Efficient Data Structures for Range-Aggregate Queries on Trees*

Hao Yuan and Mikhail J. Atallah
Department of Computer Science
Purdue University
Lafayette, IN 47907
{ yuan3, mja }@cs.purdue.edu

ABSTRACT

Graph-theoretic aggregation problems have been considered both in OLAP (grid graph) and XML (tree). This paper gives new results for MIN aggregation in a tree, where we want the MIN in a query subtree consisting of the nodes reachable from a node u along paths of length $\leq k$ (u and k are query parameters). The same problem is well solved when the aggregation is SUM rather than MIN, but the solutions rely on additive inverses for the "+" operator, and they fail for the MIN aggregation which is the topic of this paper. For the directed (rooted tree) case, we give an $O(n)$ space, constant query time solution. For the undirected case, the space complexity is $O(n \log n)$ and the query time is $O(\log n)$.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Trees

General Terms

Algorithms

Keywords

Range aggregation, query answering

1. INTRODUCTION

Range search (or aggregation) is a fundamental function in database systems. In a typical range search problem, we have a set of objects S and a commutative semigroup $(\mathbb{S}, +)$, where each object $s \in S$ is assigned a weight $w(s)$ from $\mathbb{S}$. A range counting query is in the form $\sum_{s \in S'} w(s)$, where the query subset $S' \subseteq S$ usually has some shape or proximity properties, e.g., if S is a set of points in the 2D space, then the S' can be the points that are encapsulated by a query rectangle, or are within a specified threshold distance away from a query point, etc. A range minimum query is similar to range counting except that the associative aggregation operation is the "min" rather than the "+". A range query consists of the enumeration of the elements of the query subset S'. Many geometric searching problems can be formulated under this range querying framework, see the survey paper [1] for more details.

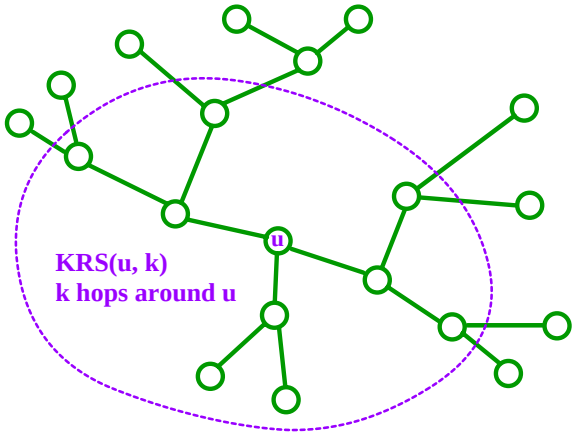
While the notions of proximity and "range" that were considered in the above-mentioned previous work were geometric, equally important are the corresponding problems in a graph-theoretic setting (where, e.g., the notion of distance is "path length" rather than Euclidean, and where a range is a subgraph having a specified shape or property, etc). The work in that graph-theoretic setting includes aggregation (both SUM and MIN) in an OLAP data cube (a high-dimensional grid graph)[12, 17, 16, 19, 6], and also in an XML or other tree structure [13, 20, 14, 8, 18] whose special case is the important range-min query of a path [9, 4] that plays an important role in many applications (most notably in sequence comparisons [10]).

In the proximity-based version of this graph-theoretic aggregation problem, the query subset S' would consist of all the nodes that are within a distance of k from a node u (both u and the nonnegative integer k are query parameters), where distance could be either along directed edges or by ignoring edge directions (i.e., using the undirected version of the graph). In a more frivolous application, such a query would correspond to the situation where, in a road network, someone wants to find the cheapest gas station that is within a distance of k to a target location u. Another, more recent motivation for considering such problems comes from integrity verification in an environment of untrusted third-party distributors (see, e.g., [2, 15, 21]). We have very recently [21] provided efficient solutions to such problems on a tree for the case where the aggregation operation is "+". The solutions in [21] make crucial use of the group structure (the existence of additive inverses) and do not apply to the case where the aggregation operation is a "min" and does not have an inverse. Solving the problem for the case where the aggregation operation is a "min" is harder, and it is the main contribution of the present paper. A more formal statement of the problem and our results is given next.

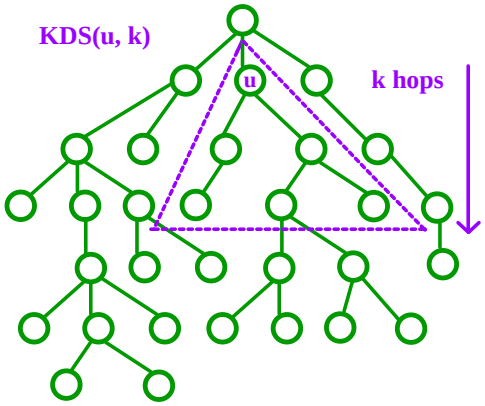
In this paper, we consider the case when S forms a tree and S' is a subtree of S. We consider both the directed

*Portions of this work were supported by Grant CNS-0627488 from the National Science Foundation, and by sponsors of the Center for Education and Research in Information Assurance and Security.

ICDT 2009, March 23–25, 2009, Saint Petersburg, Russia.



(a) K-Radius Subtree.



(b) K-Depth Subtree.

Figure 1: Example of the query subtrees ($k = 2$ in the figures).

(rooted tree) case and the undirected one. We begin with the undirected case. Particularly, we consider the subtree S' to have the following proximity properties: S' consists of all the nodes that are within a distance of k from a query node u (both u and the nonnegative integer k are query parameters). If we set the length of each tree edge to be 1, and define $d(u, v)$ to be the distance between any two nodes u and v, then we can formally define our query subtree, $\mathrm{KRS}(u, k)$, as follows

$$\mathrm{KRS}(u, k) = \{v \mid d(u, v) \leq k\}.$$

In the directed case of a rooted tree, we set $\mathrm{KDS}(u, k)$ to be the subtree which consists of node u's descendants that are within a distance of k to u, i.e.,

$$\mathrm{KDS}(u, k) = \{v \mid d(u, v) \leq k \text{ and } v \text{ is a descendant of } u\}.$$

Here, u is considered as a descendant of itself. See Figure 1 for examples of the queries.

In the setting when $\mathbb{S}$ is a group (e.g., for SUM aggregation), $O(n \log n)$-size data structures (also built in $O(n \log n)$ time) were given in [21] to sum over the group elements that are associated with $\mathrm{KRS}(u, k)$ in $O(\log n)$ time, and $O(n)$-size data structures (built in $O(n)$ time) were given to sum over those elements associated with $\mathrm{KDS}(u, k)$ in $O(1)$ time [21].

For the setting when $\mathbb{S}$ is in a total order and the aggregation operator is min (or max) over the total order, the data structures in [21] do not apply as they make crucial use of the fact that each group element has an inverse. In this paper, we follow a very different approach, and give $O(n \log n)$-size data structures (preprocessed in $O(n \log n)$ time) that can be used to compute the minimum element of $\mathrm{KRS}(u, k)$ in $O(\log n)$ time. As a subroutine for the min KRS query, the min KDS query can be solved in $O(1)$ time by preprocessing in $O(n)$ time and space. The computation model is the standard one used in the previous literature (i.e., the unit-cost RAM model).

Note that the min KDS query is a natural extension of the Range Minimum Query (RMQ) problem [9, 4], i.e., the RMQ query on an array can be treated as a special case of min KDS query when the tree considered is a chain, and the query subtree is a subchain.

