Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®
Information Processing Letters 88 (2003) 187–194
Information Processing Letters
www.elsevier.com/locate/ipl

Distributed algorithms for finding and maintaining a k-tree core in a dynamic network

Saurabh Srivastava, R.K. Ghosh *,[1]

Department of Computer Science and Engineering Indian Institute of Technology, Kanpur 208016, India

Received 4 June 2002; received in revised form 9 June 2003

Communicated by S. Albers

Abstract

A k-core C_k of a tree T is subtree with exactly k leaves for $k \leqslant n_l$, where n_l the number of leaves in T, and minimizes the sum of the distances of all nodes from C_k. In this paper first we propose a distributed algorithm for constructing a rooted spanning tree of a dynamic graph such that root of the tree is located near the center of the graph. Then we provide a distributed algorithm for finding k-core of that spanning tree. The spanning tree is constructed in two stages. In the first stage, a forest of trees is generated. In the next stage these trees are connected to form a single rooted tree. An interesting aspect of the first stage of proposed spanning algorithm is that it implicitly constructs the (convex) hull of those nodes which are not already included in the spanning forest. The process is repeated till all non root nodes of the graph have chosen a unique parent. We implemented the algorithms for finding spanning tree and its k-core. A core can be quite useful for routing messages in a dynamic network consisting of a set of mobile devices.
© 2003 Elsevier B.V. All rights reserved.

Keywords: k-core; Rooted core; Distributed algorithms; Routing; Ad hoc networks

1. Introduction

Consider a dynamic graph where nodes appear and disappear with time also the nodes move about changing connectivity at random. Such a situation can be witnessed in a network of autonomous mobile nodes. The connectivity of this graph would depend on placement of node with relative to each other. For a graph of this kind finding a rooted spanning tree itself can be challenging. In this paper first we propose a distributed algorithm for finding a rooted spanning tree with the root located being located towards the center of the graph. The algorithm works in two stages. In the first stage it finds a spanning forest. In the second stage the trees of the spanning forest are connected together to produce a tree with a single root. After we are able to obtain a spanning tree, a k-core can be constructed easily following the results from [2].

The paper is organized as follows. Section 2 deals with the problem definition and theoretical preliminar-

* Corresponding author.
E-mail addresses: saurabh.srivastava@ieee.org (S. Srivastava), rkg@cse.iitk.ac.in (R.K. Ghosh).

[1] Part of the work was done when the second named author was on a visit to INRIA, Sophia-Antipolis. The comments and suggestions of Frederic Cazals, INRIA, Sophia-Antipolis on various aspects of the work are gratefully acknowledged.

0020-0190/$ – see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/S0020-0190(03)00365-X

ies regarding the concept of the k-tree core of a tree. The distributed algorithms concerning finding a rooted spanning tree of a dynamic network are provided in Section 3. Section 4 is concerned with the proposed distributed algorithm for k-core along with its correctness issues. Section 5 deals with incremental maintenance of k-core as the network changes dynamically. Section 6 presents the results of experiments and Section 7 concludes the paper.

2. Problem definition

The distributed k-tree core algorithm involves two broad steps, namely, finding a spanning tree and then determining a k-tree core of that tree. In two subsequent sections, we discuss the two issues in detail. For the sake of completeness, in this section, we present the formal problem definition.

Given a tree T, a k-tree core [1] of T is defined to be the sub-tree T' with exactly k leaves which minimizes the sum of distances from all other nodes in T, where the distance of a node, v, from a tree is defined to be the minimum distance from v to any node in the tree. When $k = 2$, T' is a simple path and is called the 2-*core* or equivalently a *core* of the tree. More formally, let T denote an unrooted tree with vertex set $V(T)$. Let $|T|$ denote the number of vertices in T. A parent of a leaf node is defined to be a node adjacent to that leaf. For a vertex $v \in T$, we define the distance $d(v) = \sum_{u \in V(T)} d(u, v)$, where $d(u, v)$ is the length of the path from u to v. If P is a path in T, then we define the distance of P, $d(P) = \sum_{u \in V(T)} d(u, P)$, where $d(u, P) = \min_{v \in P} d(u, v)$. A path is called a *core* of T if its distance is the minimum amongst all paths in T. Let T' be a subtree of the tree T. We define the distance of T', $d(T') = \sum_{v \in V(T)} d(v, T')$, where $d(v, T') = \min_{u \in V(T')} d(u, v)$.

