

Path Minima Queries in Dynamic Weighted Trees

Gerth Stølting Brodal¹, Pooya Davoodi^{1,2}, S. Srinivasa Rao²

¹ MADALGO*, Department of Computer Science, Aarhus University.

E-mail: {gerth,pdavoodi}@cs.au.dk

² School of Computer Science and Engineering, Seoul National University, S. Korea.

E-mail: ssrao@cse.snu.ac.kr

Abstract. In the path minima problem on a tree, each edge is assigned a weight and a query asks for the edge with minimum weight on a path between two nodes. For the dynamic version of the problem, where the edge weights can be updated, we give data structures that achieve optimal query time in the comparison and the RAM models. These structures also support inserting a node on an edge, inserting a leaf, and contracting edges. When only insertion and deletion of leaves are desired, we give data structures in the comparison and the RAM models, with optimal query time that is significantly lower than when updating the weights is allowed. We also consider the problem in the semigroup model, and show lower bounds for different variants of the problem.

1 Introduction

In this paper, we study variants of the path minima problem on weighted unrooted trees, where each edge is associated with a weight. In [12], the problem is named “Bottleneck Edge Query problem”. The problem is to maintain a data structure for a collection of trees supporting the query operation:

- `pathmin(u,v)`: return the edge with minimum weight on the path between the two nodes u and v .

Dynamic versions of the problem support various subsets of the following update operations:

- `make-tree(v)`: make a single-node tree containing the node v .
- `update(e,w)`: change the weight of the edge e to w .
- `insert(e,v,w)`: split the edge $e = (u_1, u_2)$ by inserting the node v along it. The new edge (u_1, v) has weight w , and (u_2, v) has the old weight of e .
- `insert-leaf(u,v,w)`: add the node v and the edge (u, v) with weight w .
- `contract(e)`: delete the edge $e = (u, v)$, and contract u and v to one node.
- `delete-leaf(v)`: delete both the leaf v and the edge incident to it.
- `link(u,v,w)`: add the edge (u, v) with weight w to the forest, where u and v are in two different trees.
- `cut(e)`: delete the edge e from the forest.

* Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

In this paper, we consider variants of the problem, where updates can be either to only leaf edges or to arbitrary edges. We distinguish between three types of algorithms, depending on how they access the edge weights: (1) the *comparison based* algorithms where the only allowed operations on the edge weights are comparisons; (2) the *RAM* algorithms, where any standard RAM operations are allowed on the weights; (3) the *semigroup* algorithms, in which the edge weights are from a general semigroup and queries should return the product of the weights on a path, where the product is performed using the semigroup operator (The path minima problem is the special case, where the semigroup operator is minimum). Except for the computation on the edge weights, our algorithms are in the unit-cost RAM model with word size $\Theta(\log n)$ bits.

1.1 Previous work

Static weighted trees. In the static version of the problem, neither the topology of the input tree nor the edge weights change.

The static *offline* version of the problem, where m queries are given in advance together with a tree of size n , is a generalization of the minimum spanning tree verification problem. In the comparison model, the problem can be solved in $O(m + n)$ time using $O(n)$ space [19]. See [25, 20, 10, 13] for other results on the static offline version in different models.

The static online version of the problem was considered by Chazelle [9]. He solved the problem with $O(\alpha(n))$ query time, and $O(n)$ preprocessing time and space in the semigroup model, where $\alpha(n)$ is the inverse Ackermann function. Later, it was shown that with $O(nk\alpha_k(n))$ preprocessing time and $O(n)$ space, a query can be answered in $4k$ semigroup operations [1], where $\alpha_k(n)$ is a function in the inverse Ackermann hierarchy for parameter k . In the comparison model, $4k$ query time with $O(nk \log \alpha_k(n))$ preprocessing time is achieved [23].

The above upper bounds for the online problem are almost optimal. Alon and Shieber [1] proved that to answer the query using $4k$ semigroup operations, $\Omega(n\alpha_{2k}(n))$ preprocessing time is required. In the comparison model, Pettie [23] proved that $\Omega(n \log \alpha_{2k-1}(n))$ preprocessing time is required for queries using $4k$ comparisons. When only $O(n)$ preprocessing time is allowed, an $\Omega(\alpha(n))$ query lower bound is known in both models [1, 23].

Limiting the space of the data structure also gives lower bounds for the online problem. The special case of the path minima problem, where the tree is a path is the one dimensional *range minimum query* (1D-RMQ) problem. Here, the input is an array of n elements and the query asks for the position of the minimum element within a given a range. Thus, the following two lower bounds for the 1D-RMQ also hold for the path minima problem: (1) In the semigroup model, with $O(n)$ space, $\Omega(\alpha(n))$ is required to answer the query [28]; (2) In the cell probe model, with $O(n/c)$ bits of additional space, $\Omega(c)$ query time is required [7] (here, we assume that the weights are given in a read-only array).

Cartesian trees [27] are a standard data structure to solve the 1D-RMQ problem. Interestingly, Cartesian trees can be used to solve the path minima as well to achieve $O(k)$ query time with $O(n \log^{(k)} n)$ preprocessing time in

the comparison model [5, 12], where $\log^{(k)} n$ is the k -th iterated version of the logarithm function. Note that this upper bound is worse than the one of [23].

Dynamic weighted trees. Three variants of the dynamic version of the problem have been considered in the literature. None of these results consider the operation `update`, although `link` and `cut` together can simulate `update`. In the following, we mention these results.

In the case when only leaves can be inserted and deleted, Alstrup and Holm [2] gave a linear space data structure that supports `pathmin`, `insert-leaf`, and `delete-leaf` in $O(1)$ time in the RAM model (see also [17, 12]). In the comparison model, $O(1)$ query time and $O(\log n)$ amortized insertion and deletion time with $O(n)$ space is achieved by maintaining the Cartesian tree of the input tree by *dynamic trees* of Sleator and Tarjan [12].

Another variant is *incremental trees*, where the input is a forest and the only allowed update operation is `link`. In the offline setting, where a sequence of m operations are given in advance, Tarjan [26, Section 6] solved the problem in $O((m+n) \cdot \alpha(m+n, n))$ time using $O(m+n)$ space in the semigroup model. But he only considered restricted queries and links on rooted trees, where a restricted query is between a node and the root of the tree containing that node, and a restricted link adds an edge between a root and a node in two different trees. If only the sequence of links is known in advance and not the queries, for the same restricted kind of queries and links, every operation can be performed in $O(1)$ time in the RAM model [16]. In the online setting, Alstrup and Holm [2] achieved $O(\alpha(n))$ query time and $O(1)$ amortized time for `link` using $O(n)$ space in the RAM model, for arbitrary queries and restricted links in rooted trees. Later, this result was extended to arbitrary links in unrooted trees in [17].

In their seminal paper, Sleator and Tarjan [24] presented the dynamic trees that support many operations including `link`, `cut`, `root` and `evert` all in $O(\log n)$ amortized time, in the semigroup model. The operation `root` finds the root of the tree containing a given node, and `evert` changes the root of the tree containing a given node such that the node becomes the root, by turning the tree “inside out”. Essentially, this data structure can solve all the variants of the path minima problem by using a heavy machinery that supports all the update operations.

1.2 Our results

Trees with dynamic weights (Section 3). The dynamic trees of Sleator and Tarjan support the operations `pathmin` and `update` in $O(\log n)$ amortized time in the semigroup model. We present structures for the comparison and RAM models supporting the query in $O(\log n / \log \log n)$ time. They support `update` in $O(\log n / \log \log n)$ amortized time in the RAM model, and in $O(\log n)$ amortized time in the comparison model. They also support `insert`, `insert-leaf`, `contract`, `delete-leaf` with the same update times. Furthermore (in Section 5), we show that the achieved query time, and the comparison based update time are both optimal.

