Information Processing Letters 109 (2008) 161–164

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

An improved algorithm for finding a length-constrained maximum-density subtree in a tree

Hsin-Hao Su [a], Chin Lung Lu [b,c], Chuan Yi Tang [a,*]

[a] Department of Computer Science, National Tsing Hua University, Hsinchu 300, Taiwan
[b] Institute of Bioinformatics, National Chiao Tung University, Hsinchu 300, Taiwan
[c] Department of Biological Science and Technology, National Chiao Tung University, Hsinchu 300, Taiwan

ARTICLE INFO

Article history:
Received 21 April 2008
Received in revised form 8 August 2008
Available online 30 September 2008
Communicated by F.Y.L. Chin

Keywords:
Algorithms
Dynamic programming
Trees
Network design
Divide and conquer

ABSTRACT

Given a tree T with weight and length on each edge, as well as a lower bound L and an upper bound U, the so-called length-constrained maximum-density subtree problem is to find a maximum-density subtree in T such that the length of this subtree is between L and U. In this study, we present an algorithm that runs in $O(nU \log n)$ time for the case when the edge lengths are positive integers, where n is the number of nodes in T, which is an improvement over the previous algorithms when $U = \Omega(\log n)$. In addition, we show that the time complexity of our algorithm can be reduced to $O(nL \log \frac{n}{L})$, when the edge lengths being considered are uniform.

1. Introduction

Given a tree $T = (V, E)$, let $w(e)$ be a positive real function for representing the weight of an edge $e \in E$ and $l(e)$ be a positive integer function for representing the length of e. For a path $(e_1, e_2, \ldots, e_k)$ in T, its *density* is defined as $\sum_{i=1}^{k} w(e_i) / \sum_{i=1}^{k} l(e_i)$. Suppose now that we are given a tree T, as well as two positive integers L and U with $L \leqslant U$. Then the so-called *length-constrained maximum-density path* (LDP) problem is to find a maximum-density path in T such that the length of this path is between the lower bound L and the upper bound U. The LDP problem [8,10], as well as its special case in which the given T is a path [1,3,6,7,9], has applications for studying alignment and GC-content of genomic sequences in computational biology. Lin et al. were the first to study this problem but with a restriction that all the considered edges are all equal in length and also proposed two $O(nL)$-time algorithms to solve it [10], where n denotes the number of nodes in the given tree. Later, Lau et al. designed an $O(n \log^2 n)$-time algorithm for the general case of this problem with allowing the edge lengths to be any positive real numbers and further reduced the time complexity of this algorithm to $O(n \log n)$ when the lengths of edges are uniform (i.e., $l(e) = 1$ for each $e \in E$) [8]. More recently, by giving an extra parameter K that is a positive integer, Hsieh and Cheng [4] have studied an variant of the LDP problem that is to find a maximum-density path in T such that the length of this path is between L and U and the number of its edges is at least K. In this study, they proposed an $O(nKU)$-time algorithm to solve this problem.

Lau et al. [8] studied the *length-constrained maximum-density subtree* (LDT) problem, a generalization of the LDP problem, which is to find a maximum-density subtree, rather than path, in the given tree $T = (V, E)$ with the length between L and U, where the density of a subtree $T' = (V', E')$ in T is similarly defined as $\sum_{e \in E'} w(e) / \sum_{e \in E'} l(e)$. Actually, as mentioned in [8], the LDT problem has its applications in computer, traffic or logistics network design. In [5], Hsieh and Chou studied a variant of the LDT problem in which the weight and length

* Corresponding author.
E-mail address: cytang@cs.nthu.edu.tw (C.Y. Tang).