Definition. Let the number of leaves in a tree χ be denoted by *nleaves*(χ). A sub-tree, C_k, of a tree T containing $\min(k, \textit{nleaves}(C_k))$ leaves is called a k-tree core if $d(C_k) = \min_S d(S)$, where S is any subtree of T with k or less leaves.

We note that, as a consequence of optimization criterion, the leaves of a k-tree core will be the leaves of the tree T. We also note that the k-tree core of a tree need not be unique.

In [1], a centralized algorithm is presented for computing the core using the concept of the *distance saved* by a path. Let $P_{l,v}$ be a path from a vertex v to a leaf l. Then the *distance saved* by this path, *saved*($P_{l,v}$) is defined as $d(v) - d(P_{l,v})$. The value of *saved*($P_{l,v}$) is the most important measure for guiding the optimal selection of path extensions from the vertex v. From the definition, *saved*($P_{l,v}$) > *saved*($P_{l',v}$) iff $d(P_{l,v}) < d(P_{l',v})$. Consequently, minimizing $d(P_{l,r})$ is equivalent to maximizing *saved*($P_{l,r}$) over leaves $l \in T_r$.

3. A distributed algorithm for finding a spanning tree in a dynamic network

A topology graph for a mobile ad hoc network [4] can have any arbitrary structure. Hence, the first step in deriving some meaningful structure is to construct a (distributed) spanning tree.

To facilitate the description we maintain an attribute called *color* for every node which is indicative of the state of that node:

- white: parent pointer = $\emptyset$, child list = $\emptyset$.
- gray: parent pointer = $\emptyset$, child list $\neq \emptyset$.
- black: parent pointer $\neq \emptyset$.

In the following an α-cone denotes a region of space with the origin at the concerned node and bounded by two rays with an angle α between them. An α-cone is empty when there are no neighboring nodes in that region of space or if present they are all black. The nodes which are on the periphery will know their 'peripheral' status by virtue of not having any nodes, or only black nodes, in some α-*cone* as shown in Fig. 1. The algorithm has two phases as described below:

(1) Find Forest: In this phase, every node executes the procedure find_parent(). Nodes that are *black* do not accept any children since this may result in the formation of cycles. After completion of this phase a distributed forest is formed.
(2) Connect Trees: In the second phase we merge all the directed trees into a directed spanning tree. We first introduce some terminology:
   - A message, M_1, is said to have a *higher priority* than some other message, M_2, based on some

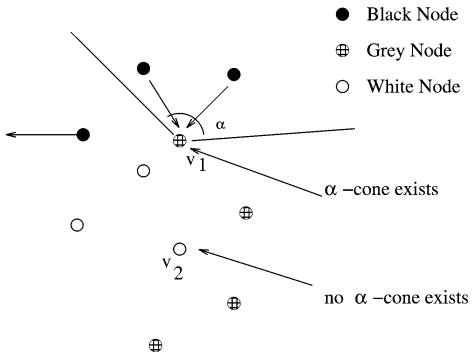

Fig. 1. A gray node, v_1, that has no white/gray node in some α-*cone* is free to choose its parent while v_2 cannot since it is unable to find any orientation of the rays which forms an empty α-cone.

property (for instance the MAC id) of the node initiating M_1 and M_2.

