Algorithms for a k-tree core of a tree

Shietung Peng, A.B. Stephens, Yelena Yesha

Computer Science Department, University of Maryland

abstract

In this paper, we define a generalization of a core which we call a k-tree core. Given a tree T and parameter k, a k-tree core is a subtree T' of T containing exactly k leaves that minimizes \( d(T') = \sum_{v \in V(T')} d(v, T') \), where \( d(v, T') \) is the distance from vertex v to subtree T'. We then give two algorithms to find a k-tree core of a tree with n vertices. The complexities of these algorithms are \( O(kn) \) and \( O(n \log n) \) respectively. This work is motivated by a resource allocation problem dealing with a partially replicated distributed database defined on a tree network. This problem is briefly described in the last section of the paper.

1 Introduction

A core of a tree is a path which solves a certain minimization problem. Specifically, for each path in a tree we measure the sums of distances from all other vertices to the path. A core is a path for which this sum of distances is minimized. For properties of cores the interested reader should see [1,3]. In 1980 Morgan and Slater [2] published a linear algorithm for finding a core of a tree. We define a generalization of a core which we call a k-tree core. A k-tree core is a subtree with exactly k leaves for which the sum of distances from all other vertices to the subtree is minimized. In our terminology the core of a tree is a 2-tree core. We give two different algorithms for constructing k-tree cores. The first algorithm, which has a complexity of \( O(kn) \), works by pruning edges from the original tree. In the course of verifying these algorithms it is shown that every core is contained in some k-tree core and every k-tree core contains a core. It is also shown that k-tree cores have a nesting property. That is, every (k-1)-tree core is contained in a k-tree core.

The notion of a k-tree core arises from the study of the placement of replicated data objects on a distributed database. The underlying problem is the placement of replicated data objects on a tree network whose edges may fail with a certain probability in such a way as to maximize the probability of a successful read operation from a randomly chosen log-in site. We will elaborate a little on this topic in the last section of the paper.

The organization of this paper is as follows. In section 2 we first give the necessary notation and definitions. In section 3 we give an algorithm for the k-tree core of a tree with n vertices which has complexity \( O(kn) \). In section 4 we give an alternative algorithm for constructing a k-tree core of a tree which is asymptotically more efficient for larger values of k. In the final section we discuss an application of our algorithms to an allocation problem in a distributed database.

2 Terminology and Definitions

In this section, we introduce some notation and definitions used throughout this paper. We let T denote an unrooted tree with vertex set \( V(T) \). Let \( |T| \) represent the number of vertices of T. We define a leaf to be a node with degree one. A parent of a leaf node is defined to be a node adjacent to that leaf. For a vertex \( v \in V(T) \), we define the distance of v, \( d(v) = \sum_{u \in V(T)} d(u, v) \). 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 as small as possible. 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 \( d(v, T') = \min_{u \in V(T')} d(u, v) \). A subtree containing exactly k leaves is called a k-tree core if its distance is as small as possible. We note that, as a consequence of the optimization criterion, the leaves of a k-tree core will be leaves of the tree T. We also note that a k-tree core of a tree need not be unique.
We now introduce some definitions relevant to a k-tree core which will be needed in the formulation of our algorithms. Let \( P_{u,l} \) be a path from a vertex \( u \) to a leaf \( l \). Then the "distance saved" by this path, \( DSAV(P_{u,l}) \), is defined as \( d(u) - d(P_{u,l}) \). The value \( DSAV(P_{u,l}) \) is the most important measure for guiding the optimal selection of path extensions from the vertex \( u \). Let the path \( P_{u,l} \) be given as \( < v_0, v_1, ..., v_r > \), where \( v = v_0 \) and \( l = v_r \). Let \( Tr(u) \) be the subtree hanging on \( u \). That is, \( Tr(u) \) is the subtree induced by the set of vertices connected to \( u \) by paths containing only \( u \) and those vertices not in path \( P_{u,l} \). Then it is easy to verify that the value of \( DSAV(P_{u,l}) \) can be computed by the following formula:

\[
DSAV(P_{u,l}) = \sum_{i=1}^{r} |Tr(u_i)|.
\]

A procedure, CORE-BRANCH(\( T_r \)), which, given a rooted tree \( T_r \), generates a path \( P_{u,B(r)} \) that minimizes \( d(P_{u,B}) \) over all \( u \in T_r \), is given below. This procedure will be used in the algorithm for finding a k-tree core of a tree presented in the next section.

