Finding Cores of Limited Length

Stephen Alstrup* Peter W. Lauridsen*
Peer Sommerlund* Mikkel Thorup*

Abstract

In this paper we consider the problem of finding a core of limited length in a tree. A core is a path, which minimizes the sum of the distances to all nodes in the tree. This problem has been examined under different constraints on the tree and on the set of paths, from which the core can be chosen. For all cases, we present linear or almost linear time algorithms, which improves the previous results. As Minieka and Patel observes (J. Algorithms, Vol. 4, 1983), the problem of finding a core of limited length would be simplified, if the core always contained the median, m. They conclude their paper by writing "we do not know if a core of length l will contain m. Unfortunately, this situation remains unexplored and as this question remains open, the development of an efficient algorithm for locating a core of a specified length remains a difficult problem." We show that the median is not necessarily included in the core and give an $O(n \min\{\log n\alpha(n,n), l\})$ algorithm for the problem, which improves the former best result $O(n \min\{n^2, l \log n\})$ (Lo and Peng, J. Algorithms Vol. 20, 1996 and Minieka, Networks Vol. 15, 1985).

1 Introduction

In 1971 Goldman [2] gave a linear time algorithm for determining a node in a tree, called a *median*, minimizing the sum of the distances to all other nodes. Later, in 1982, Slater [13] proposed to determine the path which minimizes this sum, called a *core*. More specificly, a core is path for which the total sum of distances from all nodes to the path is minimum among all paths. In 1980 Morgan and Slater [11] gave a linear time algorithm for determining a core in a tree in which all edges have length one. This algorithm is easily extended to a tree in which edge lengths are arbitrary nonnegative. A simpler linear time algorithm for finding a core in a tree with arbitrary nonnegative edge lengths was given in 1993 by Peng, Stephens and Yesha [12].

In 1983 Minieka and Patel [10] proposed the problem of finding a path of length l that minimizes the distance sum over all paths of length l. Such a path is called a core of length l. Since we cannot be certain that a path of length l exist in a tree, partial edges are usually allowed in cores of length l. Minieka and Patel do not give an algorithm for determining a core of specified length, but list a number of problems in giving such an algorithm. For instance the core of length l can have distance sum larger than a core of length $< l$. In 1985, Minieka showed that a core of a specified length can be found in $O(n^3)$ time [9]. In 1996, Lo and Peng [7] gave an $O(n \log n)$ algorithm for finding a core of a specified length in the case where all edges have length one. Furthermore they claim that their algorithm is easily extended to cases where partial edges are allowed and the edges have arbitrary nonnegative length. This is true, but if the edge weights are arbitrary nonnegative integers the complexity increases to $O(nl \log n)$, which for constant l is still $O(n \log n)$.

In 1993 Hakimi, Labbé and Schmeichel [3] examined the problem of finding a core with length $\leq l$ using either full or partial edges. This is a natural extension of the core problem since in both cases paths with length $< l$ can exist which have cost less than any path of length $= l$ [10]. For the case allowing partial edges they show that the $O(n^3)$ algorithm [9] can be used. For full edges they show the existence of a polynomial time algorithm for the problem. However it is also shown that for locating the core in an arbitrary network the problem becomes NP-hard. Finding cores in trees using parallel algorithms have also been examined, see e.g. [6, 7]. Finally we note that Minieka's $O(n^3)$ algorithm is easily modified to all cases mentioned above.

To summarize : Placing a core in a tree has been investigated for partial/full edges, core length $= l/\leq l$ and uniform/arbitrary edge weights. For the cases of uniform edge weights an $O(n \log n)$ algorithm has been

*E-mail : (stephen,waern,peso,mthorup)@diku.dk. Department of Computer Science, University of Copenhagen.

given [7]. For this case we present an $O(n)$ algorithm. For arbitrary edge weights the algorithms given so far have complexities $O(nl \log n)$ and $O(n^3)$ [7, 9]. In this paper we give two algorithms for all cases. The first determines the core in $O(nl)$ time and the second uses $O(n \log n \alpha(n,n))$ time. If only full edges are allowed the complexity of the second algorithm is $O(n \log n)$. The factor $\alpha(n,n)$ comes in because of a strong relation between the core problem and Davenport-Schinzel sequences [1]. A strong relation between algorithmic geometry and Davenport-Schinzel sequences has previously been established.

In [10], Minieka and Patel studied conditions under which there is a core containing a median. If their conditions are not satisfied, they write that "we do not know if a core of length l will contain [the median] m. Unfortunately, this situation remains unexplored and as this question remains open, the development of an efficient algorithm for locating a core of a specified length remains a difficult problem." We solve Minieka and Patel's question in Figure 1, presenting a tree with 35 nodes in which the core of length 10 does not contain the median. Examples can also be given for ternery trees, essentially replacing the high degree node with a balanced binary tree, but such ternery constructions needs more than 100 nodes.

