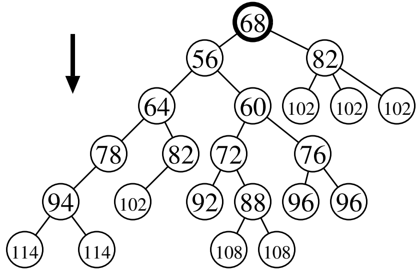
Theorem 4 A rooted trunk in a rooted tree T_r can be found in $O(n)$ time.

Proof: From the definition of rooted trunk and the formula in Lemma 1, it is easy to see that Algorithm 3 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. ❏

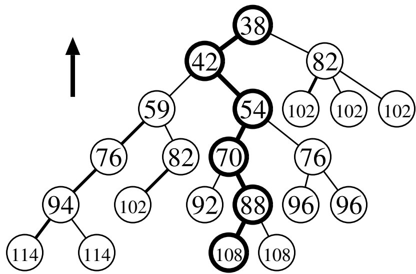
Corollary 1 A trunk in a tree T can be found in $O(n)$ time.

Proof: The rooted T_r can be obtained from an orientation of tree T in $O(n)$ time. From Theorem 4, finding a trunk in T takes $O(n)$ time. ❏

Next, we show how to partition a rooted tree efficiently. It is a bottom up computing method, similar to that of finding a rooted trunk. For each non-root node v do the fol-



(a) $\delta(l)$ of leaf



(b) min

Figure 6. Example #1 for Algorithm 3

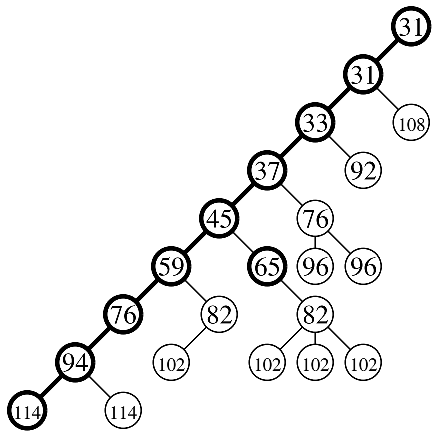


Figure 7. Example #2 for Algorithm 3

lows: Assume that node v has s children $v_1, \ldots, v_s$. Assume also that each subtree $T(v_i), 1 \leq i \leq s$, keeps record of $(Par_{v_i}, value_{v_i}, leaf_{v_i})$, where Par_{v_i} is a partition of $T(v_i) \cup \{(v, v_i)\}$, $P(leaf_{v_i}, v_i)$ is the rooted trunk of $T(v_i)$, and $vale_{v_i} = \delta(P(leaf_{v_i}, v_i)) + w(P(leaf_{v_i}, v_i))$. The algorithm finds $value_v$ and $leaf_v$ as that in Algorithm 3, and then, path $P(leaf_v, v)$ is extended to $par(v)$ if v is not the root. Par_v is constructed by union of $Par_{v_i}, 1 \leq i \leq s$, and $P(leaf_v, par(v))$ if v is not the root. The details of the recursive algorithm for partition is formally shown in Algorithm 4.

In Figure 8, we show a partition of the rooted tree T_{l_0} in Figure 5(c). In Figure 1, a 3-trunk of the tree T in Figure 5(b) obtained from the partition in Figure 8 is shown in Figure 9.