Trees with leaf updates (Section 4). For the case when only the operations insert-leaf and delete-leaf are allowed to update the tree, we dynamize the static structure of [1]. In the comparison model, our structure supports queries with $4k$ comparisons and $O(k\alpha_k(n))$ amortized update time, with $O(nk\alpha_k(n))$ pre-processing time and space in the comparison model. This structure also works in the semigroup model with the same bounds. We also show another approach to obtain $O(1)$ time for all the operations in the RAM model (which was already obtained in [2, 17]).

Lower bounds (Section 5). We show reductions from the following problems to the path minima: fully dynamic connectivity on trees, boolean union-find, and 1D-RMQ. We conclude that some of our results are optimal. Table 2 in Appendix A summarizes the previous and our results.

2 Preliminaries

Inverse-Ackermann function. In the description of our structures, we utilize a variant of inverse-Ackermann function [12] denoted by α . First, we define a sequence of functions $\alpha_0(n) = n/2$ and $\alpha_k(n) = 1 + \alpha_k(\alpha_{k-1}(n))$, for $k \geq 1$. Note that $\alpha_1(n) = \log n$, $\alpha_2(n) = \log^* n$, and $\alpha_3(n) = \log^{**} n$. Indeed, $\alpha_k(n) = \log^{***} n$, where the $*$ is repeated $k - 1$ times in the superscript for $k \geq 2$. In other words, $\alpha_k(n) = \min\{j \mid \alpha_{k-1}^{(j)}(n) \leq 1\}$, where $\alpha_k^{(1)}(n) = \alpha_k(n)$, and $\alpha_k^{(j)}(n) = \alpha_k(\alpha_k^{(j-1)}(n))$ for $j \geq 2$. The inverse-Ackermann function is defined as: $\alpha(n) = \min\{k \mid \alpha_k(n) \leq 3\}$.

Transforming unrooted trees into rooted binary trees. We change the input unrooted tree to a rooted tree by designating an arbitrary node as the root. This does not affect any of the operations that our data structures support (but it affects link). Moreover, we transform this rooted tree to a binary tree using [14]: Each node u with d children is represented by a path with $\max\{1, d\}$ nodes connected by $+\infty$ weighted edges. Each child of u becomes the left child of one of the nodes. Then, every operation insert-leaf translates to two insert-leaf operations on the binary tree, and every operation insert translates to two insert operations and moving a subtree within the binary tree. See Appendix E for more details.

Micro-macro decomposition of a binary tree. We utilize the algorithm presented in [4] to partition a tree. Given a binary tree T with n nodes and a parameter x , where $1 \leq x \leq n$, we decompose the set of nodes in T into $O(n/x)$ disjoint subsets, each of size at most x , where each subset induces a subtree of T called a *micro tree*. Furthermore, the division is constructed such that at most two nodes in a micro tree are adjacent to nodes in other micro trees that are denote by *boundary nodes*. If a micro tree has two boundary nodes, then one of the nodes is the root. We define a macro tree consisting of all the boundary nodes, such that it contains an edge between two nodes if either they are in the same micro tree or there is an edge between them in T .

Lemma 1. ([4]) *Given a rooted binary tree T with n nodes and a parameter x , where $1 \leq x \leq n$. A partitioning of T into micro trees can be performed in $O(n)$ time that satisfies: (1) each micro tree contains at most x nodes, (2) there are $O(n/x)$ micro trees, and (3) each micro tree has at most two boundary nodes.*

Cartesian trees. The Cartesian tree T_C of a weighted tree T is a binary tree, where its root corresponds to the edge e of T with minimum weight, and the two children of the root correspond to the Cartesian trees of the two components made by deleting e from T [27]. The internal nodes of T_C are the edges of T , and the leaves of T_C are the nodes of T . The query `pathmin` can be answered using T_C and an LCA structure for T_C . From [12], we have the following lemma on how to maintain a Cartesian tree under inserting leaves. See Appendix D for more details.

Lemma 2. ([12]) *The Cartesian tree of a tree with n nodes can be maintained in a data structure of size $O(n)$ that can be constructed in $O(n \log n)$ time, and supports `pathmin` in $O(1)$ time and `insert-leaf` in $O(\log n)$ time.*

3 Data structures for dynamic weights

In this section, we present two data structures for the path-minima problem that support: `pathmin`, `update`, `insert`, `insert-leaf`, and `contract`. The first data structure is in the comparison model and achieves $O(\log n / \log \log n)$ query time, $O(\log n)$ time for `update`, and $O(\log n)$ amortized time for `insert`, `insert-leaf`, and `contract`. The second data structure is in the RAM model and achieves $O(\log n / \log \log n)$ for all the above operations using Q-heaps [15] (see also Appendix B). Both the structures are similar to the ones in [17]. In the following, we first describe the comparison based structure, and then how to convert it to the RAM structure.

3.1 Comparison based structure

Decomposition. The input binary tree T is decomposed into micro trees using Lemma 1, such that each micro tree has size $O(\log^\varepsilon n)$ and at most two boundary nodes. Each micro tree is contracted to a single node. A new tree T'_1 of size $O(n / \log^\varepsilon n)$ is built containing one node for each contracted micro tree. If there is an edge in T between two micro trees, then there is an edge in T'_1 between the nodes corresponding to those micro trees. The weight of this edge in T'_1 is the minimum weight along the path between the root of the child micro tree and the root of the parent micro tree (see Fig. 3 in Appendix G). We let T_1 be a binarized version of T'_1 (see Section 2). The decomposition continues recursively on T_1 . In level i , the tree T_{i-1} is decomposed, and the tree T_i is built, for $i = 1, \dots, \ell$, where T_0 denotes T and ℓ is the number of recursive levels. The size of the micro trees in all the levels and also the size of T_ℓ is $O(\log^\varepsilon n)$, for some constant ε , where $0 < \varepsilon < 1$. The number of recursive levels, ℓ , is $O(\log n / \log \log n)$.

Data structure. The data structure consists of the following parts:

- We explicitly store all the trees T_0, \dots, T_ℓ .
- For each node in T_i , we store a pointer to the micro tree of T_i containing that node, and the local ID (insertion time) in the micro tree.
- We represent each micro tree μ with the tuple $(s_\mu, p_\mu, r_\mu, |\mu|)$ of size $o(\log n)$ bits, where s_μ , p_μ , and r_μ are arrays defined as follows. The array s_μ is the binary encoding of the topology of μ . The array p_μ maintains the local IDs of the nodes within μ , and enables us to find a given node inside μ . The array r_μ maintains the rank of the edge-weights according to the preorder traversal of μ .
- For each micro tree, we store a balanced binary search tree containing all the weights of the micro tree. This allows us to find the rank of a new weight within the micro tree under insertion in $O(\log(\log^\epsilon n))$ time.
- For each micro tree μ of T_i , we store an array of pointers that point to the original nodes in T_i given the local IDs.

Tables. We use precomputed tables to perform each of the following operations within a micro tree μ : `pathmin`, `update`, `insert`, `insert-leaf`, `contract`, `LCA`, `root` and `child-ancestor`, where `root` returns the local ID of the root of μ ; `LCA` returns the local ID of the lowest common ancestor of two given nodes in μ ; and `child-ancestor`(u, v) returns the local ID of the child of u that is also an ancestor of v (if such a child does not exist, returns null). Tables are indexed by the micro tree representation $(s_\mu, p_\mu, r_\mu, |\mu|)$ and the arguments of the corresponding operation. To perform `update`, `insert`, and `insert-leaf` within μ , we find the rank of the new weight among the edge-weights of μ using its balanced binary search tree in $O(\log |\mu|) = O(\log \log n)$ time. This rank becomes an index for the corresponding tables. The following lemma shows that the operations can be supported using the tables of size $o(n)$ bits (the proof is in Appendix F).