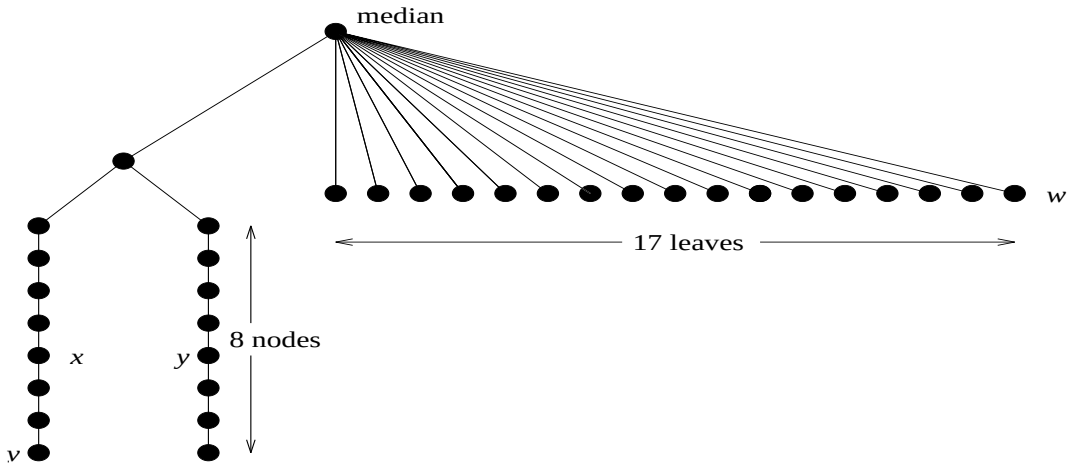

Figure 1: A sample tree, in which all edges have length one. An optimal core of length 10 through the median goes from v to w, and has cost $(8 \cdot 9)/2 + 16 = 52$. An optimal core of length 10 that is not required to go through the median goes from x to y, and has cost $2 \cdot 6 + 17 \cdot 2 + 1 = 47$.

2 Preliminaries

Let T be a tree with node set $V(T)$ and edge set $E(T)$ and let T be rooted at an arbitrary node. For a node $v \in V(T)$, T_v is the subtree in T rooted at v, hence T_v is the tree induced by v and descendants of v. With each edge $(v,w) \in E(T)$ a nonnegative integer length $length(v,w)$ is associated. The path from a node u to a node v in T is denoted $P_{u,v}$ and the length of a path P is denoted $|P|$.

For a path Q in T we use $dist(v,Q)$ to denote the distance from a node $v \in V(T)$ to Q, thus $dist(v,Q) = min_{u \in Q}|P_{v,u}|$. In the following we will allow endpoints of paths to be points on edges, thus the "u" in this definition should denote a point on Q. More precisely if $(v,w) \in E(T)$, $length(v,w) = x$ and P is a path of length $y < x$ starting at v towards w, then $dist(w,P) = x - y$. The case where points on edges are allowed is called *partial* and the case where only full edges are allowed is called *discrete*. Since we are considering distances between points we will also use the notation $dist(x,y)$ to denote the distance between the points x and y in T, hence $dist(x,y) = |P_{x,y}|$.

For a path P we define $cost_T(P) = \Sigma_{v \in V(T)} dist(v,P)$. Let $\mathcal{P}$ be a set of paths. We say $Q \in \mathcal{P}$ is a core with respect to $\mathcal{P}$, if $cost_T(Q) = min\{cost_T(P) | P \in \mathcal{P}\}$. We will use the name $Core(\mathcal{P})$ to denote a core with respect to $\mathcal{P}$. The sets $\mathcal{P}$ considered as possible core candidates in this paper satisfies the following conditions:

(a) The paths have length $= l$ or length $\leq l$, where l is a nonnegative integer.

(b) The paths are partial or discrete.

The problems can furthermore be divided into whether edge lengths are uniform (i.e. have length one) or arbitrary nonnegative integers. We thus consider eight problem instances, however in the case of uniform edge lengths, the partial and discrete problems are the same. Analogous to other path problems (e.g. shortest path) the actual problem in finding a core is not so much finding a path, but finding the cost. We will thus concentrate on finding $cost_T(Core(\mathcal{P}))$ in this paper. The algorithms described are easily extended to finding a path attaining the cost of the core.

As mentioned in the introduction the core problem is a generalization of the problem of finding a median. In the median problem it is common that nodes have costs associated. Thus, in order to get the full generalization, this should also be the case for the core problem. More specifically this would mean that for a path P, $cost_T(P) = \Sigma_{v \in V(T)} dist(v, P) * cost(v)$. For reasons of clarity we will assume that $cost(v) = 1$ for all $v \in V$ unless anything else is stated. The formulas and lemmas deriven are easily extended to the case where costs of nodes are arbitrarily nonnegative.

3 An $O(nl)$ algorithm for all cases

In this section we give an $O(nl)$ algorithm for finding the core of a tree in all cases. To simplify the description of the algorithm we assume that partial edges are allowed and the length of the core should be $= l$. With minor changes the algorithm will, within the same complexity, also solve the problem in the other cases. These changes are briefly discussed at the end of the section.

For each subtree we will compute a core containing the root of the subtree. We define $MinCost(v)$ as the cost of a core containing v in T_v. By considering $MinCost(v)$ for all v we can compute the cost of the core of T. In order to compute $MinCost(v)$ we will need the values defined as follows.

