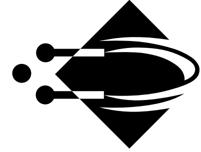


# Finding Cores of Limited Length

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

**IT-C Technical Report Series** **2000-4**

**ISSN *1600–6100*** **July 2001**

Copyright © 2000, Stephen Alstrup
Peter W. Lauridsen
Peer Sommerlun
Mikkel Thorup

The IT University of Copenhagen
All rights reserved.

Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.

ISSN *1600–6100*

ISBN *87–7949–007–7*

Copies may be obtained by contacting:

The IT University of Copenhagen
Glentevej 67
DK-2400 Copenhagen NV
Denmark

Telephone: +45 38 16 88 88
Telefax: +45 38 16 88 99
Web www.it-c.dk

# Finding Cores of Limited Length*

Stephen Alstrup[†] Peter W. Lauridsen[‡] Peer Sommerlund[§] Mikkel Thorup[¶]

June 25, 2001

**Abstract**

In this paper we consider several well-studied variants of the problem of finding a core of a prescribed length in a tree. Here a core is a path minimizing the sum of the distances to all nodes in the tree. Our most general result is an $O(n \log n\, \alpha(n))$ algorithm for the case with weighted tree edges. The previous best bound was $O(n^3)$ due to Minieka (Networks, 1985).

*A preliminary short version of this work was presented at WADS'97 [4]. In the current version, the techniques have been substantially simplified and generalized. Furthermore, we present results on related problems. Some of the work was done while the authors were at the University of Copenhagen.

[†]E-mail:stephen@it-c.dk. The IT University of Copenhagen, Glentevej 67, 2400 Copenhagen NV, Denmark.

[‡]E-mail:pwaern@usa.net. Maconomy, Denmark.

[§]E-mail:peso@netman.dk. Netman, Denmark.

[¶]E-mail:mthorup@research.att.com. AT&T Labs—Research, Shannon Laboratory, 140 Park Avenue, Florham Park, NJ 07932, USA.

# 1 Introduction

In this paper, we study the problem of finding a core of a prescribed length in a tree. Here a *core* is a path minimizing the sum of the distances to all nodes in the tree. That is, a core is a path $P$ in a tree $T$ minimizing $cost(P) = \sum_{v \in T} dist(v, P)$, where $dist(v, P)$ denotes the distance from $v$ to its nearest point in $P$. We consider several well-studied variants depending on whether the core length is $\leq \ell / = \ell$, and on whether the edges are weighted/unweighted. If the edges are weighted, we are also interested in whether the core uses partial/full edges. When using partial edges, we allow that the core may have an end-point somewhere on the middle of a tree edge. We present an $O(n \log n\, \alpha(n))$ algorithm for all cases, improving the previous $O(n^3)$ algorithm [14]. For unweighted edges, we present linear time algorithms, improving the previous $O(n \log n)$ algorithm [12].

**Previous work** The oldest version of the core problem is with length $\ell = 0$, i.e., a single node. Such a core is called a *median*, and a linear time algorithm for finding a median was presented by Goldman in 1971 [7]. In 1980-1982, Morgan and Slater studied the core problem for unlimited length ($\ell \leq \infty$), providing a linear time algorithm for unweighted edges [16, 18]. For the case of weighted edges and unlimited core length, a simple linear time algorithm was provided in 1993 by Peng, Stephens and Yesha [17]. The idea of asking for a core of a prescribed length ($= \ell$) was suggested in 1983 by Minieka and Patel [15]. As they pointed out, we cannot be certain that a path of length $\ell$ exists in a tree, so they suggested to allow for partial edges in the core. Minieka and Patel do not give any algorithm for determining a core of a prescribed length, but they list a number of problems in giving such an algorithm. For example, they point out that an optimal core of length $\ell$ can have a distance sum that is larger than a core of length $< \ell$. Also, they write "we do not know if a core of length $\ell$ 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 prescribed length remains a difficult problem." In Figure 1 we show a tree with 35 nodes in which the core of length 10 does not contain the median. Examples can also be given for ternary trees, essentially replacing the high degree node with a balanced binary tree, but such ternary constructions need more than 100 nodes.

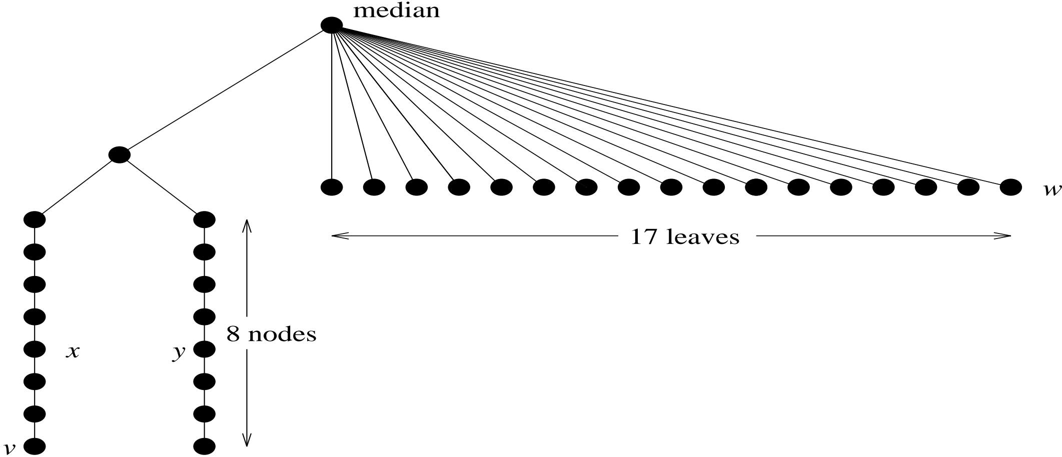


Figure 1: A sample tree, in which all edges have length one. An optimal core of length 10 through the median goes from $v$ to $w$, and has cost $(8 \cdot 9)/2 + 16 = 52$. An optimal core of length 10 that is not required to go through the median goes from $x$ to $y$, and has cost $2 \cdot 6 + 17 \cdot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

Later, in 1985, Minieka showed that a core of a prescribed length ($= \ell$) can be found in $O(n^3)$ time [14]. In 1996, Lo and Peng [12] gave an $O(n \log n)$ algorithm for finding a core of a prescribed length for the case of unweighted edges. Furthermore they claim that their algorithm is easily extended to cases where partial edges are allowed and the edges have arbitrary non-negative lengths. This is true, but if the edge lengths are arbitrary non-negative integers their complexity increases to $O(n\ell \log n)$, which for constant $\ell$ is still $O(n \log n)$.

In 1993 Hakimi, Labbé and Schmeichel [8] examined the problem of finding a core with length $\leq \ell$ 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 $= l$, as shown in [15]. For the case allowing partial edges they show that the $O(n^3)$ algorithm [14] 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 and distributed algorithms have also been examined, see e.g. [1, 10, 11, 13, 12]. Summarizing we note that Minieka's $O(n^3)$ algorithm is easily modified to all cases mentioned above.