The followings are sample applications of the subtree MIN (or MAX) queries:

- Assume that we have an XML document that stores some statistical information about a web site hierarchy. More specifically: each tree node of the XML document represents an HTML web page, and the web pages form a tree hierarchy (e.g., the main page is at the root); each node has stored the hits (number of visitors) of the corresponding web page. An interesting query can be: what is the least (or most) visited web page in a set of pages that are organized as a k-depth subtree?

- In a computer network of servers (naturally organized in a tree structure), each server has a certain amount of computing power. Consider the following scenario: we want to choose a most powerful server to install a database service, with the constraint that the chosen server v must be within a distance of k to a targeted server u (in which the major user of the the database service is located). The distance constraint may be due to the network delay requirement, in terms of the maximum acceptable number of hops (i.e., k) for u to reach the chosen v. This is indeed a max $\mathrm{KRS}(u, k)$ query.

This paper is organized as follows. In Section 2, we define some basic notation. Section 3 and Section 4 give our data structures for the K-Depth Subtree aggregation and K-Radius Subtree aggregation respectively. Section 5 concludes.

2. BASIC DEFINITIONS

In a tree $T = (V, E)$ with node set V and edge set E, we assign to each node $v \in V$ an element $w(v)$ from $\mathbb{S}$, where $\mathbb{S}$ is total ordered under $\leq$. We call w the weight function. For any node set V' of T, we define $\min V'$ to be $\min\{w(v) \mid v \in V'\}$. We use T' to represent the node set or the edge set of a subtree T' when there is no ambiguity, so $\min T'$ means the minimum weight assigned to the nodes of the subtree T'.

Each tree edge is of length 1 by default. The distance between two nodes u and v is denoted by $d(u, v)$. When $u = v$, we define $d(u, v) = 0$. It is well-known ("folklore") that the distance function for a tree can be computed in constant-time by a linear preprocessing as follows: specify a node r as the root (if it's the unrooted case), and then do a depth-first search to pre-compute the depth $d(z)$ for each node z; for an online distance query for u and v, we first compute the nearest common ancestor of u and v in constant time by the method from [11, 4](using linear space, also built in linear time); assume that the nearest common ancestor is w, then we have $d(u, v) = d(u) + d(v) - 2d(w)$, which can be computed in constant time.

In a rooted tree, we define the depth, $d(v)$, to be the distance of v to the root. Also, we define the height, $h(v)$, to be $\max\left\{d(v') - d(v) \mid v' \text{ is } v \text{ or a descendant of } v\right\}$. The complete subtree, $CS(v)$, is the subtree that is rooted at v and includes all its descendants. The children node set of v is denoted by $C(v)$.

3. K-DEPTH SUBTREE AGGREGATION

In this section, we give linear data structures to compute $\min \mathrm{KDS}(u, k)$ for any u and k in constant time. The basic idea of the algorithm is to partition the rooted tree T into $n/\log n$ blocks (each block is a sub-forest), and then process the query subtree $\mathrm{KDS}(u, k)$ in each block efficiently.

Denote the root by r, then we partition the tree into B_0, B_1, ..., $B_{n/\log n-1}$ in the following way [1]:

$$B_i = \left\{v \mid i \cdot \log n \le d(v) < (i+1) \cdot \log n\right\}, 0 \le i < n/\log n.$$

Let $B_{i_a}, B_{i_a+1}, B_{i_a+2}, \cdots, B_{i_b}$ be the consecutive blocks that the query $\mathrm{KDS}(u, k)$ overlaps with, i.e., $i_a = \lfloor d(u)/\log n \rfloor$ and $i_b = \lfloor (d(u) + k)/\log n \rfloor$. We process the query using three subqueries:

- $\mathrm{KDS}(u, k) \cap B_{i_a}$;
- $\mathrm{KDS}(u, k) \cap \bigcup_{i_a < i < i_b} B_i$;
- $\mathrm{KDS}(u, k) \cap B_{i_b}$.

For each subquery, we provide data structures that can be built in linear time and space and answer the subquery in constant time. Note that when $i_a < i_b$, the aggregation of $\mathrm{KDS}(u, k) \cap B_{i_a}$ is trivial to be done in constant time by a simple bottom-up pre-aggregation. So the following Section 3.1 is used to cover the case when $i_a = i_b$ (and in that case, the second and third subqueries are not necessary). Section 3.2 covers the second subquery, and Section 3.3 covers the third.

The reason to break the $\min \mathrm{KDS}(u, k)$ query into three subqueries is to make use of the power of the $\Theta(\log n)$ word size in the unit-cost RAM model. The first subquery benefits from the fact that the height of considered subtree is of at most $\log n$, which allows a compact representation of the query results for all possible pairs of parameters. The second subquery takes advantage of the fact that only $O(n/\log n)$ tree nodes need special processing, and we can afford a superlinear data structure for them. The third subquery is reduced to a special 2D RMQ problem. See the following subsections for details.

[1]Without loss of generality, we ignore the floor and ceiling function in this paper, and always assume that the computed results are integers.

3.1 Rooted Tree with log n Height

The $\mathrm{KDS}(u, k)$ query in B_{i_a} is indeed a special case of the KDS aggregation problem in a tree with $\log n$ height. So in this subsection, we use $\hat{T}$ instead of B_{i_a} to describe our data structures with the restriction that the maximum depth of a node in $\hat{T}$ is $\log n - 1$. Also, we denote the size of the tree $\hat{T}$ to be m. The goal is to preprocess $\hat{T}$ in $O(m)$ time and space to answer $\min \mathrm{KDS}(u, k)$ query in constant time. For each tree component in each B_i (a forest), we preprocess it using the data structure in this subsection. The total time and space for preprocessing all such tree components will be $O(n)$.

The basic idea is to locate the depth of the node that has the minimum element in $\mathrm{KDS}(u, k)$, and then do a range minimum query [9, 4] at that depth level. More specifically, let $L_i = \left\{v \mid \mathrm{d}(v) = i\right\}$ for $0 \le i < \log n$, and we want to build the following data structure (TD for TargetDepth):

$$\mathrm{TD}(v, k) = d(v'),$$

where v' has the minimum weight in $\mathrm{KDS}(v, k)$.

If there are more than two nodes whose weights are minimum, then we break the tie by choosing the one with the smallest depth. Once we have built this data structure (see Section 3.1.1), we could know that the depth of the minimum element of $\mathrm{KDS}(u, k)$ is $\mathrm{TD}(u, k)$, and then we can find out $\min \mathrm{KDS}(u, k)$ by performing a range minimum query (RMQ) [9, 4] at $L_{\mathrm{TD}(u,k)}$ (see Section 3.1.2).

3.1.1 Find the Target Depth

Storing the data structures for $\mathrm{TD}(v, k)$ ($v \in \hat{T}, 0 \le k < \log n$) directly will cost $\Theta(m \log n)$ time and space in the worst case. To save space, observe that for any k_1 and k_2 such that $k_2 = k_1 + 1$, we have either $\mathrm{TD}(v, k_2) = \mathrm{TD}(v, k_1)$ or $\mathrm{TD}(v, k_2) = d(v) + k_2$, because $\mathrm{KDS}(v, k_2)$ consists of $\mathrm{KDS}(v, k_1)$ and a portion of L_{k_2}. This observation can help achieve linear space (under the unit-cost RAM model with $O(\log n)$-size words): for each node v, we use $\log n$ bits $b_k^{(v)}$ ($0 \le k < \log n$) to represent the $\mathrm{TD}(v, k)$ structures, where $b_k^{(v)} = 0$ if $\mathrm{TD}(v, k) = \mathrm{TD}(v, k-1)$ and $b_k^{(v)} = 1$ if $\mathrm{TD}(v, k) = d(v) + k$. We use $b^{(v)}$ to denote this bit vector for node v.