Lemma 3. *Within a micro tree of size $O(\log^\epsilon n)$, we can support `pathmin`, `LCA`, `root`, `child-ancestor`, and moving a subtree inside the tree in $O(1)$ time. The operations `update`, `insert`, `insert-leaf`, and `contract` can be supported in $O(\log \log n)$ time using the balanced binary search tree of the micro tree and precomputed tables of total size $o(n)$ bits that can be constructed in $o(n)$ time.*

Query. The query `pathmin`(u, v) can be answered using the precomputed tables, if u and v are in the same micro tree in T . When u and v are not in the same micro tree, we divide the query into subqueries according to our recursive decomposition as follows. Let c be the LCA of u and v in T . There are three micro trees in T that each one contains one of u , v , and c . We solve the parts of the query that are within each of these three micro trees. In the next level, we consider three micro trees of T_1 , each one contains one of the three nodes corresponding to the three contracted micro trees that we considered in the previous level. Then, we solve the remaining parts of the query that are within these three micro trees. This query algorithm continues for $k \leq \ell$ levels, until the two micro trees containing u and v are in the same micro tree (see Fig. 4 in Appendix G). In our implementation, we first compute the LCA node of each level when we return from the previous level. In this way, we can avoid

to construct an LCA structure for each T_i . In each level, the three subqueries within the micro trees can be answered in $O(1)$ time using Lemma 3. Thus, we achieve $O(\log n / \log \log n)$ query time.

Update. We perform $\text{update}(e, w)$ by updating the data structure in all the ℓ levels. W.l.o.g. assume that $e = (u, v)$, where u is the parent of v . Let μ be the micro tree in T_0 that contains v . We start to update from the first level, where the tree is T : (1) Update the weight of e in T . (2) If v is not the root of μ , then we update μ using Lemma 3. If v is the root of μ , i.e., e connects μ to its parent micro tree, we do not need to update any micro tree. (3) Perform $\text{check-update}(\mu)$ which recursively updates the edge-weights in T_1 between μ and its child micro trees as follows. We check if pathmin along the path between the root of μ and the root of each child micro tree of μ needs to be updated. We can check this using pathmin within μ . If this is the case, for each one, we go to the next level and perform the three-step procedure on T_1 recursively. Since each micro tree has at most one boundary node that is not the root, then at most one of the child micro trees of μ can propagate the update to the next level, and therefore the number of updates does not grow exponentially. Step 2 takes $O(\log \log n)$ time, and thus update takes totally $O(\log n)$ time in the worst case.

Insertion. We perform $\text{insert}(e, v, w)$ using a three-step procedure similar to update . Let μ be the micro tree in T that contains u_2 , where $e = (u_1, u_2)$ and u_1 is the parent of u_2 . We start from the first level, where the tree is T : (1) To handle insert in the transformed binary tree, we first insert v along e in μ . Note that if u_2 is the root of μ , then v is inserted as the new root of μ . This can be done in $O(\log \log n)$ time using Lemma 3. (2) If $|\mu|$ exceeds the maximum limit $3 \log^\epsilon n$, then we split μ into $k \leq 4$ new micro trees, each of size at most $2 \log^\epsilon n + 1$ (see Appendix C). These k micro trees are contracted to nodes that should be in T_1 . One of the new micro trees that contains the root of μ corresponds to the node that is already in T_1 for μ . The other $k - 1$ new micro trees are contracted and inserted into T_1 with appropriate edge-weights, using insert recursively. Let μ' be the new micro tree that contains the boundary node of μ which is not the root of μ . We perform $\text{check-update}(\mu')$ to recursively update the edge weights in T_1 between μ' and its child micro trees. (3) Otherwise, i.e., if $|\mu|$ does not exceed the maximum limit, we do $\text{check-update}(\mu)$ to recursively update the edge weights in T_1 between μ and its child micro trees, which takes $O(\log n)$ time.

To perform $\text{insert-leaf}(u, v, w)$, we use the algorithm of insert with the following changes. In step (1), we insert v as a child of u . This can be done in $O(\log \log n)$ time. The step (3) is not required.

A sequence of n insertions into T_0 , can at most create $O(n / \log^\epsilon n)$ micro trees (since any created micro tree needs at least $\log^\epsilon n$ node insertions before it splits again). Since the number of nodes in T_0, T_1, \dots, T_ℓ is geometrically decreasing, the total number of micro tree splits is $O(n / \log^\epsilon n)$. Because each micro tree split takes $O(\log^\epsilon n)$ time, the amortized time per insertion is $O(1)$ for handling micro splits. Thus, both insert and insert-leaf can be performed in $O(\log n)$ amortized time.

Edge contraction. We perform `contract(e)` by marking v as contracted and updating the weight of e to ∞ by performing `update`. When the number of marked edges exceeds half of all the edges, we build the whole structure from scratch using `insert-leaf` for the nodes that are not marked and the edges that do not have weight of ∞ . Thus, the amortized deletion time is the same as insertion time.

Theorem 1. *There exists a dynamic path minima data structure for an input tree of n nodes in the comparison model, supporting `pathmin` in $O(\log n / \log \log n)$ time, `update` in $O(\log n)$ time, `insert`, `insert-leaf`, and `contract` in $O(\log n / \log \log n)$ amortized time using $O(n)$ space.*

3.2 RAM structure

We exploit the edge-weights in the RAM model to improve the `update` time to $O(\log n / \log \log n)$. The bottleneck in our comparison based data structure is that we maintain a balanced binary search tree for the edge-weights within a micro tree. This search tree is used to find the rank of a weight among the weights that are currently in the micro tree. The search for a weight in this search tree takes $O(\log \log n)$ time. Instead of this search tree, in the RAM model, we can maintain the edge weights of the micro tree in a Q-heap [15] to find the rank of a weight in $O(1)$ time (see Lemma 5 in Appendix B). We achieve the following data structure.

Theorem 2. *There exists a dynamic path minima data structure for an input tree of n nodes in the RAM model, which supports `pathmin` and `update` in $O(\log n / \log \log n)$ time, and `insert`, `insert-leaf` and `contract` in $O(\log n / \log \log n)$ amortized time using $O(n)$ space.*

4 Data structures for leaf updates with static weights

In this section, we consider data structures that support path minima under inserting and deleting leaves in the semigroup model (and therefore, in the comparison model) and the RAM model. The modification of existing edge weights is not allowed.

4.1 Optimal semigroup structure

Alon and Schieber [1] presented two static data structures for the problem that achieve optimal query time in the semigroup model. We observe that their structures can be made dynamic. The following theorems summarize the achieved performance (the proofs are in Appendix H)

Theorem 3. *There exists a path minima data structure in the semigroup model using $O(nk\alpha_k(n))$ space, that supports `pathmin` using $2k$ semigroup operations, and performs `insert-leaf` and `delete-leaf` in amortized $O(k\alpha_k(n))$ semigroup operations, for a parameter k , where $k \geq 1$.*

To improve the space and update time, we decompose the input tree into micro trees of size $O(\alpha^2(n))$ using the micro-macro decomposition. The macro tree is represented using the structure of Theorem 3 with $k = \alpha(n)$. By decomposing each of the micro trees to smaller micro trees of size $O(\alpha(n))$ with a micro tree of size $O(\alpha(n))$, we achieve the following structure.

Theorem 4. *There exists a path minima data structure in the semigroup model using $O(n)$ space, that supports `pathmin` in $O(\alpha(n))$ time, `insert-leaf` and `delete-leaf` in amortized $O(1)$ time.*

4.2 RAM structure

We also present a data structure in the RAM model that supports the path minima under inserting and deleting leaves. Although this structure does not give a new result (due to [2, 17]) but is a new approach to solve the problem.

We decompose the tree into *small trees* of size $O(\log n)$ and each small tree into micro trees of size $O(\log \log n)$ using the micro-macro decomposition (see Section 2). Decomposing into small trees generates a macro-macro tree of size $O(n/\log n)$, and decomposing the small trees generates $O(n/\log n)$ macro trees, each of size $O(\log n/\log \log n)$. The operations within each micro tree is supported using precomputed tables and Q-heaps. We do not store any representation for the small trees. We represent the macro-macro tree and each macro tree with a Cartesian tree (see Section 2 and Appendix I).

