PAPER

Algorithm for the Length-Constrained Maximum-Density Path Problem in a Tree with Uniform Edge Lengths[**]

Sung Kwon KIM[†*a)], ***Member***

SUMMARY Given an edge-weighted tree with n vertices and a positive integer L, the length-constrained maximum-density path problem is to find a path of length at least L with maximum density in the tree. The density of a path is the sum of the weights of the edges in the path divided by the number of edges in the path. We present an $O(n)$ time algorithm for the problem. The previously known algorithms run in $O(nL)$ or $O(n \log n)$ time.

key words: *length-constrained paths, maximum-density paths, uniform edge lengths*

1. Introduction

Let T be an edge-weighted tree with n vertices. Each edge e in T is associated with a real number, called its *weight*, w_e. Let $\pi_{u,v}$ denote the path connecting two distinct vertices u, v in T. The *weight* of $\pi_{u,v}$ is the sum of the weights of the edges in it, $w(\pi_{u,v}) = \sum_{e\in\pi_{u,v}} w_e$, its *length* $l(\pi_{u,v})$ is the number of edges in it, and its *density* is $d(\pi_{u,v}) = w(\pi_{u,v})/l(\pi_{u,v})$.

The *length-constrained maximum-density path problem* is: Given an edge-weighted tree T and a positive integer L, find a maximum-density path of length at least L in T, namely, a path π such that $d(\pi) = \max\{d(\pi_{u,v}) \mid u, v \text{ are vertices in } T \;\&\; l(\pi_{u,v}) \geq L\}$.

For this problem, Lin, Kuo, and Chao [11] presented an $O(nL)$ time, dynamic programming algorithm, and Lau, Ngo, and Nguyen [10] gave an $O(n \log n)$ time, divide-and-conquer algorithm based on centroid decomposition. In this paper, we develop an $O(n)$ time algorithm.

In Sect. 2, we review some previous results that will be used in designing our algorithm. In Sect. 3, our algorithm for the case where T is a binary tree is first presented, and then we show how to tackle the case of general trees.

A similar approach was previously used to solve another problem by the author [9].

Manuscript received January 8, 2014.
Manuscript revised March 25, 2014.

[†]The author is with the School of Computer Science and Engineering, Chung-Ang University, Seoul, Korea.
[*]Part of this work was done while the author visited School of Information Science, JAIST, Nomi-shi, 923–1211 Japan.

[**]This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (No. 2012008222) and by the Chung-Ang University research grant in 2013.

a) E-mail: skkim@cau.ac.kr.

DOI: 10.1587/transinf.2014EDP7009

2. Preliminaries

The following lemma was first presented by Huang [7] and was previously used in [11], which enables us to consider only the paths of length at least L and at most $2L - 1$ in finding a length-constrained maximum-density path.

Lemma 1: If π is a length-constrained maximum-density path, then $l(\pi) \leq 2L - 1$.

Chung and Lu [2] presented a linear time algorithm for the following problem: Given an array A of n real numbers and two positive integers $L \leq U$, find a maximum-density segment $A[j..j']$ with $L \leq j' - j + 1 \leq U$. This corresponds to an array version of the length-constrained maximum-density path problem with both lower and upper length bounds[***]. They also gave a linear time algorithm for a "location-constrained" variant of the problem in which two integer intervals $[x, y]$ and $[x', y']$ with $y < x'$ are additionally given and it is required that $j \in [x, y]$ and $j' \in [x', y']$.

Let T be a binary tree with n vertices. A *component* of T is a connected subtree of T. Two components are *adjacent* if there is an edge of T such that one of its endpoints is in one component and the other is in the other. The *external degree* of a component is the number of its adjacent components.

Frederickson [5] developed a method that partitions a binary tree into a number of components by removing some of its edges and builds a hierarchical sequence of partitions based on incremental merging of the components. Eppstein [3], [4], and Italiano and Ramaswami [8] used the method in developing algorithms for dynamically maintaining trees and graphs. We borrow the definitions from [8] with slight modification.

For an integer $2 \leq z \leq n$, a *restricted partition of order* z of T is a partition of T into components such that

1. Each component has external degree at most three.
2. Each component of external degree three consists of a single vertex.
3. Each component of external degree one or two consists of at most z vertices.
4. No two adjacent components can be combined and still satisfy 1–3.

Figure 1 shows a restricted partition of order two of a

[***]They solved a more general version of the problem: the lengths are *nonuniform*, i.e., they are positive real numbers.