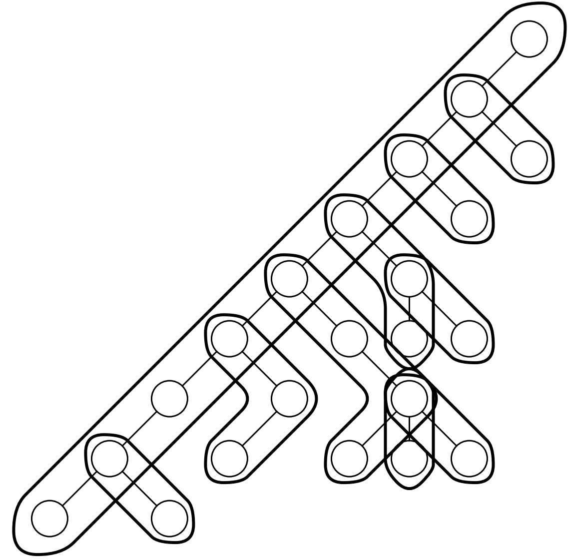


Figure 8. A partition of rooted tree T_{l_0}

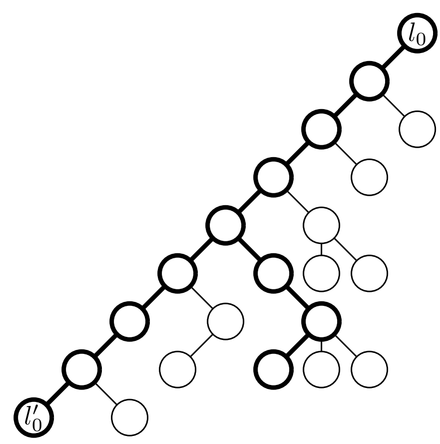


Figure 9. A 3-trunk in T

Theorem 5 Given weighted tree T, a k-trunk can be found in $O(n)$ time, where n is the number of nodes in T.

Proof: From Algorithms 3 and 4, and our discussion before, steps 1 - 4 of Algorithm 2 can be done in $O(n)$ time. To find a k-trunk, we first find the path P_{k-1} such that $\delta(P_{k-1}) + w(P_{k-1})$ is the $(k-1)$th smallest among all paths in Par_{l_0}. This can be done in $O(n)$ time. Next, we find the paths in Par_{l_0} such that $\delta(P) + w(P) \leq \delta(P_{k-1}) + w(P_{k-1})$ and let T_k be the union of these paths. It is easy to see that the union of these paths forms a k-trunk in T. ❏

Algorithm 4 (Partition(T_v))
Input: rooted tree T_v
Output: triple $(Par_v, leaf_v, value_v)$, where Par_v is the set of edge-disjoint paths and union of all paths in
$Par_v = T_v \cup \{(v, par(v))\}$ if $v \neq r$, $= T_r$, otherwise.
begin
if v is a leaf **then** return $((P(v, p(v)), v, \delta(v))$
else /* assume $v_i, 1 \leq i \leq s$ are the children of v. */
for $i \leftarrow 1$ to s **do**
Partition(T_{v_i});
endfor
$value_v \leftarrow \min_{1 \leq i \leq s}\{value_i - w(v, v_i) \times (|T_{v_i}| - 1)\}$;
$leaf_v \leftarrow leaf_{v_j}$, where $value_{v_j} - w(v, v_j) \times (|T_{v_j}| - 1) = value_v$;
if $v = r$ **then** $Par_v \leftarrow \cup_{i=1}^{s} Par(v_i)$
else $Par_v \leftarrow \cup_{i=1}^{s} Par(v_i) \cup \{P(leaf_v, p(v))\} - \{P(leaf_v, v)\}$;
return($Par_v, leaf_v, value_v$);
endif
end

5. Parallel algorithm for finding a k-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. Parallel Euler-tour and tree contraction that are the two well known techniques for parallel computation on trees.

Given a tree T with n nodes, the Euler path of T is a linear list of $2n-2$ directed edges (see Figure 10). It is well known that, by applying optimal list ranking algorithm [2] on the Euler path of T, 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. By applying Euler tour technique, $|T_v|$ for every node v in a rooted tree can also be computed in $O(\log n)$ time using $O(n/\log n)$ processors on an EREW PRAM.

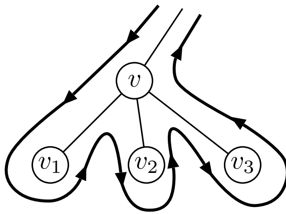


Figure 10. An Euler tour

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. In the tree contraction algorithm of Abrahamson et al. [1], 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 11).

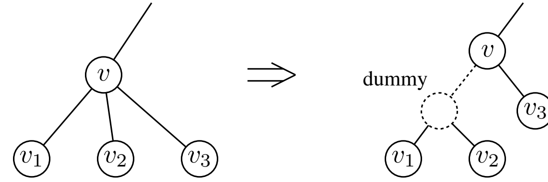


Figure 11. A binary tree presentation of a tree

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 with only one child is removed and the parent of v becomes the parent of the unique child of v.

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.