The following bottom-up procedure is the basic framework to compute the $b^{(v)}$ arrays: do a postorder tree walk starting at the root; for each node v, maintain a linked list $M_v(i)$ ($0 \le i < \log n$) whose values are $\min \mathrm{KDS}(v, i)$'s. Once we have the M_v list, it can be convert to $b^{(v)}$ efficiently according to the definition of $b^{(v)}$. The following algorithms will show that the maintenance of the M_v list for all v can be done in linear time and space in total, later we will describe how to modify the algorithm to convert the M_v list to $b^{(v)}$ without increasing the complexities.

Recall that $C(v)$ represents the set of children of node v,

and let HC(v) to be the child of v with the largest height. If $C(v) = \emptyset$, then we set HC(v) = $\emptyset$; if there is more than one child who has the largest height, then choose any of them to be HC(v). At the time when a node v is visited during the postorder tree walk, we compute its M_v list by combining the lists of its children nodes as in Algorithm 1.

Algorithm 1 Base Algorithm for Maintaining the M_v list

Procedure BaseMV(v)

1: **for** $v_c \in C(v)$ **do**
2: call BaseMV(v_c)
3: **end for**
4: initialize $M_v(i) \leftarrow w(v)$ for $0 \le i \le h(v)$
5: update $M_v(i+1) \leftarrow \min\{M_v(i+1), M_{\mathrm{HC}(v)}(i)\}$ for $0 \le i \le h(\mathrm{HC}(v))$
6: discard $M_{\mathrm{HC}(v)}$
7: **for** $v_c \in C(v) \setminus \mathrm{HC}(v)$ **do**
8: update $M_v(i+1) \leftarrow \min\{M_v(i+1),\ M_{v_c}(i)\}$ for $0 \le i \le h(v_c)$
9: update $M_v(i+1) \leftarrow \min\{M_v(i+1),\ M_{v_c}(h(v_c))\}$ for $h(v_c) < i \le h(v) - 1$.
10: discard the M_{v_c} list
11: **end for**

The space complexity for Algorithm 1 is $O(m)$. This is because: during the call of BaseMV(v), among those nodes $\{v' \mid v' \neq v$ and list $M_{v'}$ is still in the memory$\}$, no node is a descendant of the other and the length of the $M_{v'}$ list is no larger than the size of CS(v'), so the union of the active (i.e., non-discarded) $M_{v'}$'s are no bigger than the size of the whole tree.

A straightforward implementation of Algorithm 1 will require $O(\sum_v h(v)\,|C(v)|)$ time, which can be as bad as $\Theta(m \log n)$. To achieve better time complexity (especially the *update* operation at line 5, 8 and 9), we encode M_v by packing those consecutive entries which have the same value into a single entry. More specifically, we use $M'_v(j) = \langle a_j, l_j \rangle$ (call it an entry or a record) to represent that $M_v(i) = a_j$ for $\sum_{1 \le j' < j} l_{j'} \le i < \sum_{1 \le j' \le j} l_{j'}$, where the entry index j ranges from 1 to the number of unique elements in M_v and $\sum_j l_j = h(v)+1$. That is to say, the values of the first l_1 entries of M_v are a_1, the next l_2 entries are a_2, etc. To simplify the representation, we use $M'_v(j).value$ and $M'_v(j)$.length to denote the corresponding a_j and l_j of $M'_v(j)$. Using this compact representation and monotonicity of M'_v (i.e., $M'_v(j)$.value $\ge M'_v(j+1)$.value), we have Algorithm 2. In the algorithm, we use $|M'_v|$ to represent the number of entries (or records) in the linked list M'_v.

In Algorithm 2, lines 4-6 try to initialize the records of M'_v by inheriting from $M'_{HC(v)}$, and update the records by considering $w(v)$. The time complexity is proportional to

$1 +$ number of records from $M'_{HC(v)}$ that are killed by $w(v)$
$+$ number of records that are created.

Here, a record is killed by $w(v)$ means that the "value" field of the record is greater than or equal to $w(v)$. Also, the number of records that are newly created is $O(1)$ in lines 4-6. Lines 8-17 try to combine the M'_{v_c} list to M'_v. The time complexity of lines 8-11 is $O(h(v_c)+1)$, and the time

Algorithm 2 Maintaining the Compact M'_v list

Procedure CompactMV(v)

1: **for** $v_c \in C(v)$ **do**
2: call CompactMV(v_c)
3: **end for**
4: find the largest j such that $M'_{\mathrm{HC}(v)}(j)$.value $\ge w(v)$ and $M'_{\mathrm{HC}(v)}(j+1)$.value $< w(v)$
5: initialize $M'_v(1) \leftarrow \langle w(v), l \rangle$, where $l = 1 + \sum_{1 \le i \le j} M'_{\mathrm{HC}(v)}$.length
6: append the list starting from $M'_{\mathrm{HC}(v)}(j+1)$ after the first entry of M'_v, which is equivalent to setting $M'_v(1+i-j) \leftarrow M'_{\mathrm{HC}(v)}(i)$ for $j < i \le |M'_{\mathrm{HC}(v)}|$
7: **for** $v_c \in C(v) \setminus \mathrm{HC}(v)$ **do**
8: convert M'_{v_c} to the non-compact form M_{v_c}
9: find the largest j such that $\sum_{1 \le i < j} M'_v(j)$.length $\le 2 + h(v_c)$; set $l \leftarrow \sum_{1 \le i < j} M'_v(i)$.length
10: split $M'_v(j) = \langle a_j, l_j \rangle$ into two parts: $m'_1 = \langle a_j, 2 + h(v_c) - l \rangle$ and $m'_2 = \langle a_j, l_j - (2 + h(v_c) - l) \rangle$
11: convert $M'_v(1), M'_v(2), \ldots, M'_v(j-1)$ and m'_1 to the non-compact form, and then update the non-compact $M_v(i)$ by $\min\{M_v(i), M_{v_c}(i-1)\}$ for $1 \le i \le h(v_c)+1$; convert the updated $M_v(i)$ $(0 \le i \le h(v_c)+1)$ back to the compact form, denote this new compact list by m''_1
12: find the smallest $j' \ge j$ such that $M'_v(j')$.value $< m''_1(|m''_1|)$.value; if no such j' exists, set j' to be $|M'_v(j')| + 1$
13: **if** $j' > j$ **then**
14: extend the length of the last entry of m''_1 by m'_2.length $+ \sum_{j \le i < j'} |M'_v(i)|$.length; append $M'_v(i)$ (for $i \ge j'$) to the m''_1 list
15: **else**
16: append m'_2 and $M'_v(i)$ (for $i > j'$) to the m''_1 list
17: **end if**
18: replace the M'_v list by m''_1
19: discard the M_{v_c} list
20: **end for**

complexity of lines 12-17 is in proportional to

$$1 + \text{number of records that are killed} + \text{number of records that are created.}$$

Note that in lines 8-17, only $O(1)$ number of records are created (including the splitted records at line 10). Combing the analysis above, the number of records that are created during the whole algorithm is $O\big(\sum_v (1 + |C(v)|)\big) = O(m)$. This implies that the number of records that are killed is also $O(m)$. The total complexity of lines 8-11 over all the loops is $O\big(\sum_v \sum_{v_c \in C(v) \setminus \mathrm{HC}(v)} (h(v_c) + 1)\big)$, which is indeed $O(m)$ according the following Lemma 1. Therefore, the total time complexity of Algorithm 2 is $O(m)$.