The query can be solved in $O(1)$ time by dividing it according to the three levels of the decomposition. The new leaves are inserted into the micro trees. When the size of a micro tree exceeds its maximum limit, we split it, and insert the new boundary nodes into the appropriate macro tree. Our main observation is the following.

Lemma 4. *When a micro tree is split, we can insert the new boundary nodes by performing `insert-leaf` using the Cartesian tree of the corresponding macro tree.*

By representing the Cartesian trees by the comparison based structure of [12], with $O(1)$ query time and logarithmic leaf insertion and deletion time, Lemma 4 allows us to achieve the following.

Theorem 5. *There exists a dynamic path minima data structure for an input tree of n nodes using $O(n)$ space that supports `pathmin` in $O(1)$ time, and supports `insert-leaf` and `delete-leaf` in amortized $O(1)$ time.*

5 Lower bounds

In this section, we show some lower bounds for both the query time and update time of different variants of the problem by giving reductions from other problems. The parameters t_q , t_u , t_l , and t_c denote the running time of the operations `pathmin`, `update`, `link`, and `cut` respectively. Let $t = \max\{t_u, t_l, t_c\}$.

In the cell probe model, we prove that if we want to support link and cut in a time within a constant factor of the query time, then $t_q = \Omega(\log n)$. Moreover, if we want a fast query time $t_q = o(\log n)$, then one of link or cut cannot be supported in $O(\log n)$ time, e.g., if $t_q = O(\log n / \log \log n)$, then $t = \Omega(\log^{1+\varepsilon} n)$ for some $\varepsilon > 0$. We also show that $O(\log n / \log \log n)$ query time is the best achievable for polylogarithmic update time, e.g., a faster query time $O(\log n / (\log \log n)^2)$ causes t to blow-up to $(\log n)^{\Omega(\log \log n)}$. In the semi-group and the comparison models, we also show two trivial reductions for the variant that only update is required.

Supporting link and cut. In the following, we reduce the *fully dynamic connectivity* and *boolean union-find* problems to the path minima problem with link and cut.

- The fully dynamic connectivity problem on forests is defined as follows. We have to maintain a forest of undirected trees under three operations **connect**, **link**, and **cut**, where **connect**(x, y) returns true if there exists a path between the nodes x and y , and returns false otherwise. Let t_{con} be the running time of **connect**, and t_{update} be the maximum of the running times of link and cut. Patrascu and Demaine [22] proved the lower bound $t_{\text{con}} \log(2 + t_{\text{update}}/t_{\text{con}}) = \Omega(\log n)$ in the cell probe model.

This problem is reduced to the path minima as follows. We put a dummy root r on top of the forest, and connect r to an arbitrary node of each tree with an edge of weight $-\infty$. Thus the forest becomes a tree. For this tree, we construct a path minima data structure. The answer to **connect**(x, y) is false iff the answer to **pathmin**(x, y) is an edge of weight $-\infty$. To perform **link**(x, y), we first run **pathmin**(x, r) to find the edge e of weight $-\infty$ on the path from r to x . Then we remove e and insert the edge (x, y) . To perform **cut**(x, y), we first run **pathmin**(x, r) to find the edge e of weight $-\infty$. Then we change the weight of e to zero, and the weight of (x, y) to $-\infty$. Now, by performing **pathmin**(x, r), we figure out that x is connected to r through y , or y is connected to r through x . W.l.o.g. assume that x is connected to r through y . Therefore, we delete the edge (x, y) , insert (x, r) with weight $-\infty$, and change the weight of e back to $-\infty$.

Thus, we obtain the trade-off $t_q \log \frac{t_q+t}{t_q} = \Omega(\log n)$. From this, we e.g., conclude that if $t_q = O(\log n / \log \log n)$, then $t = \Omega(\log^{1+\varepsilon} n)$, for some $\varepsilon > 0$. We can also show that if $t = O(t_q)$, then $t_q = \Omega(\log n)$.

- The boolean union-find problem is maintaining a collection of disjoint sets under the following operations: **find**(x, A): returns true if $x \in A$, and returns false otherwise; **union**(A, B): returns a new set containing the union of the disjoint sets A and B . Kaplan et al. [18] proved the trade-off $t_{\text{find}} = \Omega(\frac{\log n}{\log t_{\text{union}}})$ for this problem in the cell probe model, where t_{find} and t_{union} are the running time of **find** and **union**.

The incremental connectivity problem is the fully dynamic connectivity problem without the operation cut. The boolean union-find problem is triv-

ially reduced to the incremental connectivity problem. The incremental connectivity problem is reduced to the path minima problem with the same reduction used above.

Therefore, we obtain $t_q = \Omega(\frac{\log n}{\log(t_q+t)})$. We can conclude that when $t_q = O(\log n/(\log \log n)^2)$, slightly less than $O(\log n/\log \log n)$, then the running time of t blows-up to $(\log n)^{\Omega(\log \log n)}$.

Supporting only update. Since the 1D-RMQ problem is a special case of the path minima problem, the following three lower bounds apply to the path minima problem, when the operation **update** is required.

1. Alstrup et al. [3, Section 2.2] proved that on a cell probe model with word size b bits, 1D-RMQs on an array of size n require $\Omega(\log n/\log(t_u b \log n))$ time, where t_u is the time to update an entry of the array. This implies, e.g., that update time $(\log n)^{O(1)}$ implies query time $\Omega(\log n/\log \log n)$.
2. In the comparison model, Brodal et al. [6] proved the following lower bound for maintaining the minimum of a dynamic set (i.e., the 1D-RMQ problem with queries only for the complete array): if insertions and deletions perform at most t_u comparisons then find-min queries require at least $n/(e^{2t_u} - 1)$ comparisons. This implies, e.g., if minimum queries (and therefore 1D-RMQ and path minima queries) use $(\log n)^{O(1)}$ comparisons then updates require $\Omega(\log n)$ comparisons.
3. For the generalization to the semigroup model, where a query returns the product of the elements of a subarray in time t_q , Pătraşcu and Demaine [22] proved the lower bounds $t_q \log(t_u/t_q) = \Omega(\log n)$ and $t_u \log(t_q/t_u) = \Omega(\log n)$, e.g., implying that with update time $O(\log n)$, queries would require time $\Omega(\log n)$ and vice versa.

References

1. N. Alon and B. Schieber. Optimal preprocessing for answering on-line product queries. Technical report, Department of Computer Science, School of Mathematical Sciences, Tel Aviv University, 1987.
2. S. Alstrup and J. Holm. Improved algorithms for finding level ancestors in dynamic trees. In *Proc. 27th International Colloquium on Automata, Languages and Programming*, pages 73–84. Springer-Verlag, 2000.
3. S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. In *Proc. 39th Annual Symposium on Foundations of Computer Science*, page 534, Washington, DC, USA, 1998. IEEE Computer Society.
4. S. Alstrup, J. Secher, and M. Spork. Optimal on-line decremental connectivity in trees. *Information Processing Letters*, 64(4):161–164, 1997.
5. P. Bose, A. Maheshwari, G. Narasimhan, M. Smid, and N. Zeh. Approximating geometric bottleneck shortest paths. *Computational Geometry*, 29(3):233–249, 2004.
6. G. S. Brodal, S. Chaudhuri, and J. Radhakrishnan. The randomized complexity of maintaining the minimum. *Nordic Journal of Computing*, 3(4):337–351, 1996.