Definition 1 *We define $Size_{down}(v)$ to be the number of nodes in T_v. Furthermore $Sum_{down}(v)$ is defined as $\Sigma_{w \in V(T_v)} |P_{w,v}|$. Analogously $Size_{up}(v)$ is the number of nodes in $V(T) \backslash V(T_v)$ and $Sum_{up}(v) = \Sigma_{w \in V(T) \backslash V(T_v)} |P_{w,v}|$. Finally we define the extended Sum, $Sum^*_{down}(v)$ as $\Sigma_{w \in V(T_v)} |P_{w,parent(v)}|$, hence $Sum^*_{down}(v) = Sum_{down}(v) + length(parent(v), v) * Size_{down}(v)$.*

Using recursion formulas the $Size$ and Sum values can be found in linear time by traversing T in bottom-up and top-down fashions. Below we give the recursion formula for Sum.

Let v be a node in T and let $w_1, \ldots, w_j$ be the children of v.
$Sum_{down}(v) = 0$, if v is a leaf.
$Sum_{down}(v) = \Sigma_{1 \leq i \leq j}(length(v, w_i) * Size_{down}(w_i) + Sum_{down}(w_i))$, otherwise.
$Sum_{up}(v) = 0$, if v is the root.
$Sum_{up}(v) = length(v, parent(v)) * Size_{up}(v) + Sum_{up}(parent(v))$
$\quad + (Sum_{down}(parent(v)) - Sum_{down}(v))$, otherwise.

$MinCost(v)$ can be found by combining the cost of two paths that start at v and propagates towards two different children of v. The cost of such paths will be denoted as $DownCost(v, k)$. More precisely we have

Definition 2 *$DownCost(v, k)$ is the cost of the minimum cost path in T_v of length k, which starts at the root of T_v, hence $DownCost(v, k) = min\{cost_{T_v}(P_{v,x}) \mid x$ is a point in $T_v, |P_{v,x}| = k\}$. The extended Downcost, $DownCost^*(v, k)$, is the cost of the minimum cost path of length k, which starts at the root of $T_v \cup \{(parent(v), v)\}$.*

In order to compute $MinCost(v)$ we will compute $DownCost^*(w, k)$, for $k = 0..l$ for all children w of v. Note that since the largest distance from a leaf to v could be $< l$, we can only compute $DownCost$ values for $k = 0..min\{l, max\{|P_{v,x}| \mid x \in T_v\}\}$. In order to make formulas more simple we will w.l.o.g. ignore this in the following. The lemma below shows how the $DownCost$ values can be computed bottom-up.

Lemma 3 *Let $v \in V(T)$ and let c denote the length of the edge from v to its parent in T. We have the following:*

$DownCost^*(v,k) = Sum_{down}(v) + (c-k) * Size_{down}(v)$, *if* $c \geq k$
$DownCost^*(v,k) = DownCost(v,k-c)$, *otherwise.*
$DownCost(v,k) = 0$, *if* v *is a leaf.*
$DownCost(v,k) = Sum_{down}(v) - max\{Sum^*_{down}(w) - DownCost^*(w,k) \mid w$ *is a child of* $v\}$, *otherwise.*□

If we traverse T in a bottom-up fashion and apply the formulas in lemma 3, we can compute $DownCost$ values for a node v with j children in $O(j * l)$ time. We can therefore compute $DownCost$ values for all nodes in $V(T)$ in $O(nl)$ time.

We will now show how to compute $MinCost(v)$. If v has only one child $MinCost(v) = DownCost(v,l)$. Assume that v has only two children, w_1 and w_2. We then have values $DownCost^*(w_1,k)$ and $DownCost^*(w_2,k)$ for $k = 0..l$. We can thus in $O(l)$ time compute $MinCost(v)$ using the formula

$$MinCost(v) = min\{DownCost^*(w_1,k) + DownCost^*(w_2,l-k) \mid k = 0..l\} \quad (1)$$

In the general case where v has more than two children we do the following. Assume that $w_1, ..., w_j$ are the children of v. We cannot use formula 1 directly, since the best core involving nodes from two subtrees, say T_{w_1} and T_{w_2}, is $min\{DownCost^*(w_1,k) + DownCost^*(w_2,l-k) \mid k = 0..l\} + \Sigma_{3 \leq i \leq j} Sum^*_{down}(w_i)$. In order to get a simple formula we will instead compare how much is saved by using a path in any subtree T_{w_i}. To be more specific we could express formula 1 as

$$MinCost(v) = Sum_{down}(v) - max\{Save^*(w_1,k) + Save^*(w_2,l-k) \mid k = 0..l\} \quad (2)$$

where

$$Save^*(w,k) = Sum^*_{down}(w) - DownCost^*(w,k)$$

We now proceed as follows. First we compute a core candidate for the first two children by applying formula 2. We have thus computed a list of $Save(\cdot,k)$ values for both w_1 and w_2. These lists are now merged into one, by taking $min\{Save^*(w_1,k), Save^*(w_2,k)\}$ for each $k = 0..l$. By using the merged list of $Save$ values together with $Save^*(w_3,k)$ values in formula 2, we can compute $MinCost$ in the tree $T_{w_1} \cup T_{w_2} \cup T_{w_3} \cup \{v\}$. By continuing this process we find $MinCost(v)$.