LEMMA 1.

$$\sum_{v \in \hat{T}} \sum_{v_c \in C(v) \setminus HC(v)} (h(v_c) + 1) \leq m$$

PROOF. Perform a *Longest Path Assignment* as follows: do a bottom-up tree walk; when a leaf node v is visited, assign $\mathrm{LP}(v) \leftarrow \{v\}$; when an internal node v is visited, choose any child node v' who has the highest height, then assign $\mathrm{LP}(v) \leftarrow \mathrm{LP}(v') \cup \{v\}$.

We have $|\mathrm{LP}(v)| = h(v) + 1$ according to the definition. Let $\mathrm{UP}(v)$ be the union of the paths assigned to the nodes in the set $v_c \in C(v) \setminus \mathrm{HC}(v)$, then it is sufficient to prove that $\sum_v |\mathrm{UP}(v)| \leq m$. This can be proved by showing that $\mathrm{UP}(v_1) \cap \mathrm{UP}(v_2) = \emptyset$ for any two v_1 and v_2. To show this, we consider the following cases.

- Case 1, $\mathrm{CS}(v_1)$ does not overlap with $\mathrm{CS}(v_2)$. In this case, the argument holds as $\mathrm{UP}(v) \subseteq \mathrm{CS}(v)$ for any v.
- Case 2, $\mathrm{CS}(v_1) \cap \mathrm{CS}(v_2) \neq \emptyset$. Without loss of generality, we assume that v_2 is a descendant of v_1.
  - Case 2a, v_2 is in $\mathrm{LP}(v_1)$. By definition, $\mathrm{UP}(v_1)$ does not include $\mathrm{LP}(v_1)$, therefore, $\mathrm{CS}(v_2)$ does not overlap with $\mathrm{UP}(v_1)$, which implies that $\mathrm{UP}(v_2) \cap \mathrm{UP}(v_1) = \emptyset$.
  - Case 2b, v_2 is not in $\mathrm{LP}(v_1)$. In this case, v_2 must belong to $\mathrm{LP}(v_c)$ for one of v_1's child node v_c. We have

$$\begin{aligned}
&\mathrm{UP}(v_2) \cap \mathrm{UP}(v_1) \\
=&\mathrm{UP}(v_2) \cap \mathrm{LP}(v_c) \\
\subseteq&\mathrm{UP}(v_2) \cap \big(\big(\mathrm{LP}(v_c) \setminus \mathrm{LP}(v_2)\big) \cup \mathrm{LP}(v_2)\big) \\
=&\big(\mathrm{UP}(v_2) \cap (\mathrm{LP}(v_c) \setminus \mathrm{LP}(v_2))\big) \\
&\cup \big(\mathrm{UP}(v_2) \cap \mathrm{LP}(v_2)\big) \\
=&\emptyset \cup \emptyset \\
=&\emptyset.
\end{aligned}$$

□

A slight modification of Algorithm 2 can enable it to get $b^{(v)}$ for all v: at any place where an entry of M'_v is changed, we update its corresponding bits in $b^{(v)}$ accordingly. Each update can done in constant time due to the power of the unit-cost RAM model. Therefore, we have a $O(\left|\hat{T}\right|)$ time/space algorithm to preprocess the $b^{(v)}$ lists, and then subsequently find the depth of the minimum element of $\mathrm{KDS}(u, k)$ for any u and k in constant time.

3.1.2 RMQ at Target Depth

Once the $b^{(v)}$ lists are computed, we can locate the depth of the minimum weighted node in $\mathrm{KDS}(u, k)$. Let $i = \mathrm{TD}(u, k)$ be the target depth, then we can do a range minimum query (RMQ) at L_i if the nodes at L_i are sorted by their preorder number (or postorder number), because the nodes in $\mathrm{KDS}(u, k) \cap L_i$ are in consecutive positions.

Let $V_i(1), V_i(2), \ldots, V_i(|L_i|)$ represent the the sorted array of size $|L_i|$ where $V_i(j)$ represents the node in L_i with the j_{th} smallest preorder number. The orders of those nodes in the array are still the same if we sort them by their postorder numbers. Let $w_i(j) = w(V_i(j))$ for $1 \leq j \leq |L_i|$. The range minimum query should be performed on the array w_i between two indices, i.e., we need to find out the two boundary nodes $\mathrm{LD}(u, i)$ and $\mathrm{RD}(u, i)$ respectively, where node $\mathrm{LD}(u, i)$ is immediately to the left of the node that has the smallest preorder number in $\mathrm{KDS}(u, k) \cap L_i$, and node $\mathrm{RD}(u, i)$ is the node that has the biggest postorder number in $\mathrm{KDS}(u, k) \cap L_i$. Once we locate the indices j_1 and j_2 where $\mathrm{LD}(u, i) = V_i(j_1)$ and $\mathrm{RD}(u, i) = V_i(j_2)$, we can do a $\mathrm{RMQ}(j_1 + 1, j_2)$ on the array w_i to get $\min \mathrm{KDS}(u, k) \cap L_i = \min_{j_1 < j \leq j_2} w_i(j)$ in constant time. The total time and space for precomputing the data structures for RMQ is linear [4].

To compute $\mathrm{LD}(u, i)$ and $\mathrm{RD}(u, i)$ (where $i = \mathrm{TD}(u, k)$) in constant time, in the following paragraphs, we reduce the problem to the *Level Ancestor Problem* [7, 5], which can be preprocessed in linear time and space, and subsequently answer a query in constant time. The reduction is done in the preprocessing stage and uses linear time and space.

The definition of the functions $\mathrm{LD}(u, i)$ and $\mathrm{RD}(u, i)$ (restricted to $i \geq d(u)$) can be reformulated in the following way without changing its meaning: let $\mathrm{Preorder}(v)$ be the preorder number and $\mathrm{Postorder}(v)$ be the postorder number during the tree walk, then $\mathrm{LD}(u, i)$ is the node that has the largest preorder number in the set

$$\{v \mid v \in L_i \text{ and } \mathrm{Preorder}(v) < \mathrm{Preorder}(u)\ \};$$

and $\mathrm{RD}(u, i)$ is the node that has the largest postorder number in the set

$$\{v \mid v \in L_i \text{ and } \mathrm{Postorder}(v) \leq \mathrm{Postorder}(u)\ \}.$$

Note that we need to consider the special cases when $\mathrm{LD}(u, i)$ or $\mathrm{RD}(u, i)$ does not exist, which is trivial to handle.

Here, we will show how to reduce the problem to the level ancestor problem for computing $\mathrm{RD}(u, i)$, and similar approach can be applied to compute $\mathrm{LD}(u, i)$. Build an auxiliary graph for the given tree as follow: for every node v, we link an arc from v to $\mathrm{RD}(v, d(v) + 1)$. This can be done in linear time during the postorder tree walk:

- At each depth level l during the tree walk, we maintain

LastVisit(l), the last visited node on depth l. Initially, we set LastVisit(l) = $\emptyset$; after every visit of v, we set LastVisit($d(v)$) $\leftarrow v$.

- When a node v is visited, we link an arc to LastVisit($d(v)+1$). If LastVisit($d(v)+1$) is $\emptyset$, we treat it specially: RD(v, l) does not exist for $l > d(v)$.