### 1.1 This Paper

The problem of locating a core of a tree has previously been investigated for cores containing partial/full edges, core length $= l / \leq l$ and uniform/arbitrary edge lengths. For all combinations of the mentioned conditions, we improve the previous best results. For the cases of uniform edge lengths an $O(n \log n)$ algorithm has been given [12]. In section 4 we present an $O(n)$ algorithm for these cases. For arbitrary edge lengths the algorithms given so far have complexities $O(nl \log n)$ and $O(n^3)$ [12, 14]. In this paper we give two algorithms for all cases. The first is presented in section 3 and determines the core in $O(nl)$ time. The second is presented in section 5 and uses $O(n \log(n) \alpha(n))$ time, where $\alpha(n)$ is the inverse of Ackermann's function. The factor $\alpha(n)$ comes in by establishing a strong relation between the core problem and Davenport-Schinzel sequences [5]. Our $O(n \log(n) \alpha(n))$ has been compared experimentally to the original $O(n^3)$ algorithm of Minieka [14]. In the experiments, our new asymptotically faster but more complicated algorithm dominated Minieka's algorithm already from around 100 nodes. A detailed report on the experiments can be found in [20]. For the construction of the $O(n \log(n) \alpha(n))$ we need a balanced tree structure. For this purpose we use top trees [2, 3]. Using top trees the solution can be presented more elegantly. The top trees were originally developed to be used for a general version of dynamic trees [19], but here we use top trees to solve our static problem. Using the top tree we present a black box for bottom computation in trees which we use for our solution. In appendix C we show how to improve other problems [22] using the black box. For the sake of completeness we also give the details of constructing top trees in appendix A. The algorithms presented in the litterature for finding a median are for weighted nodes, whereas the algorithms given for finding a core are for unweighted nodes. In our presentation, for simplicity, we assume that the nodes are unweighted. Generalizing to weighted nodes with the core minimizing the weighted distance sum to all nodes in the tree is straightforward. In appendix B we show how to generalize when finding a core of unlimited length, since we need this algorithm for the unweighted case.

### 1.2 Subsequent work

After our results were first announced in [4], a paper in [21] present a weaker result, giving an $O(n^2)$ algorithm for finding a core $= l$ using partial edges and edge weight.

## 2 Preliminaries

We adopt the general convention that $\min \emptyset = \infty$. In all our computations, we assume that the core length $\ell$ is not 0. Otherwise, this is just the median problem, which is solved in linear time in [7].

For a tree $T$, $\mathcal{V}(T)$ and $\mathcal{E}(T)$ denote the node and edge set of $T$. In the case of weighted edges, we may consider a point $p$ inside an edge $(v, w)$. Then the sum of the distances from $p$ to $v$ and $w$ is the weight of $(v, w)$. We will use $\mathcal{P}(T)$ to denote the set of all points in $T$. For any two points $p, q \in \mathcal{P}(T)$, $p \cdots q$ denotes the unique path from $p$ to $q$ in $T$. The length of a path $P$ in $T$ is denoted $|P|$. With this notation, the core cost of a path $P$ in $T$ is

$$cost_T(P) = \sum_{v \in \mathcal{V}(T)} \min_{p \in \mathcal{P}(P)} |v \cdots p|.$$

From now on, let $T$ denote a fixed tree for which we wish to find a core. Our first preliminary step is to root $T$ in a median thus minimizing the average distance from the nodes to the root. As described in [7], the median can be found in linear time. For each internal node $v$ in $T$, we assume that the children are ordered, and then $v(i)$ denotes the $i$th child. Further, for each node $v \in \mathcal{V}(T)$, $T_v$ denotes the subtree of $T$ rooted at $v$. In many of our algorithms, we assume the following preprocessing:

**Lemma 1** *We can tabulate the following functions over vertices $v$ in linear time:*

$$\begin{aligned}
SizeDown(v) &= |\mathcal{V}(T_v)| \\
SizeUp(v) &= |\mathcal{V}(T) \setminus \mathcal{V}(T_v)| \\
SumDown(v) &= \sum_{w \in \mathcal{V}(T_v)} |w \cdots v| \\
SumDown^*(v) &= \sum_{w \in \mathcal{V}(T_v)} |w \cdots parent(v)| \\
SumUp(v) &= \sum_{w \in \mathcal{V}(T) \setminus \mathcal{V}(T_v)} |w \cdots v|
\end{aligned}$$

**Proof:** Computing the *Size* functions is trivial. Assuming this is done, we compute the *Sum* functions as follows.

First we compute *SumDown* bottom-up using the formulas:

$$\begin{aligned}
SumDown(v) &= \sum_{w \in Children(v)} (SumDown^*(w)) \\
SumDown^*(v) &= SumDown(v) + |(v, parent(v))| \times SizeDown(v).
\end{aligned}$$

Having computed the *SumDown* functions, we can compute the *SumUp* functions top-down using the formula:

$$\begin{aligned}
SumUp(v) = \; & SumUp(parent(v)) + |(v, parent(v))| \times SizeUp(v) \\
& + (SumDown(parent(v)) - SumDown^*(v)).
\end{aligned}$$

■

## 3 An $O(n\ell)$ algorithm for all cases

In this section, we present an $O(n\ell)$ algorithm for finding the core of a tree in all cases. First we present the algorithm for the case of partial edges with core length $\leq \ell$. Then we mention the minor changes needed to deal with each of the other cases.

For each subtree we compute a core containing the root of the subtree. We define $MinCost(v)$ as the cost of a core containing $v$ in $T_v$. Using the $SumUp$ table from Lemma 1 this suffices to identify the minimum cost of a core in $T$.

**Lemma 2** *The minimum cost of a core in $T$ is $\min_{v \in \mathcal{V}(T)} MinCost(v) + SumUp(v)$.*

**Proof:** By examining $MinCost(v)$ for all $v \in T$, we examine all core candidates except for paths starting at a point $p$ inside an edge and only going down in the rooted tree. However, since the root is a median, the cost of such a path can only decrease if we pull it up to the nearest node above $p$ in $\mathcal{V}(T)$. ■

Given the $SumDown$ and $SizeDown$ tables from Lemma 1, we are going to compute $MinCost(v)$ bottom-up. We need the following auxiliary values to facilitate an efficient computation:

$DownCost(v,k)$ denotes the minimum cost of a path in $T_v$ of a length $\leq k$ starting at $v$.

$DownCost^*(v,k)$ denotes the minimum cost of a path in $T_v \cup \{(parent(v), v)\}$ of length $\leq k$ starting at $parent(v)$.

$DownCost(v,i,k)$ denotes the minimum cost of a path in $T_v$ of length $\leq k$ starting at $v$ and using an edge towards one of the first $i$ children of $v$. For $i = 0$, $DownCost(v,i,k) = SumDown(v)$.

Having computed the above values, we can compute $MinCost(v)$ as the minimum cost for $i = 1, ..., |Children(v)|$ and $k = 0, ..., \ell$ of a path having one side going $k$ down from $v$ towards or past $v(i)$ and having the other side going $\ell - k$ down from $v$ towards or past one of the children preceding $v(i)$, if any. Thus $MinCost(v)$ is computed as

$$\min_{\substack{i=1,...,|Children(v)| \\ k=0,...,\ell}} \{DownCost^*(v(i),k) - SumDown^*(v(i)) + DownCost(v, i-1, \ell - k)\}$$

Having a path of length $\ell - k$ from $v$ downwards one of its first $i-1$ child, $DownCost^*(v(i),k) - SumDown^*(v(i))$ gives us the saving of the cost of the path by extending it $k$ downwards $v(i)$.

