Path Minima Queries in Dynamic Weighted Trees

Gerth Stølting Brodal[1], Pooya Davoodi[1], S. Srinivasa Rao[2]

[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 trees each tree 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 on a tree, where the edge-weights can be updated, we give comparison-based and RAM data structures that achieve optimal query time. These structures support inserting a node on an edge, inserting a leaf, and contracting edges. When only insertion and deletion of leaves in a tree are needed, we give two data structures that achieve optimal and significantly lower query times than when updating the edge-weights is allowed. One is a semigroup structure for which the edge-weights are from an arbitrary semigroup and queries ask for the semigroup-sum of the edge-weights on a given path. For the other structure the edge-weights are given in the word RAM. We complement these upper bounds with lower bounds for different variants of the problem.

1 Introduction

We study variants of the path minima problem on weighted unrooted trees, where each edge is associated with a weight. The problem is to maintain a data structure for a collection of trees (a forest) supporting the query operation:

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

In the dynamic setting, a subset of the following update operations is provided:

- 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.

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

– cut(e): delete the edge e from the forest, splitting a tree into two trees.

We present path minima data structures to maintain trees under either updating leaf edges or updating arbitrary edge-weights. Additionally, we study the complexity of path minima queries on forests that can be updated by link and cut.

We define three different models depending on how data the algorithms can operate on the edge-weights: (1) the *comparison-based* model, where the only allowed operation on the edge-weights are comparisons; (2) the *RAM* model, where any standard RAM operation is allowed on the edge-weights; and (3) the *semigroup* model, in which the edge-weights are from an arbitrary semigroup and queries ask to compute the semigroup-sum of the edge-weights along a path (notice that a data structure over the semigroup $(\mathbb{R}, \max)$ can be used to make a comparison-based structure for the path minima problem). Except for the computation on the edge-weights, our algorithms are in the unit-cost word RAM with word size $\Theta(\log n)$ bits, where n is the total number of nodes in the input trees.

1.1 Previous work

Static weighted trees. The minimum spanning tree verification problem on a spanning tree of a graph can be solved by asking a sequence of offline path minima queries in the tree, which can be supported using amortized $O(1)$ comparisons for each query [17]. In the online setting, Pettie proved that $\Omega(\alpha(n))$ amortized comparisons are required to answer each query [21], which is a tight lower bound due to [8].

For online queries, Alon and Sheiber [1] considered trade-offs between the preprocessing time and the query time in the semigroup model. They presented a data structure that supports queries in at most $4k - 1$ semigroup operations with $O(nk\alpha_k(n))$ preprocessing time and space for a parameter $k \geq 1$, where $\alpha_k(n)$ is a function in the inverse-Ackermann hierarchy (defined in Section 1.3). They also proved that $\Omega(n\alpha_{2k}(n))$ preprocessing time is required to obtain such a query time in the worst case. Notice that $\alpha_{2k}(n) = o(\alpha_k(n))$ for $k > 1$ (Section 1.3). Similar trade-offs are known in the comparison-based model [21].

Dynamic weighted trees. In the comparison-based model, Demaine, Landau, and Weimann [10] showed how to maintain an input tree under inserting and deleting leaves in $O(\log n)$ amortized time and supporting queries in $O(1)$ time using linear space. In the RAM model, $O(1)$ amortized update time with the same query time and space can be achieved [2, 15].

Tarjan was interested in maintaining a collection of rooted trees under the operation link (incremental trees) to support a class of path minima queries, where a root is one of the end points of the query paths [23, Section 6]. In the semigroup model, for a sequence of m offline queries and updates, he obtained $O((m + n) \cdot \alpha(m + n, n))$ time using $O(m + n)$ space. In the RAM model, this running time can be improved to $O(1)$ for each online query and amortized $O(1)$ for each offline update [14]. Alstrup and Holm showed that arbitrary queries can be supported in $O(\alpha(n))$ time while online updates are performed in constant amortized time in the RAM model [2]. Finally, Kaplan and Shafrir generalized this result to arbitrary links in unrooted trees [15].

Dynamic trees (link-cut trees) of Sleator and Tarjan support many operations including pathmin, link, cut, root, and evert in $O(\log n)$ amortized time, in the semigroup model [22]. 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 given node becomes the root, by turning the tree "inside out". Essentially, this data structure can solve all the variants of the dynamic path minima problem.

Relation to range minimum queries. A special case of the path minima problem, is the one-dimensional *range minimum query* (1D-RMQ) problem, where the input tree is a path. In this problem, we are given an array containing n elements, and we have to find the position of the minimum element within a query range.

The following lower bounds are derived from known lower bounds by reduction from the 1D-RMQ problem: (1) In the semigroup model, with linear space, $\Omega(\alpha(n))$ semigroup operations are required to answer a path minima query [25]; (2) In the RAM model, using $O(n/c)$ bits of additional space, the number of cell probes required to answer a path minima query is at least $\Omega(c)$, for a parameter $1 \leq c \leq n$ [7] (here, we assume that the edge-weights are given in a read-only array); (3) In the RAM model, if we want to update the edge-weights in polylogarithmic time, $\Omega(\log n/\log\log n)$ cell probes are required to support path minima queries [3, Section 2.2]; (4) In the comparison-based model, if we want to answer a path minima query in polylogarithmic time, $\Omega(\log n)$ comparisons are required to update the edge-weights [6]; (5) In the semigroup model, logarithmic time to support path minima queries implies $\Omega(\log n)$ time to update the edge-weights, and vice versa [20].

