ELSEVIER

Information Processing Letters 69 (1999) 63–67

Information Processing Letters

An efficient algorithm for the length-constrained heaviest path problem on a tree

Bang Ye Wu [a,1], Kun-Mao Chao [b,2], Chuan Yi Tang [a,*]

[a] *Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan*

[b] *Department of Computer Science and Information Management, Providence University, Shalu, Taiwan*

Received 10 June 1998; received in revised form 13 November 1998

Communicated by F.Y.L. Chin

Abstract

Given a tree with weight and length on each edge, this paper presents an efficient algorithm for locating the length-constrained heaviest path on the tree. The time complexity of the algorithm is $O(n \log^2 n)$ and can be reduced to $O(n \log n)$ if the edge lengths are all integers in the range 1 to $O(n)$, where n is the number of vertices. It is also shown that several similar problems can be solved by the same algorithm.

Keywords: Algorithms; Network design; Trees; Divide and conquer

1. Introduction

Consider the following network design problem. Given a tree network with length and weight on each edge, we want to upgrade the network by replacing a path with high speed edges. The length of an edge may represent the building cost and the weight may represent the profit (maybe the traffic load). We are also given a budget constraint which limits the length of the path to be upgraded. Thus, our goal is to find a length-constrained path, and we hope the weight (profit) of the path is as large as possible.

We call such a problem the *length-constrained heaviest path* problem on a tree network. Let n denote the number of vertices of the input tree. Since there are $O(n^2)$ paths on the tree, a direct method is to evaluate all the paths and find the optimal solution. In this paper, we present an efficient algorithm for the problem. Our algorithm uses the *divide-and-conquer* strategy and runs in $O(n \log^2 n)$ time. Furthermore, if the edge lengths are all integers in the range 1 to $O(n)$, the time complexity can be reduced to $O(n \log n)$.

Our algorithm is a recursive one. It first roots the tree at its *centroid* (defined later), finds the best path containing the centroid, and then find the best path within each subtree by recursively calling the algorithm. Such a technique had been used to solve some other path problems on a tree. In [4], the technique was used to find the kth longest path in a tree, and in [5], it was used to find the core of a tree with specified length.

The length-constrained heaviest path problem can have several variants. For example, when the profit is the most important factor, we may want to find the path whose weight must exceed a given lower bound and in addition the path length is as short as possible.

* Corresponding author. Email: cytang@cs.nthu.edu.tw.

[1] Email: dr838305@cs.nthu.edu.tw.

[2] Email: kmchao@csim.pu.edu.tw.

0020-0190/99/$ – see front matter
PII: S0020-0190(98)00194-X

In the above situation, the edge weights and lengths are all nonnegative. However, our algorithm works for the more general case in which the weights and the lengths can be any real numbers. With such a property, we shall show that our algorithm can solve the following four problems with the same time complexity. Given a tree with edge length and edge weight, find a path on the tree such that the path weight is maximum (or minimum) subject to that the edge length is no more than (or no less than) a given value.

The organization of the paper is as follows. In Section 2, we define some notations and describe some basic operations which will be used in the algorithm. In Section 3, we present the algorithm. In Section 4, we give concluding remarks.

2. Preliminaries

In this paper, a tree $T = (V, E)$ is an undirected tree with vertex set V and edge set E. Let w and l be two edge functions mapping each edge to a real number (not necessarily nonnegative). We call w the edge weight and l the edge length. For any graph G, $V(G)$ denotes its vertex set, and $E(G)$ denotes the edge set. Let u and v be two vertices of a tree T. We use $P = path(u, v)$ to denote the unique simple path between u and v on T. We assume that any path contains at least one edge. Define

$$l(P) = \sum_{e \in E(P)} l(e)$$

be the path length and

$$w(P) = \sum_{e \in E(P)} w(e)$$

be the path weight of P. Our problem is formally defined as follows:

Definition 1. Given a tree $T = (V, E)$, an edge weight function w, an edge length function l, and a real number B, the *length-constrained heaviest path* (LCHP) problem is to find a path P such that

$$w(P) = \max_{u,v \in V} \{w(path(u, v)) \mid l(path(u, v)) \leqslant B\}.$$

We use $Hpath(T, l, w, B)$ to denote the weight of the optimal path.

For a rooted tree T, the subtree rooted at vertex i is denoted by T_i. For any tree T, the centroid of T is a vertex $m \in V(T)$ such that if we delete m and the edges connected to m, each resulting subtree will contain no more than $|V(T)|/2$ vertices. The centroid always exists for any tree. Linear time algorithms for locating the centroid appeared in [2,3]. A basic algorithm can be briefly described as follows. Root T at any vertex, and visit the vertices in the postorder sequence. When visit a vertex i, compute $|V(T_i)|$ using the following recurrence relation: $|V(T_i)| = 1$ if i is a leaf, and $|V(T_i)| = \sum_{j \in child(i)} |V(T_j)| + 1$ otherwise, in which $child(i)$ is the vertex set of the children of i. In the traversal, the first vertex i with $|V(T_i)| \geqslant |V(T)|/2$ is the centroid of T.