For each node $v$ visited in bottom-up order and for $k = 0, ..., \ell$, the $DownCost$ values are computed as follows. First we set

$$DownCost^*(v,k) = \begin{cases} DownCost(v, k - |(v, parent(v))|), \text{ if } |(v, parent(v))| \leq k \\ SumDown(v) + (|(v, parent(v))| - k) \times SizeDown(v), \text{ otherwise} \end{cases}$$

Next, for $i = 1, ..., |Children(v)|$, we set

$$DownCost(v,i,k) = \min \begin{cases} DownCost^*(v(i),k) - SumDown^*(v(i)) + SumDown(v) \\ DownCost(v, i-1, k) \end{cases}$$

Finally, we set

$$DownCost(v, k) = DownCost(v, |Children(v)|, k).$$

Note, that if $v$ is a leaf, $DownCost(v, k) = DownCost(v, 0, k) = SumDown(v) = 0$. For each node, we spend $O(\ell)$ time plus $O(\ell)$ time for each child, so the total time spend over all nodes is $O(n\ell)$.

**Theorem 3** *We can solve the core problem with length $= \ell / \leq \ell$ with weighted tree edges and for partial/full tree edges in the core in $O(n\ell)$ time.*

**Proof:** The above construction settles the theorem for core length $\leq \ell$ and partial edges. To deal with a core of length $= \ell$, and hence with paths of length $= k$ in each of the above $DownCost$ values, the only change needed for our formulas is to set $DownCost(v, 0, k) = \infty$ if $k > 0$.

In the case of full edges and a core of length $\leq \ell$, we take our construction for partial edges and a core of length $\leq \ell$ and set $DownCost^*(v, k) = DownCost^*(v, 0) = SumDown^*(v)$ if $k < |(v, parent(v))|$.

In the case of full edges and a core of length $= \ell$, we take our construction for partial edges and a core of length $\leq \ell$ and set $DownCost^*(v, k) = \infty$ for $0 < k < |(v, parent(v))|$.

None of the above changes affect our time bounds. ■

These results improve the previous $O(n\ell \log n)$ bound of Lo and Peng [12]. In particular, it gives linear time algorithms if $\ell = O(1)$.

# 4 A linear time algorithm for the unweighted cases

In this section, we present a linear time algorithm for finding cores when the tree edges are unweighted. Note that unweighted tree edges imply that partial edges are not relevant.

## 4.1 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. This is done by processing nodes with only one child in constant time.

Let $v$ be a node with exactly one child $w$. No matter whether we are seeking a core of length $\leq \ell$ or a core of length $= \ell$, then $DownCost(v, k) = DownCost^*(w, k)$ and $MinCost(v) = DownCost^*(w, \ell)$. Moreover, $DownCost^*(v, 0) = SumDown^*(v)$ while $DownCost^*(v, k) = DownCost(v, k - 1)$ for $k > 0$. The auxiliary variables $DownCost(v, i, k)$ are not relevant in this case.

The above observations suggest that we represent $DownCost^*(w, \cdot)$ as a sorted doubly linked list with $DownCost^*(w, k)$ the $(k+1)$th element. We store pointers to both the first and the last element of $DownCost^*(w, \cdot)$. Then $DownCost(v, \cdot)$ takes over the list from $DownCost^*(w, \cdot)$ in constant time. Since $MinCost(v) = DownCost^*(w, \ell)$, it is found in constant time as the last element of $DownCost^*(w, \cdot)$. Now, $DownCost(v, \cdot)$ is no longer needed, so we compute $DownCost^*(v)$ simply by putting $DownCost^*(v, 0) = SumDown^*(v)$ in front of the list of $DownCost(v, \cdot)$ and removing the last element.

For all other nodes, that is, for the leaves and the internal nodes with multiple children, it is not a problem with the above list representation since we have $O(\ell)$ time for each list involved.

**Lemma 4** *For an unweighted tree, we can find a core of length $\leq \ell / = \ell$ in $O(b\ell + n)$ time where $b$ is the number of leaves in $T$.*

**Proof:** We spend constant time on each node with one child, adding up to $O(n)$ time. For all other nodes we spend $O(\ell)$ plus $O(\ell)$ time per child, which adds up to $O(b\ell)$ time. ■

### 4.2 A linear time algorithm

In this section we show how the above $O(b\ell + n)$ algorithm can be modified to a linear time algorithm. This is done by taking special care of subtrees of size $< \ell$.

We define a *leaf tree* as a subtree in $T$ in which every node has $< \ell$ descendants, including itself. Let $R$ be the tree from which all leaf trees have been removed.

**Lemma 5** *The tree $R$ has at most $n/\ell$ leaves.*

**Proof:** Each leaf in $R$ must be a node in $T$ with at least $\ell$ descendants, and no two leaves of $R$ share any descendants in $T$. ■

Our goal is to spend constant time per node in a leaf tree plus constant time per node with degree one in $R$. For all other nodes in $R$, we are willing to spend $O(\ell)$ time, as in Section 3.

First we consider the problem of finding a core within a leaf tree. Since a leaf tree has $< \ell$ nodes, there is no such core of length $= \ell$. If we are seeking a core of length $\leq \ell$, we take each leaf tree with root $v$ and find the best core of unlimited length in the leaf tree. This can be done as follows : let the node $v$ have weight $SizeUp(v) + 1$, and the remaining nodes have weight 1. Next, we find the best weighted core with unlimited length in the leaf tree and add to its cost value $SumUp(v)$. The weighted core with unlimited length in a tree $T$ have cost $\min_{v,w \in \mathcal{V}(T)} \sum_{z \in \mathcal{V}(T)} weight(z) dist(z, P)$, where $P = v \cdots w$. It is straightforward to generalize any of the existing linear time algorithms for finding a core with unlimited length in a tree without node weight to a tree with node weights. In appendix B we give the details.

As in Lemma 2, the minimum cost of a core containing a node in $R$ is the minimum value of $MinCost(v) + SumUp(v)$ with $v \in R$. The $SumUp$ values are available from Lemma 1, so it only remains to find $MinCost(v)$ for each $v \in R$.

We will use the formulas from Section 3. To get started, for each root $w$ of leaf tree, we have to find $DownCost(w, \cdot)$. Assuming we are seeking a core of length $= \ell$, for $k \leq \min\{height(T_w), \ell\}$,

$$DownCost(w, k) = \min_{u \in \mathcal{V}(T_w),\ |u \cdots w| = k} cost_{T_w}(u \cdots w).$$

Now, $cost_{T_w}(w \cdots w) = DownCost(w, 0) = SumDown(w)$, and for all $u \neq w$, we can compute $cost_{T_w}(u \cdots w)$ top-down in constant time using

$$cost_{T_w}(u \cdots w) = cost_{T_w}(parent(u) \cdots w) - SizeDown(u).$$

By considering the nodes in $T_w$ breadth-first, minimizing over each layer, we compute $DownCost(w, k)$ for each $k \leq height(T_w)$ in $O(|\mathcal{V}(T_w)|)$ time.