Before we state the main theorem of this section we look at the case, in which a core should have length $\leq l$. In order to find $MinCost(v)$ in this case we only need to ensure that the $DownCost$ values are correct. More precisely we should have $DownCost^*(v,k) = min\{DownCost^*(v,j) \mid 0 \leq j \leq k\}$ for any node v. By using the $DownCost$ values computed using lemma 3 as a basis, this is however easily obtained in $O(l)$ time. Finally we can modify the algorithm to handle the discrete case, by changing the first part of the formula for $DownCost^*(v,k)$ in lemma 3 to: $DownCost^*(v,k) = \infty$, if $length(parent(v),v) \geq k$.

Theorem 4 *Given a tree* T *and a nonnegative integer* l*, let* $\mathcal{P}$ *be a set of discrete or partial paths with length* $= l$ *or* $\leq l$*. We can compute* $cost(Core(\mathcal{P}))$ *in* $O(nl)$ *time.*

Proof. The computation of the $Size$ and Sum values are done in $O(n)$ time using bottom-up and top-down traversals. The computation of $DownCost$ and $MinCost$ values is done by traversing T bottom-up. In this traversal we use $O(l)$ time for each child in T. This computation thus takes $O(nl)$ time. Since the cost of a core in T is $MinCost(v) + Sum_{up}(v)$ for some v, we compute $cost(Core(\mathcal{P}))$ in linear time.□

4 An almost linear algorithm for all cases

In this section we give an $O(n \log n \gamma(n))$ for finding a core in all cases listed in section 2. The function $\gamma(n)$ depends on whether paths are partial, in which case $\gamma(n) = O(\alpha(n,n))$, or discrete, in which case $\gamma(n) = O(1)$. We first present an algorithm with complexity $O(n * h * \gamma(n))$, where h is the height of the tree, measured in the number of edges. Secondly we show how to compress the tree so that $h = O(\log n)$, which establishes the promised complexity.

In this section we will assume that T is binary. If this is not the case we do the following: Pick any node v with more than two children and let w_1, w_2 and w_3 be children of v. We insert a new node u as the parent

of w_1 and w_2 and insert u as a child of v. We repeat this process until no node can be found with more than two children. This process gives a binary tree with at most twice the number of nodes as T. In order to ensure that the binary tree has the same properties relating to cores as the original, we have to look at the generalized problem in which nodes have costs associated. We set the cost of a node to be 1 if it is an original nodes and 0 otherwise. Furthermore we set the length of all inserted edges, that is the edges from an inserted node to its parent, to be 0. This will ensure the preservation of properties relating to cores. In the following subsections, we will restrict our attention to looking for a core should of length $= l$ allowing for partial edges. At the end of the section, we will outline how to deal with the other cases.

4.1 The structure of $DownCost$

In the following, we will discuss how to represent $DownCost$ so as to facilitate an efficient computation of $MinCost$.

For a given node v, we can draw $DownCost(v,\dot{)}$ in a coordinate system as follows. First, for every point a in T_v, where a may be on the middle of an edge, insert the point $(dist(v,a), cost_{T_v}(P_{v,a}))$. Clearly, the points on an edge form a straight line. $DownCost(v,\cdot)$ is now the lower envelope of the inserted points, i.e. $DownCost(v,k)$ is the minimum inserted y-value for $x = k$. Now

Lemma 5 ([5, 14]) *The lower envelope of q straight line segments jumps at most $\Theta(q\alpha(q))$ times between the segments.*□

Thus $DownCost(v,\cdot)$ is a piecewise linear function, dividing into $m = O(|T_v|\alpha(|T_v|))$ pieces. Such a function could, say, be represented as a sequence of point pairs:

$$((x_0,y_1),(x_1,z_1)),((x_1,y_2),(x_2,z_2))\cdots,((x_{m-1},y_m),(x_m,z_m))$$

where (x_{i-1},y_i) and (x_i,z_i) are the boundary points of the ith piece. Thus $DownCost(v,x_i) = \min\{z_i, y_{i+1}\}$ and if $x_{i-1} < x < x_i$, $DownCost(v,x) = y_i + \frac{(z_i-y_i)}{(x_i-x_{i-1})}(x - x_{i-1})$. If $x < x_0$ or $x > x_m$, we define $DownCost(v,x) = \infty$. We refer to the x_i as *break points*. In the following we will think of any piecewise linear function f, as being of the above form, and by $|f|$ we then denote the number of break points.

Observation 6 *In the following, let f and g be piecewise linear functions and let $\delta, \Delta, a, b \in \Re$.*

- *Define $f_1 : x \mapsto f(x+\delta)+\Delta$. Then $|f_1| = |f|$ and f_1 can be constructed in time $O(|f|)$.*
- *Define $f_2 : x \mapsto \min\{f(x), g(x)\}$. Then $|f_2| \le 2(|f|+|g|)-1$, and f_2 can be constructed in time $O(|f|+|g|)$.*
- *Define $f_3 : x \mapsto f(x)+g(x)$. Then $|f_3| \le |f|+|g|$, and f_3 can be constructed in time $O(|f|+|g|)$.*
- *$\mu = \min_{a\le x\le b} f(x)$ is found in time $O(|f|)$.*

Proof. Concerning f_1, note that we just need to subtract δ from all the x_i, and add Δ to all the y_i and z_i.