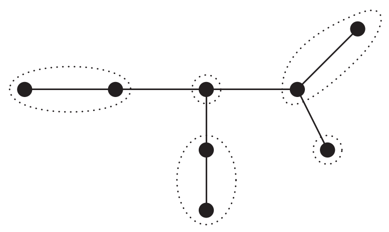

Fig. 1 A restricted partition of order two.

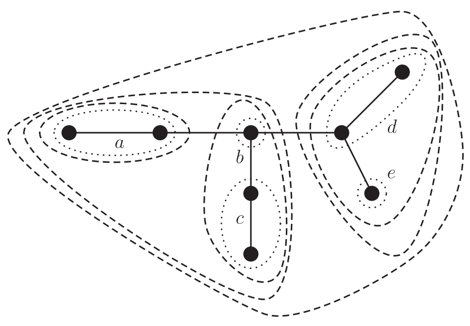

Fig. 2 A restricted multilevel partition of order two.

tree.

The following gives a hierarchical method of reconstructing T from the components by restoring the removed edges, which has certain nice properties (detailed later). A *restricted multilevel partition of order z* of T is a hierarchical sequence of partitions of T such that:

1. The components at level 1 (called *basic components*) are those of a restricted partition of order z of T.
2. Each component at level $i > 1$ is either a copy of a component at level $i-1$ or is formed by combining two adjacent components at level $i-1$ with the addition of the edge between them. This is done by finding a restricted partition of order *two* of the tree obtained after contracting the components at level $i-1$.
3. There is only one component, T, at the topmost level.

Figure 2 depicts a tree and its restricted multilevel partition of order two.

A rooted binary tree, called a *topology tree* for T, can naturally be formed from a restricted multilevel partition of T.

1. A node[†] at level i in the topology tree represents a component at level i in the restricted multilevel partition.
2. A node at level $i > 1$ has at most two children that represent the components at level $i-1$ whose union is the component represented by the node.

Fredrickson [5] proved that

1. a restricted partition of order z of T can be found in $O(n)$ time and the number of components in the partition is $O(n/z)$,
2. a topology tree has $O(n/z)$ leaf nodes and is of height

[†]Nodes are used in a topology tree to distinguish them from vertices in an input tree.

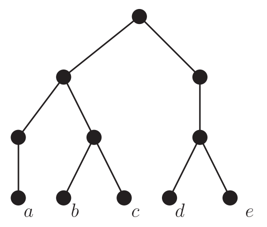

Fig. 3 The topology tree for the tree in Fig. 2.

$O(\log(n/z))$, and

3. a topology tree can be constructed in $O(n)$ time.

Figure 3 illustrates the topology tree for the tree in Fig. 2.

A solution for the following *path sum query* problem is employed in the design of our algorithm. Let T be a tree in which the edges are associated with real values. The *path sum* of the path $\pi_{u,v}$ for vertices u, v in T, denoted by $s(\pi_{u,v})$, is the sum of the values of the edges in $\pi_{u,v}$. In our application, $s(\pi_{u,v})$ is $l(\pi_{u,v})$ or $w(\pi_{u,v})$. The path sum query problem is: Preprocess T so that, after preprocessing, for any query pair of vertices u, v in T, the path sum $s(\pi_{u,v})$ can be answered quickly. A solution for the problem consists of two parts: preprocessing and query-answering.

The following solution is well-known, whose preprocessing works as follows:

1. Pick up a vertex t in T and convert T into a rooted tree with root t.
2. Compute $s(\pi_{t,v})$ for each vertex v in T.
3. Preprocess T so that, given any query pair of vertices u, v in T, their lowest common ancestor can be found quickly.

After the preprocessing, given any pair of vertices u and v, the query-answering finds their lowest common ancestor t' and computes $s(\pi_{u,v}) = s(\pi_{t,u}) + s(\pi_{t,v}) - 2 \cdot s(\pi_{t,t'})$. Since solutions with $O(n)$ preprocessing time and $O(1)$ query answering time for the lowest common ancestor problem are given by Harel and Tarjan [6], and Schieber and Vishkin [12], the path sum query problem can be solved within the same complexity.

3. Algorithm

Let T be an edge-weighted binary tree with n vertices for which we want to solve the length-constrained maximum-density path problem. Let L be a positive integer that corresponds to lower length bound.