7. G. S. Brodal, P. Davoodi, and S. S. Rao. On space efficient two dimensional range minimum data structures. In *Proc. 18th Annual European Symposium on Algorithms*, volume 6347 of *LNCS*, pages 171–182. Springer-Verlag, 2010.
8. B. Chazelle. A theorem on polygon cutting with applications. In *Proc. 23rd Annual Symposium on Foundations of Computer Science*, pages 339–349, 1982.
9. B. Chazelle. Computing on a free tree via complexity-preserving mappings. *Algorithmica*, 2:337–361, 1987.
10. B. Chazelle and B. Rosenberg. The complexity of computing partial sums off-line. *Int. J. Comput. Geometry Appl.*, 1(1):33–45, 1991.
11. R. Cole and R. Hariharan. Dynamic lca queries on trees. *SIAM Journal on Computing*, 34(4):894–923, 2005.
12. E. D. Demaine, G. M. Landau, and O. Weimann. On cartesian trees and range minimum queries. In *Proc. 36th International Colloquium on Automata, Languages and Programming*, volume 5555 of *LNCS*, pages 341–353. Springer-Verlag, 2009.
13. B. Dixon, M. Rauch, and R. E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM J. Comput.*, 21(6):1184–1192, 1992.
14. G. N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 14(4):781–798, 1985.
15. M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533–551, 1994.
16. D. Harel. A linear time algorithm for finding dominators in flow graphs and related problems. In *Proc. 17th Annual ACM Symposium on Theory of Computing*, pages 185–194. ACM Press, 1985.
17. H. Kaplan and N. Shafrir. Path minima in incremental unrooted trees. In *Proc. 16th Annual European Symposium on Algorithms*, volume 5193 of *LNCS*, pages 565–576. Springer-Verlag, 2008.
18. H. Kaplan, N. Shafrir, and R. E. Tarjan. Meldable heaps and boolean union-find. In *Proc. 34th annual ACM symposium on Theory of computing*, pages 573–582. ACM Press, 2002.
19. V. King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18(2):263–270, 1997.
20. J. Komlós. Linear verification for spanning trees. *Combinatorica*, 5(1):57–65, 1985.
21. D. M. Neto. *Efficient cluster compensation for lin-kernighan heuristics*. PhD thesis, University of Toronto, Toronto, Ontario, Canada, Canada, 1999.
22. M. Pătraşcu and E. D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing*, 35(4):932–963, 2006.
23. S. Pettie. An inverse-ackermann type lower bound for online minimum spanning tree verification. *Combinatorica*, 26(2):207–230, 2006.
24. D. Sleator and R. Endre Tarjan. A data structure for dynamic trees. *Journal of computer and system sciences*, 26(3):362–391, 1983.
25. R. Tarjan. Complexity of monotone networks for computing conjunctions. *Algorithmic aspects of combinatorics*, page 121, 1978.
26. R. E. Tarjan. Applications of path compression on balanced trees. *J. ACM*, 26(4):690–715, 1979.
27. J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.
28. A. C.-C. Yao. Space-time tradeoff for answering range queries (extended abstract). In *Proc. 14th annual ACM symposium on Theory of computing*, pages 128–136. ACM Press, 1982.

Appendix

A Tables of previous and new results

Table 1. Results for the static version of the problem in both the offline and online settings. The lower bounds should be read like a conditional sentence, e.g., the result from [23] states that if the preprocessing time is $O(n)$, the query time $\Omega(\alpha(n))$ is required. The size of the input tree is denoted by n , and k and c are arbitrary parameters, where $1 \leq k \leq \alpha(n)$ and $1 \leq c \leq n$. In the last row of the table, t denotes the number of bits required to explicitly store the input tree. Note that the semigroup lower bounds of [10, 1] are in stronger models.

Ref.	Time		Space	Model	
Offline for m queries					
[20]	$O(m+n)$ comparisons		–	Comparison	
[19]	$O(m+n)$		$O(n)$	Comparison	
[13]	$O(m+n)$		$O(n)$	RAM	
[25]	$\Omega((m+n) \cdot \alpha(m+n, n))$		–	Semigroup	
[10]	$\Omega((m+n) \cdot \alpha(m+n, n))$		–	Faithful Semigroup	
Online					
Ref.	Preprocessing time	Query time	Space	Update time	Model
[9]	$O(n)$	$O(\alpha(n))$	$O(n)$	-	Semigroup
[1]	$O(n \cdot k \cdot \alpha_k(n))$	k	$O(n \cdot k \cdot \alpha_k(n))$	-	Semigroup
[12]	$O(n \log^{(k)} n)$	$O(k)$	$O(n)$	-	Comparison
[28]	–	$\Omega(\alpha(n))$	$O(n)$	-	Semigroup
[1]	$\Omega(n \cdot \alpha_k(n))$	k	-	-	Commutative Semigroup
[1]	$\Omega(n)$	$O(\alpha(n))$	-	-	Commutative Semigroup
[23]	$O(n)$	$\Omega(\alpha(n))$	-	-	Comparison
[23]	$O(n \cdot \alpha_k(n))$	$\Omega(k)$	-	-	Comparison
[7]	-	$\Omega(c)$	$O(n/c) + t$ bits	-	RAM

Table 2. Results for the dynamic versions of the problem. The lower bounds should be read like a conditional sentence, e.g., the result from [6] states that if the query time is $\log^{O(1)} n$, then $\Omega(\log n)$ update time is required. The size of the input tree is denoted by n , the size of the universe, for integer weights, is denoted by u , and k is an arbitrary parameter, where $1 \leq k \leq \alpha(n)$.

Ref.	Preprocessing time	Query time	Space	Update time	Model
insert-leaf, delete-leaf					
Theorem 3	$O(n \cdot k \cdot \alpha_k(n))$	$2k$	$O(n \cdot k \cdot \alpha_k(n))$	$O(k \cdot \alpha_k(n))$	Semigroup, Comparison
Theorem 4	$O(n)$	$O(\alpha(n))$	$O(n)$	$O(1)$	Semigroup, Comparison
[12]	$O(n \log n)$	$O(1)$	$O(n)$	$O(\log n)$	Comparison
[12]	$O(n \log n)$	$O(1)$	$O(n)$	$O(\log \log u)$	RAM
[2, 17], Theorem 5	$O(n)$	$O(1)$	$O(n)$	$O(1)$	RAM
update, insert, insert-leaf, contract					
Theorem 1	$O(n)$	$O(\log n / \log \log n)$	$O(n)$	$O(\log n)$	Comparison
Theorem 2	$O(n)$	$O(\log n / \log \log n)$	$O(n)$	$O(\log n / \log \log n)$	RAM
[22]	-	$\Omega(\log n)$	-	$O(\log n)$	Semigroup
[22]	-	$O(\log n)$	-	$\Omega(\log n)$	Semigroup
[6]	-	$\log^{O(1)} n$	-	$\Omega(\log n)$	Comparison
[3, Section 2.2]	-	$\Omega(\log n / \log \log n)$	-	$\log^{O(1)} n$	RAM
link					
[2, 17]	$O(n)$	$O(\alpha(n))$	$O(n)$	$O(1)$	RAM
link, cut					
[24, Section 5]	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$	Semigroup

B Q-heap

Fredman and Willard [15] presented a dynamic data structure called Q-heap which, for a small enough set of integers, supports the operations search (unsuccessful search returns the predecessor), rank (which returns the number of elements in the set that are less than the query element), insert, and delete in constant time in the RAM model. The following lemma summarizes this data structure.

Lemma 5. ([15]) *For any integer $n < 2^b$, and a set of m b -bit integers, where b is the word size of the machine and m is at most $(\log n)^{1/4}$, there exists a data structure of size $O(m)$ supporting insertion, deletion, and rank operations in $O(1)$ worst case time using a lookup table of size $O(n)$ constructed in $O(n)$ time.*

C Splitting a tree

Given a binary tree T with n nodes and at most two boundary nodes, we decompose the set of nodes of T into at most four disjoint subsets inducing four

subtrees of T in $O(n)$ time, where each subtree has at most $1 + 2n/3$ nodes, and at most two boundary nodes including the old boundary nodes. Note that T can also be decomposed by using Lemma 1, but then the old boundary nodes do not necessarily remain as the boundary nodes of the newly created subtrees. Therefore, we present another algorithm for this decomposition as follows.