0020-0190/$ – see front matter
doi:10.1016/j.ipl.2008.09.027

functions were defined on nodes rather than edges. However, their algorithms can still be applied to solve the LDT problem we considered in this study. In fact, as pointed out in [11], the LDT problem with general integer edge lengths can be shown to be NP-hard by a simple reduction from the knapsack problem [2]. Currently, to the best of our knowledge, the LDT problem can be solved in $O(nU^2)$ time using the algorithm proposed in [5,8] when the edge lengths are positive integers and, in addition, it can be solved in $O(nL^2)$ time using the algorithm designed in [8] when the edge lengths are uniform. It should be noted here that neither of algorithms in [5,8] for solving the LDT problem with any integer edge lengths are polynomial but pseudo-polynomial. Basically, these two algorithms utilized the same idea that is first to transform the given tree into a rooted binary tree and then use the dynamic programming approach to find all maximum-weight subtrees with all possible lengths on the transformed binary tree. Further note that the LDT problem with uniform edge lengths can be simply considered as the problem of finding a *size-constrained maximum-density subtree* (SDT) in a given tree [10]. In [8], Lau et al. have shown that finding an SDT in a general graph is still NP-hard. However, as mentioned above, this problem can be solved in polynomial time using the algorithm proposed in [8] when the given graph is just a tree, where notably the time-complexity of this algorithm is $O(nL^2)$ that is polynomial because $L \leqslant n$.

In this study, we propose an improved algorithm on the basis of a combination of divide-and-conquer and dynamic programming to solve the LDT problem. If the edge lengths are positive integers, then our algorithm can solve the LDT problem in $O(nU \log n)$ time, which is an improvement over the previous algorithms when $U = \Omega(\log n)$. If the edge lengths are uniform, then the time-complexity of our algorithm can be further reduced to $O(nL \log \frac{n}{L})$. Finally, we also show that the maximum-weight subtrees of all sizes can be computed in $O(n^2)$ time.

The rest of this paper is organized as follows. Section 2 presents the main idea of the algorithm we used to solve the LDT problem with positive integer lengths. Section 3 shows its further improvement in the case when all edge lengths are uniform. Section 4 concludes the study with some remarks.

2. Algorithm for solving the LDT problem in a tree

In the following, we propose a method to improve the algorithms that were designed in [5] and [8] based on the dynamic programming approach for solving the LDT problem. The basic steps of these two algorithms are as follows. First, the algorithms transform the given tree into a rooted binary tree. Next, for each node x in this binary tree, they allocate a table of size U in which the ith entry, where $1 \leqslant i \leqslant U$, represents the weight of a maximum-density subtree whose root is at x and whose length is restricted as i. Suppose that y and z are the two children of x. Then a subtree rooted at x of length i can be constructed by joining a subtree of length j rooted at y and a subtree of length $i - l(x, y) - l(x, z) - j$ rooted at z together with edges (x, y) and (x, z). Based on this property, we can see that there are $O(U)$ possible choices for the computation of each entry in the table associated with x and therefore the total computation of the table takes $O(U^2)$ time.

The basic idea we used in our dynamic programming algorithm is as follows. We first transform the input tree into a rooted tree by choosing a node r as the root and then we compute a maximum-density subtree containing r. Note that by removing r and all edges incident with it, we can yield several subtrees, say $T_1, T_2, \ldots, T_{\deg(r)}$. If the optimal solution does not contain r, then it must be a subtree of T_i, where $1 \leqslant i \leqslant \deg(r)$. Actually, this idea was also used in [8,11] to find a length-constrained maximum-density path or a length-constrained heaviest path in a tree.

Based on the above idea, we are dedicated to design a dynamic programming algorithm, as described below, to find a length-constrained maximum-density subtree containing the root r. For each node x in the rooted tree T, we use $A_x[i]$ to store the weight of a maximum-weight subtree of length i that contains r and x, where A_x is a table of size U. Initially, we set $A_r[0] = 0$ and $A_r[i] = -\infty$ for $i = 1, 2, \ldots, U$ and $A_x[i] = -\infty$ for $i = 0, 1, 2, \ldots, U$ for every non-root node x. Then we traverse the tree T rooted at r in a depth-first search manner. In the traversing process, there are two different directions to visit y: (1) descending direction from a parent node x to y and (2) ascending direction from a child node x to y. Depending on the traversing direction, we then assign a value to $A_y[i]$, as described as follows, when we visit node y. If the direction we visit y is descending, then we let $A_y[i] = A_x[i - l(x, y)] + w(x, y)$ for $i = l(x, y), l(x, y) + 1, \ldots, U$. If the direction we visit y is ascending, then we let $A_y[i] = \max\{A_x[i], A_y[i]\}$ for $i = 0, 1, \ldots, U$.