Briefly, our algorithm works as follows: We first divide T into components by removing some of the edges of T, and separately solve the problem for each of the components. These components and their solutions are at level 1. To reach the topmost level where only one component T itself and its solution exist, we explain how to go from level $i-1$ to level i for $i > 1$. We select some distinct pairs of adjacent components out of those at level $i-1$. For each pair selected, we merge the two components by restoring

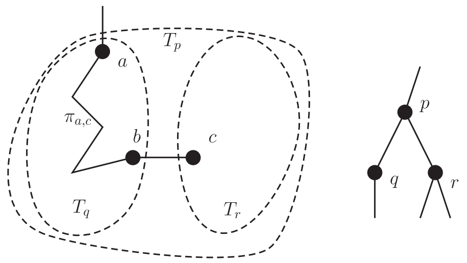

Fig. 4 The case of p having two children q and r.

```
computeW(p)
    $a$ = the connector of $T_p$;
    return recursion$(p, a)$;

recursion$(p, a)$
    if ($p$ is a leaf), then compute $W_p^a$ "directly" and return it;
    if ($p$ has a single child $q$), then return $W_q$;
    // $p$ has two children $q, r$ and assume that $a$ is in $T_q$. //
    compute $W_r^a$ as explained in the text;
    $W_q^a$ = recursion$(q, a)$;
    for ($i = 0$ to $2L - 2$)
        $W_p^a[i] = \max\{W_q^a[i], W_r^a[i]\}$;
    return $W_p^a$;
```

Fig. 5 Algorithm for computing W_p.

the edge between them and combine their solutions at level $i-1$ to get one at level i. The unselected components and their solutions at level $i-1$ are copied and stored at level i. We repeat this procedure until the topmost level reaches. Described later are details about how large the components at level 1 are, which pairs are selected from the components at level $i-1$, and how the two solutions at level $i-1$ are combined to get one at level i.

We first perform preprocessing of the path sum query on T with respect to both length and weight of edges so that, after preprocessing, $l(\pi_{u,v})$ and $w(\pi_{u,v})$ for any two vertices u, v in T can be found in $O(1)$ time. This preprocessing takes $O(n)$ time as mentioned in Sect. 2.

We next build in $O(n)$ time a topology tree T^* that corresponds to a restricted multilevel partition of order $z = L$ of T. As explained in Sect. 2, T^* is a rooted binary tree of height $O(\log(n/L))$ with $O(n/L)$ leaf nodes, and each node of T^* represents a component of T. Let T_p for a node p in T^* denote the component of T that p represents. If p is a leaf, then T_p is a basic component and has at most L vertices. If p has a single child q only, then $T_p = T_q$. If p has two children q and r, then $T_p = T_q \cup T_r \cup \{(b, c)\}$, where (b, c) is the edge between T_q and T_r, and b and c are vertices in T_q and T_r, respectively. See Fig. 4. In other words, two components T_q and T_r are combined to produce T_p by restoring the edge. We call b (respectively, c) the *connector* of T_q (respectively, T_r). If p has a single child q, then we define the connector of T_q to be equal to the connector of T_p. So, each T_p has one connector.

T^* requires $O(n)$ memory as proved in [5]. We augment T^* by storing the basic component T_p at each leaf node p, and the connector of T_p at each non-root node p. This augmented T^* clearly requires $O(n)$ memory.

For a node p in T^* and a vertex a in T, define W_p^a to be an array of size $2L-1$ such that, for $0 \le i \le 2L-2$, $W_p^a[i] = \max\{w(\pi_{a,v}) \mid v \text{ is a vertex in } T_p \ \&\ l(\pi_{a,v}) = i\}$. (Refer to Lemma 1.) $W_p^a[i]$ is the maximum of the weights of the paths of length i in T between a and the vertices of T_p, and $W_p^a[i] = -\infty$ if there is no such path.

If a belongs to T_p, then W_p^a can be computed in $O(\max\{|T_p|, L\})$ time by the following "direct" method. Set $W_p^a[i] = -\infty$ for all $0 \le i \le 2L-2$. Transform T_p into a rooted tree by naming a its root. Traverse T_p in preorder and for each currently visited vertex v, set $W_p^a[l(\pi_{a,v})] = \max\{w(\pi_{a,v}), W_p^a[l(\pi_{a,v})]\}$ if $l(\pi_{a,v}) \le 2L-2$. Note that $l(\pi_{a,v})$ and $w(\pi_{a,v})$ can be found in $O(1)$ time using the path sum query.

