PAPER Special Section on Foundations of Computer Science

Path Maximum Query and Path Maximum Sum Query in a Tree*

Sung Kwon KIM[†a)], ***Member***, **Jung-Sik CHO**[†], ***and*** **Soo-Cheol KIM**[†], ***Nonmembers***

SUMMARY Let T be a node-weighted tree with n nodes, and let $\pi(u, v)$ denote the path between two nodes u and v in T. We address two problems: (i) Path Maximum Query: Preprocess T so that, for a query pair of nodes u and v, the maximum weight on $\pi(u, v)$ can be found quickly. (ii) Path Maximum Sum Query: Preprocess T so that, for a query pair of nodes u and v, the maximum weight sum subpath of $\pi(u, v)$ can be found quickly. For the problems we present solutions with $O(1)$ query time and $O(n \log n)$ preprocessing time.

key words: *path maximum query, path maximum sum query, tree*

1. Introduction

Let T be a rooted tree with n nodes. The lowest common ancestor of two nodes u and v in T, denoted by $LCA_T(u, v)$, is the node that is a common ancestor of u and v and is as far as possible from the root.

LCA (Lowest Common Ancestor) Preprocess T so that, for a query pair of nodes u and v, $LCA_T(u, v)$ can be found efficiently.

Solutions with $O(n)$ preprocessing time and $O(1)$ query time are presented in Harel and Tarjan [5], and Schieber and Vishkin [6].

Let $A[1..n]$ be an array with n real numbers.

RMQ (Range Maximum Query) Preprocess A so that, for a query pair of indices $i \le j$, $RM_A(i, j) = \max_{i \le k \le j} A[k]$ can be found efficiently.

Bender *et al.* [1] and Gabow *et al.* [4] show that LCA and RMQ are linearly equivalent. So, solutions with $O(n)$ preprocessing time and $O(1)$ query time are from [5], [6].

Define $S(i, j) = A[i] + \cdots + A[j]$ for $1 \le i \le j \le n$.

RMSQ (Range Maximum Sum Query) Preprocess A so that, for a query pair of indices $i \le j$, $RMS_A(i, j) = \max_{i \le k \le l \le j} S(k, l)$ can be found efficiently.

Chen and Chao [2] proves the linear equivalence between RMQ and RMSQ, and, as a consequence, gives a solution with $O(n)$ preprocessing time and $O(1)$ query time.

This paper extends RMQ and RMSQ from arrays to trees and studies PMQ (path maximum query) and PMSQ (path maximum sum query).

Let $T = (V, E)$ be a size n rooted tree with the node set V and the edge set E. Assume that $V = \{1, 2, \ldots, n\}$. Each node $v \in V$ is weighted with a real value $A[v]$. For a pair of nodes u and v, there exists a unique path in T that connects them. Let $\pi(u, v)$ denote the path.

PMQ (Path Maximum Query) Preprocess T so that, for a query pair of nodes u and v, $PM_T(u, v) = \max_{w \in \pi(u,v)} A[w]$ can be found efficiently.

Define $S(u, v) = \sum_{w \in \pi(u,v)} A[w]$.

PMSQ (Path Maximum Sum Query) Preprocess T so that, for a query pair of nodes u and v, $PMS_T(u, v) = \max_{w,x \in \pi(u,v)} S(w, x)$ can be found efficiently.

We are interested in solutions with $O(1)$ query time whose preprocessing time is as small as possible.

In Sect. 2 we describe our solution to the path maximum query, and in Sect. 3 we present our solution to the path maximum sum query. We give concluding remarks in Sect. 4.

Notation: (u, v) denotes an (undirected) edge, and $\langle u, v \rangle$ denotes a directed edge where v is the parent of u. $par(v)$ denotes the parent of v.

2. Path Maximum Query

We want to preprocess $T = (V, E)$ so that path maximum queries can be answered quickly. For that, we first consider a restricted version of PMQ. A path $\pi(u, v)$ is *lineal* if u is an ancestor of v or vice versa.

LPMQ (Lineal Path Maximum Query) Preprocess T so that, for a query pair of nodes u and v with v being an ancestor of u, $LPM_T(u, v) = \max_{w \in \pi(u,v)} A[w]$ can be found efficiently.