We first find a *centroid edge* e of T , i.e., an edge whose removal partitions T into two trees of size at most $1 + 2n/3$ each. It is well-known that for a given non-empty binary tree, such an edge exists and can be found in $O(n)$ time [8]. We remove e from T to obtain two components T_1 and T_2 , each of size at most $1 + 2n/3$. We should maintain the property that each component has at most two boundary nodes. Let c_1 and c_2 be the incident nodes of e , existing in T_1 and T_2 respectively. There are three cases based on the location of the boundary node(s) of T : (1) Let b be the only boundary node of T . Assume w.l.o.g. that b is in T_1 . Then T_1 is a micro tree with two boundary nodes b and c_1 , and T_2 is a micro tree with one boundary node c_2 . (2) If nodes b_1 and b_2 are the two boundary nodes of T , and if b_1 is in one component and b_2 is in the other component, then each component has two boundary nodes. (3) If nodes b_1 and b_2 are the two boundary nodes of T , and if w.l.o.g. both of b_1 and b_2 are in T_1 . The component T_2 has one boundary node c_2 . The component T_1 has three boundary nodes c_1 , b_1 , and b_2 . Then we split T_1 into three components as follows. Let $(c_1, \dots, b, x_1, \dots, b_1)$ be the path from c_1 to b_1 , and $(c_1, \dots, b, x_2, \dots, b_2)$ be the path from c_1 to b_2 , where b is the last common node in these two paths. We remove the edges (b, x_1) and (b, x_2) which partition T_1 into three components. Each component has exactly two boundary nodes which are the pairs (b_1, x_1) , (b_2, x_2) , and (c_1, b) .

D More on Cartesian trees

An example of the Cartesian tree of a tree is shown in Fig. 1. The path minima query between two nodes u and v in T corresponds to finding the LCA of the leaves u and v in the Cartesian tree of T . It has been proved that an $O(n)$ space data structure to maintain the Cartesian tree of a tree with n nodes, can be constructed in $O(n \log n)$ time and comparisons to support path minima queries in $O(1)$ time ([21, Section 3.3.2], [5, Section 2], and [12, Theorem 2]).

Since the construction of the Cartesian tree of a star tree (i.e., a tree where one node is adjacent to all remaining nodes) corresponds to sorting the edges, it follows that an explicit construction of a Cartesian tree will require $\Omega(n \log n)$ comparisons [12].

In Fig. 2, we show how to maintain a Cartesian tree T_C of a tree T with n nodes under the insertion of a new leaf in T in $O(\log n)$ worst case time. Recall that every edge of T is an internal node in T_C , and every node (including leaves) of T is a leaf in T_C . Let ℓ be a new leaf which we want to insert into T as a new child of v . Therefore, v is a leaf and the edge $e = (\ell, v)$ is a new node in T_C . In T_C , the edge e should be inserted in the path from v to the root such that the weights along the path remain heap ordered, and ℓ becomes a child of e . Using

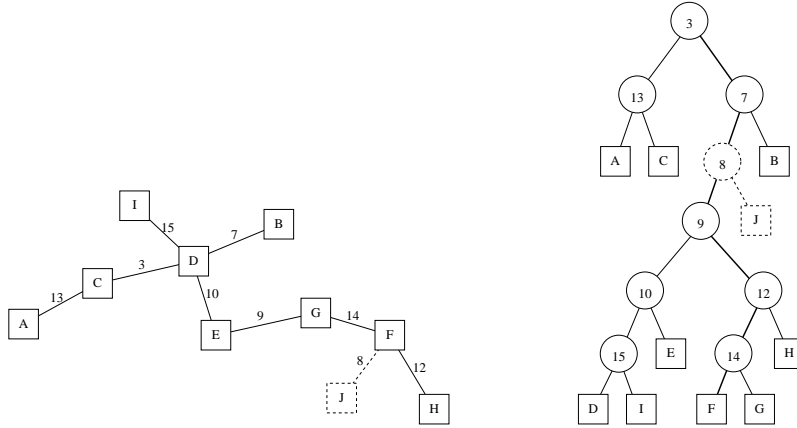


Fig. 1. A tree T (left) and the Cartesian tree of T (right). Inserting the new leaf J as a neighbor of F with an edge of weight 8 in T causes 8 to be inserted as a node in the Cartesian tree on the path from F to the the root such that heap order is preserved.

link-cut trees of Sleator and Tarjan [24] to represent T_C , the position of e in the path from v to the root of T_C can be computed in $O(\log n)$ worst case time.

E Transforming trees into rooted binary trees

We transform a rooted tree to a binary tree using the following well-known transformation (see also [14]). For every node u in the tree with d children v_1, v_2, \dots, v_d , where $d \geq 2$, we represent u by d nodes w_1, w_2, \dots, w_d . We make w_{i+1} the left child of w_i , v_i the right child of w_i , and replace u by w_i as the child of the parent of u . The left child of w_d is empty, which acts as a place holder for inserting new children for u .

The operation $\text{insert-leaf}(u, v, w)$ can be performed by doing $\text{insert-leaf}(w_d, w_{d+1}, \infty)$ for the left child of w_d , and $\text{insert-leaf}(w_{d+1}, v, w)$ for the right child of w_{d+1} . The operation $\text{insert}(e, v, w)$ can be performed by a single insert operation.

F Proof of Lemma 3

Proof. Let μ be the micro tree. The size of the lookup table used to perform pathmin is analyzed as follows. Each entry of the table is a pointer to an edge of μ which can be stored using $O(\log \log n)$ bits. The index to the table consists of (i) $(s_\mu, p_\mu, r_\mu, |\mu|)$, and (ii) two indexes in the range $[1 \cdots |\mu|]$ which represent two pointers to query nodes. The number of different arrays s_μ is $2^{|\mu|}$. The number of different arrays p_μ and r_μ is $O(|\mu|!)$. Therefore, the table is stored in $O(2^{|\mu|} \cdot (|\mu|!) \cdot (|\mu|^3) \cdot (\log |\mu|)) = o(n)$ bits.

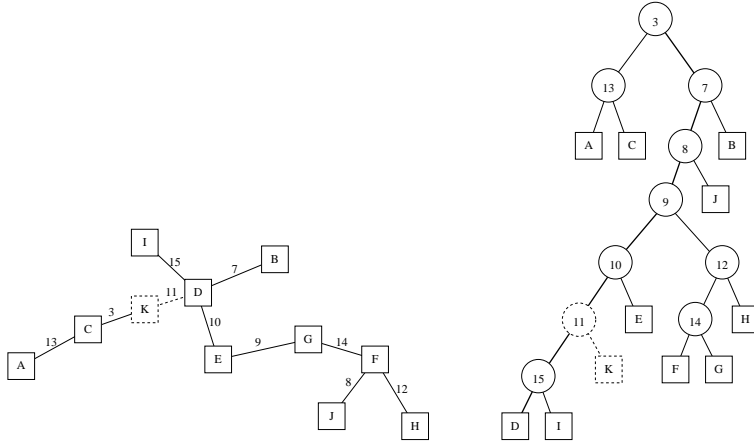


Fig. 2. In the tree T (left), a new node K is inserted on the edge (C,D) with weight 3. The new edge (K,D) has weight $11 \geq 3$. In the Cartesian tree T_C (right), the new edge with weight 11 is inserted as a node on the path from D to the node $LCA(C,D)=3$ such that heap order is preserved.

In the lookup table used for update-weight, each entry is an array r_μ which maintains the rank of the edge-weights of μ after updating a weight. The index to the table consists of (i) $(s_\mu, p_\mu, r_\mu, |\mu|)$, (ii) an index in the range $[1 \cdots |\mu|]$ to an edge to be updated, and (iii) the rank of the new weight. Therefore, the table can be stored in $O(2^{|\mu|} \cdot (|\mu|!) \cdot (|\mu|^4) \cdot (\log |\mu|)) = o(n)$ bits.

In the lookup table used for add-leaf, each entry is a four-tuple $(s_\mu, p_\mu, r_\mu, |\mu|)$ which maintains the representation of μ after adding the new leaf. The index to the table consists of (i) $(s_\mu, p_\mu, r_\mu, |\mu|)$, (ii) an index in the range $[1 \cdots |\mu|]$ to a vertex adjacent to the new edge, and (iii) the rank of the new weight. Therefore, the table can be stored in $O(2^{|\mu|} \cdot (|\mu|!) \cdot (|\mu|^4) \cdot (\log |\mu|)) = o(n)$ bits.