Another basic operation used in the algorithm is to find $l(path(m, i))$ for every vertex i and a specified vertex m. The operation can be easily done in linear time as follows. Root T at m, and visit the vertices in a preorder sequence. When we visit vertex i, we compute

$$l\big(path(i, m)\big) = l\big(path\big(parent(i), m\big)\big) + l\big(i, parent(i)\big),$$

in which $parent(i)$ denotes the parent of i.

3. The algorithm

For the sake of simplicity, our algorithm is written only to return the weight of the optimal path. It is easy to modify the algorithm such that it can also return the optimal path. The algorithm is based on the divide-and-conquer strategy. If we root T at any vertex v, an optimal path either contains v or is completely contained within one of the subtrees rooted at the children of v. The algorithm finds the best path containing v and the solutions in subtrees are found by recursively calling the algorithm. For the sake of efficiency, the algorithm roots the input tree at its centroid m.

To find the best path containing the root, for each vertex i, we find the best path starting at i and passing through the root m. We first compute the length and weight of $path(m, i)$ for each vertex i. Then, for each $i \in V(T)$, we find another vertex j such that $w(path(m, j))$ is maximum subject to that j is not in the same subtree as i and $l(path(m, j)) \leqslant$

$B - l(path(m, i))$, where B is the length constraint. A direct method is to store the paths separately: one array for one subtree. However, this method leads to a higher time complexity when the degree of m is large.

Our algorithm uses a trick to overcome this difficulty. Such a technique has been used in [6]. For each possible length x, we find two vertices u and v such that

$$w(path(m,u)) = \max_{i \in V(T)} \{w(path(m,i)) \mid l(path(m,i)) \leqslant x\}$$

and

$$w(path(m,v)) = \max_{i \in V(T) \setminus V(Y)} \{w(path(m,i)) \mid l(path(m,i)) \leqslant x\},$$

where Y is the subtree containing vertex u. For a vertex i, if x is the possible longest length such that $l(path(m,i)) + x \leqslant B$, then the best path starting at i and passing through m can be quickly determined as follows: If i is not in the subtree Y, then the best path is $path(i,m) \cup path(m,u)$. Otherwise, the best path will be $path(i,m) \cup path(m,v)$.

The algorithm is listed below. The edge weight w, edge length l, and length constraint B will not be changed, and can be treated as global variables.

Algorithm LCHP

Input: A tree $T = (V, E)$.
/* Assume $V = \{1..n\}$ */
Output: $Hpath(T, l, w, B)$.
Step 0: **if** T contains no edge, **return** $-\infty$.
Step 1: Root T at its centroid m.
Assume $\{s_i \mid 1 \leqslant i \leqslant k\}$
be the children of m.
Step 2: /* Find the best solution containing m. */
/* X is an array of records and each record $X[v]$ is for one path $path(m,v)$.
$X[v].length = l(path(m,v))$,
$X[v].weight = w(path(m,v))$, and
$X[v].subtree = s_j$ if $v \in V(T_{s_j})$. */
Step 2.1: For each $v \in V$, compute $l(path(m,v))$, $w(path(m,v))$, and the subtree containing v. Store the results in array X.
Step 2.2: /* Find the best path with one endpoint at m. */
/* $hweight$ is used to store the best solution found so far. */
$hweight = \max\{X[i].weight \mid X[i].length \leqslant B,\ 1 \leqslant i \leqslant n\}$;
Step 2.3: Sort and relabel the vertices such that $X[i].length \leqslant X[i+1].length$.
Step 2.4: For each $1 \leqslant i \leqslant n$,
$pm_1[i] = X[u].weight$;
$f_1[i] = X[u].subtree$;
$pm_2[i] = X[v].weight$;
$f_2[i] = X[v].subtree$;
where
$X[u].weight =$
$\max\{X[j].weight \mid j \leqslant i\}$, and
$X[v].weight =$
$\max\{X[j].weight \mid j \leqslant i$ and $X[j].subtree \neq X[u].subtree\}$.
If v does not exist, then $pm_2[1] = -\infty$ and $f_2[1] = -1$;
Step 2.5: /* Scan the array to find the best pair of paths. */
$j = n$;
for $i = 1$ **to** n **do**
while $(X[j].length > B - X[i].length)$
decrease j by 1;
endwhile
if $j < 1$ **then** goto Step 3;
if $f_1[j] \neq X[i].subtree$ **then**
$hweight =$
$\max\{hweight, X[i].weight + pm_1[j]\}$
else
$hweight =$
$\max\{hweight, X[i].weight + pm_2[j]\}$
endif
endfor
Step 3: $hweight =$
$\max\{hweight, \mathrm{LCHT}(T_{s_i}) \mid 1 \leqslant i \leqslant k\}$
Step 4: **return** $hweight$.