Actually, after the traversal is finished, we can prove that $A_r[i]$ is the weight of the maximum-weight subtree of length i containing r according to Lemma 1, as described below. Based on this property, therefore, the density of a length-constrained maximum-density subtree containing r is $\max_{L \leqslant i \leqslant U} A_r[i]/i$. Note that during the traversing process, we record the sequence of the nodes we have visited and denote it by $(r, \ldots, x)$, where x is the currently visiting node. Clearly, the graph induced by these nodes is a subtree of T and for convenience we denote it by $T(r, \ldots, x)$.

Lemma 1. *When we currently visit x, let* $(r, \ldots, x)$ *be the sequence of the nodes we have traversed. Then* $A_x[i]$ *stores the weight of the maximum-weight subtree in* $T(r, \ldots, x)$ *of length i that contains both r and x until new* $A_x[i]$ *is computed.*

Proof. We prove this lemma by induction on the sequence of the visited nodes. Initially, $T(r)$ is a subtree that contains an isolated node r and clearly the lemma holds. Next, we assume that the lemma holds for $(r, \ldots, x)$ and let y be the next node we are going to visit. Then there are two cases to be considered.

(1) Suppose that y is a child node of x. Then y is a leaf in the induced subtree $T(r, \ldots, x, y)$. Therefore, if we remove y from any subtree of $T(r, \ldots, x, y)$ that contains r and y, the resulting tree must be a subtree of $T(r, \ldots, x)$ that contains r and x. According to the assumption, it is clear that $A_x[i - l(x, y)] + w(x, y)$, which equals to the

value of $A_y[i]$ based on our method described above, is the weight of a maximum-weight subtree in $T(r,\ldots,x,y)$ of length i that contains r and y.

(2) Suppose that y is a parent of x. Let P be a maximum-weight subtree of $T(r,\ldots,x,y)$ with length i containing r and y. Since y is the parent of x, our traversal sequence must be in the order of $(r,\ldots,y,x,\ldots,x,y)$. Let Q be the subtree that is rooted at x. If $P\cap Q=\emptyset$ then P is a subtree of $T(r,\ldots,y,x,\ldots,x,y)\backslash Q = T(r,\ldots,y)$ passing through r and y, indicating that the weight of P has already been stored in $A_y[i]$. If P contains a node in Q, then P must contain x. In this case, P is a maximum-weight subtree of $T(r,\ldots,y,x,\ldots,x)$ passing through r and x and its weight was stored in $A_x[i]$. According to our method, we will select the maximum one between $A_x[i]$ and old $A_y[i]$ and assign it to new $A_y[i]$. In other words, $A_y[i]$ keeps the weight of maximum-weight subtree of $T(r,\ldots,x,y)$. □

When the traversal is completed, $A_r[i]$ stores the weight of a maximum-weight subtree in $T(r,\ldots,r)=T$ of length i that contains r according to Lemma 1. Therefore, the density of a length-constrained maximum-density subtree containing r is $\max_{L\leqslant i\leqslant U} A_r[i]/i$.

Given a tree $T=(V,E)$, there exists a node c called centroid such that deleting c results in several subtrees each containing no more than $|V|/2$ nodes. We can find this node by rooting T at some node first. Then we perform a postorder traversal on T to count the total number of nodes below each node. The first node that has more than $|V|/2$ nodes below it is a centroid. This can be done in linear time.

Now, we describe our algorithm that proceeds as follows:

1. Find a centroid c of T.
2. Use the dynamic programming method described before to find a length-constrained maximum-density subtree that contains c.
3. Separate T into several subtrees, say $T_1, T_2, \ldots, T_{\deg(c)}$, by removing c and all edges incident with it from T, and recursively repeat steps 1 and 2 on these subtrees.
4. Compare $\deg(c)+1$ length-constrained maximum-density subtrees with containing the centroid we obtained in steps 2 and 3 and choose the one with the highest density as the output.

Below, we analyze the time complexity of our algorithm. Let $T(n)$ be the time complexity of the algorithm when the size of the input tree is n. Then step 1 can be done in $O(n)$, as described above. In step 2, we take $O(U)$ time to update the table associated with each node in the input tree. Thus the time complexity of step 2 is $O(nU)$. Clearly, according to our algorithm, $T(n)$ can be written as a recursive function as follows, where notably n_i denotes the size of T_i.

$$T(n)=\sum_{i=1}^{\deg(c)} T(n_i)+O(nU).$$

Since $n_i\leqslant n/2$, it is not hard to derive that $T(n)=O(nU\log n)$.