For notational simplicity, W_p is used to denote W_p^a if a is the connector of T_p. We compute W_p for the nodes p of T^*. If we use the "direct" method for every node p, it takes at least $O(n\log(n/L))$ time as T^* has $O(\log(n/L))$ levels and at each level $O(n)$ time is sufficient to apply the "direct" method to the nodes at the level. We show that this can be done in $O(n)$ time by presenting an algorithm that works in bottom-up fashion.

If p is a leaf, then T_p, which is a basic component, contains at most L vertices and thus W_p can be computed in $O(L)$ time by the "direct" method. If p has a single child q only, then $W_p = W_q$.

Consider the case where p has two children q and r. Both W_q and W_r have been computed and are available for reference. As mentioned before, $T_p = T_q \cup T_r \cup \{(b, c)\}$, where b and c are the connectors of T_q and T_r, respectively. Let a be the connector of T_p. Then, $W_p[i] = W_p^a[i] = \max\{W_q^a[i], W_r^a[i]\}$ for $0 \le i \le 2L-2$.

In order to explain how to compute W_q^a and W_r^a, assume, without loss of generality, that a belongs to T_q. We first compute W_r^a from W_r. Note that $W_r = W_r^c$ as c is the connector of T_r. Since a is not in T_r, any path from a to a vertex of T_r contains $\pi_{a,c}$ as its subpath (See Fig. 4). Thus, $W_r^a[i] = -\infty$ for $0 \le i \le \min\{l(\pi_{a,c})-1, 2L-2\}$, and $W_r^a[i] = w(\pi_{a,c}) + W_r[i - l(\pi_{a,c})]$ for $\min\{l(\pi_{a,c}), 2L-1\} \le i \le 2L-2$. Since both $l(\pi_{a,c})$ and $w(\pi_{a,c})$ can be obtained in $O(1)$ time, W_r^a can be found in $O(L)$ time.

W_q^a can be computed recursively. Both T_p and T_q contain a in Fig. 4. So, computing W_p^a and computing W_q^a are basically identical except that T_p and T_q, respectively, are involved. The method that computes W_p^a can be used to compute W_q^a if p is replaced by q (or, if T_p is replaced by T_q). The recursion stops if a leaf node is reached. Summarizing the explanations given so far, our algorithm for computing W_p is shown in Fig. 5. To compute W_p for the nodes p of T^*, traverse T^* in postorder and call *computeW*(p) for each node p.

We analyze the time complexity of computing W_p for the nodes p of T^*. Remember that T^* is a rooted binary tree of height $h = O(\log(n/L))$, and has $O(n/L)$ leaf nodes at level 1 and its root at level h. Moreover, Frederickson [5] proved that the number of nodes at level i is at most $5/6$ of the number of nodes at level $i-1$. So, if m is the number of leaf nodes of T^*, then the number of nodes at level i is at most $(5/6)^{i-1}m$.

For a node p at level i, a call to *computeW*(p) makes at most i recursive calls to *recursion*$(\cdot)$ until a leaf is reached. So, the total number of calls to *recursion*$(\cdot)$ while computing W_p for the nodes p of T^* is at most $\sum_{i=1}^{h} i(5/6)^{i-1}m$, which is less than $36m$. Note that $\sum_{i=1}^{\infty} i\alpha^i = \alpha/(1-\alpha)^2$ for $0 < \alpha < 1$. Since each call to *recursion*$(\cdot)$ takes $O(L)$ time and $m = O(n/L)$, the time of computing W_p for the nodes p of T^* is at most $30m \cdot c_1L \leq 30 \cdot c_2(n/L) \cdot c_1L = O(n)$ for constants $c_1, c_2 > 0$.

We now have W_p for every node p of T^*. To locate a length-constrained maximum-density path in T we need to consider only the paths of length at least L and at most $2L-1$ by Lemma 1. Our idea is to traverse T^* in postorder and to find a maximum-density path π_p in T_p of length at least L and at most $2L-1$ and its density d_p for each node p of T^*. Then, d_{root} for the *root* of T^* is the density of a length-constrained maximum-density path of T. For ease of explanation, our algorithm is designed to compute the density d_p only, and an easy modification of the algorithm locates the path π_p itself.

If p is a leaf node in T^*, then T_p has at most L vertices and thus contains no path of length at least L. So, set $d_p = -\infty$. If p has a single child q in T^*, then $T_p = T_q$ and so set $d_p = d_q$.