To solve this problem, we first introduce an artificial node $r^* = 0$ with $A[r^*] = \infty$. Let $T^* = (V^*, E^*)$ with $V^* = V \cup \{r^*\}$ and $E^* = E \cup \{\langle r, r^* \rangle\}$. r is the root of T, and r^* is the root of T^* and the parent of r. Define $B[v]$ to be the *bounding ancestor* of v, which is the node x such that $x \in \pi(r^*, par(v))$, $A[x] \ge A[v]$, and the level of x is as large as possible. The level of x is the number of edges in $\pi(r^*, x)$.

Figure 1 shows our algorithm for computing the array $B[1..n]$. Q is a queue that stores nodes, insert(x, Q) appends node x to the rear end of Q, and delete(Q) removes the node at the front end of Q and returns it. The algorithm traverses T^* in level-order. Let v be the currently visited node. At this point, we may assume that $B[w]$ has been computed for each $w \in \pi(r^*, v)$. Suppose it has Δ_v children $u_1, \ldots, u_{\Delta_v}$. We are to compute $B[u_1], \ldots, B[u_{\Delta_v}]$. Relabel so that $A[u_1] \le \ldots \le A[u_{\Delta_v}]$.

Manuscript received March 18, 2008.
Manuscript revised June 23, 2008.

[†]The authors are with the Department of Computer Science and Engineering, Chung-Ang University, Seoul, Korea. S.K. Kim is the correspondence author.

*This research was supported by the Chung-Ang University Research Scholarship in 2008.

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

DOI: 10.1587/transinf.E92.D.166

```
A[r*] ← ∞;
B[r] ← r*;
insert(r, Q);
while(Q ≠ ∅)
    v ←delete(Q);
    if(v is not a leaf)
        let u_1, ..., u_{Δ_v} be the children of v;
        sort A[u_1], ..., A[u_{Δ_v}] and relabel the indices so that
            A[u_1] ≤ ··· ≤ A[u_{Δ_v}];
        B[u_1] ← v;
        for(i = 1 ~ Δ_v)
            while(A[u_i] > A[B[u_i]])
                B[u_i] ← B[B[u_i]];
            insert(u_i, Q);
            if(i < Δ_v)
                B[u_{i+1}] ← B[u_i];
```

Fig. 1 Computing $B[1\,..\,n]$.

Consider the sequence $v, B[v], B[B[v]], \ldots, B[\cdots B[v]\cdots] = r^*$. Notice that $A[v] \le A[B[v]] \le \cdots \le A[r^*]$. We merge the sorted list $A[u_1], \ldots, A[u_{\Delta_v}]$ with another sorted list $A[v], A[B[v]], \ldots, A[r^*]$. From the merged list we can easily compute $B[u_1], \ldots, B[u_{\Delta_v}]$. The algorithm does not explicitly merge the lists, but this is done implicitly.

$B[1\,..\,n]$, computed by the algorithm in Fig. 1, may have duplicate integers. Let $b_1 < \cdots < b_k$ be the distinct integers appearing in B. Obviously, $b_1 = 0$. Let $B_i = \{j \mid B[j] = b_i\} = \{j_{i,1}, \ldots, j_{i,|B_i|}\}$ for $1 \le i \le k$. Rearrange the integers in each B_i so that $A[j_{i,1}] \le \cdots \le A[j_{i,|B_i|}]$. In the sorted list $j_{i,1}, \cdots, j_{i,|B_i|}$, let each element point to the element to its right by assigning $L[j_{i,1}] \leftarrow j_{i,2}$, $L[j_{i,2}] \leftarrow j_{i,3}$, …, and $L[j_{i,|B_i|-1}] \leftarrow j_{i,|B_i|}$. Finally, let the last element of the list point to b_i by assigning $L[j_{i,|B_i|}] \leftarrow b_i$.

Let T_L be the tree defined by L. Its node set is V^* and an edge $\langle j, j'\rangle$ exists if $L[j] = j'$. An edge $\langle j, j'\rangle$ is *vertical* if j and j' belong to the same B_i (i.e., if $j, j' \in B_i$ for some i), and *nonvertical*, otherwise. Note that T_L is a binary tree. If a node in T_L has two children, one of them is connected through a vertical edge and the other through a nonvertical edge.

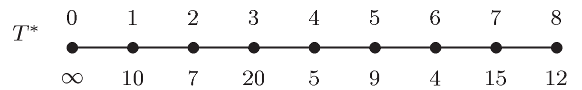
Figure 2 shows how our algorithm computes T_L when T is a tree consisting of a single path of length n; note that this actually corresponds to the case of RMQ. Fig. 3 depicts a geometric description of how $B[1\,..\,n]$ is computed for the case of a single-path tree.