The size of the other two tables used for LCA and child-ancestor is analyzed similarly. Since the total number of entries in all the tables is less than $o(2^{|\mu|^2})$ and each entry can be computed in time $O(|\mu|)$, all the tables can be constructed in $o(n)$ time. \square

G Figures for the structure of Section 3

Fig. 3 and 4.

H Optimal semigroup structure for leaf updates with static weights

In this section we first develop a dynamic structure for the path minima problem which uses $O(nk\alpha_k(n))$ space, and supports queries in $O(k)$ time, and leaf inser-

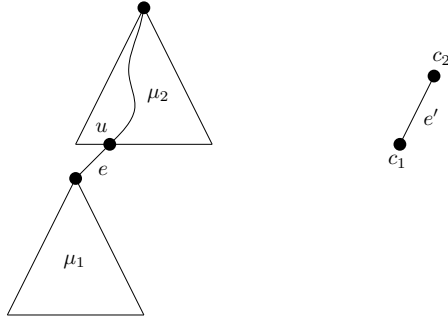


Fig. 3. The micro trees μ_1 and μ_2 are contracted to the nodes c_1 and c_2 respectively. The weight of the edge e' is calculated as follows: $w(e') = \min\{w(e), w(\text{pathmin}(u, r(\mu_2)))\}$.

tions in $O(k\alpha_k(n))$ amortized time. The only operations allowed to perform on the edge weights are comparisons. Using this structure, we then develop another dynamic structure using linear space, which supports `pathmin` queries in $O(\alpha(n))$ worst case time, and leaf insertions in $O(1)$ amortized time. Similar to the static structure of Alon and Schieber [1], we perform the micro-macro decomposition on the input tree. The following theorem presents the first structure.

Theorem 6. *For an input tree with n nodes, there exists a data structure using $O(nk\alpha_k(n))$ space, which supports `pathmin` queries in $O(k)$ time, and leaf insertions in amortized $O(k\alpha_k(n))$ time, for all integers n and k , where $k \geq 1$ and $n > 0$.*

Proof. We prove the following statement by induction on k : There exists a representation of an input tree T with n nodes which uses at most $c_s nk\alpha_k(n)$ space, and supports `pathmin` queries in at most $c_q k$ time, and leaf insertions in amortized $O(k\alpha_k(n))$ time, for all integers k and n , and for some constants c_s and c_q , where $k \geq 1$, $n > 0$, $c_s > 1$, and $c_q > 1$.

When $k = 1$, we use the dynamic path minima structure of Demaine et al. [12] for the input tree T . This data structure uses $O(n)$ space and supports queries in $O(1)$ time and leaf insertions in $O(\log n)$ amortized time. Note that $\alpha_1(n) = \log n$. We choose the constants c_s and c_q to be the constant factors in the space bound and query time respectively, of this structure.

Assume that the theorem is true for $1, 2, \dots, k - 1$. We prove that it is true for k . We use the micro-macro decomposition to partition T into $O(n/\alpha_{k-1}(n))$ micro trees, each of size $O(\alpha_{k-1}(n))$. The size of the macro tree is $O(n/\alpha_{k-1}(n))$. Each of these micro trees are then split recursively. Therefore, in the second level of recursion, each micro tree of size $O(\alpha_{k-1}(n))$ is partitioned into $O(\alpha_{k-1}(n)/\alpha_{k-1}(\alpha_{k-1}(n)))$ micro trees, each of size $O(\alpha_{k-1}(\alpha_{k-1}(n)))$. Furthermore, in the second level, there are $O(n/\alpha_{k-1}(n))$ macro trees, each of size $O(\alpha_{k-1}(n)/\alpha_{k-1}(\alpha_{k-1}(n)))$. In the last level of the recursion, the size of each micro tree is constant. Hence, the number of levels is $O(\alpha_k(n))$ (see Section 2).

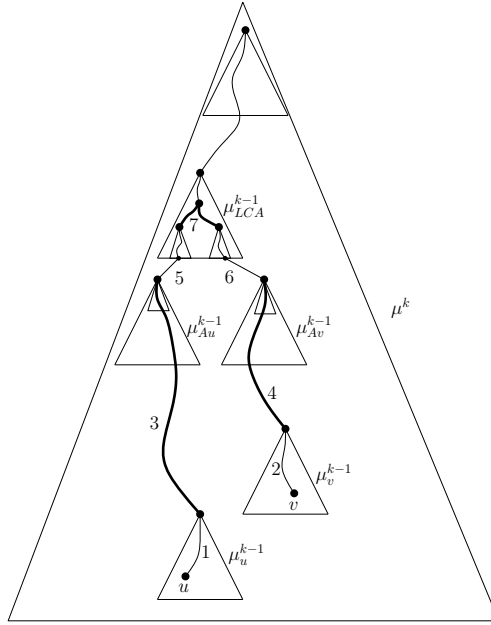


Fig. 4. Query decomposition

Before explaining the data structure and the insert operation, we intuitively describe the query algorithm.

In the following, for a micro tree μ , we denote its root by r_μ .

Querying. The given query $\text{pathmin}(x, x')$ is decomposed into two subqueries $\text{pathmin}(\text{LCA}(x, x'), x)$ and $\text{pathmin}(\text{LCA}(x, x'), x')$. Thus, it is enough to only consider pathmin queries of the form $\text{pathmin}(x, y)$, where x is an ancestor of y . We find the top most level of the recursion in which x and y fall into different micro trees μ_1 and μ_2 respectively. The query is divided into three subqueries $\text{pathmin}(x, b_{\mu_1})$, $\text{pathmin}(b_{\mu_1}, r_{\mu_2})$, and $\text{pathmin}(r_{\mu_2}, y)$, where b_{μ_1} is the lowest boundary node of μ_1 which is an ancestor of r_{μ_2} . In the case when x and y fall into the same micro tree in the last level of the recursion, the query is answered by scanning the path between x and y .

Data structure. Each macro tree in the decomposition is maintained using the data structure for $k - 1$. For each micro tree μ in the decomposition of T , let y be a node in μ . We store the value $\text{pathmin}(r_\mu, y)$ in y . Also, if μ has a boundary node b_μ which is different from r_μ , we store the value $\text{pathmin}(x, b_\mu)$ for all the ancestors x of b_μ within μ . This information is used to answer the two subqueries $\text{pathmin}(x, b_{\mu_1})$ and $\text{pathmin}(r_{\mu_2}, y)$ mentioned in the query algorithm.

We also maintain a dynamic LCA structure [11] for T , which supports LCA queries and leaf insertions in constant worst case time.

In addition, we construct another tree L_T with height $\alpha_k(n)$ which has a node corresponding to each micro tree at every level of recursion. More precisely, the root corresponds to the input tree T . It has $O(n/\alpha_{k-1}(n))$ children, each corresponding to a first level micro tree. In general, a level- i node in this tree corresponds to a level- i micro tree μ , and has d children if μ is decomposed into d micro trees in level $i + 1$. The last level corresponds to the nodes in the input tree T . We maintain a dynamic LCA structure [11] for L_T . Given two nodes x and y , this structure can be used to find the top most level of recursion in which x and y fall into different micro trees, in constant time, which is required in the query algorithm.

Insertion. Suppose that we want to insert a new leaf ℓ adjacent to a node y in T such that the weight of the new edge between ℓ and y is w . We insert ℓ into each of the micro trees containing y in different levels of the decomposition. Let μ be the micro tree containing y in level i of the decomposition. We store the value $\text{pathmin}(r_\mu, \ell)$ in ℓ which is determined by $\min\{w, \text{weight}(\text{pathmin}(r_\mu, y))\}$. We also insert ℓ into the tree L_T as a sibling of y (and update the LCA structure of L_T).