The resulting graph is indeed a tree (denoted by T^*), because each node has only one out-going arc to a node at a deeper level. The root of this auxiliary tree is the rightmost node at the deepest level of the original tree. Denote this root by T^*_r.

A useful property of the RD function is that: for any node v and depth level l_1, l_2 where $d(v) \le l_1 \le l_2$, we have

$$\mathrm{RD}(v, l_2) = \mathrm{RD}(\mathrm{RD}(v, l_1), l_2).$$

This implies an iterative method to compute RD(v, l) for any $l \ge d(v)$, as follows:

- first, we compute $v_1 \leftarrow \mathrm{RD}(v, \mathrm{d}(v) + 1)$;
- then $v_2 \leftarrow \mathrm{RD}(v_1, \mathrm{d}(v_1) + 1)$, which is RD($v, \mathrm{d}(v) + 2$);
- we can get $v_j \leftarrow \mathrm{RD}(v_{j-1}, d(v_{j-1}) + 1)$ for $j \ge 2$, and the computed v_j is indeed RD($v, d(v) + j$).

By definition, the node RD($v, d(v) + 1$) in the tree $\hat{T}$ is the parent node of v in the auxiliary tree T^*. So the above iterative method can be directly reduced to the level ancestor problem in T^* as follows: to compute RD(u, i), we need to find the ancestor of u in T^* such that the ancestor is $i - d(u)$ levels above u in T^*. Here, the depth $d(u)$ is still computed in $\hat{T}$.

A similar approach can be used to do the reduction for LD(u, i) if a preorder tree walk is performed instead of the postorder tree walk when we build the auxiliary graph. Alternatively, a different way to achieve constant-time computation of LD(u, i) and RD(u, i) is discussed in Ben-Amram's work [3]. Combining the data structures in this subsection, we have the following theorems to solve the first subquery for KDS(u, k).

THEOREM 1. *For a tree $\hat{T}$ whose height is at most $\log n$, we can preprocess it in $O(|\hat{T}|)$ time and space to answer the* min *KDS(u, k) query in $O(1)$ time for any node $u \in \hat{T}$ and nonnegative integer k.*

THEOREM 2. *We can preprocess the tree T in $O(n)$ time and space to answer the top subquery of any* min *KDS(u, k) query in $O(1)$ time.*

3.2 The Middle Subquery

The middle subquery is $\mathrm{KDS}(u, k) \cap \bigcup_{i_a < i < i_b} B_i$. The basic idea of our solution is to compute the minimum weight of the following two parts separately and then combine them

$$\text{Part 1: } \mathrm{KDS}(u, k) \cap \bigcup_{i_a+1 \le i \le i_a+2^q} B_i;$$

$$\text{Part 2: } \mathrm{KDS}(u, k) \cap \bigcup_{i_b-2^q \le i < i_b} B_i;$$

where $q = \lfloor \log_2(i_b - i_a - 1) \rfloor$. These two parts completely cover the middle subquery, and the min element for each part can be computed efficiently by the algorithm given in this subsection.

3.2.1 Preprocessing Stage

We need to preprocess $f(v, p) = \min \mathrm{KDS}(v, 2^p \log n)$ for $v \in Z$ and $0 \le p \le \log(n/\log n) \le \log n$ where

$$Z = \bigcup_{0 \le j \le n/\log n} L_{j \log n}.$$

The preprocessing can be done by a dynamic programming approach.

First, perform a preorder tree walk to initialize $f(v, 0)$ (which is $\min \mathrm{KDS}(v, \log n)$: whenever we visit a node $v \in Z$, set $f(v, 0) \leftarrow w(v)$ and the working node $v_{\text{working}} \leftarrow v$; when we visit a node $v \notin Z$, we update

$$f(v_{\text{working}}, 0) \leftarrow \min\{f(v_{\text{working}}, 0), w(v)\}.$$

Assume that $f(v, p-1)$ is computed for all $v \in Z$, then we can let $f(v, p)$ for each $v \in Z$ equal to

$$\min\{f(v, p-1), \min_{v'} f(v', p-1)\}$$

where v' goes over all the descendants of v at depth level $d(v) + 2^{p-1} \log n$. Since all the descendants of v at that depth level occupy consecutive positions, we can build a RMQ data structure at each $j \log n$ ($0 \le j \le n/\log n$) depth level, where the underlying array element is $f(v, p-1)$, to speed up the computation. Similar to Section 3.1.2, we need to identify the boundary nodes LD(v, l) and RD(v, l) to do the RMQ query, where $l = d(v) + 2^{p-1} \log n$.

A naive interpretation of the dynamic programming above shows it will need $O(|Z| \log(n/\log n)) = O(n \log n)$ time and space. To speed this up, we will show that the actual number of relevant v, p pairs for computing and storing $f(v, p)$ are $O(n)$ by a more refined analysis. Define $Z_0 = \{v \mid v \in Z \text{ and } h(v) < \log n\}$ and $Z_1 = \{v \mid v \in Z \text{ and } h(v) \ge \log n\}$. We have $Z = Z_0 \cup Z_1$. For a node $v \in Z_0$, there is no need to compute $f(v, p)$ for $p \ge 1$, because v has no descendant at any depth level bigger than or equal to $d(v) + \log n$. Hence, the total number of relevant pairs associated with Z_0 is bounded by n. For a node $v \in Z_1$, the corresponding p can be as large as $\log(n/\log n)$, but the size of Z_1 is bounded by $O(n/\log n)$. Here is a short proof: for each $v \in Z_1$, assume that $v \in L_i$ for an i, then there are at least $\log n$ descendants of v, choose $\log n - 1$ such descendants with minimal depths (with ties broken arbitrarily) along with v to form a set A_v, where $|A_v| = \log n$; for any $v_1, v_2 \in Z_1$, we have $A_{v_1} \cap A_{v_2} = \emptyset$, therefore, $|Z_1| \cdot \log n = \sum_{v \in Z_1} |A_v| = \left|\bigcup_{v \in Z_1} A_v\right| \le n$, which is equivalent to $|Z_1| \le n/\log n$. Hence, the number of relevant v, p pairs that are associated with Z_1 is bounded by

$n/\log n \cdot \log(n/\log n) \le n$. The linear number of v, p pairs shows that the dynamic programming uses linear time and space.

3.2.2 Query Stage

Both parts of the middle subquery can be done by doing a RMQ at the pre-computed data structures. More specifically, for the first part, we identify the two boundary nodes $\mathrm{LD}(u, (i_a+1)\log n)$ and $\mathrm{RD}(u, (i_a+1)\log n)$, and let their positions in the sorted $L_{(i_a+1)\log n}$ be j_1 and j_2, then we can get the first part by

$$\min \mathrm{KDS}(u,k) \cap \bigcup_{i_a < i \le i_a + 2^q} B_i = \min_{j_1 < j \le j_2} f(V_{(i_a+1)\log n}(j),\ q).$$

Part 2 is done in a similar way: we identify the two boundary nodes $\mathrm{LD}(u, (i_b - 2^q)\log n)$ and $\mathrm{RD}(u, (i_b - 2^q)\log n)$, and let their positions in the sorted $L_{(i_b-2^q)\log n}$ be j_1 and j_2, then

$$\min \mathrm{KDS}(u,k) \cap \bigcup_{i_b - 2^q \le i < i_b} B_i = \min_{j_1 < j \le j_2} f(V_{(i_b-2^q)\log n}(j),\ q).$$

The above computations require locating four boundary nodes and two range minimum queries, all of which can be done in constant time. Finally, the result for the middle subquery can be computed by choosing the minimum of the results for those two parts. Therefore, we have the following theorem:

THEOREM 3. *For the middle subquery of KDS(u,k), we can preprocess the tree in $O(n)$ time and space to answer the subquery in $O(1)$ time.*

3.3 The Bottom Subquery

The bottom subquery is $\mathrm{KDS}(u,k) \cap B_{i_b}$. We will reduce this subquery to what we call the *Skyline RMQ* problem, for which we give a constant time query solution with linear time/space preprocessing.