Preprocess T_L for the LCA queries. Then, we claim that for a query lineal path $\pi(u, v)$, $LPM_T(u, v) = LCA_{T_L}(u, v)$. Figure 4 summarizes our algorithm for LPMQ preprocessing.

The following lemma shows that when T consists of a single path only, T_L can be used to answer PMQs correctly.

Lemma 1: Let T be a tree consisting of a single path only. Then, for a query pair of nodes u, v, $LPM_T(u, v) = LCA_{T_L}(u, v)$.

Proof: Since T is a path, we let $T = (1, 2, \ldots, n)$, where $r = 1$ and j is the child of $j-1$ for $2 \le j \le n$. Construct T_L on the array $A[1\,..\,n]$. In T_L, the root r^* has a single child, which is $j_{1,|B_1|}$. In Fig. 2, $j_{1,|B_1|} = 3$. First of all, it is easy to see that



	0	1	2	3	4	5	6	7	8
A	∞	10	7	20	5	9	4	15	12

	1	2	3	4	5	6	7	8
B	0	1	0	3	3	5	3	7

$b_1 = 0 \qquad B_1 = \{1, 3\}$
$b_2 = 1 \qquad B_2 = \{2\}$
$b_3 = 3 \qquad B_3 = \{4, 5, 7\}$
$b_4 = 5 \qquad B_4 = \{6\}$
$b_5 = 7 \qquad B_5 = \{8\}$

	1	2	3	4	5	6	7	8
L	3	1	0	5	7	5	3	7

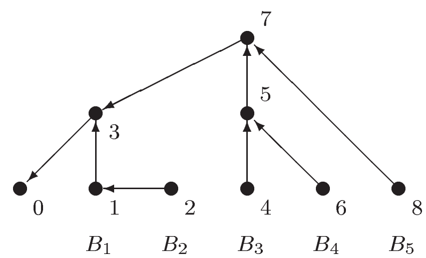


Fig. 2 Computing L and T_L when T is a single path.

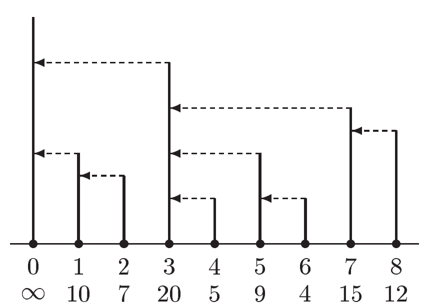


Fig. 3 Geometric description of computing $B[1\,..\,n]$ in Fig. 2. The heights of segments correspond to $A[1\,..\,n]$.

(1) Compute B.
(2) Compute $b_1, \ldots, b_k$ and $B_1, \ldots, B_k$.
(3) Sort each B_i so that $A[j_{i,1}] \le \cdots \le A[j_{i,|B_i|}]$.
(4) Compute L and T_L.
(5) Preprocess T_L for LCA queries.

Fig. 4 Preprocessing T with node values A for answering LPM queries.

$A[j_{1,|B_1|}] = \max A[1\,..\,n]$. See Fig. 3. Assume that $j_{1,|B_1|}$ has two children, p and q, and the edge $\langle p, j_{1,|B_1|}\rangle$ is vertical and the edge $\langle q, j_{1,|B_1|}\rangle$ is nonvertical. Then, $p = j_{1,|B_1|-1}$, and $q = j_{l,|B_l|}$ where l is the integer such that $b_l = j_{1,|B_1|}$. In Fig. 2, $p = 1$ and $q = 7$, $\langle 1, 3\rangle$ is vertical and $\langle 7, 3\rangle$ is nonvertical. Again, by the definition of B it is easy to see that

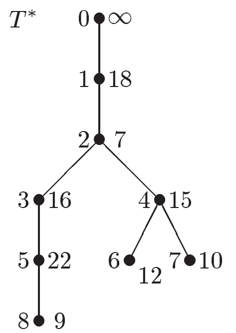


Fig. 5 Example tree.