After inserting ℓ , if the size of a micro tree in some level i exceeds the maximum allowed size for the micro trees of level i , then we split it into a constant number of smaller micro trees (see Section C). The new boundary nodes resulting from the split are inserted into the macro tree of level i . Then, the appropriate micro trees and macro trees in all the levels below level i are also split. The number of these micro trees is constant in each level.

Query time. As explained in the query algorithm, a query is split into at most three subqueries: at most two subqueries within micro trees, and one within a macro tree. For the queries within micro trees, one of the query nodes is a boundary node, and hence the answer is explicitly stored with one of the query nodes. The time to answer the query within the macro tree, by induction hypothesis, is $c_q(k - 1)$. By performing two additional operations, we can answer the given query. Thus, choosing $c_q = 2$ makes the overall query time to be $c_q k$.

Insert time. We now analyze the amortized cost of an insertion. The time to insert a node into a micro tree with n_i nodes at any level of recursion, is $O(1)$. By induction hypothesis, the time to insert a node into a macro tree with n_i nodes, at any level of recursion, is $O((k - 1)\alpha_{k-1}(n_i))$. But insertion into that macro tree happens after $O(\alpha_{k-1}(n_i))$ insertions into one of the micro trees at that level. Hence the amortized cost of the insertion into the macro tree is $O(k - 1)$. The time required to split the constant number of micro and macro trees that need to be split at all the levels below cost only $O(1)$ amortized time, as the sum of the sizes of all those trees is negligible compared to the size of the micro tree at the current level (which was split). Since an insert operation is performed by inserting a leaf into an appropriate micro tree at each level of recursion, the total time over all recursion levels is amortized $O(k\alpha_k(n))$.

Finally, inserting a node into L_T and updating it takes $O(1)$ worst-case time. Thus the overall insertion time is amortized $O(k\alpha_k(n))$.

Space. The space requirement of a micro tree with n_i at any level of recursion is $O(n_i)$. Thus the space for all the micro trees at all the $\alpha_k(n)$ levels of recursion is $O(n\alpha_k(n))$. By the induction hypothesis, the space requirement of a macro tree with n_i nodes at any level of recursion is $c_s n_i(k-1)$. Hence the space for all micro trees at all the levels of recursion is $c_s n(k-1)\alpha_k(n)$. Thus by choosing c_s appropriately, we can make the overall space (for all the micro and macro trees) to be at most $c_s n k \alpha_k(n)$. \square

Linear time dynamic path minima structure with $\alpha(n)$ query time. We now describe a linear space data structure which supports queries in $O(\alpha(n))$ time, and updates in $O(1)$ amortized time.

Theorem 7. *There exists a data structure maintaining an edge-weighted tree with n nodes under insertions and deletions of leaves, which uses $O(n)$ space and supports `pathmin` queries in $O(\alpha(n))$ worst-case time, insertions and deletions in $O(1)$ amortized time.*

Proof. Let α denote the value $\alpha(n)$. We describe a structure assuming that $\alpha(n)$ remains the same through out the updates. Whenever $\alpha(n)$ changes, we create a new structure with the changed $\alpha(n)$ value, and this only requires $O(1)$ amortized time.

Let T be the input tree. Using the micro-macro decomposition, we partition T into $O(n/\alpha^2)$ micro trees, each of size $O(\alpha^2)$. We maintain each of the micro trees using the dynamic path minima structure described below.

Each micro tree μ of size $O(\alpha^2)$ is represented as follows. We maintain μ as $O(\alpha)$ micro trees, each of size $O(\alpha)$, and a macro tree of size $O(\alpha)$ that maintains all the boundary nodes of the micro trees. To support `pathmin` queries on μ , we simply traverse the path between the two query nodes and answer the query in $O(\alpha)$ time. Inserting leaves into μ can be supported in $O(1)$ amortized time, by simply performing the inserts on the appropriate micro tree of μ , and splitting if required. Thus μ can be stored in $O(\alpha^2)$ space to support queries in $O(\alpha)$ time and inserts in $O(1)$ amortized time.

We maintain all the boundary nodes of the micro trees of T in a macro tree of size $O(n/\alpha^2)$. This macro tree is again stored using the structure of Theorem 3 with $k = 2\alpha$. The space used by this macro tree is $O((n/\alpha^2) \cdot 2\alpha \cdot \lambda(2\alpha, n)) = O(n/\alpha)$, as $\lambda(2\alpha(n), n) = O(1)$. The structure supports queries on the macro tree in $O(\alpha)$ time and updates in $O(2\alpha \cdot \lambda(2\alpha, n)) = O(\alpha)$ amortized time. As updates to the macro tree are performed only when a micro tree splits, this cost can be amortized over all the updates performed on the micro tree before it splits. Thus the overall update time can be shown to be $O(1)$ amortized. \square

I RAM structure for leaf updates with static weights

In this section, we present a path minima data structure in the RAM model, that supports the query in $O(1)$ time, and the operations `insert-leaf` and `delete-leaf` in $O(1)$ amortized time. This result was already known [2, 17], but we give a simpler data structure to achieve it.

We decompose the tree into *small trees* of size $O(\log n)$ and each small tree into micro trees of size $O(\log \log n)$ using the micro-macro decomposition (see Section 2). Decomposing into small trees generates a macro-macro tree of size $O(n/\log n)$, and decomposing the small trees generates $O(n/\log n)$ macro trees, each of size $O(\log n/\log \log n)$. The operations within each micro tree is supported using precomputed tables. We do not store any representation for the small trees. We represent the macro-macro tree and each macro tree with a Cartesian tree (see Section 2).

Query algorithm. The query is divided according to the micro-macro decomposition of the input tree such that each subquery is contained within a micro tree, a macro tree or the macro-macro tree. The subqueries within the micro trees are answered using precomputed tables in $O(1)$ time. The subqueries within the macro trees and the macro-macro tree are answered using the corresponding Cartesian trees in $O(1)$ time. Since the total number of subqueries is $O(1)$, we can compute the answer in $O(1)$ time.

Leaf insertion. To perform `insert-leaf`, we add the new leaf into the appropriate micro tree using precomputed tables. Then, if the size of the micro tree exceeds its maximum limit, we split it into at most four micro trees (see Section C). The new boundary nodes are inserted into the appropriate macro tree (described later). If the size of the macro tree also exceeds its maximum limit, we split it into at most four macro trees similarly. During the split of a macro tree that we remove at most three edges, we also split the micro trees containing these three edges and distribute the micro trees among the new macro trees. Then the nodes that connect the new macro trees to each other, are inserted into the macro-macro tree.

We have shown that new leaves can be inserted into Cartesian trees in logarithmic time (Lemma 2). The following lemma shows that, during the split, the new nodes can be inserted into the macro trees as leaves. This is similarly true for the macro-macro tree.

Proof of Lemma 4. Lemma 2 explains how we can insert a leaf into the Cartesian tree in logarithmic time. We only need to show that the new boundary nodes can be inserted by performing the same operation `insert-leaf`. Let b_1 and b_2 be the two boundary nodes of a micro tree, and let x be a new boundary node as a result of splitting the micro tree. Recall that the edge (b_1, b_2) in the macro tree is the path minima along the path between b_1 and b_2 in the micro tree. Let w be

the weight of (b_1, b_2) . If x is not on the path between b_1 and b_2 , then it is a leaf in the macro tree. Otherwise, x splits the edge (b_1, b_2) into two edges (b_1, x) and (b_2, x) . Obviously the weight of one of these two edges, w.l.o.g. (b_1, x) , is equal to w , and the other one has a weight w' , where $w' \geq w$. Consider the subtree S of the Cartesian tree rooted at the child of (b_1, b_2) that has b_2 as a descendant. Then x is a leaf, adjacent to b_2 , in the part of the macro tree corresponding to S . Thus, (b_2, x) can be inserted into the Cartesian tree as a leaf. \square

The precomputed tables and the representation of the micro trees are similar to Section 3.1. To perform `delete-leaf`, we simply mark the deleted leaves because they have no effect on the result of future operations. Using global rebuilding for `delete-leaf` and the amortized analysis, we achieve the following data structure.