For $k > height(T_w)$, $DownCost(w, k) = \infty$. That is, the last $\min\{0, \ell - height(T_w)\}$ elements of $DownCost(w, \cdot)$ are $\infty$. We only wish to spend constant time and space on these last elements,

so we generally introduce multi-elements $a^i$, representing $i$ iterations of the element $a$, but only storing $a$ and $i$. Thus, if $\ell > height(T_w)$, we complete $DownCost(w, \cdot)$ by placing $\infty^{\ell - height(T_w)}$ after $DownCost(w, height(T_w))$. Consequently, for core length $= \ell$, we can compute the lists $DownCost(w, \cdot)$ for each root $w$ of a leaf tree in linear total time.

If instead we are seeking a core of length $\leq \ell$, we first compute $DownCost(w, k)$, $k \leq \min\{height(T_w), \ell\}$, as above, but subsequently, for $k = 0, ..., \min\{height(T_w), \ell\}$, we set $DownCost(w, k) = \min\{DownCost(w, k-1), DownCost(w, k)\}$. Further, if $k > height(T_w)$ we append $DownCost(w, height(T_w))^{\ell - height(T_w)}$. The time bounds are unchanged.

Having computed $DownCost(w, \cdot)$ for each root $w$ of a leaf tree, we compute $DownCost^*(w, \cdot)$ as in the subsection 4.1, simply by putting $DownCost^*(w, 0) = SumDown^*(w)$ in front of $DownCost(w, \cdot)$, and removing the last element. In conclusion we can compute $DownCost^*(w)$ for each root $w$ of a leaf tree in linear total time.

To combine with the ideas from the previous subsection, for each node in $R$, we assume that all children from $R$ come first in the children list.

Further, with each element in a $DownCost$ list, we store a $\delta$-value to be added to the element and to all succeeding elements in the list. This allows us to add the same value to all elements in a list in constant time. Also, we store the sum $\Delta$ of these $\delta$-values. Now, to get the correct value of the last element of a list, we simply add $\Delta$, and if we want to remove the last element, we just have to subtract its $\delta$-value from $\Delta$.

Now, in the processing of a node $v \in R$, first we set

$$\begin{aligned} DownCost(v, 1, \cdot) &= DownCost^*(v(1), \cdot) - SumDown^*(v(1)) + SumDown(v), \text{ and} \\ MinCost(v) &= DownCost(v, 1, \ell). \end{aligned}$$

Above, the addition of $-SumDown^*(v(1)) + SumDown(v)$ to all elements in $DownCost^*(v(1), \cdot)$ is implemented by addition to both $\Delta(DownCost^*(v(1), \cdot))$ and $\delta(DownCost^*(v(1), 0))$.

All of this takes constant time. The constant time is important if $v$ has only one child $w$ in $R$. Since children in $R$ are listed first, $v(1) = w$, and hence we spend only constant time on this child, as required.

Now, let $i$ increase from 2 to $|Chidren(v)|$. For each value of $i$, first, for $k = 0, ..., height(T_{v(i)})$, we set

$$MinCost(v) = \min \begin{cases} MinCost(v) \\ DownCost(v, i-1, \ell - k) + DownCost^*(v(i), k) - SumDown^*(v(i)) \end{cases}$$

Now, $DownCost(v, i-1, \cdot)$ is no longer needed, and we use it as an initial value for $DownCost(v, i, \cdot)$. Afterwards, for $k = 0, ..., \min\{\ell, height(T_{v(i)})\}$, we set

$$DownCost(v, i, k) = \min \begin{cases} DownCost^*(v(i), k) - SumDown^*(v(i)) + SumDown(v) \\ DownCost(v, i-1, k) \end{cases}$$

Finally, when done with $DownCost(v, |Children(v)|, \cdot)$, we assign it to $DownCost(v, \cdot)$.

Thus, the time spent at $v$ is

$$O(1 + \sum_{i=2,..,|Children(v)|} \min\{\ell, height(T_{v(i)})\})$$

The $\ell$ bound is interesting if $v(i)$ is in $R$, for the first child from $R$, if any, takes constant time, and since $R$ has at most $n/\ell$ leaves, this adds up to a total of $O(n)$. The bound of $height(T_{v(i)})$ is interesting if $v(i)$ is the root of a leaf tree. Each leaf tree is only involved once, and hence this also adds up to $O(n)$. In conclusion,

**Theorem 6** *For unweighted tree edges, we can find a core of length $= \ell / \leq \ell$ in $O(n)$ time.* ■

# 5 An $O(n \log(n) \alpha(n))$ algorithm for all cases

In the following we present an algorithm for all cases, with $O(n \log(n)\alpha(n))$ time complexity and $O(n\alpha(n))$ space complexity. The function $\alpha(n)$ is the inverse of Ackermann's function. The result is achieved by establishing a strong relation to Davenport-Schinzel sequences. In all our developments, we focus on finding a core of length $= \ell$.

## 5.1 Core cost and Davenport-Schinzel sequences

For any node $v \in \mathcal{V}(T)$ and connected subgraph $S$ of $T$, define

$$cost_T(v, S, k) = \min\{cost_T(v \cdots p) \mid p \in \mathcal{P}(S),\ |v \cdots p| = k\}$$

As an example from the previous section, we have

$$DownCost(v, k) = cost_{T_v}(v, T_v, k)$$

We will argue that any function $cost_T(v, S, \cdot)$ is piece-wise linear and divides into $O(|\mathcal{E}(S)|\alpha(|\mathcal{E}(S)|))$ straight line segments. To see this, consider the function $\mu : \mathcal{P}(T) \to \mathbb{R}^2$ mapping $p \in \mathcal{P}(T)$ to $(|v \cdots p|, cost_T(v \cdots p))$. Then the image $\mu(e) = \{\mu(p) \mid p \in \mathcal{P}(e)\}$ of an edge $e = (u, w)$ is the straight line segment from between $(|v \cdots u|, cost_T(v \cdots u))$ and $(|v \cdots w|, cost_T(v \cdots w))$. Moreover, $cost_T(v, S, \cdot)$ is the lower envelope of the image of $T_v$, that is,

$$cost_T(v, S, x) = \min\{y \mid (x, y) = \mu(p),\ p \in \mathcal{P}(T_v)\}.$$

Figure 2 shows an example where three edges define three segments, and *DownCost* is found as the lower envelope of these three segments.

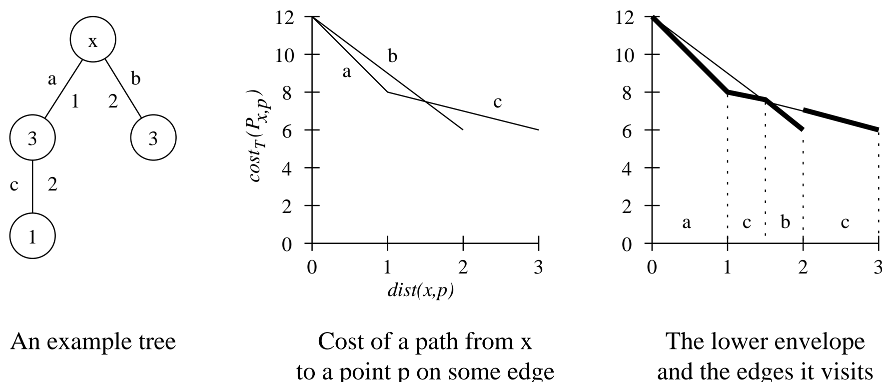


Figure 2: An example of how *DownCost* is computed as the lower envelope of a number of linear segments defined by edges. Note that the nodes have weights. This is to keep the tree small, and yet have an interesting example.