Theorem 1. *The LDT problem can be solved in $O(nU\log n)$ time.*

3. Algorithms for solving the SDT problem in a tree

We here define the size of a tree as the number of edges it contains and therefore the size-constrained maximum-density subtree problem is equivalent to the length-constrained maximum-density subtree problem with uniform edge lengths. As pointed out in [8], if the size is constrained between L and U, there exists a maximum-density subtree with a size less than $3L$. For completeness, we give a more detailed proof below.

Lemma 2. *A tree T of a size greater than or equal to $3L$ can be separated into two edge-disjoint subtrees at least of the size L.*

Proof. Let the tree T be rooted at some node r. Then we perform a postorder traversal on T to compute the size of subtree rooted at every node. Let x be the first node we traverse in postorder such that the subtree rooted at x has at least the size L. Let $x_1,x_2,\ldots,x_k$ be the children of x. Denote the subtree rooted at x_i by $T(x_i)$ and its size by $\text{size}(T(x_i))$. Let j be the smallest positive integer such that $\sum_{i=1}^{j}\text{size}(T(x_i))+j\geqslant L$. Such j must exist, since $\sum_{i=1}^{k}\text{size}(T(x_i))+k$ is the size of the subtree rooted at x, which is at least L. Let $P=\bigcup_{i=1}^{j}T(x_i)\cup(x,x_i)$. Then P is a subtree at least of the size L. But the size of P is no greater than $2L$. Suppose $\text{size}(P)>2L$. Then $\sum_{i=1}^{j-1}\text{size}(T(x_i))+j-1=\text{size}(P)-\text{size}(T(x_j))-1$. Recall that x is the first node whose induced subtree has a size at least L, implying $\text{size}(T(x_j))<L$. As a result, $\sum_{i=1}^{j-1}\text{size}(T(x_i))+j-1>L$, which again contradicts to the assumption that x is the first node whose induced subtree has a size greater than or equal to L. Therefore, all the edges which are in T but not in P induce another subtree at least of a size L. □

If the size of T exceeds $3L-1$, then T can be separated into two edge-disjoint subtrees, say T_1 and T_2, each with size of at least L, according to Lemma 2. Then it is clear that the higher density between T_1 and T_2 must be higher than or equal to the density of T. To find a size-constrained maximum-density subtree in a given tree, we actually can employ the same algorithm for finding a length-constrained maximum-density subtree described in the previous section. For each node v in the given tree, however, we only need to compute the table of A_v ranging from 1 to $3L-1$, rather than U, if U is greater than $3L-1$. In addition, we do not need to consider those maximum-density subtree whose size is less than L, since all these subtrees do not satisfy the size constraint required to be at least L. Therefore, we can reformulate the recursive function of the time complexity as follows, so that we can get a tighter time bound.

$$T(n)=\begin{cases}\sum_{i=1}^{\deg(c)}T(n_i)+O(nL), & n-1\geqslant L,\\ O(1), & \text{otherwise.}\end{cases}$$

Note that in the above recursive function, n_i denotes the size of the subtree induced by the ith child of the initial centroid c and the size of T is $n-1$.

Corollary 1. *A size-constrained maximum-density subtree in a tree can be find in $O(nL\log\frac{n}{L})$ time when $n-1 \geqslant L$.*

Proof. Suppose by induction that $T(m) < knL\lg\frac{m}{L}$ for all $m < n$, where $k > 0$ is a constant. Then we have $T(n) \leqslant kL\sum_{n_i-1\geqslant L} n_i \lg\frac{n_i}{L} + \sum_{n_i-1<L} e + dnL$, for some constants $d > 0$ and $e > 0$. It should be noted that if all the subtrees have a size less than L, then the first term on the right-hand side of the above inequality is zero. Actually, we can express the above inequality as $T(n) \leqslant kL\sum_i n_i \lg\frac{n_{\max}}{L} + enL + dnL$, where $n_{\max} = \max\{n_i \mid n_i - 1 \geqslant L\}$, and consequently $T(n) \leqslant knL\lg\frac{n_{\max}}{L} + (d+e)nL \leqslant knL\lg\frac{n}{2L} + (d+e)nL$. Here, we choose k such that $k > d+e$. If the first term is zero, then $T(n) < knL \leqslant knL\lg\frac{n}{L}$. If the first term exists, then $T(n) \leqslant knL\lg\frac{n}{L} + (d+e-k)nL < knL\lg\frac{n}{L}$. □