	0	1	2	3	4	5	6	7	8
A	∞	18	7	16	15	22	12	10	9

	1	2	3	4	5	6	7	8
B	0	1	1	1	0	4	4	5

$$\begin{array}{ll} b_1 = 0 & B_1 = \{1, 5\} \\ b_2 = 1 & B_2 = \{2, 4, 3\} \\ b_3 = 4 & B_3 = \{7, 6\} \\ b_4 = 5 & B_4 = \{8\} \end{array}$$

	1	2	3	4	5	6	7	8
L	5	4	1	3	0	4	6	5

Fig. 6 Computing L for the tree in Fig. 5.

$A[p] = \max A[1 \mathinner{..} j_{1,|B_1|}-1]$ and $A[q] = \max A[j_{1,|B_1|}+1 \mathinner{..} n]$. In Fig. 3, $A[1] = \max A[1 \mathinner{..} 2]$ and $A[7] = \max A[4 \mathinner{..} 8]$.

This corresponds to the definition of a Cartesian tree [7]. The Cartesian tree on A is defined as follows: find $A[j] = \max A[1 \mathinner{..} n]$ and make $A[j]$ the root; find $A[p] = \max A[1 \mathinner{..} j-1]$ and $A[q] = \max A[j+1 \mathinner{..} n]$, and make $A[p]$ and $A[q]$ the children of $A[j]$; and recursively repeat this procedure until all elements of A are contained in the tree.

T_L is a Cartesian tree on A. Hence, by the definition of a Cartesian tree, for a query pair of nodes u, v, we have $LPM_T(u, v) = LCA_{T_L}(u, v)$. □

Figure 5 shows a tree T. Applying the algorithm in Fig. 4 to T computes B and L in Fig. 6, and T_L in Fig. 7 (a).

Consider a leaf node y of T. Let $T' = \pi(r, y)$ be the root-to-leaf path between r and y. Apply to T' our algorithm in Fig. 4 to get T'_L. Let $b'_1, \ldots, b'_{k'}$ and $B'_1, \ldots, B'_{k'}$ be the b_i's and B_i's.

Figure 7 (b) depicts T'_L for $y = 8$ and Fig. 7 (c) for $y = 7$.

Lemma 2: For any pair of nodes $u, v \in T'$, $LCA_{T'_L}(u, v) = LCA_{T_L}(u, v)$.

Proof: It is sufficient to show that, for any pair of nodes $w, x \in T'$, there is a directed edge $\langle w, x\rangle \in T'_L$ if and only if there is a directed path $\langle w, \ldots, x\rangle$ in T_L such that the nodes on the path except w and x are all in $T - T'$.

For example, in Fig. 7 the vertical edge $\langle 2, 3\rangle$ in (b) corresponds to the path $\langle 2, 4, 3\rangle$ in (a), and the nonvertical edge

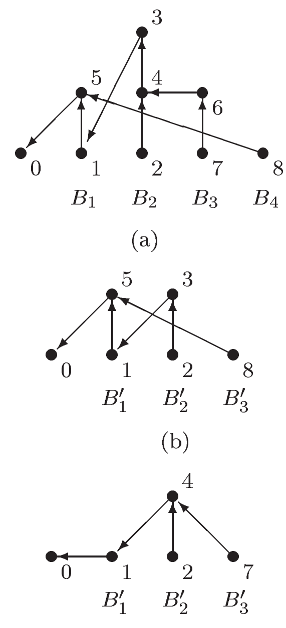


Fig. 7 (a) T_L for the tree in Fig. 5 and L in Fig. 6. (b) T'_L for $y = 8$. (c) T'_L for $y = 7$.

$\langle 4, 1\rangle$ in (c) corresponds to the path $\langle 4, 3, 1\rangle$ in (a).

$\Rightarrow$) i) $\langle w, x\rangle$ is a vertical edge:
Since $\langle w, x\rangle$ is a vertical edge in T'_L, there is an integer l' such that $w, x \in B'_{l'}$. In the sorted list $B'_{l'}$, w appears immediately before x and no other integer lies between them. By the definition of B, it is easy to show that $w, x \in B_l$ for l such that $b_l = b'_{l'}$. In B_L, w appears before x. By the definition of L there is a directed path from w to x in T_L, consisting of vertical edges only. As w and x are adjacent in $B'_{l'}$, no node in T' except w and x can appear on the path.

ii) $\langle w, x\rangle$ is a nonvertical edge:
By the definition of a nonvertical edge, $w \in B'_{l'}$ for l' such that $x = b_{l'}$, and w is the last integer in the sorted list $B'_{l'}$. Let l be the integer such that $x = b_l$. Then, $w \in B_l$. In T_L, there is a directed path $\langle w, \ldots, j_{l,|B_l|}, x\rangle$. The nodes on the path except w and x are not in T' as w is the last one in $B'_{l'}$.