Before showing the correctness of the algorithm, we give an example to demonstrate how the algorithm finds the best path containing the centroid. The input tree is shown in Fig. 1. The algorithm computes the following Table 1. (The vertices are already sorted by their lengths to the root.)

Step 2.5 runs as follows (only iterations for $i = 1, 2$ and 4 are listed):

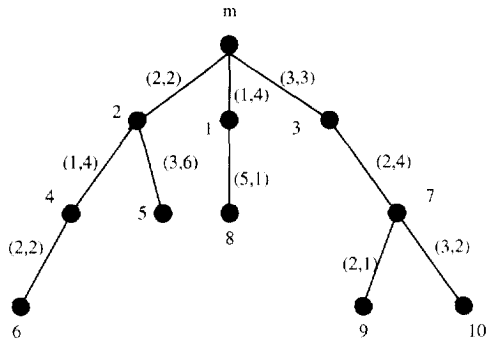

Fig. 1. The root is the centroid m of the tree, the numbers on nodes are their labels, and the mark on each edge represents (length, weight). The length constraint is 8 and the optimal path is from 4 to 7.

Table 1

v	1	2	3	4	5	6	7	8	9	10
$X[v].length$	1	2	3	3	5	5	5	6	7	8
$X[v].weight$	4	2	3	6	8	8	7	5	8	9
$X[v].subtree$	1	2	3	2	2	2	3	1	3	3
$pm_1[v]$	4	4	4	6	8	8	8	8	8	9
$f_1[v]$	1	1	1	2	2	2	2	2	2	3
$pm_2[v]$	$-\infty$	2	3	4	4	4	7	7	8	8
$f_2[v]$	-1	2	3	1	1	1	3	3	3	2

(1) $i = 1$: Since $X[1].length + X[9].length = 8$, $j = 9$. Since $f_1[9] = 2 \neq X[1].subtree = 4$, $hweight = X[1].weight + pm_1[9] = 12$. This solution is the path from node 1 to 5 (or 6).
(2) $i = 2$: $j = 8$. Since $f_1[8] = 2 = X[2].subtree$ (it means that node 6 is in the same subtree as node 2 and it is not a feasible solution), the best solution (starting from node 2 and passing the root) is $X[2].weight + pm_2[8] = 9$ and $hweight = \max\{12, 9\} = 12$.
(3) $i = 4$: $j = 7$. Since $f_1[7] = 2 = X[4].subtree$, the possible best solution is $X[4].weight + pm_2[7] = 13$ and $hweight = \max\{12, 13\} = 13$. It corresponds to the path from node 4 to 7.

The next lemma shows the correctness of the algorithm.

Lemma 1. *Algorithm* LCHP *correctly finds the* LCHP(T, l, w, B).

Proof. Since any path either contains or does not contain the centroid m and our algorithm is a recursive one, we only need to show that Step 2 finds the best path containing m. Let $path(u, v)$ be the path that we want to find. If $u = m$ or $v = m$, it is checked in Step 2.2. Otherwise, u and v are in two different subtrees and $path(u, v) = path(u, m) \cup path(m, v)$. For each possible length $X[i].length$, the algorithm finds the heaviest path $path(m, p)$ whose length is no more than $X[i].length$, and stores $w(path(m, p))$ in $pm_1[i]$. The algorithm also finds another path $path(m, q)$ which is the heaviest path with the same length constraint and q is not in the same subtree as p. The weight of $path(m, q)$ is stored at $pm_2[i]$. At Step 2.5, for each vertex i, the algorithm finds the best feasible path starting at i and containing m. To obtain the other endpoint of the desired path, it first finds the possible longest length $X[j].length$. Since $pm_1[j]$ records the heaviest path with length no more than $X[j].length$, the best solution is $(X[i].weight + pm_1[j])$ if the two endpoints are in two different subtrees. Otherwise, the best solution is $(X[i].weight + pm_2[j])$. Therefore, we conclude that the algorithm finds the solution correctly. □

We are now ready to present the main result of the paper.