3.3.1 Reduction to Skyline RMQ

The Skyline RMQ problem:

INPUT: Given a 2D array $M(x,y)$ ($1 \le x \le n_x$ and $0 \le y < n_y$) with the following property: let $\mathbb{S}' = \mathbb{S} \cup \{\top\}$ where $\top \notin \mathbb{S}$ and $s \le \top$ for any $s \in \mathbb{S}$, i.e., we add a top (or maximal) element to $\mathbb{S}$, then the 2D array must satisfy

- *Skyline Property*: for any x, y such that $M(x,y) \ne \top$, we have $M(x,y') \ne \top$ for all $0 \le y' \le y$;
- *Non-Increasing Property:* for those $M(x,y) \ne \top$ where $y > 0$, we have $M(x,y) \le M(x,y-1)$;
- $M(x,0) \ne \top$ for all $1 \le x \le n_x$.

Use $|M|$ to denote the number of non-$\top$ elements in the 2D array. Because of the skyline property, the array can be represented by a vector of variable-length vectors using only $O(|M| + n_x + n_y)$ space.

QUERY: a three sided range minimum query is asked in the form of $Q(x_1, x_2, y_1)$ ($1 \le x_1 \le x_2 \le n_x$ and $0 \le y_1 < n_y$) whose result should be

$$\min_{x_1 \le x \le x_2, 0 \le y \le y_1} M(x,y).$$

Let $i = i_b \log n$, then observing that

$$\mathrm{KDS}(u,k) \cap B_{i_b} = \bigcup_{j_1 < j \le j_2} \mathrm{KDS}(V_i(j), d(u) + k - i),$$

where j_1 and j_2 are the indices of $\mathrm{LD}(u,i)$ and $\mathrm{RD}(u,i)$ at array V_i (see 3.1.2 for the related definitions), a 2D Skyline RMQ instance can be constructed by mapping $\min \mathrm{KDS}(V_i(j), d(u) + k' - i)$ to $M(j, d(u) + k' - i)$ for $1 \le j \le |L_i|$ and $i - d(u) \le k' \le h(u)$, with all other entries to be $\top$. Obviously, the resulting 2D array satisfies the skyline property, and it also satisfies the non-increasing property because $\mathrm{KDS}(V_i(j), d(u) + k' - i) \le \mathrm{KDS}(V_i(j), d(u) + k' - 1 - i)$ (the greater the depth of the subtree, the smaller the result).

The time and space complexity of this reduction are $O(|B_{i_b}|)$, because we can do a top-down aggregation starting at depth level i. Also, the size of the resulting skyline array is $|B_{i_b}|$, i.e. $|M| = |B_{i_b}|$; and the corresponding width and height satisfy $n_x \le |B_{i_b}|$, $n_y \le |B_{i_b}|$. Therefore, if we can preprocess the Skyline RMQ problem in $O(|M| + n_x + n_y)$ time and space and do constant-time querying, then the bottom subquery of $\mathrm{KDS}(u,k)$ can also be done in constant-time with linear preprocessing for every B_i.

3.3.2 Constant-time Query

3.2.1 For a three sided range query on the skyline array, we consider two cases: in the first case, both $M(x_1, y_1)$ and $M(x_2, y_1)$ are not $\top$; in the second case, either $M(x_1, y_1)$ or $M(x_2, y_1)$ is $\top$ (or both).

For the first case, we can solve the problem by building a RMQ data structure for each level by associating the $\top$-runs with the minimum elements below the them. Here, let $M(x_1', x_2', y')$ where $x_1' \le x_2'$ represent a run of horizontal entries: $\{M(x_1', y'), M(x_1'+1, y'), \cdots, M(x_2', y')\}$; we call it a $\top$-run if all of the entries are $\top$. A maximal-length $\top$-run is defined based on a $\top$-run with the additional restriction: $M(x_1' - 1, y') \ne \top$ and $M(x_2' + 1, y') \ne \top$. Now, for each y', we build a RMQ data structures on top of an auxiliary array for $M(\cdot, y')$. The auxiliary array is built based on $M(x, y')$ ($1 \le x \le n_x$) but with each maximal-length $\top$-run $M(x_1', x_2', y')$ replaced by $\min_{x_1' \le x \le x_2', 0 \le y < y'} M(x,y)$. Each replacement can be done in constant-time in the following way:

- pre-build a **roof array** $R(\cdot)$, where
$$R(x) = \min_{0 \le y < n_y} M(x,y)$$
for $1 \le x \le n_x$; the roof array can be built in $O(|M| + n_x + n_y)$ time by a bottom-up scanning;
- for the replacement of a maximal-length $\top$-run $M(x_1', x_2', y')$, we have
$$\min_{x_1' \le x \le x_2', 0 \le y < y'} M(x,y) = \min_{x_1' \le x \le x_2'} R(x),$$
which can be computed by a range minimum query on the roof array.

The total number of maximal-length $\top$-runs in $M(\cdot,\cdot)$ is bounded by $O(|M| + n_y)$, because each maximal-length $\top$-run $M(x_1', x_2', y')$ should have a non-$\top$ entry $M(x_1' - 1, y')$ to

its left, or it is in the form of $M(1, x_2', y')$. The former $\top$-run to non-$\top$ entry mapping is one-to-one, and the number of non-$\top$ entries is $O(|M|)$; the latter forms have at most n_y instances. Combining with all the non-$\top$ entries, the sum of the sizes of the auxiliary arrays are $O(|M|+n_y)$. Hence, the preprocessing time and space complexity for the auxiliary arrays are $O(|M| + n_x + n_y)$. To answer the three sided range minimum query, we just need to locate $M(x_1, y_1)$ and $M(x_2, y_1)$'s positions in the auxiliary array for $M(\cdot, y_1)$, and do a standard range minimum query between those positions in constant time.

The second case is reduced to the first case as follows. Locate the smallest $\hat{x}_1$ such that $\hat{x}_1 \geq x_1$ and $M(\hat{x}_1, y_1) \neq \top$. Locate the biggest $\hat{x}_2$ such that $\hat{x}_2 \leq x_2$ and $M(\hat{x}_2, y_1) \neq \top$. We then have

$$Q(x_1, x_2, y_1) = \min\big\{Q(x_1, \hat{x}_1 - 1, y_1),\ Q(\hat{x}_1, \hat{x}_2, y_1),\ Q(\hat{x}_2 + 1, x_2, y_1)\big\}.$$

In the equation, $Q(\hat{x}_1, \hat{x}_2, y_1)$ is indeed the first case, while $Q(x_1, \hat{x}_1 - 1, y_1)$ and $Q(\hat{x}_2 + 1, x_2, y_1)$ can be solved by range minimum queries on the roof array due to the skyline property, i.e.,

$$Q(x_1, \hat{x}_1 - 1, y_1) = \min_{x_1 \leq x \leq \hat{x}_1 - 1} R(x);$$
$$Q(\hat{x}_2 + 1, x_2, y_1) = \min_{\hat{x}_2 + 1 \leq x \leq x_2} R(x).$$

So the question is how to locate $\hat{x}_1$ and $\hat{x}_2$ in constant time. This can be done by a construction similar to the LD and RD functions that we used in Section 3.1.2. Because locating $\hat{x}_1$ is symmetric to locating $\hat{x}_2$, here we only show how to locate $\hat{x}_1$, and locating $\hat{x}_2$ can be done in the same way.