- Edges which are part of the trees and those which are outside the trees constructed in phase 1, are called *tree edges* and *non-tree edges*, respectively.
- Let *root*(v) denote the root of the tree containing v.

```
procedure find_parent() {
  set color ← white
  while (there is no empty α-cone) {
    receive message c from child
    if (c != NULL) {
      set color ← gray
    }
  }
  /* an empty α-cone was found */
  if (non-black neighbor, n, exists)
  {
    set parent ← n
    set color ← black
  } else {
    /* root */
    set parent ← self
    set color ← black
  }
}
```

Each node individually ensures that after the completion of this phase all non-tree edges connect to neighbors which are in the same tree. To ensure this using message passing, each root, r, multicasts a Usurp-Root$_r$ message down its tree. Every node, v, in the network waits until a Usurp-Root$_{root(v)}$ is received by it and by all its neighbors, u_i's, which are connected by non-tree edges. If Usurp-Root$_{root(v)}$ = Usurp-Root$_{root(u_i)}$ for all i, then v copies Usurp-Root$_{root(v)}$ to all its children. If on the other hand some neighbor, u_k, received a different message then u_k and v decide which root has a higher priority and 'temporarily connect' this non-tree edge. This temporary connection is converted into a tree edge when an acknowledgment traverses it. The message from the root with higher priority, let us say Usurp-Root$_v$, is forwarded to u_k and the parent (child) pointer in the node receiving the lower (higher) priority message is updated (added). The node u_k forwards the message to both its children as well as its parent. Thus, the Usurp-Root message from the root with highest priority makes a complete round trip in the network. Two cases, which can occur while a message is being forwarded in the tree, are:

(1) Two Usurp-Roots from the same root collide. The message with the larger hops traversed is allowed to proceed.
(2) Two Usurp-Roots from different roots collide. The message from the root with higher priority proceeds.

The nodes which temporarily connected their edges only acknowledge if they successfully receive acknowledgments from all their new children. In case they receive another Usurp-Root then the same procedure is repeated. An example run of the algorithm is illustrated in Fig. 2. Messages from the root r_1 (which has the highest priority) reach the nodes v_b and v_a which forward it the nodes v_c (in tree T_3) and r_2 (in tree T_2), respectively, and trees T_2 and T_3 get oriented accordingly.

4. A distributed algorithm for construction of a k-core

Our algorithm first finds a rooted core as an intermediate step. This rooted core is then used to construct a k-core. All steps of the algorithm are animated by message exchanges over the given network.

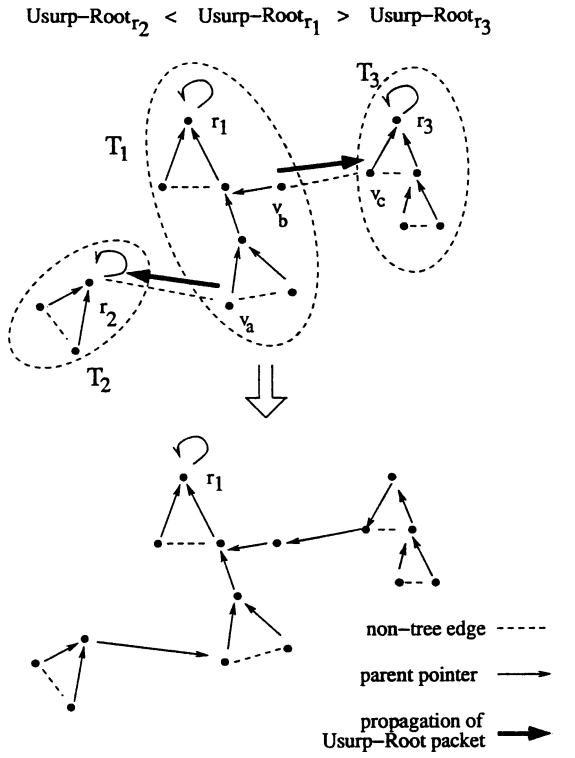

Fig. 2. Connecting the trees T_1, T_2 and T_3 of the forest.

4.1. A rooted core

We begin by defining a rooted core.

Definition. A rooted core of a rooted tree T_r is a path which minimizes the distance $d(P_{l,r})$ amongst all leaves l, where $P_{l,r}$ is a path from the root r to a leaf l.

A parallel algorithm for computing k-tree core has been presented in [3]. One can easily find the core of a tree by applying the following results from [2], which provide the basis for our distributed implementation of finding the k-tree core backbone in the network.