Theorem 2. *Algorithm* LCHP *finds* $Hpath(T, l, w, B)$ *in* $O(n \log^2 n)$ *time.*

Proof. The correctness of the algorithm is shown in Lemma 1. We shall now analyze the time complexity. It is trivial to see that Steps 1, 2.1 and 2.2 take $O(n)$ time. Step 2.3 takes $O(n \log n)$ time for sorting. For Step 2.4, we set $pm_1[1] = X[1].weight$ and $pm_2[1] = -\infty$ initially, and for $i \geqslant 2$, $pm_1[i]$ and $pm_2[i]$ can be found among $\{pm_1[i-1], pm_2[i-1], X[i].weight\}$ by comparing their values and the subtrees they come from. Thus, Step 2.4 can be done in $O(n)$ time. For Step 2.5, since the variable j is never increased, it also takes $O(n)$ time. Let $Q(n)$ denote the time complexity in which n is the number of vertices of the input tree. We have $Q(1) = O(1)$ and $Q(n) = \sum_{i \in child(m)} Q(|V(T_i)|) + O(n \log n)$ for $n > 1$, where m is the centroid of input tree. From the property of centroid, $|V(T_i)| \leqslant n/2$ and $\sum_{i \in child(m)} |V(T_i)| =$

$n-1$. Solve the recurrence relation and we have $Q(n) = O(n \log^2 n)$. □

In the above proof, we can see that the time complexity is dominated by sorting. All steps except the sorting can be done in $O(n)$ time. If the edge lengths are all integers in the range 1 to $O(n)$, then Step 2.3 can be done in $O(n)$ time by an integer-sorting algorithm, e.g., *counting sort* [1].

Corollary 3. *The* LCHP *problem can be solved in* $O(n \log n)$ *time if the edge lengths are all integers in the range* 1 *to* $O(n)$, *where n is the number of vertices.*

Proof. Since the sorting procedure can be done in $O(n)$ time in such a case, we have $Q(1) = O(1)$ and $Q(n) = \sum_{i \in child(m)} Q(|V(T_i)|) + O(n)$ for $n > 1$, where m is the centroid of the input tree. From the property of centroid, $|V(T_i)| \leqslant n/2$ and

$$\sum_{i \in child(m)} |V(T_i)| = n - 1.$$

Solve the recurrence relation and we have $Q(n) = O(n \log n)$. □

Corollary 4. *Given a tree T, an edge weight function w, an edge length function l, and a real number B, the following problems can be also solved in* $O(n \log^2 n)$ *time, where n is the number of vertices in T. Furthermore, if the edge lengths are all integers in the range* 1 *to* $O(n)$, *the problems can be solved in* $O(n \log n)$ *time.*

(1) *Find a path P such that* $l(P) \geqslant B$ *and* $w(P)$ *is maximum.*
(2) *Find a path P such that* $l(P) \leqslant B$ *and* $w(P)$ *is minimum.*
(3) *Find a path P such that* $l(P) \geqslant B$ *and* $w(P)$ *is minimum.*

Proof. We shall show that all the three problems can be solved by algorithm LCHP by an input transformation. Let T, l, w, B be the input of the problems. For the first problem, we first construct an edge function l^- in which $l^-(e) = -l(e)$ for each $e \in E(T)$. Then, find $Hpath(T, l^-, w, -B)$ by algorithm LCHP. It is easy to see that it is the optimal solution of the first problem. Similarly, the solution of the second and the third problem are $Hpath(T, l, w^-, B)$ and $Hpath(T, l^-, w^-, -B)$, respectively, where $w^-(e) = -w(e)$ for each $e \in E(T)$. □

4. Concluding remarks

In this paper, we present an efficient algorithm for the length-constrained heaviest path problems on a tree. An interesting generalization of the problem is to find the heaviest subtree subject to a length constraint. However, even when the input tree is a star, the problem can be easily shown to be NP-hard by observing that the knapsack problem can be reduced to this problem. It is interesting to find a good approximation algorithm for the problem.

References

[1] T.H. Coremen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA, 1994.
[2] A.J. Goldman, Optimal center location in simple networks, Transportation Sci. 5 (1971) 212–221.
[3] O. Kariv, S.L. Hakimi, An algorithmic approach to network location problems. I: The p-centers, SIAM J. Appl. Math. 37 (1979) 513–538.
[4] N. Megiddo, A. Tamir, E. Zemel, R. Chandrasekaran, An $O(n \log^2 n)$ algorithm for the kth longest path in a tree with applications to location problems, SIAM J. Comput. 10 (2) (1981) 328–337.
[5] S. Peng, W.-T. Lo, Efficient algorithms for finding a core of a tree with a specified length, J. Algorithms 20 (1996) 445–458.
[6] B.Y. Wu, C.Y. Tang, An $O(n)$ algorithm for finding an optimal position with relative distances in an evolutionary tree, Inform. Process. Lett. 63 (1997) 263–269.