To prove $|f_2| \le 2(|f|+|g|)-1$, consider any break point p of f_2 which is neither a break point of f nor of g. Then p is the intersection of two straight-line segments of f and g, but then f_2 cannot break again until either f or g has broken. Similarly, the breakpoint before p in f_2 must be from f or g.

Now f_2 is constructed by a merge style procedure, where in each step, we either identify a new piece of f_2, or finish the processing of a piece from f or g. The time of this procedure is $O(|f_2|+|f|+|g|) = O(|f|+|g|)$.

To prove $|f_3| \le |f|+|g|-1$, we simply observe that any break point in f_3 must be a breakpoint in f or in g. The construction of f_3 is done in a merge style procedure in time $O(|f|+|g|)$. □

4.2 An $O(nh\alpha(n))$ algorithm

In this subsection, we are so far still restricting our attention to cores of a length $= l$, on which partial edges are allowed. In particular, this means that we are working with piecewise linear functions

Lemma 7 *Let v be a node with children w_1 and w_2. Given (representations of) $DownCost(w_1, \cdot)$ and $DownCost(w_1, \cdot)$, we can construct $DownCost^*(w_1, \cdot)$, $DownCost^*(w_2, \cdot)$, $DownCost(v, \cdot)$, and $MinCost(v)$ in time $O(|DownCost(w_1, \cdot)| + |DownCost(w_2, \cdot)|) = O(|T_v|\alpha(|T_v||)$.*

Proof. We find $DownCost^*(w_1, x)$ as the concatenation of $((0, Sum^*_{Down}(w_1)))$, $(length(v, w_1), Sum_{Down}(w_1))$ and $DownCost(w_1, x - length(v, w_1))$. $DownCost^*(w_2, \cdot)$ is constructed symmetrically.

Now $DownCost(v, x)$ is found as

$$\min\{DownCost^*(w_1, x) + Sum^*_{Down}(w_2), DownCost^*(w_2, x) + Sum^*_{Down}(w_1)\}.$$

Finally

$$MinCost(v) = \min_{0 \le k \le l} DownCost^*(w_1, k) + DownCost^*(w_2, l - k)$$

Thus, by Observation 6, all the desired values are found in time $O(|DownCost(w_1, \cdot)|+|DownCost(w_2, \cdot)|)$. By Lemma 5,

$$O(|DownCost(w_1, \cdot)| + |DownCost(w_2, \cdot)|) = O(|T_{w_1}|\alpha(|T_{w_1}|) + |T_{w_2}|\alpha(|T_{w_2}|)) = O(|T_v|\alpha(|T_v|))$$

□

Theorem 8 *The core of a tree can be computed in $O(n\alpha(n)h)$, where h is the height of the tree.*

Proof. We apply Lemma 7 bottom-up on all vertices v. For a given vertex v, the computation time is $O(|T_v|\alpha(|T_v|))$. Since no vertex w participates in more than h trees T_v, the result follows. □

4.3 An almost linear algorithm

In the above algorithms, whenever we visit a node v, we look for an optimal core with v as the top-most node. As a result, a node is involved in a computation every time we visit one of its ancestors, and hence our complexity has the height h of the tree as a multiplicative factor.

In this section, we will replace the h-factor by a $(\log n)$-factor. When visiting a node v, we will still look for a core containing v, but the "subtree" it is restricted to, will no longer just be descending from v. Based on balancing techniques, we will assign subtrees to nodes, so that each node is involved in only $O(\log n)$ subtrees. The previous algorithms are then modified to work with these new subtrees.

4.3.1 Flattening a tree

Consider a tree T with n nodes. We will now construct a *flattened* version $F(T)$ of T of height $O(\log n)$. The tree $F(T)$ for T will only be used to describe in which subtree we look for a core. Thus, cost, length etc. of additional nodes/edges in $F(T)$ have no meaning, since the additional nodes are only used to explain in which the subtree computation is done.

In order to obtain a tree with height limited to $O(\log n)$ we use heavy path division, as described by Harel and Tarjan [4]. First for each internal $v \in T$, let $heavy(v)$ denote the child of v with the maximum number of descending nodes. In case of a tie, we pick $heavy(v)$ as the left child. The other child of v is called *light*. We say that the edge from a light child to its parent is a light edge. Edges which are not light are called heavy. The heavy edges partition the nodes in T on heavy paths, such that each node belongs to exactly one heavy path (a leaf, which is not a heavy child, is a heavy path with one node). To compress the heavy path we will not follow the standard by Harel and Tarjan, but instead use the following lemma.