$\Leftarrow$) i) $w, x \in B_l$ for some l:
In this case, $w, x \in B'_{l'}$ for l' such that $b'_{l'} = b_l$. Moreover, w and x appear adjacently because on the path $\langle w, \ldots, x\rangle$ all nodes except w and x are in $T - T'$. So, the vertical edge $\langle w, x\rangle$ is in T'_L.

ii) $w \in B_l$ and $x \in B_m$ for some $l \neq m$:
Since $w \in B_l$, we have $B[w] = b_l$. By the definition of B, b_l is an ancestor of w in T. So, $b_l \in T'$. Consider the directed path from w to x in T_L, $\pi = \langle w, \ldots, j_{l,|B_l|}, b_l, \ldots, x\rangle$. If $x \neq b_l$, then $b_l \in T'$ appears on π. This contradicts to the assumption that no node in T' except w and x appears on π. So, $x = b_l$. In T'_L, $w \in B'_{l'}$ for l' such that $b'_{l'} = x$, and w is the last element in $B'_{l'}$. Hence, $\langle w, x\rangle$ is a nonvertical edge in T'_L. □

From Lemmas 1 and 2 we have the following theorem.

Theorem 1: For any pair of nodes u, v with v being an ancestor of u, $LPM_T(u, v) = LCA_{T_L}(u, v)$.

For a query pair of nodes u, v, $PM_T(u, v)$ can be found as follows:

- Find the node w such that $w = LCA_T(u, v)$.
- Compute $a = LPM_T(u, w)$.
- Compute $b = LPM_T(v, w)$.
- Return $\max\{a, b\}$.

Since each step takes $O(1)$ time, the query time is $O(1)$.

Let us analyze the time complexity of our preprocessing algorithm in Fig. 4. In Step (1), the sorting requires $O(\Delta_v \log \Delta_v)$ time for each node v and the other requires $O(\Delta_v)$ time. Let $\Delta = \max_{v \in V} \Delta_v$. Since $\sum_{v \in V} \Delta_v \log \Delta_v \leq n \log \Delta$, Step (1) can be done in $O(n \log \Delta)$ time. Step (3) calls a sorting for each B_i, and thus needs $O(|B_i| \log |B_i|)$ time for each sorting, and $O(n \log \Delta')$ time in total, where $\Delta' = \max_{1 \leq i \leq k} |B_i|$. The other steps require $O(n)$ time. Thus, the preprocessing time is $O(n \log(\max\{\Delta, \Delta'\}))$, and since $\Delta \leq n$ and $\Delta' \leq n$, it is $O(n \log n)$.

Theorem 2: There is a solution for the path maximum query with $O(1)$ query time and $O(n \log n)$ preprocessing time.

LPMIQ (Lineal Path Minimum Query) Preprocess T so that, for a query pair of nodes u and v with v being an ancestor of u, $LPMI_T(u, v) = \min_{w \in \pi(u,v)} A[w]$ can be found efficiently.

Finding the lineal path minimum can also be solved in a similar way as finding the lineal path maximum. We introduce LPMIQ here to use in the next section.

3. Path Maximum Sum Query

In this section we preprocess T so that path maximum sum queries can be answered efficiently. Again, we are first interested in a restricted version.

LPMSQ (Lineal Path Maximum Sum Query) Preprocess T so that, for a query pair of nodes u and v with v being an ancestor of u, $LPMS_T(u, v) = \max_{w,x \in \pi(u,v)} S(w, x)$ can be found efficiently.

To solve this restricted version efficiently, we first let $C[v] = S(r, v)$ for each $v \in V$. $C[v]$ is the cumulative sum of the path from the root to v. Then, for any two nodes u, v with v being an ancestor of u, $S(u, v) = C[u] - C[par(v)]$.