The problem of identifying the number of jumps of the lower envelope was posed by Davenport and Schinzel in 1965 [5], and was resolved in 1986 by Hart and Sharir:

**Theorem 7 ([9])** *The lower envelope of $q$ straight line segments jumps $O(q\alpha(q))$ times between the segments.* ■

For any piece-wise linear function $f$, let $\|f\|$ be the number of linear segments of $f$.

**Corollary 8** *For any node $v \in \mathcal{V}(T)$ and connected subgraph $S$ of $T$, $\|cost_T(v, S, \cdot)\| = O(|\mathcal{E}(S)|\alpha(|\mathcal{E}(S)|))$.*
■

Since all edge (and node) weights are integers, $\mu$ maps each node $w \in \mathcal{V}(S)$ to a point $(|v \cdots w|, cost_T(v \cdots w))$ where both coordinates are integers. Based on this, it follows that the coordinates of the end-point of the straight-line segments of $cost_T(v, S, \cdot)$ can be represented exactly as integer fractions, thus leading to an exact representation in $O(\|cost_T(v, S, \cdot)\|)$ space. In the rest of the paper we will ignore this representation issue, pretending that coordinates are computed directly as real numbers.

### 5.2 A simple case study

We will now discuss a strongly polynomial implementation of the algorithm from Section 3, except that we are looking for a core of length $= \ell$ instead of $\leq \ell$. For simplicity, we assume that the tree $T$ is binary. If $h$ is the height of the tree, we will get an $O(nh\alpha(n))$ algorithm.

Since $\ell$ can be arbitrarily large, we are no longer satisfied using $O(\ell)$ space on a function $DownCost(v, \cdot)$. Instead, we use the $O(|\mathcal{E}(T_v)|\alpha(|\mathcal{E}(T_v)|))$ space representation for all $k$ from the last subsection.

For a leaf $v$, we have $DownCost(v, 0) = 0$ and $DownCost(v, k) = \infty$ if $k > 0$. For any node $v$, given $DownCost(v, \cdot)$, we compute

$$DownCost^*(v, k) = \begin{cases} DownCost(v, k - |(v, parent(v))|), & \text{if } |(v, parent(v))| \leq k \\ SumDown(v) + (|(v, parent(v))| - k) \times SizeDown(v), & \text{otherwise} \end{cases}$$

This computation amounts to a simple shift of $DownCost(v, \cdot)$ plus putting a straight-line segment in front. This takes $O(\|DownCost(v, \cdot)\|) = O(|\mathcal{E}(T_v)|\alpha(\mathcal{E}(|T_v|))$ time.

Now, let $v$ be an interior node. Since $T$ is binary,

$$MinCost(v) = \min_{k \in [0, \ell]} DownCost^*(v(1), k) + DownCost^*(v(2), \ell - k),$$

which is straightforward to compute in $O(\|DownCost^*(v(1), \cdot)\| + \|DownCost^*(v(2), \cdot)\|) = O(|\mathcal{E}(T_v)|\alpha(\mathcal{E}(|T_v|))$ time.

Finally, we have to compute

$$DownCost(v, k) = \min \begin{cases} DownCost^*(v(1), k) + SumDown^*(v(2)) \\ DownCost^*(v(2), k) + SumDown^*(v(1)) \end{cases}$$

which again is straightforward to do in $O(\|DownCost^*(v(1), \cdot)\| + \|DownCost^*(v(2), \cdot)\|) = O(|\mathcal{E}(T_v)|\alpha(\mathcal{E}(|T_v|))$ time.

Thus, to process a node $v$, we pay at most $O(\alpha(n))$ for each edge below it, and since each edge appears below at most $h$ nodes, the total cost of the computation is $O(nh\alpha(n))$.

## 5.3 Top Trees

The naive $O(nh\alpha(n))$ algorithm from the previous section works in $O(n\log(n)\alpha(n))$ time for a balanced binary tree of depth $O(\log n)$. To achieve this time bound for an arbitrary $T$, we run an algorithm of the same flavor over a top tree [2, 3], which is a balanced binary tree $\tau$ defined on top of our tree $T$. Top trees are designed for dynamically changing trees, but in this paper, we demonstrate that they have some additional merit for static problems.

**Definition 9** *Let $T$ be a tree with $n$ nodes. For a connected subgraph $C$ of $T$, we call a node in $C$ with a neighbor in $T$ outside $C$ a* boundary node. *A* cluster *is a connected subgraph of $T$ with at most two boundary nodes. Two clusters $A$ and $B$ intersecting in a single node can be* merged *into $C = A \cup B$ if $C$ is a cluster. A* top tree *$\tau$ over $T$ is a binary tree with the following properties:*

1. *The nodes of $\tau$ represent some of the clusters of $T$.*
2. *The leaves of $\tau$ represent the edges of $T$.*
3. *Any internal node of $\tau$ represents a cluster $C$ merged from the clusters $A$ and $B$ represented by its children.*
4. *The root of $\tau$ represents $T$.*
5. *The height of $\tau$ is $O(\log n)$.*

*For each cluster $C$, $\partial C$ denotes the set of boundary nodes. The path between the boundary nodes is called the* cluster path, *denoted $\pi(C)$. If $C$ has only one boundary node $v$, $\pi(C)$ is the 0 length path from $v$ to $v$.*

The different cases of merging are illustrated in Figure 3.

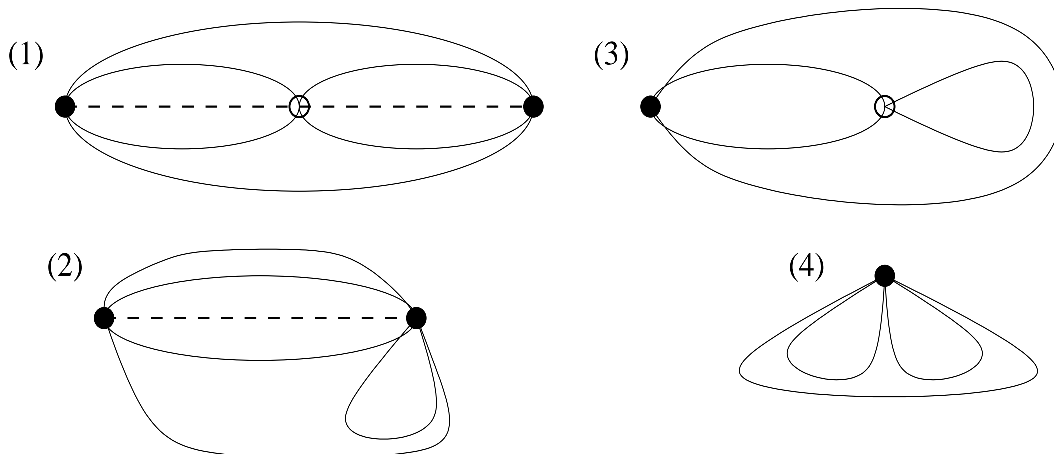


Figure 3: The cases of merging two clusters into one. The • are the boundary nodes of the merged cluster and the ∘ are the boundary nodes of the children clusters that did not become boundary nodes of the merged cluster. Finally the dashed line is the cluster path of the merged cluster.