**Procedure UPDATE(\( u \))**

begin
FOR every \( u \in S \) DO
Let \( v_1, ..., v_r \) be the children of \( u \).
\[ |T(u)| = \sum_{i=1}^{r} |T(u_i)| + 1; \]
\[ DSAV(u) = \max_{1 \leq i \leq r} \{ |T(u_i)| + DSAV(v_i) \}; \]
Let \( v_k \) be the vertex such that \( |T(u_k)| + DSAV(v_k) \) is the maximum.
\[ B(u) := B(v_k); \]
Mark \( u \)
end.
**Procedure CORE-BRANCH(\( T_r \))**

begin
For every leaf \( l \in T_r \) do
Mark \( l \);
\[ |T(l)| := 1, \]
\[ DSAV(l) := 0; \]
\[ B(l) := l; \]
REPEAT
\( S := \{ u \mid u \) is unmarked and all its children are marked\};
UPDATE(\( S \))
UNTIL \( S = \emptyset \);
Return(\( P_{r,B(r)} \))
end.

3 An algorithm for finding a k-tree core of a tree

The first algorithm for the k-tree core problem is based on the algorithm for finding a core and a greedy strategy for extending a core to a k-tree core. Let \( < v_0, v_1, ..., v_r > \) be a core of tree \( T \). Let \( Tr(v_i) \), \( i = 1, 2, ..., r \), be the subtrees hanging on the core via \( v_i \). The \( Tr(v_i) \) are pairwise disjoint and \( \bigcup_{i=1}^{r} V(Tr(v_i)) = V(T) \). The greedy strategy is to select a maximum among all \( DSAV(P_{u,l}) \), where \( P_{u,l} \) is a path from \( u \) to a leaf \( l \in Tr(v_i) \). The algorithm is presented as follows.

**Algorithm k-TREE-CORE-I**

**INPUT:** Tree \( T \) and \( k < \epsilon \)
**OUTPUT:** A k-tree core of tree \( T \)

begin
Find a core \( P = < v_0, v_1, ..., v_r > \) from the core algorithm presented in section 3;
Let \( Tr(v_i), i = 1, 2, ..., r \), be the subtrees hanging on the core via \( v_i \);
\[ ktree core := P; \]
\[ CANSET := \{ v_0, v_1, ..., v_r \}; \]
\[ j := k - 2; \]
FOR every vertex \( v \in CANSET \) DO

\[ DSAV(v) := \max_{u \in Tr(v)} DSAV(P_{u,l}); \]

WHILE \( j > 0 \) DO
Let \( v_q \) be the vertex such that
\[ DSAV(v_q) = \max_{v \in CANSET} DSAV(v); \]
Let \( P_q = < v_q, v_{q_1}, ..., v_{q_s} > \) be the path in \( Tr(v_q) \) that achieves the value \( DSAV(v_q) \), where \( v_{q_0} = v_q \);
Let \( Tr(v_{q_i}), i = 1, 2, ..., s \), be the subtrees hanging on the path \( P_q \);
Remove edge \( < v_{q_0}, q_1 > \) from \( Tr(v_q) \);
\[ ktree core := ktree core \cup P_q; \]
\[ CANSET := CANSET \cup \{ v_{q_1}, ..., v_{q_s} \}; \]
FOR \( i := 1 \) to \( s \) DO
\[ DSAV(v_i) := \max_{u \in Tr(v_{q_i})} DSAV(P_{u,l}); \]
\[ j := j - 1; \]
output \( ktree core \)
end.

Figure 1 illustrates how the algorithm works. The same example will be used through the remaining sections. First, we assume that a core \( C \) given by \( < 1, 11, 15, 16, 20, 17, 18, 8 > \) is found. Then, the maximum distance savings for the subtrees hanging on the vertices on the path \( C \) are listed in the following table.