Lemma 1 [2]. *Let tree T be oriented into a tree T_r rooted at r. Let $P_{domleaf(r),r}$ be a rooted core of T_r, where domleaf(r) is the leaf of this rooted core, which we call the dominating leaf. Then there exists a core of T which starts from domleaf(r).*

Theorem 1. *The problem of finding a core of a tree can be reduced to that of finding a rooted core of a rooted tree.*

Proof. The problem of finding a core of a tree T can be solved as follows:

(1) Orient T into a tree T_r rooted at r.
(2) Find a rooted core $P_{domleaf(r),r}$ of T_r.
(3) Re-orient T into a tree $T_{r'}$ rooted at $r' =$ *domleaf*(r).
(4) Find and output a rooted core of $T_{r'}$.

By Lemma 1 and the definition of a rooted core, it is easy to see that a rooted core of $T_{r'}$, $P_{domleaf(r'),domleaf(r)}$, is a core of the tree T. □

4.2. The algorithm for rooted core

The algorithm uses the result of Theorem 1 and each step of the proof, except the first one, is animated through messages from the root to the leaves or in the opposite direction. The first step of the proof would have been accomplished when the spanning tree was constructed and the rest of the phases are described below.

4.2.1. Claim dominating leaf status

The tree is already rooted from its construction in Section 3. Therefore we only need to find the *dominating leaf*, which is the leaf of the rooted core in this rooted tree. To do this each leaf sends a *claim-dominating-leaf* message up its parent in the tree. The message consists of the identity of the leaf along with its *distance saved* value, and the size of the sub-tree, which get updated as the message moves up the tree. Each node v with children $\{a_1, a_2, \ldots, a_t\}$ will choose the leaf (in the sub-tree of its child a_j) with the largest *saved* value and as its *dominating child* and will calculate its the new *saved* value as, $saved(v) = saved(a_j) + \sum_i size(a_i)$. This iterative computation of the savings is due to the following result.

Lemma 2. *Let $P_{l,u}$ be a path in subtree T_u of tree T_r and v be the vertex in $P_{l,u}$ adjacent to u. Then $saved(P_{l,u}) = saved(P_{l,v}) + |T_v|$, where $|T_v|$ is the number of nodes in the subtree T_v. (See Fig. 3.)*

Proof.

$$\begin{aligned} &saved(P_{l,u}) - saved(P_{l,v}) \\ &\quad = \big(d(u) - d(v)\big) + \big(d(P_{l,v}) - d(P_{l,u})\big) \\ &\quad = \big(|T_v| - \big(|T_r| - |T_v|\big)\big) + \big(|T_r| - |T_v|\big) \\ &\quad = |T_v|. \end{aligned}$$

□

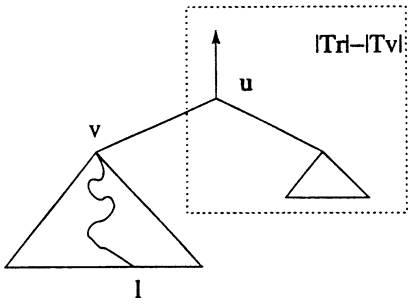

Fig. 3. Bottom up computation of *saved* value.

Ties in the *saved* value of two children is resolved based on the unique id of the leaf. When the root chooses its dominating child it sends an acknowledgment down to its children which is propagated.

(1) Re-orient with *dominating leaf* as root.
The dominating leaf, *domleaf*, initiates a Usurp-Root$_{domleaf}$ and forwards the message to all tree neighbors. This message traverses the whole tree and every tree edge is oriented in the direction opposite to the movement of the message.
(2) New leaves again contend for *dominating leaf* status. Each leaf in this new tree again sends a *Claim-Dominating-Leaf* message up the tree and an acknowledgment is sent back to a leaf by the root if its savings are the largest in this new tree. The path which the acknowledgment traverses defines the core of the tree and all node on the core will be able to construct a distributed core.