Let u and v be two nodes with v being an ancestor of u. Let $x_0 = par(v)$. To find $LPMS_T(u, v)$, we first locate $x_1 \in \pi(u, v)$ such that $C[x_1] = \max_{w \in \pi(u,v)} C[w]$, and then locate $y_1 \in \pi(par(x_1), x_0)$ such that $C[y_1] = \min_{w \in \pi(par(x_1),x_0)} C[w]$. For $i = 2, \ldots$, locate $x_i \in \pi(u, x'_{i-1})$ such that $C[x_i] = \max_{w \in \pi(u,x'_{i-1})} C[w]$ where $x'_{i-1} \in \pi(u, v)$ and $par(x'_{i-1}) = x_{i-1}$, and then locate $y_i \in \pi(par(x_i), x_{i-1})$ such that $C[y_i] = \min_{w \in \pi(par(x_i),x_{i-1})} C[w]$. Refer to Fig. 8. Let $x_0, x_1, \ldots, x_l$ and $y_1, \ldots, y_l$ be the sequences of nodes thus obtained. Obviously $x_l = u$.

Lemma 3: $LPMS_T(u, v) = \max_{1 \leq i \leq l}\{C[x_i] - C[y_i]\}$.

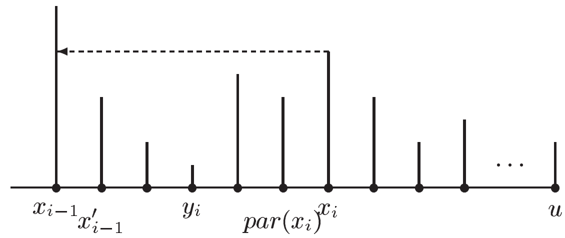


Fig. 8 x_i corresponds to the tallest one to the right of x_{i-1}, and y_i corresponds to the shortest one between x_{i-1} and $par(x_i)$.

Proof: Let x and y be the nodes such that $x, y \in \pi(u, v)$ and $S(x, y) = LPMS_T(u, v)$. Assume that y is an ancestor of x. Then, $S(x, y) = C[x] - C[par(y)]$. We shall show that $x = x_i$ and $par(y) = y_i$ for some i. Note that $\pi(par(x_l), x_0)$ is partitioned into l paths, $\pi(par(x_l), x_{l-1}), \ldots, \pi(par(x_1), x_0)$. Assume that $par(y) \in \pi(par(x_i), x_{i-1})$ for some i. Since $C[par(y)]$ should be as small as possible, we have $par(y) = y_i$ as $C[y_i]$ is the minimum of $C[par(x_i)], \ldots, C[x_{i-1}]$. Since $C[x]$ should be as large as possible, we have $x = x_i$ as $C[x_i]$ is the maximum of $C[u], \ldots, C[x'_{i-1}]$. □

In Sect. 2, $B[v]$ is defined with respect to A. Here, we redefine $B[v]$ with respect to C; $B[v]$ is the node x such that $x \in \pi(r^*, par(v))$, $C[x] \geq C[v]$, and the level of x is as large as possible.

In the sequence $x_1, \ldots, x_l$, $B[x_i] = x_{i-1}$ for $2 \leq i \leq l$ as $C[x_{i-1}] > C[x_i]$ and $C[x_i]$ is the maximum among those to the right of x_{i-1}. The definition of y_i for $2 \leq i \leq l$ can be rephrased as $C[y_i] = \min_{w \in \pi(par(x_i),B[x_i])} C[w]$.

Define $M[v]$ to be the *matching ancestor* of v, which is the node x such that $x \in \pi(par(v), B[v])$, $C[x] = \min_{w \in \pi(par(v),B[v])} C[w]$, and the level of x is as large as possible. Note that $M[x_i] = y_i$ for $2 \leq i \leq l$. Lemma 3 can be rewritten as $LPMS_T(u, v) = \max\{C[x_1] - C[y_1], \max_{2 \leq i \leq l}\{C[x_i] - C[M[x_i]]\}\}$.

For brevity, let $D[v] = C[v] - C[M[v]]$ for $v \in V - \{r\}$, and $D[r] = A[r]$. Then, $D[x_i] \geq D[w]$ for any node $w \in \pi(x_i, x'_{i-1})$. Hence, $\max_{w \in \pi(u,x'_1)} D[w] = \max_{2 \leq i \leq l} D[x_i]$.

Lemma 4: $LPMS_T(u, v) = \max\{C[x_1] - C[y_1], \max_{w \in \pi(u,x'_1)} D[w]\}$.