<table>
<thead>
<tr>
<th>( u_i )</th>
<th>1</th>
<th>11</th>
<th>15</th>
<th>16</th>
<th>20</th>
<th>17</th>
<th>18</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSAV</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>7</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

From the table, it is easy to see that the path \( < 3, 13, 14, 16 > \) should be added to the output and the vertices 3,13,14 should be put into the candidate set. Then, from the following new table, the path \( < 6, 19, 17 > \) is added at the next iteration and the algorithm terminates since \( k = 4 \).
Figure 1: An example for Ctree core

<table>
<thead>
<tr>
<th>v_i</th>
<th>1</th>
<th>11</th>
<th>15</th>
<th>20</th>
<th>17</th>
<th>18</th>
<th>8</th>
<th>3</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSAV</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Lemma 1
Every k-tree core intersects every core.

Lemma 2
Every k-tree core contains a core.

Theorem 1
Algorithm k-TREE-CORE.I correctly generates a k-tree core of the tree T.

Theorem 2
The time complexity of the k-tree core algorithm is \(O(kn)\).

4 An alternative method of finding a k-tree core of a tree

In the previous section, we developed an efficient algorithm for finding a k-tree core of a tree. If \(k\) is a constant then the time complexity of the algorithm is \(O(n)\). However, when \(k\) is not a constant, say \(k = n^\alpha\), \(0 < \alpha < 1\), the time complexity of the algorithm will be \(O(n^{1+\alpha})\). Does there exist a more efficient algorithm when \(k\) is large? In this section we will answer this question by providing a new algorithm which takes only \(O(n \lg n)\) time. The algorithm works from outside inward as follows. For each leaf \(l\), we travel inward, starting from \(l\), until a vertex with degree greater than two is found. This node can be viewed as the head of the branch ending at leaf \(l\). Then, we compute the distance saved by adding the branch, say \(ds(l)\), in the sense described in the previous section. Second, we construct a min-heap of all leaves according to their \(ds\) values. Third, we remove the branch with minimum \(ds\) value, one a time. If necessary, after the removal, we extend some branch and update its associated \(ds\) value. For any removal or extension, we also need to update the heap. This process is continued until the number of leaves left equals to \(k\). Notice that, following the way that all leaves travel, we can orient all subtrees which have been visited as rooted subtrees. Therefore, it makes sense to say one vertex is an ancestor or a descendant of another vertex in these subtrees.

For each leaf \(l\) let \(head(l)\) be the head of the branch ending at \(l\). That is, \(head(l)\) is an ancestor of \(l\) and the degree of its parent, \(p(head(l))\), is greater than two. Let \(|T(head(l))|\) be the size of the subtree rooted at \(head(l)\). Let \(ds(l)\) be the distance saved by adding the branch ending at \(l\). That is, \(ds(l) = DSAV(P_{p(head(l))},l)\) using the notation introduced at section 2. For each node \(v\) in a branch, let \(leaf(v)\) represent the leaf in the branch containing \(v\). The algorithm can now be stated formally as follows. Again, we consider only the case \(k < e\) as in algorithm k-tree-core.I.

algorithm k-tree-core.II

begin
REPEAT
\[ T(p(v)) := |T(v)| + |T(p(v))|; \]
leaf(p(v)) := leaf(v);
\[ v := p(v); \]
p(v) := p(p(v));
\[ ds(leaf(v)) := ds(leaf(v)) + |T(v)|; \]
UNTIL \(\deg(p(v)) > 2\)
head(leaf(v)) := v;
end.

Step 1: { Initialization }
FOR each \(v \in T\) DO
IF \(v\) is a leaf THEN \(leaf(v) := v;\)
\[ ds(v) = |T(v)| = 1; \]
\[ a(v) = v; \]
ELSE \(leaf(v) := nil;\)
\[ s(v) = 1; \]
FOR each leaf \(l \in T\) DO
IF \( \deg(p(l)) = 2 \) THEN \( \text{COMPUTE}(l) \);

Construct a priority queue \( Q \) (e.g., a heap) of all leaves based on their \( ds \) values;

* The procedures \( \text{DEL}(Q) \) and \( \text{UPDATE}(Q, \text{leaf}(u)) \) are standard subroutines used to delete an element in \( Q \) and to update \( Q \) while one of it elements changing its weight, i.e., its \( ds \) value.*

\( r := m - k \); where \( m \) is the number of leaves in the tree.