Using tree contraction technique, we can show that $\delta(v)$, for all $v \in V(T)$, can be computed optimally in parallel.

Lemma 6 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: We show that $\delta(v)$ for all $v \in V(T_r)$ can be computed in parallel using tree contraction and tree expansion (the reverse process of tree contraction) techniques. First, we show that $\delta(r)$ can be computed in parallel using tree contraction as follows: For binary tree T_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 T_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 $v \in T_r$. ❏

To find a rooted trunk of a rooted tree in parallel, we use the tree contraction 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 T_i, a binary tree in the sequence of trees generated by the tree contraction, we compute two functions, $f(v)$ and $g(v)$. If v is a leaf in T_i, the value of function $f(v)$ represents $\min\{\delta(P(v, l')) + w(P(v, l'))\}$, where l_v is a leaf in T_v, the subtree of T_r rooted at v. Therefore, when tree contraction ends, we have $f(r) = \delta(P(r, l_r)) + w(P(r, l_r))$, where $P(r, l_r)$ is a rooted trunk in T_r. 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 T_r, 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 5.

Figure 12 shows how the algorithm works for the example tree T_r in Figure 5(b). Notice that BT_3 comes from BT_2 by three $bypass$ operations. We explain the computations for $bypass(v)$ as shown in trees BT_2 and BT_3. From the algorithm, we get $f(p(v)) = 72 - (22 - 5 - 1) = 56$, $w(u, p(v)) = 1 + 1 = 2$, and $g(u) = 5 - 3 = 2$. During the $prune(u)$ operation in BT_3, we get $f(p(u)) = \min\{56, 88 - 2(22 - 3 - 1) + 2\} = 54$ in BT_4. Finally, $BT_6 = \{r\}$ comes from BT_5 by a $prune$ operation, and we get $f(r) = \min\{65, 42 - (22 - 17 - 1) + 0\} = 38$.

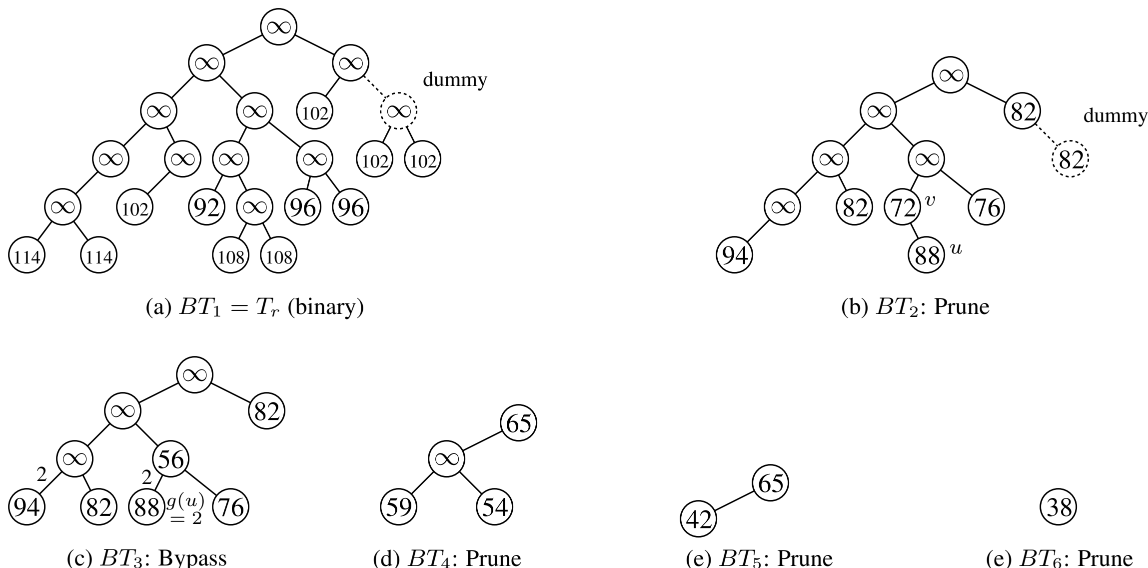
Theorem 6 Given a rooted tree T_r, $\min\{\delta(P(r, l)) + w(P(r, l))\}$, where l is any leaf in T_r, can be computed in $O(\log n)$ time using $O(n/\log n)$ processors on an EREW PRAM.