It is also possible to compute the maximum-weight subtrees of all sizes in $O(n^2)$ time. The algorithm is the same as the one we described above. But when applying the dynamic programming approach on each tree, we consider the table up to the size of the tree instead of $3L-1$. In this way, the maximum-weight subtree of the size i is the maximum of maximum-weight subtree of the size i in each tree that has a size greater than or equal to i. In this case, the recursive function for the time complexity of algorithm becomes as $T(n) = \sum_{i=1}^{\deg(c)} T(n_i) + O(n^2)$.

Corollary 2. *It takes $O(n^2)$ time to find the maximum-weight subtrees of all sizes.*

Proof. Clearly, $T(n) \leqslant \sum_{i=1}^{\deg(c)} T(n_i) + en^2$ for some constant $e > 0$. Suppose that $T(n) < dn^2$. Then

$$T(n) < \sum_{i=1}^{\deg(c)} dn_i{}^2 + en^2 < \sum_{i=1}^{\deg(c)} dn_i\frac{n}{2} + en^2 \leqslant \frac{dn^2}{2} + en^2 < dn^2,$$

if we choose $d > 2e$. Therefore, $T(n) = O(n^2)$. □

4. Concluding remarks

In this study, we have proposed an $O(nU\log n)$-time algorithm for solving the maximum-density subtree problem, which is better than the previous algorithms when bound $U = \Omega(\log n)$. In addition, we have shown that the time complexity of this algorithm can be reduced to $O(nL\log\frac{n}{L})$ when the edge lengths in the given tree are uniform. Actually, the idea behind the dynamic programming we designed in this study can be used to compute a length-constrained or size-constrained optimal subtree with other objective functions in a tree, such as length-constrained bottleneck subtree. As a future work, it would be interesting to know if the space complexity of the length-constrained maximum-density subtree problem can be reduced to $O(n+U)$.

Acknowledgements

This work was supported in part by National Science Council of Republic of China under grant NSC-97-2815-C-007-041-E.

References

[1] K.M. Chung, H.I. Lu, An optimal algorithm for the maximum-density segment problem, SIAM Journal on Computing 34 (2004) 373–387.
[2] M.R. Garey, D.S. Johnson, Computers and Intractability—A Guide to the Theory of NP-Completeness, Freeman, San Francisco, 1979.
[3] M.H. Goldwasser, M.Y. Kao, H.I. Lu, Linear-time algorithms for computing maximum-density sequence segments with bioinformatics applications, Journal of Computer and System Science 70 (2005) 128–144.
[4] S.Y. Hsieh, C.C. Cheng, Finding a maximum-density path in a tree under the weight and length constraints, Information Processing Letters 105 (2008) 202–205.
[5] S.Y. Hsieh, T.Y. Chou, Finding a weight-constrained maximum-density subtree in a tree, in: Proceedings of the 16th International Symposium on Algorithms and Computation (ISAAC 2005), in: Lecture Notes in Computer Science, vol. 3827, 2005, pp. 944–953.
[6] X. Huang, An algorithm for identifying regions of a DNA sequence that satisfy a content requirement, Computer Applications in the Biosciences 10 (1994) 219–225.
[7] S.K. Kim, Linear-time algorithm for finding a maximum-density segment of a sequence, Information Processing Letters 86 (2003) 339–342.
[8] H.C. Lau, T.H. Ngo, B.N. Nguyen, Finding a length-constrained maximum-sum or maximum-density subtree and its application to logistics, Discrete Optimization 3 (2006) 385–391.
[9] Y.L. Lin, T. Jiang, K.M. Chao, Efficient algorithms for locating the length-constrained heaviest segments, Journal of Computer and System Sciences 65 (2002) 570–586.
[10] R.R. Lin, W.H. Kuo, K.M. Chao, Finding a length-constrained maximum-density path in a tree, Journal of Combinatorial Optimization 9 (2005) 147–156.
[11] B.Y. Wu, K.M. Chao, C.Y. Tang, An efficient algorithm for the length-constrained heaviest path problem on a tree, Information Processing Letters 69 (1999) 63–67.