4.3. Extending the algorithm to k-core

The algorithm for finding the *k-core* is a direct derivation from the algorithm presented above. The only change is in phase 3 (see Theorem 1 of Section 4.1) of the core finding algorithm. Instead of sending the dominating leaf to their parent, each intermediate node in the tree sends $k-1$ messages from its children to the parent, i.e., the top $k-1$ messages when ordered by their savings value. Moreover, the savings value in the message with the largest savings (the dominating leaf in that sub-tree) is incremented by the size of that sub-tree. The root then chooses the first $k-1$ of the leaves as the end points of the k-core and multicasts this down the tree. The root itself forms one leaf of the k-core.

For proving the correctness of this method of finding the k-core we first make the observation that our algorithm is essentially a greedy path adding algorithm. Starting from a core of the tree, at the $(k-2)$th step of we add a path to some leaf not already in the $(k-1)$-core to get the k-core. It was shown in [1] that this simple greedy strategy works.

The core discovery process consists of a set of phases as described in Section 4.2. We can enumerate the type of messages required for core discovery as follows:

(1) Create spanning tree
  (a) *Create Forest* (CHOOSE_PARENT)
  (b) *Join all trees in forest* (USURP_ROOT)
  (c) *Acknowledge finish of join* (ACK_JOIN)
(2) Find rooted core
  (a) *Claim Dominating Leaf Status* (CLAIM_DOM)
  (b) *Acknowledge Dominating Leaf* (ACK_DOM)
(3) Re-orient tree with root at leaf of rooted core found earlier
  (a) *Re-orient with root at dominating leaf* (USURP_ROOT)
  (b) *Acknowledge finish of Re-orientation* (ACK_REORIENT)
(4) Find k leaves with highest savings (Find k-core)
  (a) *Claim k Dominating Leaf Status* (CLAIM_DOM)
  (b) *Acknowledge the k leaves with highest savings* (ACK_DOM)

5. Incremental update of k-core

Since we are considering networks with nodes moving arbitrarily, the core computation done at some time cannot be expected to be a valid measure of the tree that minimizes the sum of distances at some arbitrary time in the future. Hence it is required that there be some mechanism to *refresh* k-core at appropriate instances in time. If the application requires a correct core then it may lead to flooding the network with messages from the above phases very frequently. On the other hand, if we can tolerate with an approximate k-core, then a mechanism that will incrementally correct the core can be developed. In fact, there are certain

applications, such as routing data packets over a network from a source to destination, that can do with approximate core maintenance. Besides that we note that a spanning tree of the network after all is not unique. So maintaining an approximate core can be useful in a number of circumstance with the flip side of the approach being a saving in associated overheads.

The core needs to be maintained in the event of changes in the network topology. The following cases are handled as follows:

(1) *Addition of a node*: The node contacts its nearest neighbor and asks to be adopted, giving its savings. The parent compares this with the savings of its dominating child. If they are more than the savings of the dominating child then the dominating child is updated and the message passed to its parent. This goes on till this message terminates at some node. If it becomes a core leaf then the core is recomputed.
(2) *Deletion of an existing edge*: If the deleted edge was a non-tree edge then no changes are required while on the other hand if it was a tree edge then the following procedure is executed by the two nodes, p and c, which were the parent and child, respectively, on the deleted edge.
The node p updates its savings. The orphaned child, c, searches for a new parent and initiates addition as described earlier. If there is no parent to choose then it sends a Usurp_Root down its sub-tree to discover any possible connections to a larger tree in the network. If a descendant exists with a connection (that is the graph has not become partitioned) then it initiates Connect Trees phase as described in Section 3.
(3) *Deletion of a node*: Equivalent to deletion of an edge and addition of the orphaned children.
(4) *Addition of an edge*: This edge is treated as a non-tree edge and no changes in the core are required. But this has an effect on the cluster connectivities which influence shortest path computations.

6. Results of experiments

In this section we present the results of our experiments with the proposed algorithms for finding spanning tree and k-core and maintaining it over time.