Proof: We apply tree contraction on tree T_r with $O(1)$ additional computations of functions f and g on $T_i, 1 \leq i \leq s$, while performing $prune$ and $bypass$ operations. Initially, we set $g(v) = 0$ and $f(v) = \delta(v)$ if v is a leaf in T_r; otherwise $f(v) = \infty$. If v is a dummy 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 for Lemma 3, 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. ❏

Algorithm 5 (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 $BT_1 = T_r$ and $BT_s = \{r\}$;
  /* BT_i is obtained from BT_{i-1} by $prune(v)$ and $bypass(v)$ 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



(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 12. A working example for Algorithm 5

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. ❏

The parallel algorithm for finding a k-trunk is similar to that of finding a trunk. The difference is that, after second orientation of tree T into rooted tree T_{l_0}, we use the following Parallel_Partition algorithm which is a parallel version of partitioning T into a set of edge-disjoint paths. Then,

Algorithm 6 (Parallel_Partition(T_r))
Input: rooted tree T_r
Output: $Par(r)$ /* Partition of T_r into edge-disjoint paths. */
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 $BT_1 = T_r$ and $BT_s = \{r\}$;
    /* BT_i is obtained from BT_{i-1} by $prune(v)$ and $bypass(v)$ operations. */
    for each $prune(v)$ **do**
        $f(p(v)) = \min\{f(p(v)), f(v) - w(v, p(v)) \times (n - |T_v| - 1)\} + g(v)$;
        if $f(v) - w(v, p(v)) \times (n - |T_v| - 1)\} + g(v) < f(p(v))$ **then** $leaf(p(v)) = leaf(v)$;
    endfor
    for each $bypass(v)$ **do**
        $f(p(v)) = \min\{f(p(v)), f(v) - w(v, p(v)) \times (n - |T_v| - 1)\} + g(v)$;
        if $f(v) - w(v, p(v)) \times (n - |T_v| - 1)\} + g(v) < f(p(v))$ **then** $leaf(p(v)) = leaf(v)$;
        $w(u, p(v)) = w(v, p(v)) + w(u, v)$; /* u is the unique child of v. */
        $g(u) = g(u) + w(v, p(v)) \times (|T_v| - |T_u|)$
    endfor
    Use Euler tour technique to find the node v_l closest to the root with $leaf(v) = l$, for every leaf $l \in T_r$;
    $Par = \cup_{l \text{ is a leaf and } v_l \neq r} \{P(p(v_l), l)\}$
    return(Par);
end

the parallel algorithm for finding a k-trunk is described in Algorithm 6.

Theorem 7 Given weighted tree T, a k-trunk can be found in $O(n/p)$ time, using $p = O(n/\log n)$ processor on EREW PRAM, where n is the number of nodes in T and $p \leq n/\log n$.

Proof: From Algorithms 5 and 6 and our discussion above, steps 1 - 4 of Algorithm 2 can be done in $O(n/p)$ time using $p = O(n/\log n)$ processors on EREW PRAM. Since finding path P_{k-1} such that $\delta(P_{k-1}) + w(P_{k-1})$ is the $(k-1)$th smallest in Par_{l_0} can be done in $O(n/p)$ time using $p = O(n/\log n)$ processors on EREW PRAM, it is easy to see from Algorithm 2 that a k-trunk in T can be found in $O(n/p)$ time using $p = O(n/\log n)$ processors on EREW PRAM. ❑

6. An application and concluding remarks

The concept of trunk in tree networks has applications on efficient multicast for mobile 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 k-trunk in the tree can be used to reduced the cost for maintenance due to its simplicity. Moreover, the parameter k can be computed such that the maintenance cost and the performance are balanced. The total cost for multicast using a subtree T' contains two parts: the cost for broadcasting inside T' and the cost for unicasting from the nodes inside T' to the nodes outside. Formally speaking, the cost of the first part is $\sum_{e \in T'} w(e)$ and the cost of the second part is $\sum_{v \in V(T')} d(v, P)$. Therefore, the problem of finding a subtree T_k with k leaves that minimizes communication cost is exactly the problem of finding a k-trunk in T. Figure 13 shows a 4-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.