Suppose that p has two children q and r. Then, $T_p = T_q \cup T_r \cup \{\hat{e}\}$, where $\hat{e}$ is the edge connecting T_q and T_r. At this point, we have d_q and d_r. Since d_q (respectively, d_r) is the density of a length-constrained maximum-density path of T_q (respectively, T_r), we need d', the density of a length-constrained maximum-density path such that one of its endpoints is in T_q and the other is in T_r. Then, $d_p = \max\{d_q, d_r, d'\}$.

To compute d', we employ the linear time algorithm for the "location-constrained" version of the problem of Chung and Lu [2] reviewed in Sect. 2. From W_q and W_r, construct an array $A[-2L+2\,..\,2L-2]$ of size $4L-3$ as follows: $A[0] = w_{\hat{e}}$, $A[-i] = W_q[i] - W_q[i-1]$ for $1 \leq i \leq 2L-2$, and $A[i] = W_r[i] - W_r[i-1]$ for $1 \leq i \leq 2L-2$. For $j \in [-2L+2, -1]$ and $j' \in [1, 2L-2]$, we have $\sum_{i=j}^{j'} A[i] = W_q[-j] + w_{\hat{e}} + W_r[j']$, which is the maximum of the weights of the paths of length $-j+j'+1$ such that one of their endpoints is in T_q and the other is in T_r. After setting $U = 2L-1$, $[x, y] = [-2L+2, -1]$, and $[x', y'] = [1, 2L-2]$, find a solution on A using the algorithm, which is d'. This takes $O(L)$ time.

Since computing d_p for a node p of T^* takes $O(L)$ time and there are $O(n/L)$ nodes in T^*, the computation of d_p for the nodes p of T^* takes $O(n)$ time.

Our algorithm can be summarized as follows:

1. Perform the preprocessing of the path sum query on T with respect to both edge length and edge weight.
2. Build a topology tree T^* which corresponds to a restricted multilevel partition of order $z = L$ of T.
3. Compute W_p for the nodes p of T^* by traversing T^* in postorder.
4. Compute d_p for the nodes p of T^* by traversing T^* in postorder. Then, d_{root} is the final solution.

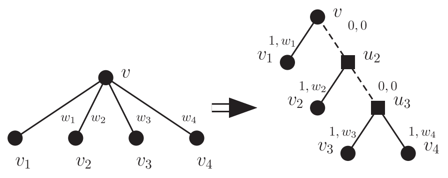

Fig. 6 v has four children v_1, v_2, v_3 and v_4. w_1, w_2, w_3 and w_4 are edge weights. Square vertices and dashed edges are new. $0, 0$ denote the length and weight of a new edge and $1, w_i$ denote the length and weight of an old edge.

Theorem 1: A length-constrained maximum-density path in a binary tree can be computed in $O(n)$ time.

To find a length-constrained maximum-density path in a (general) tree, we transform the tree into a binary tree. Given an edge-weighted tree T_0, a binary tree T is obtained as follows [13]: Select a vertex of degree one of T_0 and make it the root of the rooted version of T_0. For each vertex v of T_0 with $k \geq 3$ children, $v_1, \ldots, v_k$, replace v with a path $(v, u_2, \ldots, u_{k-1})$, where $u_2, \ldots, u_{k-1}$ are new vertices and $(v, u_2), (u_2, u_3), \ldots, (u_{k-2}, u_{k-1})$ are new edges. Each of these new edges has $l_e = w_e = 0$. Replace the edges $\{(v, v_i) \mid 2 \leq i \leq k-1\}$ with the edges $\{(u_i, v_i) \mid 2 \leq i \leq k-1\}$ of corresponding weights, and replace the edge (v, v_k) with the edge (u_{k-1}, v_k) of corresponding weight. The edge (v, v_1) remains unchanged. Each of these "old" edges has $l_e = 1$. See Fig. 6. A proof that every path in T_0 has a corresponding path in T with the same length and weight, and vice versa can be found in [1]. This transformation runs in $O(n)$ time.

Now, T is an edge-weighted binary tree with *binary* edge lengths. The *length* of a path is redefined to be the sum of the lengths of the edges in the path, or the number of edges with $l_e = 1$. Theorem 1 is about binary trees with *uniform* edge lengths (all $l_e = 1$) while the current T is a binary tree with *binary* edge lengths. We focus on explaining what must be changed in the description of the algorithm above when the edge lengths are not uniform but binary.