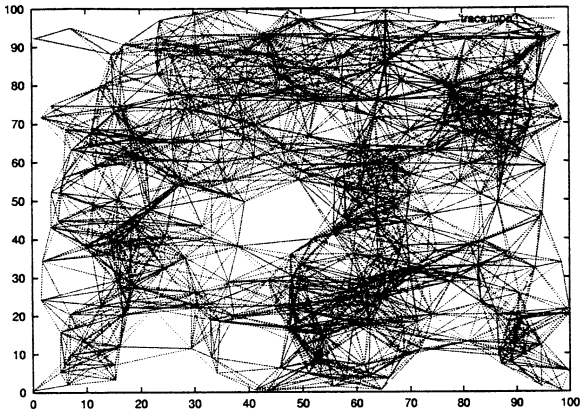

Fig. 4. The topology being considered for observing the effect of α.

6.1. Effect of the parameter α

Each edge in a spanning tree can be viewed as partitioning the set of vertices into two parts. We refer to an important edge in the spanning tree as the one which has a large number of vertices on both sides. We simulated the effect of varying the value of α and saw that the results were consistent with the fact that a low value of α gives a spanning tree which has important edges towards the periphery. This is not desirable. The topology being considered is shown in Fig. 4 which shows a network of 350 nodes, and the cores obtained for values of $\alpha = 1.1$ and $\alpha = 3.1$ are depicted in Figs. 5 and 6, respectively.

If one carefully analyzes the algorithm for construction of spanning tree, the effect of α is immediately apparent.When the value of α is around π, a node which has no other node to present in one half plane then that node is essentially a vertex on the convex hull of the set of nodes in the network. Therefore, at each iteration the algorithm actually finds a convex hull of the remaining set nodes which have not yet chosen their respective parents. When we have a convex hull with no internal nodes then the algorithm terminates.

6.2. Effect of k on structure of core

It has been shown that each $(k-1)$-tree core is a subset of a k-tree core. Hence, increasing the value of k beyond a point only adds redundant edges which are not really required as can be seen in Fig. 7, while Fig. 6 shows a core with a k value of 5 which seems

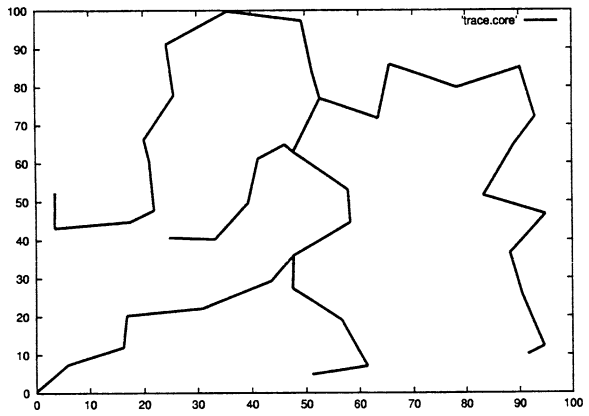

Fig. 5. K-core for $\alpha = 1.1$, $k = 5$.

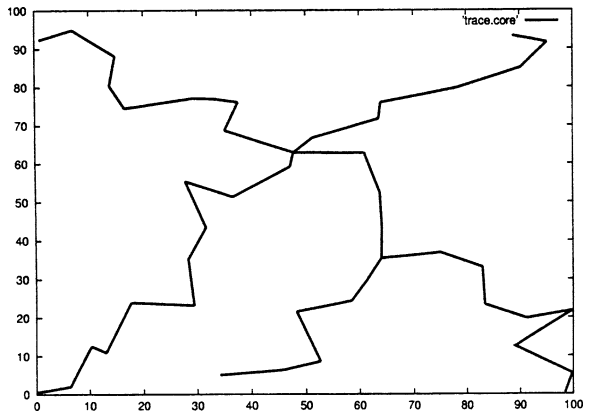

Fig. 6. K-core for $\alpha = 3.1$, $k = 5$.

optimal. The topology being considered is the same, as in Fig. 4.