Step 2: \{ Remove minimum branches \}

REPEAT
\( u := \text{DEL}(Q) \);
\( |T(p(\text{head}(u)))| := |T(p(\text{head}(v)))| + |T(\text{head}(v))| \);
\( \deg(p(\text{head}(v))) := \deg(p(\text{head}(v))) - 1 \);
Remove edge \( < \text{head}(v), p(\text{head}(v)) > \);
\( r := r - 1 \);
IF \( \deg(p(\text{head}(v))) = 2 \) and only one of its two neighbors, say \( u \), satisfies \( \text{leaf}(u) \neq \text{nil} \)
THEN \( \text{COMPUTE}(u) \) & \( \text{UPDATE}(Q, \text{leaf}(u)) \);
UNTIL \( r = 0 \)
end.

We illustrate how the algorithm works using the example in Figure 1. First, we establish a priority queue based on the \( ds \) values for all leaves of the tree as shown in the following table.

<table>
<thead>
<tr>
<th>vertex</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ds ) value</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Then vertices 4, 5, 10, 7, 9 are deleted from the queue and the \( d \) values of vertices 3, 6, 8 are updated. The new table is constructed as follows.

<table>
<thead>
<tr>
<th>vertex</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ds ) value</td>
<td>3</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Now the vertex 2 (or vertex 1) will be deleted and the algorithm terminates.

**Theorem 3**
For a tree with \( n \) vertices, algorithm \( k\text{-tree.core.JI} \) finds a \( k \)-tree core in time \( O(n \log n) \).

**5 Application**

In this section we discuss an application of our algorithms. We consider a distributed database sitting on a network with a tree topology with \( n \) nodes. Each of the edges in the tree may fail with probability \( p \). We assume that users login at random vertices of the network. In order to increase the availability of a particular data object \( O \) we assume that it has been replicated and that \( k \) copies of it, \( C_1, \ldots, C_k \), exist on the network, where \( k < n \). In order to successfully read the object \( O \) we must be able to read at least one copy of the object. Then the probability of successfully reading the data object is the probability of there being an unbroken path from the random login site to at least one copy of the data. We wish to place the copies on the tree in such a way as to maximize the probability of a successful read operation.

A placement of the copies \( C_i \) at nodes \( c_i \) induces a copy subtree \( T_c \). This subtree is the union of paths \( P_{ij} \) where \( P_{ij} \) is the unique path joining \( c_i \) and \( c_j \). Each \( x_j \) in \( T_c \) is the root an attached subtree \( T(x_j) \) whose vertices (except \( x_j \)) do not belong to \( T_c \).

Let \( q = 1 - p \). For a given \( T_c \), let \( P_k(q, T_c) \) represent the probability of a successful read operation from a random login site. Then for \( q \) sufficiently near 1, minimizing \( P_k(q, T_c) \) for all possible subtrees \( T_c \) is a necessary condition for maximizing \( P_k(q, T_c) \). It can be shown that for \( p \) sufficiently close to 0, \( P_k(q, T_c) \) is minimized by those subtrees \( T_c \) which minimize

\[
\sum_{x_j \in V(T_c)} h(x_j, T(x_j))
\]

where \( h(x_j, T(x_j)) \) is the sum of all distances in \( T(x_j) \) to \( x_j \). It can also be shown that this minimum must occur when all copies of \( O \) are located at leaves of the tree network. Consequently, a necessary condition for optimizing the probability of a successful read, is that \( k \) copies be placed on certain leaves of the tree in order to minimize the sum of distances of nodes to the subtree induced by the placement of the \( k \) copies. In other words, the copies must be placed so as to form a \( k \)-tree core. In the example tree in figure 1, Four copies of the data objects should be put in nodes 1, 3, 6, and 8 (the leaves of a 4-tree core).

**References**