From [2, 3] we know that a top tree $\tau$ over $T$ can be constructed in linear time. The construction from [2, 3], which is based on a reduction to Frederickson's topology trees [6], is very

complicated. However, the construction addresses the more general problem of maintaining top trees for a dynamically changing forest. Here, we only need to compute a static top tree $\tau$ once, and this is much simpler. A direct construction is given in the appendix. This direct construction was used for the implementations of our algorithm in [20].

In our usage of top trees, for each cluster $C$ in $\tau$ we are going to compute information such as the minimum cost of a path located in cluster $C$. The computation is done bottom-up in the top tree, such that when computing the information for a cluster, the information is already computed for its children. Since the root cluster represents $T$, this will give us the desired core cost. Before describing the exact information needed for computing the core cost, we will present a general accounting scheme for computing information via a top tree.

Let $Info(C)$ be the desired information for a cluster $C$. The previous definition of *merging* two clusters $A$ and $B$ into cluster $C$ is hereafter extended to include computing $Info(C)$ given $Info(A)$ and $Info(B)$.

Given a top tree $\tau$ of $T$.

- Let $t_{single}$ and $s_{single}$ be the amount of time and space required to compute $Info(C_e)$ given the cluster $C_e$ represented by a leaf node $e$ from $\tau$.
- Let $t_{merge}(|\mathcal{E}(C)|)$ and $s_{merge}(|\mathcal{E}(C)|)$ be the time and space required to merge clusters $A$ and $B$ into $C$, where these three clusters are represented by nodes in $\tau$.

Then we have the following lemma:

**Lemma 10** *Let $t_{single}$ and $t_{merge}$ be defined as above. Assume $\frac{t_{merge}(x)}{x}$ and $\frac{s_{merge}(x)}{x}$ are non-decreasing. Given a tree $T$ with $n$ nodes and a top tree $\tau$ of $T$, we can compute $Info(T)$ in $O(nt_{single} + t_{merge}(n)\log(n))$ time and $O(ns_{single} + s_{merge}(n))$ space.*

**Proof:** A top tree has height $O(\log n)$, so each edge is part of $O(\log n)$ merges. Since $\frac{t_{merge}(x)}{x}$ is non-decreasing, the maximal cost per edge for participating in a merge is $t_{merge}(n)/n$. Thus the total merge cost is $O(n \log n t_{merge}(n)/n) = O(t_{merge}(n) \log n)$. For the space bound, we assume that the merges are done bottom-up, only one layer at the time. Here a layer is constituted from nodes on the same depth in $\tau$. In each layer, each edge participates in only one merge, and the maximal space per edge is $s_{merge}(n)/n$. Hence the maximal space for processing a layer is $O(s_{merge}(n))$. ∎

Lemma 10 should be held against the dynamic results on top trees from [2, 3] stating that we can maintain top trees with information for a dynamic forest in $O(t_{single} + \log(n)t_{merge}(n))$ time per edge insertion or deletion. This bound is similar to ours for computing the information from scratch. However, the dynamic bound does not require that $t_{merge}(n)/n$ is non-decreasing, but only that $t_{merge}(n)$ is non-decreasing. This is important because $t_{merge}$ and $t_{single}$ are constant in [2, 3], leading to a dynamic $O(\log n)$ bound.

### 5.4 The cluster information

We will now describe what information we wish to compute for the clusters. The text below covers the case where partial edges are allowed and we search for a core of length $= \ell$. The modifications required to cover the remaining cases are discussed at the end of the section.

For each cluster $C$, we will compute $CoreCost(C)$ which is the minimum cost in $T$ of a core contained in $C$. Since $C = T$ for the root cluster, this will gives us the minimum cost of a core in

$T$. Moreover, we will compute some auxiliary variables facilitating a fast bottom-up computation in the top tree. We will compute $PathCost(C)$ denoting $cost_T(\pi(C))$, and for each boundary node $v \in \partial C$, we will compute the function $BoundaryCost(v, C, \cdot)$ that maps $k$ into the lowest cost of a path of length $k$ that extends from $v$ into $C$. That is,

$$BoundaryCost(v, C, k) = cost_T(v, C, k),$$

where $cost_T(v, C, k)$ are as defined in subsection 5.1.

In our computations of information, we use $NodeCost(v)$ to denote the core cost of the length 0 path $v$ and $EdgeCost((v, w))$ to denote the core cost of the path $v \cdots w$. Then $NodeCost(v) = SumUp(v) + SumDown(v)$ and $EdgeCost((v, w)) = NodeCost(v) - |(v, w)|SizeDown(w)$ assuming $v$ is closest to the root of $T$. Hence $NodeCost$ and $EdgeCost$ are both readily available by Lemma 1.

**Leaf clusters** We now proceed to show how to compute the cluster values for the edges of $T$, which are the clusters represented by the leaves of our top tree $\tau$. Let $e = (v, w)$ be an edge with $v$ closest to the root of $T$. If $k \leq |e|$, we have

$$BoundaryCost(v, e, k) = EdgeCost(e) + (|e| - k) \times SizeDown(w)$$

and

$$BoundaryCost(w, e, k) = EdgeCost(e) + (|e| - k) \times SizeUp(w).$$

For $k > |e|$, $BoundaryCost(\cdot, e, k) = \infty$.

Since $v$ is closest to our median root,

$$CoreCost(e) = BoundaryCost(v, e, \ell).$$

If $u \in \{v, w\}$ is the only boundary node of $e$, $u$ is the 0-length cluster path of $e$, and $PathCost(e) = NodeCost(u)$. Otherwise, $PathCost(e) = EdgeCost(e)$. Each of the above computations are done in constant time and space, so we conclude that $t_{single}, s_{single} = O(1)$.

**Merged clusters** Now, consider a cluster $C$ created from merging the two clusters $A$ and $B$. Let $v$ be the intersection point between $A$ and $B$. Then

$$CoreCost(C) = \min\{CoreCost(A), CoreCost(B), MinCost(v)\}$$

where $MinCost(v)$ is the minimum cost of a core containing $v$, computed as

$$MinCost(v) = \min_{k \leq \ell} BoundaryCost(v, A, k) + BoundaryCost(v, B, \ell - k) - NodeCost(v).$$

The computation of $MinCost(v)$ is done in $O(\|BoundaryCost(v, A, \cdot)\| + \|BoundaryCost(v, B, \cdot)\|)$ time, which by Corollary 8 is $O(|\mathcal{E}(C)|\alpha(|\mathcal{E}(C)|)$ time.

Now we compute $PathCost$ as follows. If $v \notin \pi(C)$ (Figure 3 (3)), $\pi(C)$ is the unique boundary node $c$ of $C$, and then

$$PathCost(C) = NodeCost(c)$$

Otherwise, $\pi(C) = \pi(A) \cup \pi(B)$, and hence

$$PathCost(C) = PathCost(A) + PathCost(B) - NodeCost(v)$$

In either case, *PathCost* is computed in constant time.