Since the preprocessing of the path sum query can be done with respect to any *real* values, step 1 is not affected by the fact that the edge lengths are binary. Since a restricted multilevel partition and a topology tree are constructed on the *unweighted* version of T, step 2 can be executed as is.

In Fig. 5, when p is a leaf, W_p^a can be computed using the "direct" method without any change since each $l(\pi_{a,v})$ is from the path sum query. Similarly, the computation of W_r^a is unchanged because the path sum query enables us to find $l(\pi_{a,c})$. So, we can execute step 3 as in the algorithm above.

In step 4, the only part that needs explanation is about A. We construct an array $A[-2L+2\,..\,2L-2]$ of size $4L-3$ from W_q and W_r as we did previously. For $j \in [-2L+2, -1]$ and $j' \in [1, 2L-2]$, $\sum_{i=j}^{j'} A[i] = W_q[-j] + w_{\hat{e}} + W_r[j']$, which is the maximum of the weights of the paths of length

$-j + j' + l_{\hat{e}}$ such that one of their endpoints is in T_q and the other is in T_r. If $l_{\hat{e}} = 0$, then $w_{\hat{e}} = 0$ and thus $\sum_{i=j}^{j'} A[i] = W_q[-j] + W_r[j']$. So, step 4 correctly works.

As each of steps 1–4 again takes $O(n)$ time, we have proved the following theorem.

Theorem 2: A length-constrained maximum-density path in a tree can be computed in $O(n)$ time.

References

[1] B. Bhattacharyya and F. Dehne, "Using spine decompositions to efficiently solve the length-constrained heaviest path problem for trees," Inf. Process. Lett., vol.108, pp.293–297, 2008.

[2] K.M. Chung and H.I. Lu, "An optimal algorithm for the maximum-density segment problem," SIAM J. Comput., vol.34, pp.373–387, 2004.

[3] D. Eppstein, "Asymptotic speed-ups in constructive solid geometry," Algorithmica, vol.13, pp.462–471, 1995.

[4] D. Eppstein, "Clustering for faster network simplex pivots," Networks, vol.35, pp.173–180, 2000.

[5] G.N. Frederickson, "Data structures for on-line updating of minimum spanning trees, with applications," SIAM J. Comput., vol.14, pp.781–798, 1985.

[6] D. Harel and R.E. Tarjan, "Fast algorithms for finding nearest common ancestors," SIAM J. Comput., vol.13, no.2, pp.338–355, 1984.

[7] X. Huang, "An algorithm for identifying regions of a DNA sequence that satisfy a content requirement," Comput. Appl. Biosci., vol.10, pp.219–225, 1994.

[8] G.F. Italiano and R. Ramaswami, "Maintaining spanning trees of small diameter," Algorithmica, vol.2, pp.275–304, 1998.

[9] S.K. Kim, "Linear-time algorithm for the length-constrained heaviest path problem in a tree with uniform edge lengths," IEICE Trans. Inf. & Syst., vol.E96-D, no.3, pp.498–501, March 2013.

[10] H.C. Lau, T.H. Ngo, and B.N. Nguyen, "Finding a length-constrained maximum-sum or maximum-density subtree and its application to logistics," Discrete Optim., vol.3, pp.385–391, 2006.

[11] R.R. Lin, W.H. Kuo, and K.M. Chao, "Finding a length-constrained maximum-density path in a tree," J. Comb. Optim., vol.9, pp.147–156, 2005.

[12] B. Schieber and U. Vishkin, "On finding lowest common ancestors: Simplification and parallelization," SIAM J. Comput., vol.17, pp.1253–1262, 1988.

[13] A. Tamir, "An $O(pn^2)$ algorithm for the p-median and related problems on tree graphs," Oper. Res. Lett., vol.19, pp.59–64, 1996.

[14] B.Y. Wu, K.M. Chao, and C.Y. Tang, "An efficient algorithm for the length-constrained heaviest path problem on a tree," Inf. Process. Lett., vol.69, pp.63–67, 1999.

Sung Kwon Kim received his B.S. degree from Seoul National University, Korea, his M.S. degree from KAIST, Korea, and his Ph.D. degree from University of Washington, Seattle, U.S.A. He is currently with School of Computer Science and Engineering, Chung-Ang University, Seoul, Korea.