To locate $\hat{x}_1$, we build the following data structures: for each (x, y) pair such that $M(x, y) \neq \top$, we map it to the pair $\mathrm{NT}(x, y) = (x', y + 1)$, where x' is the smallest number that satisfies $x' \geq x$ and $M(x', y+1) \neq \top$; if the x' does not exist, then set $x' = n_x + 1$. Define the iterative function $\mathrm{NT}^{(j)}$ by $\mathrm{NT}^{(j)}(x, y) = \mathrm{NT}^{(j-1)}(x, y)$ and $\mathrm{NT}^{(1)}(x, y) = \mathrm{NT}(x, y)$. Due to the skyline property, either we have $\mathrm{NT}^{(y_1)}(x_1, 0) = (\hat{x}_1, y_1)$, or $\hat{x}_1$ does not exist because of $M(x, y_1) = \top$ for $x \geq x_1$. In the latter case, $\mathrm{NT}^{(y_1)}(x_1, 0)$ will result in a pseudo-pair $(n_x + 1, y_1)$, so we can treat $\hat{x}_1 = n_x + 1$ in such a case. Note that if $\hat{x}_1 > x_2$, then we have $M(x, y_1) = \top$ for $x_1 \leq x \leq x_2$, which means that the query $Q(x_1, x_2, y_1)$ can be reduced to the range minimum query on the roof array, i.e., $Q(x_1, x_2, y_1) = \min_{x_1 \leq x \leq x_2} R(x)$.

The $\mathrm{NT}(x, y)$ mapping can be built in $O(|M| + n_x + n_y)$ time by a simple right-to-left and top-to-down scanning as shown in Algorithm 3.

The iterative calculation of $\mathrm{NT}^{(j)}$ can be reduced to the level ancestor problem [7, 5]. Build an auxiliary tree as follows: treat each pair (x,y) in the set $\{(x, y) \mid M(x, y) \neq \top\} \cup \{(n_x + 1, y) \mid 0 \leq y \leq n_y\}$ as a node of the auxiliary tree; the parent node of (x, y) is $\mathrm{NT}(x, y)$; the root of the auxiliary tree is $(n_x + 1, n_y)$. In this auxiliary tree, $\mathrm{NT}^{(j)}(x, y)$ is actually the ancestor of (x, y) at j levels above it. Since the level ancestor problem can be solved in constant time with linear preprocessing, the reduction shows that $\mathrm{NT}^{(j)}(x, y)$ can be computed in constant time with $O(|M| + n_x + n_y)$ preprocessing. Therefore, the $\hat{x}_1$ (or $\hat{x}_2$) can be located with the same complexities.

Algorithm 3 Build the $\mathrm{NT}(x, y)$ mapping.

Define $h(x)$ to be the maximum y that satisfies $M(x, y) \neq \top$.

```
1: Set LastVisit(y) ← n_x + 1 for all 0 ≤ y ≤ n_y
2: for x = n_x down to 1 do
3:    for y = h(x) down to 0 do
4:       NT(x, y) ← (LastVisit(y + 1), y + 1)
5:       LastVisit(y) ← x
6:    end for
7: end for
```

Combining the data structures for the two cases above, the Skyline RMQ problem is efficiently solved, and we have the following theorem.

THEOREM 4. *For the bottom subquery of* $\min$ *KDS*(u, k)*, we can preprocess the tree in* $O(n)$ *time and space to answer the subquery in* $O(1)$ *time.*

With Theorem 2, 3 and 4, we have solved the $\mathrm{KDS}(u, k)$ query efficiently.

THEOREM 5. *The* $\min$ *KDS*(u, k) *query can be solved in* $O(1)$ *time by preprocessing the tree in* $O(n)$ *time and space.*

4. K-RADIUS SUBTREE AGGREGATION

In this section, we give an $O(n \log n)$-space data structure to compute $\min \mathrm{KRS}(u, k)$ for any u and k in $O(\log n)$ time. The basic idea is to divide and conquer with the help of *Heavy Path Decomposition*, which was introduced by Harel and Tarjan in [11]. Our $\mathrm{KRS}(u, k)$ algorithm uses the $\mathrm{KDS}(u, k)$ algorithm discussed in the previous section as a key routine.

Given a tree T, choose one of its nodes to be the root r. Then starting at the root, we do a Heavy Path Decomposition [11] by calling Decomposition(r), which is described in Algorithm 4. In the algorithm, $C(v)$ represent the set of children of node v, and $\mathrm{CS}(v)$ represents the complete subtree rooted at v.

For each heavy path HeavyPath(v_s) created during Algorithm 4, let v_l be the leaf node in the path, we build the KDS data structure for $\mathrm{CS}(v_s)$ rooted at v_s, and then we build another KDS data structure for $\mathrm{CS}(v_s)$ rooted at v_l (i.e., make v_l the root of $\mathrm{CS}(v_s)$). This preprocessing would result in the following Lemma.

LEMMA 2. *We can compute* $\min$ *KRS*$(u, k) \cap$ *CS*(v_s) *for any* $u \in$ *HeavyPath*(v_s) *in constant time.*

PROOF. This is based on the fact that $\mathrm{KRS}(u, k) \cap \mathrm{CS}(v_s) = \mathrm{KDS}_1(u, k) \cup \mathrm{KDS}_2(u, k)$ for any $u \in \mathrm{HeavyPath}(v_s)$, where

Algorithm 4 Heavy Path Decomposition

Procedure Decomposition(v_s)

```
initialize HeavyPath(v_s) = {v_s}
set v ← v_s
while C(v) ≠ ∅ do
    let v_c be the child of v that has the largest number of
    descendants, i.e.,
    v_c = arg max_{v'∈C(v)} |CS(v')|
    HeavyPath(v_s) = HeavyPath(v_s) ∪ {v_c}
    for v'_c ∈ C(v) \ {v_c} do
        call Decomposition(v'_c)
    end for
    set v ← v_c
end while
```

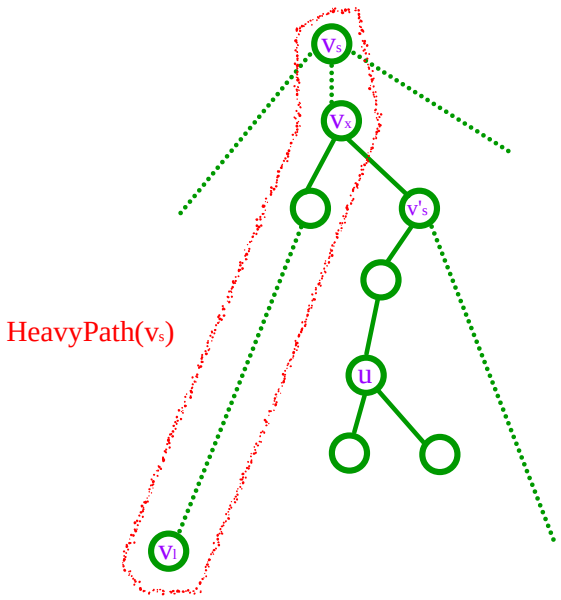


Figure 2: Labels of important nodes.