Finally, we need to compute $BoundaryCost(u, C, \cdot)$ for each boundary node $u$ of $C$. By symmetry, we may assume that $u \in \partial A$

If $u = v$ (Figure 3 (2) [with $u$ the boundary node on the right] and (4))

$$BoundaryCost(u, C, k) = \min\{BoundaryCost(u, A, k), BoundaryCost(u, B, k)\}$$

The computation is done in $O(\|BoundaryCost(u, A, \cdot)\| + \|BoundaryCost(u, B, \cdot)\|)$ time, which by Corollary 8 is $O(|\mathcal{E}(C)|\alpha(|\mathcal{E}(C)|)$ time.

If $u \in \partial A \setminus \partial B$, (Figure 3 (1), (2) [with $u$ the boundary node on the left] and (3))

$$BoundaryCost(u, C, k) = \begin{cases} BoundaryCost(u, A, k) \text{ if } k \le |\pi(A)|\text{; otherwise:} \\ \min \begin{cases} BoundaryCost(u, A, k) \\ BoundaryCost(B, v, k - |\pi(A)|) \\ +PathCost(A) - NodeCost(v) \end{cases} \end{cases}$$

The computation is done in $O(\|BoundaryCost(u, A, \cdot)\| + \|BoundaryCost(v, B, \cdot)\|)$ time, which by Corollary 8 is $O(|\mathcal{E}(C)|\alpha(|\mathcal{E}(C)|)$ time. Thus, $t_{merge}(x), s_{merge}(x) = O(x\alpha(x))$, so Lemma 10 gives the following theorem for the case of partial tree edges in the core and core length $= \ell$:

**Theorem 11** *We can find a core of length $\le \ell / = \ell$ allowing partial edges in $O(n \log(n)\alpha(n))$ time and $O(n\alpha(n))$ space.*

*If we only allow full edges in the core, we can find it in $O(n \log(n))$ time and $O(n)$ space.*

**Proof:** We have already dealt with the case of partial edges and core length $= \ell$. Still dealing with partial edges, if we seek a core of length $\le \ell$, the only change to the computation is to set $CoreCost(e) = EdgeCost(e)$ if $\ell > |e|$ and $BoundaryCost(\cdot, e, k) = EdgeCost(e)$ if $k > |e|$.

In the case with full edges and cores of length $= \ell$, we redefine $BoundaryCost(v, C, k) = \min_{w \in C}\{cost_T(v \cdots w) \mid |v \cdots w| = k\}$. In our computations, this means that we have to set $CoreCost(e) = \infty$ if $\ell \ne |e|$ and $BoundaryCost(\cdot, e, k) = \infty$ if $k \notin \{0, |e|\}$. Now $BoundaryCost(v, C, k)$ is the lower envelope of $|\mathcal{V}(C)|$ points, so the lower envelope consists of $\le |\mathcal{V}(C)|$ points, and hence all computations over $C$ are done in $O(|\mathcal{E}(C)|)$ time, leading to $O(n \log n)$ total time by Lemma 10.

In the case with full edges and cores of length $\le \ell \ne 0$, we redefine $BoundaryCost(v, C, k) = \min_{w \in C}\{cost_T(v \cdots w) \mid |v \cdots w| = k\}$. In our computations, this means that for an edge $e = (v, w)$, we have to set $CoreCost((v, w)) = \min\{NodeCost(v), NodeCost(w)\}$ if $\ell < |e|$, and $CoreCost((v, w)) = EdgeCost(e)$ if $\ell \ge |e|$. Similarly, we set $BoundaryCost(v, e, k) = NodeCost(v)$ if $k < |e|$, and $BoundaryCost(v, e, k) = EdgeCost(e)$ if $k \ge |e|$. Now $BoundaryCost(v, C, k)$ is the lower envelope of $|\mathcal{E}(C)|$ horizontal lines, so the lower envelope consists of $\le |\mathcal{E}(C)|$ horizontal lines, and hence all computations over $C$ are done in $O(|\mathcal{E}(C)|)$ time, leading to $O(n \log n)$ time by Corollary 10. ■

## References


[1] E. A. Albacea. Parallel algorithm for finding a core of a tree network. *Information Processing Letters*, 51(5):223–226, 1994.

[2] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Minimizing diameters of dynamic trees. In *Automata, Languages and Programming, 24th International Colloquium*, volume 1256 of *Lecture Notes in Computer Science*, pages 270–280. Springer-Verlag, 7–11 July 1997.

[3] S. Alstrup, J. Holm, and M. Thorup. Maintaining median and center in dynamic trees. In *7th Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 1851 of *Lecture Notes in Computer Science*, pages 46–56. Springer-Verlag, 2000.

[4] S. Alstrup, P.W. Lauridsen, P. Sommerlund, and M. Thorup. Finding cores of limited length. In *Algorithms and Data Structures, 5th International Workshop*, volume 1272 of *Lecture Notes in Computer Science*. Springer, 1997.

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

[6] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Computing*, 14(4):781–798, 1985.

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

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

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

[10] E. Jennings. Distributed algorithm for finding a core of a tree network. *SOFSEM'95*, 1012:385–390, 1995.

[11] 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.

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

[13] W.-T Lo and S. Peng. A simple optimal parallel algorthm for a core of a tree. *Journal of parallel and distributed computing*, 20:388–392, 1994.

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

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

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

[17] 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.

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

[19] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *JCSS*, 26(3):362–391, 1983. See also STOC'81.

[20] P. Sommerlund. Finding cores of limited length. Master's thesis, Computer Science, University of Copenhagen, 1998.

[21] B-F. Wang. Efficient parallel algorithms for optimally locating a path and a tree of a specified length in a weighted tree network. *Journal of algorithms*, 34:90–108, 2000. The paper was submitted in 1997.

[22] B. Y. Wu, K. Chao, and C. Y. Tang. An efficient algorithm for the length-constrained heaviest path problem on a tree. *Information processing Letters*, 69(2):63–67, 1999.


# A Static construction of top trees

In this section we show how to construct a top tree in linear time. The leaves in a top tree are the clusters at level 0, representing the edges from $T$. Next we consider how to merge these clusters, consisting of a single edge. Two edges $(a, v)$, $(b, v)$ can be merged to a cluster at the next level

(i) if $v$ has degree 2, see Figure 3, Case (1) and (3), where $v$ is a $\circ$ node.

or

(ii) if either $a$ or $b$ is a leaf, see Figure 3 Case (2) and (3), where $v$ is a $\bullet$ node.

Now, given that a cluster can only participate in one merge, we can show :

**Lemma 12** *Given a tree $T$ with $n$ nodes, we can divide it into $n/c$ clusters, consisting of at most two edges, for a constant $c > 1$.*

**Proof:** For $n > 1$ a tree has at least $n/2 + 1$ nodes with degree below 3. Each of these nodes can be involved in a merge of type (i) or/and (ii). Choosing one of these nodes to be involved in a merge for constructing a cluster, can exclude another node to be involved in a merge. E.g., on a path with an odd number of edges, each of the nodes can be selected for a merge, however we can not select all the nodes, since an edge at can be involved in one merge only. However, this is only the case for simple paths consisting of an odd number of nodes with degree 2, and a node incident with an odd number of leafs. In both cases at least 2/3 of the nodes can be chosen to be involved in a merge. Hence, of the $n/2 + 1$ nodes with degree below 3, at least 2/3 of these can be involved in a merge, giving that at least $2n/3$, for $n > 1$, nodes can be involved in a merge. Since at most three nodes are involved in a single merge, at least $4n/9$ of the edges can be merged. ■

Given a tree $T$ we can divide it into $n/c$ clusters for a constant $c > 1$. These clusters represent the clusters at level 1 in the top tree and induced another tree $T'$ with $n/c$ edges. For (i) an edge $(a, b)$, for (ii) an edge $(a, v)$ if $b$ is leaf, or $(b, v)$ if $a$ is leaf (arbitrary choice if both $a$ and $b$ are leafs). For an edge $e$ not merged with another edge, $e$ is included in the induced tree.