The algorithm in Fig. 9 computes the arrays $C[1..n]$, $B[1..n]$, $M[1..n]$ and $D[1..n]$. It works in a similar way as the one in Fig. 1 does. Let v be the current node. We assume that $B[w]$ and $M[w]$ have been computed for $w \in \pi(r^*, v)$. Suppose it has Δ_v children, $u_1, \ldots, u_{\Delta_v}$. Compute $C[u_i] = C[v] + A[u_i]$ for $1 \leq i \leq \Delta_v$, and sort them in ascending order. Consider the sequence $v, B[v], B[B[v]], \ldots, B[\cdots B[v] \cdots] = r^*$. Let $v_1 = v, v_2 = B[v], \ldots, v_m = r^*$. To compute $B[u_i]$, we assume that $B[u_{i-1}]$ has already been computed. Let $B[u_{i-1}] = v_p$. We scan the list $v_p, v_{p+1}, \ldots, v_m$ until we find v_q such that $C[v_q] \geq C[u_i]$.

To compute $M[u_i]$, we also may assume that $M[u_{i-1}]$ has already been computed. Let $B[u_{i-1}] = v_p$ and $B[u_i] = v_q$ for $p < q$. $M[u_i]$, by definition, is the node in $\pi(v, v_q)$ such that $C[M[u_i]] = \min_{w \in \pi(v,v_q)} C[w]$. This is equivalent to $C[M[u_i]] = \min_{1 \leq j \leq q-1} C[M[v_j]]$.

```
C[r*] ← ∞;
C[r] ← A[r];
B[r] ← r*;
D[r] ← A[r];
insert(r, Q);
while(Q ≠ ∅)
    v ←delete(Q);
    if(v is not a leaf)
        let u_1, ..., u_{Δ_v} be the children of v;
        for(i = 1 ∼ Δ_v)
            C[u_i] ← C[v] + A[u_i];
        sort C[u_1], ..., C[u_{Δ_v}] and relabel the indices so that
            C[u_1] ≤ ··· ≤ C[u_{Δ_v}];
        B[u_1] ← M[u_1] ← v;
        for(i = 1 ∼ Δ_v)
            while(C[u_i] > C[B[u_i]])
                if(C[M[u_i]] > C[M[B[u_i]]])
                    M[u_i] ← M[B[u_i]];
                B[u_i] ← B[B[u_i]];
            D[u_i] ← C[u_i] − C[M[u_i]];
            insert(u_i, Q);
            if(i < Δ_v)
                B[u_{i+1}] ← B[u_i];
                M[u_{i+1}] ← M[u_i];
```

Fig. 9 Computing C, B, M and D.

(1) Compute C and D using the algorithm in Fig. 9.
(2) Preprocess T for LCA queries.
(3) Preprocess T with node values D for answering $LPM_{T,D}(u,v)$ using the algorithm in Fig. 4.
(4) Preprocess T with node values C for answering $LPM_{T,C}(u,v)$ using the algorithm in Fig. 4.
(5) Preprocess T with node values C for answering $LPMI_{T,C}(u,v)$ using the algorithm in Fig. 4.

Fig. 10 Preprocessing for PMSQs and LPMSQs.

Since $C[M[u_{i-1}]] = \min_{1\le j\le p-1} C[M[v_j]]$, we have $C[M[u_i]] = \min\{C[M[u_{i-1}]], \min_{p\le j\le q-1} C[M[v_j]]\}$. Starting with $M[u_i] = M[u_{i-1}]$, the algorithm scans $M[v_p], \ldots, M[v_{q-1}]$ to find the minimum of their C values.

Figure 10 shows our algorithm for preprocessing T for PMS queries and LPMS queries. $LPM_{T,D}(u,v)$, $LPM_{T,C}(u,v)$, and $LPMI_{T,C}(u,v)$ denote $\max_{w\in\pi(u,v)} D[w]$, $\max_{w\in\pi(u,v)} C[w]$, and $\min_{w\in\pi(u,v)} A[w]$, respectively.

Given a query pair of nodes u and v with v being an ancestor of u, we can find $LPMS_T(u,v)$ using the following query-answering algorithm, which is based on Lemma 4.

- Find $x_1 \in \pi(u,v)$ such that $C[x_1] = LPM_{T,C}(u,v)$.
- Find $y_1 \in \pi(par(x_1), par(v))$ such that $C[y_1] = LPMI_{T,C}(par(x), par(v))$.
- Find $z \in \pi(u, x_1')$ such that $D[z] = LPM_{T,D}(u, x_1')$.
- Return $\max\{C[x_1] - C[y_1], D[z]\}$.