Lemma 9 ([8]) *Given a sequence $d_1 \cdots d_m$ of positive real weights, in time $O(m)$, we can construct an ordered binary tree, such that the depth of the ith leaf is $O(\lceil \log D - \log d_i \rceil)$ where $D = \sum_{i=1}^m d_i$.*□

$F(T)$ is constructed by taking each heavy path, and replace it by the weight balanced tree described in the above lemma. More specifically, let $P = v_1 \cdots v_m$ be a heavy path in T and let d_i denote the number of descendants of the light child of v_i, below denoted as $w(v_i)$. We then construct an ordered binary tree as described in the above lemma and let the i'th leaf be v_i.

Lemma 10 *$F(T)$ has height $O(\log n)$.*

Proof. Let $v_1, \ldots, v_k$ be the nodes from T that we meet on a path from a leaf v_1 to the root in $F(T)$. For each heavy path on this path we meet one node v_i, thus $k \leq \log_2 n$. To get from v_i to v_{i+1} we generally traverse an edge from one balanced tree to another, and $O(\lceil \log w(v_{i+1}) - \log w(v_i) \rceil)$ balance edges, that is, $O(2 + \log w(v_{i+1}) - \log w(v_i))$ edges. Thus a total of

$$\sum_{i=1}^{k} O(2 + \log w(v_{i+1}) - \log w(v_i)) = O(\log n + \log w(v_k)) = O(\log n)$$

□

We note that $F(T)$ only is an abstraction we will use to explain the complexity of finding a core in T. In Figure 2 the construction of $F(T)$ is illustrated.

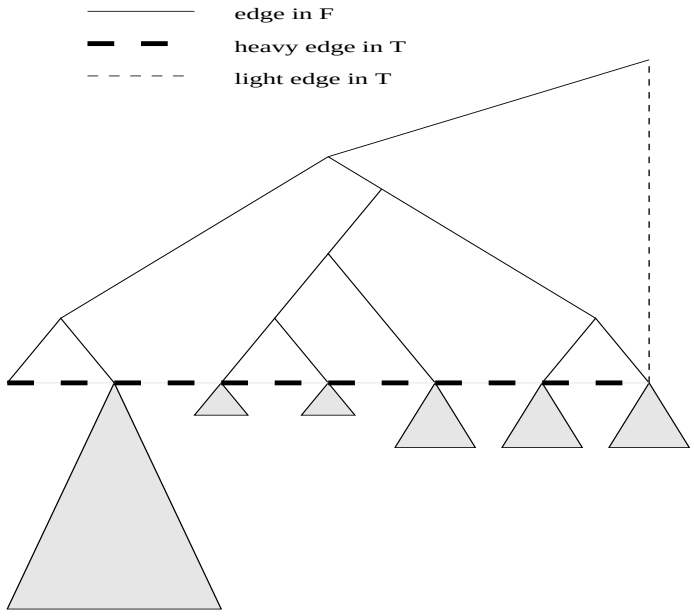

Figure 2: The weight balanced tree replacing a heavy path.

4.3.2 Flattened core computation

Consider the weight balanced tree t over some heavy path $P = v_1 \cdots v_m$ where v_1 is the node nearest the root. Let x be a node in t, and let $P(x) = v_i \cdots v_j$, $i \leq j$, be the segment of P descending from x. Let $tree(x)$ be the subtree in T descending from v_i excluding the subtree rooted at the heavy child of v_j, that is, $tree(x)$ consists of $P(x)$ as well as all light subtrees of nodes on $P(x)$. For a node x let v_i and v_j be as above, $lower(x) = v_j$ and $upper(x) = v_i$. Let T^v be the subtree obtain by deleting the heavy child of v and its descendant from T. If v is a leaf $T^v = T$. Thus, $tree(x) = T^{lower(x)} \cap T_{upper(x)}$. Note that if $x \in T$, $P(x) = x$.

We say that a path Q in T *belongs* to x if it is contained in $tree(x)$ and there is no child y of x in the flattened tree such that Q is contained in $tree(y)$. Clearly, for any path Q in T we have a unique node x that Q belongs to. Thus, we identify the core, if for each node x in the flattened tree, we find the path belonging to it, minimizing the cost in all of T. In order to facilitate a bottom-up computation of cores in the flattened tree, we need some functions analogous to the function $DownCost$ from the previous section.

For any node x in F, define the restricted upcost, $RestrUpCost(x, k)$ as the minimum cost in $T^{lower(x)}$ of a path of length k in $tree(x)$ starting in $lower(x)$. Similarly, define the restricted DownCost, $RestrDownCost(x, k)$, as the minimum cost in $T_{upper(x)}$ of a path of length k in $tree(x)$ starting in $upper(x)$. The realization

that both cost functions are needed is a main point in deriving an efficient algorithm. Observe that both $RestrUpCost$ and $RestrDownCost$ are the lower envelopes of a set of straight line segments, one for each edge in $tree(x)$. Hence both of them have $O(|tree(x)|\alpha(|tree(x)|)$ break points.

We are now ready to describe the bottom-up computation of optimal cores. The vertices x of $F(T)$ are visited in bottom-up order. For each x, we make a series of computations, each taking $O(|tree(x)|\alpha(|tree(x)|)$ time. In the computation, it is important whether x is a node in T. Note that the computation will determine $DownCost(v,\cdot)$ for each root of a heavy path in T.