**Theorem 13** *Given a tree $T$ with $n$ edges it is possible to construct a top tree $\tau$ of $T$ in $O(n)$ time.*

**Proof:** Applying Lemma 12 to a tree, gives another induced tree whos size is only a constant fraction $(1/c)$ of the original tree. Hence, applying the lemma $O(\log n)$ times, we have a top tree of height $O(\log n)$. Since the division can be done in $O(n)$ time for a tree of size $n$, we get a total complexity $\sum_{i=0}^{O(\log n)} O(n/c^i) = O(n)$. ■

In the above calculation we have been sloppy, since we were only interested in giving the bound $O(\log n)$ for the height of a top tree.

# B Computing a core with unlimited length in a tree with node weights

In this section we show how to compute a core with unlimited length in a tree with weights on the nodes. The core of such a core is $\min_{v,w\in\mathcal{V}(T)} \sum_{z\in\mathcal{V}(T)} weight(z) dist(z,P)$, where $P = v\cdots w$.

First we change the functions from Lemma 1 to take the weights on the nodes into account.

$$\begin{aligned}
SizeDown(v) &= \sum_{v\in\mathcal{V}(T_v)} weight(v) \\
SizeUp(v) &= \sum_{v\in(\mathcal{V}(T)\setminus\mathcal{V}(T_v))} weight(v) \\
SumDown(v) &= \sum_{w\in\mathcal{V}(T_v)} weight(w)|w\cdots v| \\
SumDown^*(v) &= \sum_{w\in\mathcal{V}(T_v)} weight(w)|w\cdots parent(v)| \\
SumUp(v) &= \sum_{w\in\mathcal{V}(T)\setminus\mathcal{V}(T_v)} weight(w)|w\cdots v|
\end{aligned}$$

These functions can still be computed bottom-up in linear time.

Next we compute $DownCost^*(v) = DownCost^*(v,\infty)$ bottom up in the tree. For a leaf this value is 0. Let $Save(w) = DownCost^*(w) - SumDown^*(w)$. For an internal node $v$, $DownCost^*(v) = \min_{w\in Children(v)} SumDown(v) - Save(w)$, which can be computed in a time linear to the number of children, which in total is $n$. Finally, for each node, we determine which of its two children $x, y$ has the largest $Save$ value, and compute $MinCost(v) = SumUp(v) + SumDown(v) - Save(x) - Save(y)$, and return the $MinCost$ of the node minimizing this value over all nodes in the tree.

# C Applications of top trees techniques

In [22] they study the length-constrained heaviest path problem in a tree. Let $w$ and $l$ be two edge functions mapping each edge to a real number. Further for a path $P$ we define $l(P) = \sum_{e\in\mathcal{E}(P)} l(e)$ and $w(P) = \sum_{e\in\mathcal{E}(P)} w(e)$. Then the length-constrained heaviest path problem is to find a path $P$ such that $w(P) = \max_{u,v\in\mathcal{V}(T)}\{w(u\cdots v) | l(u\cdots v) \le F\}$, for a given $F$. In [22] an $O(n(\log n)^2)$ time algorithm is given to solve the problem. In [22] they also consider the special case where the $l$

function map edges to integers in range $1 \cdots O(n)$. In this special case they give an algorithm with complexity $O(n \log n)$.

Using the top tree techniques we get an $O(n \log n)$ algorithm for the general case. For the special case, we use the leaf trees techniques achieving a linear time algorithm. Hence, in both cases we improve [22] with a factor of $O(\log n)$. Both improvements are straightforward using the techniques given in this paper, here we give the details for the general case.

For each cluster $C$ in the top tree we will compute:

$$ClusterCust(C) = \max_{u,v \in \mathcal{V}(C)} \{w(u \cdots v) | l(u \cdots v) \leq F\}.$$

In order to do this we use following auxiliary functions:

$$WPath(C) = w(\pi(C))$$

$$LPath(C) = l(\pi(C))$$

$$BoundaryCost(v, C, k) = \max_{u \in \mathcal{V}(C)} \{w(u \cdots v) | l(u \cdots v) \leq k\}, \text{ for a boundary node } v \text{ in } C.$$

For a specific cluster $C$ the first two auxiliary functions are constant values, and the last one is represented as a picewise linear function. The space cost of $BoundaryCost(v, C, k)$ is clearly linear in $C$.

We will compute the information bottom up in the top tree as in the previous sections, however since we are dealing with a simple problem, we do not need to compute $SizeUp$, $SizeDown$, etc.

**Leaf Clusters :** For a leaf cluster $e$ we have $WPath(e) = w(e)$, $LPath(e) = l(e)$, and $BoundaryCost(v, e, k) = w(e)$, for $k \geq l(e)$; otherwhise these values are 0. Finally, for an edge $e = (v, w)$, $ClusterCost((v, w)) = BoundaryCost(v, e, F)$. Hence, a leaf clusters can be initialized in constant time.

**Merged Clusters :** Next we consider two clusters, $A$ and $B$, intersecting in $v$, to be merged to a cluster $C$.

$$ClusterCost(C) = \max \begin{cases} ClusterCost(A) \\ ClusterCost(B) \\ \max_{k \leq F} BoundaryCost(v, A, k) + BoundaryCost(v, B, F - k)\}. \end{cases}$$

If $C$ hase only one boundary node $WPath(C) = LPath(C) = 0$, otherwhise $WPath(C) = WPath(A) + WPath(B)$ and $LPath(C) = LPath(A) + LPath(B)$.

Recall, $A$ and $B$ intersect in $v$. If $a \neq v$ is a boundary node for $C$ and $A$, as in Figure 3 (1), (2) and (3), then

$$BoundaryCost(a, C, k) = \begin{cases} BoundaryCost(a, A, k), \text{ for } k \leq LPath(A) \\ \max \begin{cases} BoundaryCost(a, A, k) \\ WPath(A) + BoundaryCost(v, B, k - LPath(A))\}, \text{ for } k > LPath(A) \end{cases} \end{cases}$$

If $v$ is a boundary node for $C$, as in Figure 3 (2) and (4), then

$$BoundaryCost(v, C, k) = \max \begin{cases} BoundaryCost(v, A, k) \\ BoundaryCost(v, B, k) \end{cases}$$

Since the space cost of $BoundaryCost(v, C, k)$ is linear in $C$, we can compute its information in $|C|$ time.

Now, using Lemma 10 we get the required result.

In the paper [22] a series of problems are proven to have the same complexity as the length-constrained heaviest path problem. Using our solutions we have improved these problems too.

Final remark : In the above computation we had two measures for edges. The core problem could also be generalized to have two measures for edges. One measure to limit the length of the core, and one measure to the distance between nodes. If this should be required our core solution can easily be generalized without increasing its complexity.