6.3. Effect of mobility on maintenance of core

Assuming we can do with some imperfections in the core for some period of time until the changes in the network become very drastic, it is possible to control overhead for maintenance of core. Notice that imperfection in core can mean a rise in the cost for using that core for transmission or traversal of message from one point of the graph to the other. So, it may be good idea to balance that cost with the cost of maintaining core instantaneously as the graph witnesses a change. Fig. 8 illustrates the relative amount of message exchanges when we experimented with different the threshold

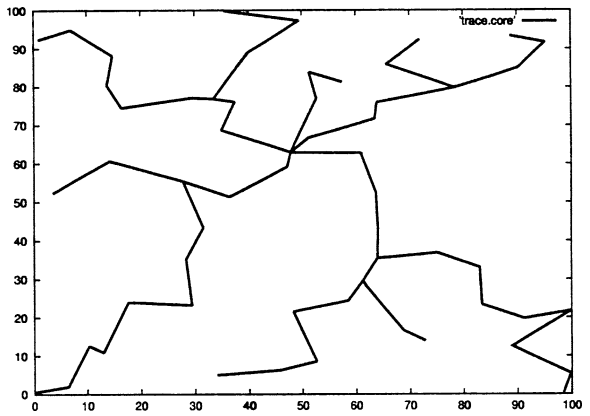

Fig. 7. K-core for $k = 10$, $\alpha = 3.1$.

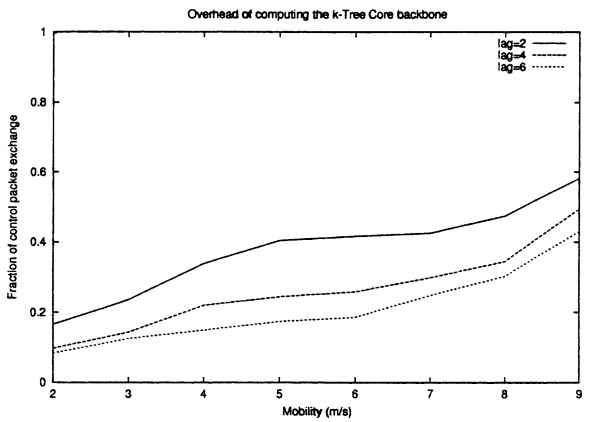

Fig. 8. Overhead of core computation for different values of lag (i.e., the number of changes in the topology required for recomputation of the core).

values that must be reached before the core is recomputed.

7. Conclusions

In this paper, we have proposed a distributed algorithm for constructing a rooted spanning tree of a dynamic graph that has root toward the center of the graph. The construction of k-core from the spanning tree involves first finding a rooted 2-core then reorienting the root of the core at dominating leaf, and finally converting it into a k-core by sending out claim k dominating leaf status messages. All stages of the algorithm are animated through messages. We also

provide specific steps to maintain k-core as the nodes move about, come alive or go dead. We notice that the process of finding spanning tree implicitly involves construction of (convex) hull of the set of nodes of the dynamic graph. Initially the set consists of all nodes. At each stage of the algorithm a subset of nodes choose their parents, and these nodes are precisely those lying on the hull. After the nodes have chosen their respective parent they are eliminated from further considerations and the process is repeated. When the value of α is 180 degrees, the technique of identifying peripheral nodes of the graph can be seen to constitute finding of a convex hull. So, we believe if a less costly method of identifying convex hull can be found then our algorithm can speed up. As we mentioned initially, finding and maintaining a k-core in general can be quite useful for routing and regulating flow of traffic in a dynamic network.

References

[1] S. Peng, A.B. Stephens, Y. Yesha, Algorithms for core and k-core of a tree, J. Algorithms 15 (1993) 143–159.
[2] S. Peng, W.-T. Lo, A simple optimal parallel algorithm for a core of a tree, J. Parallel Distributed Comput. 20 (1994) 388–392.
[3] S.-C. Ku, W.-K. Shih, B.-F. Wang, Efficient parallel algorithms for optimally locating a k-leaf tree in a tree network, in: Proc. 1997 Internat. Conf. on Parallel Processing, ICPP '97.
[4] E.M. Royer, C.-K. Toh, A review of current routing protocols for ad-hoc mobile wireless networks, IEEE Magazine Personal Commun. 17 (8) (1999) 46–55.