- Let x be a leaf in T. For $k=0$, $RestrDownCost(x,k)=0$ and $RestrUpCost(x,k)=Sum_{up}(x)$. For $k>0$, each of the above values become ∞.
- If x is in T but not a leaf and w is the light child of x, $RestrDownCost(x,k)=DownCost^*(w,k)+Sum^*_{down}(heavychild(x))$ and $RestUpCost(x,k)=DownCost^*(w,k)+Sum_{up}(x)$. If x does not have a light child, replace $DownCost^*(w,k)$ by 0 in the above formulas.
- If x is the root of the balanced tree over some heavy path, $DownCost(upper(x),k)=RestrDownCost(x,k)$. $DownCost^*(upper(x),\cdot)$ is computed from $DownCost(upper(x),\cdot)$ as described in the previous section.
- If x is not in T and x has children y_1 and y_2 where $P(y_1)$ is above $P(y_2)$,

  $$RestrDownCost(x,k)=\min\{RestrDownCost(y_1,k),RestrDownCost(y_2,k-\delta)+\Delta,f(x)\}.$$

  Here $\delta=dist(upper(y_1),upper(y_2))$ and

  $$\Delta=\sum\{Sum^*_{down}(w)|w \text{ is a light child of a node on } path(y_1)\}.$$

  Finally f is the function corresponding to the edge $(lower(y_1),upper(y_2))$ which has boundary points $(dist(upper(y_1),lower(y_1)),\Delta+Sum^*_{Down}(lower(y_1)))$ and $(dist(upper(y_1),upper(y_2)),\Delta+Sum_{Down}(lower(y_1)))$.

  The function $RestrDownCost(x,\cdot)$ is computed symmetrically.

  Assuming that we are not in the (trivial) case, where the core is contained in the edge $(lower(y_1),upper(y_2))$, we can finally compute $MinCost(x)$ as

  $$\min_k\{RestrDownCost(y_2,k)+RestUpCost(y_1,l-k-length(upper(y_2),lower(y_1)))\}.$$

The above computation takes time $|tree(x)|\alpha(|tree(x)|,|tree(x)|)$. Since each node participates in $O(\log n)$ $tree(x)$-values, we conclude

Theorem 11 *For a tree T with n nodes, lengths on the edges, and weights on the nodes, we can solve the discrete core problem in $O(n\log n)$ time and the partial core problem in $O(n\log n\alpha(n))$ time.*□

We will now briefly consider the other cases of cores. For the case where cores should have length $\leq l$, for each leaf w in T_v, for all $k\geq dist(v,w)$, we have to insert the point (k,c), where c is the cost in T_v of the path from v to w. That is, each leaf gives rise to an extra horizontal line in our coordinate system. Asymptotically, this does not affect any of the above bounds, so again we get that $DownCost(v,\cdot)$ is a piecewise linear function, dividing into $O(|T_v|\alpha(|T_v|))$ pieces.

In the discrete case, we only need to store the at most $|T_v|$ points (k,c) where k is the distance from v to some node in T and c is the least cost of a path to such a node. Thus, we derive the same complexity for these cases.

5 A linear time algorithm for the uniform cases

In this section we assume that all edges have length one. This means that the discrete and partial cases are the same. We present a linear time algorithm for finding a core of length $=l$ or $\leq l$. As an intermediate step we first present an algorithm which speeds up the algorithm from section 3 in case T has few leaves.

5.1 A faster algorithm for trees with few leaves

In this subsection we show how to speed up the algorithm from section 3 if the number of leaves of T is small. We will do this by processing simple paths in the tree differently.

Before we describe the algorithm we note the differences in the deriven formulas of section 3 for the uniform case: The Sum formulas remain unchanged except $length(e) = 1$ for all $e \in E(T)$. The formula for $DownCost^*$ in lemma 3 is simplified to: $DownCost^*(v,k) = DownCost(v,k-1)$ for $k > 0$ and $DownCost^*(v,0) = Sum_{down}(parent(v))$.

Let v be a node with exactly one child w. Since w is the only child of v we have $DownCost(v,k) = DownCost(w,k-1)$ for $k > 0$ and $DownCost(v,0) = Sum_{down}(v)$. We can thus get the $DownCost(v,k)$ values for $k = 1..l$ by copying the $DownCost(w,k-1)$ values. In the following we give a more detailed description of this process.

As before, let v be a node with exactly one child w. First assume that w has more than one child. By using lemma 3 we can compute $DownCost(w,k)$ for $k = 0...l$. These values are inserted in a *cost-list* for w ordered by increasing k. According to the relation between $DownCost(v,k)$ and $DownCost(w,k)$ stated above, we can remove the last element from this list and insert $Sum_{down}(v)$ at the start and obtain a list containing the values $DownCost(v,k)$ for $k = 0..l$. Now assume that w has only one child. We then have a cost-list containing the values $DownCost(w,k)$ for $k = 0..l$. By repeating the described process - deleting the last element from the list and adding the element $Sum_{down}(v)$ as the first element - we obtain a list containing the values $DownCost(v,k)$ for $k = 0..l$.