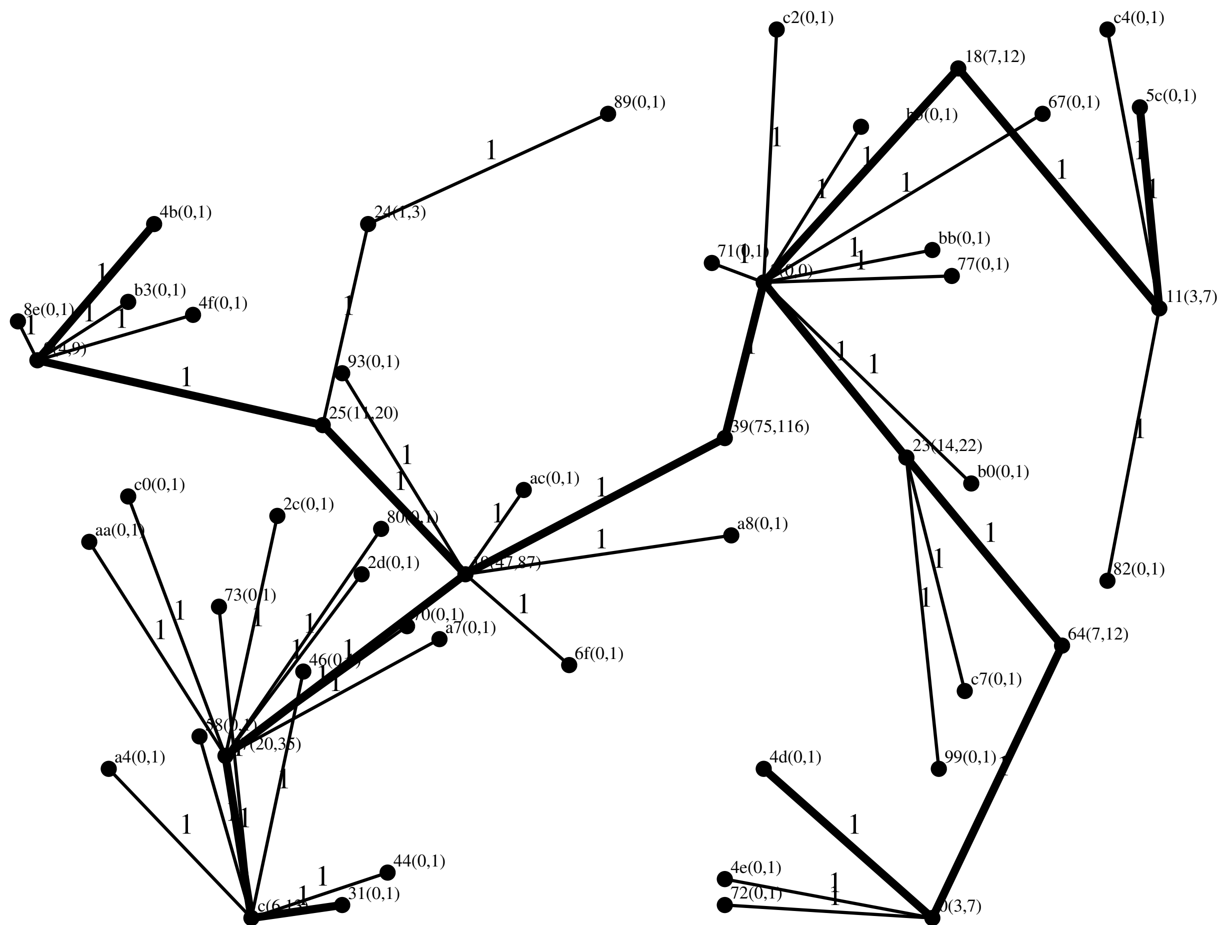


Figure 13. A 4-trunk of a spanning tree on virtual mesh

Some possible future work are as follows: To develop distributed algorithms that run asynchronously and use local information only, for constructing a k-trunk in tree networks; and to develop communication algorithms based on the algorithms proposed in this paper for applications on wireless ad hoc networks and do simulations for performance analysis. For example, multicast in wireless sensor networks in which the cost to be minimized is the energy used due to the fact that each node in a sensor network has very limited amount of energy supported by a portable battery.

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 and 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.