$\mathrm{KDS}_1(u,k)$ is the k-depth subtree of $\mathrm{CS}(v_s)$ with v_s as the root, and $\mathrm{KDS}_2(u,k)$ is the k-depth subtree of $\mathrm{CS}(v_s)$ with v_l as the root. Hence, two constant-time KDS queries are sufficient. □

Now, consider the general $\mathrm{KRS}(u,k)$ query within the subtree $\mathrm{CS}(v_s)$, where v_s is the root of a heavy path. There are two cases: $\mathrm{KRS}(u,k) \cap \mathrm{HeavyPath}(v_s) = \emptyset$ or $\mathrm{KRS}(u,k) \cap \mathrm{HeavyPath}(v_s) \neq \emptyset$. Here is the method to test which case applies: let v_l be the leaf node in $\mathrm{HeavyPath}(v_s)$, then we first compute the *lowest common ancestor* of u and v_l in constant time [11, 4], denote this ancestor to be v_x (see Figure 2); if $d(u,v_x) \leq k$, then the first case applies, otherwise the second case applies.

In the first case, we can recursively reduce the query to be $\mathrm{KRS}(u,k) \cap \mathrm{CS}(v'_s)$, where v'_s is the only child node of v_x such that v'_s is an ancestor of u. We can locate v'_s in constant-time by a *level ancestor query* [7, 5], i.e., v'_s is the ancestor of u at depth level $d(v_x)+1$. This is recursive because there is a heavy path starting at v'_s.

For the second case, observing that

$$\mathrm{KRS}(u,k) \cap \mathrm{CS}(v_s) = \big(\mathrm{KRS}(u,k) \cap \mathrm{CS}(v'_s)\big) \cup \big(\mathrm{KRS}\big(v_x, k-d(u,v_x)\big) \cap \mathrm{CS}(v_s)\big),$$

we can compute the query for $\mathrm{KRS}(u,k) \cap \mathrm{CS}(v'_s)$ and $\mathrm{KRS}\big(v_x, k-d(u,v_x)\big) \cap \mathrm{CS}(v_s)$ separately, and then choose the minimum of them. The minimum element of $\mathrm{KRS}(u,k) \cap \mathrm{CS}(v'_s)$ can be computed recursively like the first case, while $\mathrm{KRS}\big(v_x, k-d(u,v_x)\big) \cap \mathrm{CS}(v_s)$ can be answered in constant time by Lemma 2 because $v_x \in \mathrm{HeavyPath}(v_s)$. The recursion should stop once $d(u,v_x) = 0$ (ie., u is on the heavy path starting from the working root v_s), as there is no need to recurse.

Based on the above discussions, we can compute $\mathrm{KRS}(u,k)$ recursively: the starting case is $\mathrm{KRS}(u,k) \cap \mathrm{CS}(r)$, where r is the chosen root of the whole tree. The time and space required to preprocess the data structures is $O(n \log n)$ because the recursion depth of Algorithm 4 is $O(\log n)$ due to the property of the Heavy Path Decomposition [11]. For the same reason, the time complexity for the recursive query answering is $O(\log n)$.

THEOREM 6. *We can preprocess a tree T in $O(n \log n)$ time and space to answer the* min *KRS(u,k) query in $O(\log n)$ time for any node $u \in T$ and nonnegative integer k.*

5. CONCLUSIONS

We gave efficient data structures and algorithms for processing MIN aggregation queries on subtrees of a tree, where the query subtree is specified as a node u and integer k, and consists of the nodes that are within a distance of k from node u. Both directed (rooted tree) and undirected versions of the problem were considered. Our main result for the directed case is an $O(n)$-space data structure that can be used to compute, in constant time, the minimum element of a k-depth subtree rooted at a node u (for any u,k pair). For the undirected case, we give an $O(n \log n)$-space data structure that can be used to compute, in $O(\log n)$ time, the minimum element within a distance of k from node u. Future research will consider the case when the tree is dynamic, and also consider classes of graphs other than trees.

6. REFERENCES

[1] P. Agarwal and J. Erickson. Geometric range searching and its relatives. *Advances in Discrete and Computational Geometry, volume 23 of Contemporary Mathematics, 1–56. American Mathematical Society Press, Providence, RI, 1999.*

[2] M. J. Atallah, Y. Cho, and A. Kundu. Efficient data authentication in an environment of untrusted third-party distributors. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 696–704, 2008.

[3] A. M. Ben-Amram. The Euler path to static level-ancestors. *Unpublished manuscript.*

[4] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proceedings of the 4th Latin American Symposium on Theoretical Informatics*, pages 88–94, 2000.

[5] M. A. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004.

[6] F. Bengtsson and J. Chen. Space-efficient range-sum queries in OLAP. In Y. Kambayashi, M. K. Mohania, and W. Wöß, editors, *DaWaK*, volume 3181 of *Lecture Notes in Computer Science*, pages 87–96. Springer, 2004.

[7] O. Berkman and U. Vishkin. Finding level-ancestors in trees. *J. Comput. Syst. Sci.*, 48(2):214–230, 1994.

[8] B. Chazelle. Computing on a free tree via complexity-preserving mappings. In *25th Annual Symposium on Foundations of Computer Science, 24-26 October 1984, Singer Island, Florida, USA*, pages 358–368, 1984.

[9] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 135–143, New York, NY, USA, 1984. ACM.

[10] D. Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.

[11] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[12] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in OLAP data cubes. In J. Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 73–88. ACM Press, 1997.

[13] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In M. J. Franklin, B. Moon, and A. Ailamaki, editors, *SIGMOD Conference*, pages 133–144. ACM, 2002.

[14] J. Komlós. Linear verification for spanning trees. In *25th Annual Symposium on Foundations of Computer Science, 24-26 October 1984, Singer Island, Florida, USA*, pages 201–206, 1984.

[15] A. Kundu and E. Bertino. Secure dissemination of XML content using structure-based routing. In *EDOC*, pages 153–164, 2006.

[16] H.-G. Li, T. W. Ling, and S. Y. Lee. Range-max/min query in OLAP data cube. In M. T. Ibrahim, J. Küng, and N. Revell, editors, *DEXA*, volume 1873 of *Lecture Notes in Computer Science*, pages 467–476. Springer, 2000.

[17] W. Liang, H. Wang, and M. E. Orlowska. Range queries in dynamic OLAP data cubes. *Data Knowl. Eng.*, 34(1):21–38, 2000.

[18] S. Pettie. An inverse-ackermann style lower bound for the online minimum spanning tree. In *43rd Symposium on Foundations of Computer Science (FOCS 2002), 16-19 November 2002, Vancouver, BC, Canada, Proceedings*, pages 155–, 2002.

[19] C. K. Poon. Orthogonal range queries in OLAP. In J. V. den Bussche and V. Vianu, editors, *Database Theory - ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings*, volume 1973 of *Lecture Notes in Computer Science*, pages 361–374. Springer, 2001.

[20] R. E. Tarjan. Sensitivity analysis of minimum spanning trees and shortest path trees. *Inf. Process. Lett.*, 14(1):30–33, 1982.

[21] H. Yuan and M. J. Atallah. Efficient distributed third-party data authentication for tree hierarchies. In *Proceedings of The 28th International Conference on Distributed Computing Systems (ICDCS 2008).*