For the case where the length of the core is $\leq l$, $DownCost(v,k)$ is the lowest cost of a path of length $\leq k$. Since $Sum_{down(v)} \geq DownCost(v,k)$ for all k, the method described above also holds in this case.

For any node v with only one child we will use the processing described above. We compute $MinCost(v)$ as $DownCost(v,l)$.

Lemma 12 *Given a tree T in which edges have length one, let $\mathcal{P}$ be the set of paths with length $\leq l$ or $= l$ in T. We can compute $cost(Core(\mathcal{P}))$ in $O(bl+n)$ time, where b is the number of leaves in T.*

Proof. For each node v with at most one child we use $O(1)$ time and for any other node we use $O(l)$ time for each of its children. Since the number of children of nodes with more than one child is $O(b)$, we use $O(bl+n)$ time all together.□

5.2 A linear time algorithm

In this section we show how the $O(bl+n)$ algorithm from section 5.1 can be modified to a linear time algorithm. Generally speaking, this is done by ignoring subtrees of size $< (l/2)-1$.

We define a *leaf tree* as a subtree in T, in which all nodes has less than $(l/2)-1$ descendants. Let R be the tree from which all leaf trees has been removed.

Lemma 13 *The tree R has $O(n/l)$ leaves.* □

By lemma 12 and 13 we have the following:

Corollary 14 *We can compute $MinCost(v)$ for each node $v \in R$ in $O(n)$ time.*□

By corollary 14 we can obtain a linear time algorithm if we can process the leaf trees in linear time and at the same time combine these with R. We first observe that any path in a leaf tree has length $< l$. We can therefore compute the best core of length $\leq l$ in any leaf tree by using the linear time algorithm for finding a core of unlimited length [11]. It thus only remains to find the core candidates, which contain nodes from both R and leaf trees. In order to do this, we should compute $DownCost(r,\cdot)$ for any root, r, of a leaf tree. This is done in a time linear to the size of the tree in the following way: For each node w at level k, i.e. $|P_{r,w}| = k$, we compute $cost_{T_r}(P_{r,w})$ top-down using the formula $cost_{T_r}(P_{r,w}) = cost_{T_r}(P_{r,parent(w)}) - Size_{down}(w)$. Then we compute $DownCost(v,k)$ as $\min\{cost_{T_r}(P_{r,w}) \mid |P_{r,w}| = k\}$.

Let v be a node which has a removed child r. The child r has been removed because it is the root of a leaf tree t. Furthermore let s denote the size of t, thus $s = Size_{down}(r)$. As stated in the discussion above we can

compute $DownCost^*(r,k)$, $k = 0..height(r)$, values in $O(s)$ time. We compute $MinCost(v)$, in which we also consider leaf trees, by using the same approach as in section 3. However in order to keep the complexity at $O(s)$ we only consider $Save(r,k)$ for $k = 0..height(r)$ when including t in the set of possible locations of a core. Finally we note that the merging of $Save^*$ values described in section 3 is also done in $O(s)$ time for each leaf tree. The inclusion of leaf trees in the $MinCost$ computations in R is thus done in a time linear to the combined size of the leaf trees.

The discussion above yields the following:

Theorem 15 *In the case where all edges have length one, we can compute $cost(Core(\mathcal{P}))$ in $O(n)$ time.*□

References

[1] H. Davenport and A. Schinzel. A combinatorial problem connected with differential equations. *Amer. J. Math.*, 87:684–694, 1965.

[2] A.J. Goldman. Optimal center location in simple networks. *Transportation Sci.*, 5:212–221, 1971.

[3] S.L. Hakimi, M. Labbé, and E.F. Schmeichel. On locating path- or tree-shaped facilities on networks. *Networks*, 23:543–555, 1993.

[4] D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestors. *Siam J. Comput*, 13(2):338–355, 1984.

[5] S. Hart and M. Sharir. Nonlinearity of davenport-schinzel sequences and of general path compression schemes. *Combinatorica*, 6:151–177, 1986.

[6] W. Lo and S. Peng. An optimal parallel algorithm for a core of a tree. In *International conference on Parallel processing*, pages 326–329, 1992.

[7] W. Lo and S. Peng. Efficient algorithms for finding a core of a tree with a specified length. *J. Algorithms*, 20:445–458, 1996.

[8] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching.* EATCS. Springer, 1 edition, 1984.

[9] E. Minieka. The optimal location of a path or tree in a tree network. *Networks*, 15:309–321, 1985.

[10] E. Minieka and N.H. Patel. On finding the core of a tree with a specified length. *J. Algorithms*, 4:345–352, 1983.

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

[12] S. Peng, A.B. Stephens, and Y. Yesha. Algorithms for a core and k-tree core of a tree. *J. Algorithms*, 15:143–159, 1993.

[13] P.J. Slater. Locating central paths in a graph. *Transportation Sci.*, 16:1–18, 1982.

[14] A. Wiernik. Planar realizations of nonlinear davenport-schinzel sequences by segments. In *Foundations of Computer Science*, pages 97–106, 1986.