Since each step takes $O(1)$ time, the query time for $LPMS_T(u,v)$ is $O(1)$.

We are now ready to answer path maximum sum queries. Given a query pair of nodes u and v, $PMS_T(u,v)$ can be computed as follows:

- Find w such that $w = LCA_T(u,v)$.
- If $u = w$, return $LPMS_T(v,w)$.
- If $v = w$, return $LPMS_T(u,w)$.
- Compute $a = LPMS_T(u,w)$.
- Compute $b = LPMS_T(v,w)$.
- Find $x \in \pi(u,w)$ such that $C[x] = LPM_{T,C}(u,w)$.
- Find $y \in \pi(v,w)$ such that $C[y] = LPM_{T,C}(v,w)$.
- Compute $c = C[x] - C[par(w)] + C[y] - C[w]$.
- Return $\max\{a, b, c\}$.

Given u and v, we first locate $w = LCA_T(u,v)$. If $u = w$ or $v = w$, then $\pi(u,v)$ is a lineal path and its maximum sum can be found by calling either $LPMS_T(v,w)$ or $LPMS_T(u,w)$. Otherwise, we compute a, b and c, and return their maximum. a (resp., b) is the sum of a maximum sum path, both of whose end nodes are on $\pi(u,w)$ (resp., $\pi(v,w)$). c is the sum of a maximum sum path, one of whose end nodes, x, is on $\pi(u,w)$ and the other, y, is on $\pi(v,w)$.

It is easy to see that the query time is $O(1)$ as each step takes only $O(1)$ time. The time complexity of the preprocessing in Fig. 10 is as follows: Step (1) takes $O(n\log\Delta)$ time. Step (2) takes linear time. Steps (3), (4) and (5) use the same algorithm in Fig. 4 to the same tree T with different node values (C or D) or different objective functions (maximum or minimum). Step (3) takes $O(n\log(\max\{\Delta, \Delta_3'\}))$, where $\Delta_3' = \Delta'$ and Δ' is defined in the previous section. Similarly, Steps (4) and (5) take $O(n\log(\max\{\Delta, \Delta_4'\}))$ and $O(n\log(\max\{\Delta, \Delta_5'\}))$, respectively. The preprocessing requires $O(n\log(\max\{\Delta, \Delta_3', \Delta_4', \Delta_5'\})) = O(n\log n)$ time as $\Delta, \Delta_3', \Delta_4', \Delta_5' \le n$.

Theorem 3: There is a solution for the path maximum sum query with $O(1)$ query time and $O(n\log n)$ preprocessing time.

4. Concluding Remarks

We have presented solutions with $O(1)$ query time for both the path maximum query and the path maximum sum query. One immediate future work is to reduce the preprocessing time.

References

[1] M.A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin, "Lowest common ancestors in tree and directed acyclic graphs," J. Algorithms, vol.57, pp.79–94, 2005.
[2] K.Y. Chen and K.M. Chao, "On the range maximum-sum segment query problem," Discrete Appl. Math., vol.155, pp.2043–2052, 2007.
[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms, Second ed., MIT Press and McGraw-Hill, 2001.
[4] H. Gabow, J. Bentley, and R.E. Tarjan, "Scaling and related techniques for geometry problems," STOC, pp.135–143, 1984.
[5] D. Harel and R.E. Tarjan, "Fast algorithms for finding nearest common ancestors," SIAM J. Comput., vol.13, no.2, pp.338–355, 1984.
[6] B. Schieber and U. Vishkin, "On finding lowest common ancestors: Simplification and parallelization," SIAM J. Comput., vol.17, pp.1253–1262, 1988.
[7] J. Vuillemin, "A unifying look at data structures," Commun. ACM, vol.23, pp.229–239, 1980.

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 Department of Computer Science and Engineering, Chung-Ang University, Seoul, Korea.

Jung-Sik Cho received his B.E. degree from Kang-nam University, Korea in 2003 and his M.E. degree from Chung-Ang University, Korea in 2005, respectively, all in computer science and engineering. He is currently a Ph.D. course student at Chung-Ang University. His areas of research interest are security of RFID system, security of sensor network, cryptography, and information security.

Soo-Cheol Kim received his B.E. and M.E. degrees from Chung-Ang university, Korea, in 2004 and 2007, respectively, all in computer sciences and engineering. He is currently a Ph.D. course student at Chung-Ang University. His areas of research interest are security of RFID system, security of sensor network, cryptography, and information security.