Cartesian trees [24] are a standard structure that along with lowest common ancestor structures can support range minimum queries in constant time with linear space and preprocessing time [13]. Cartesian trees can be also defined for weighted trees as follows: The Cartesian tree T_C of a weighted tree T is a binary tree, where the root of T_C corresponds to the edge e of T with minimum weight, and the two children of the root in T_C correspond to the Cartesian trees of the two components made by deleting e from T. The internal nodes of T_C are the edges of T, and the leaves of T_C are the nodes of T.

Similar to range minimum queries, Cartesian trees can be used to support path minima queries in $O(1)$ time using linear space [18, Section 3.3.2], [5, Section 2], and [10, Theorem 2]. But constructing a Cartesian tree requires $\Omega(n \log n)$ comparisons derived by a reduction from *sorting* [10]. This lower bound implies a logarithmic lower bound to maintain the Cartesian tree under inserting new leaves in the original tree, which is tight due to the following lemma.

Lemma 1. ([10]) *The Cartesian tree of a tree with n nodes can be maintained in a linear space data structure that can be constructed in $O(n \log n)$ time, and supports path minima queries in $O(1)$ time and inserting leaves in $O(\log n)$ time.*

1.2 Our results

In Section 2, we present a comparison-based data structure that supports path minima queries in $\Theta(\log n/\log\log n)$ time, and supports updates to the edge-weights in $\Theta(\log n)$ amortized time. In the RAM model, the update time is improved to $O(\log n/\log\log n)$ amortized. Both data structures support the operations insert, insert-leaf, and contract with the same update times.

In Section 3, we dynamize the data structure of Alon and Shieber [1] in the semigroup model, to support path minima queries in at most $7k-4$ semigroup operations using $O(n\alpha_k(n))$ space, while supporting insertions and deletions of leaves in $O(\alpha_k(n))$ amortized time. Using Lemma 1, we can obtain a RAM structure that supports path minima queries in constant time and inserting and deleting leaves in constant amortized time, giving an alternative approach to achieve a known result [2, 15].

In Section 4, we provide cell probe lower bounds for query-update trade-offs when data structures are served by the operations pathmin, link and cut, in the RAM model. We prove that if we want polylogarithmic update time, we cannot hope for answering path minima queries in faster than $\Omega(\log n/\log\log n)$ time. We also show that with logarithmic update time, $\Theta(\log n)$ query time achieved by the dynamic trees of Sleator and Tarjan is the best possible. Furthermore, we prove that with sub-logarithmic query time, obtaining logarithmic update time is impossible.

1.3 Preliminaries

In Sections 2 and 3, we design our data structures for rooted trees, though every unrooted tree can be transformed to a rooted tree by choosing an arbitrary node as the root. Notice that all the update operations except link play the same role on rooted trees, whereas link in rooted trees is restricted to add new edges between a root and another node. Moreover, we transform rooted trees to binary trees using a standard transformation [11]: 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, the operations in rooted trees translate to a constant number of operations in binary trees.

Since we make our data structures for rooted trees, we can divided each path minima query into two subqueries as follows. Every pathmin(u, v) is reduced to two subqueries pathmin(c, u) and pathmin(c, v), where c is the lowest common ancestor (LCA) of u and v. It is possible to maintain a tree under inserting leaves and internal nodes, deleting leaves and internal nodes with one child, and determining the LCA of any two nodes all in worst-case $O(1)$ time [9]. Therefore, we only consider queries pathmin(u, v), where u is an ancestor of v.

In our data structures, we utilize a standard decomposition of binary trees denoted by *micro-macro decompositions* [4]. Given a binary tree T with n nodes and a parameter x, where $1 \le x \le n$, the set of nodes in T is decomposed into $O(n/x)$ disjoint subsets, each of size at most x, where each subset is a connected 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. These nodes are denoted by *boundary nodes*. The root of every micro tree is a boundary node except for the micro tree that contains the root of T. The macro tree is a tree of size $O(n/x)$ 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.

We use a variant of the inverse-Ackermann function α defined in [10, 19]. First, we define the inverse-Ackermann hierarchy for integers $n \geq 1$: $\alpha_0(n) = \lceil n/2 \rceil$, $\alpha_k(1) = 0$, 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 for $k \geq 2$, $\alpha_k(n) = \log^{**\cdots*} n$, where the $*$ is repeated $k-1$ times in the superscript. 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: $\alpha(n) = \min\{k \mid \alpha_k(n) \leq 3\}$. The two-parameter version of the inverse-Ackermann function for integers $m, n \geq 1$ is defined as follows: $\alpha(m, n) = \min\{k : \alpha_k(n) \leq 3 + m/n\}$. This definition of the function satisfies $\alpha(m, n) \leq \alpha(n)$ for every m and n.

2 Data structures for dynamic weights

In this section, we present two path minima data structures that support all the update operations except link and cut in an input tree. The first data structure is in the comparison-based model and achieves $\Theta(\log n / \log\log n)$ query time, $\Theta(\log n)$ time for update, and $O(\log n)$ amortized time for insert, insert-leaf, and contract. The second data structure improves the update time to $O(\log n / \log\log n)$ in the RAM model. Both the structures are similar to the ones in [15]. In the following, we first describe the comparison-based structure, and then we explain how to convert it to the RAM structure.

2.1 Comparison-based data structure

We decompose the input binary tree T into micro trees of size $O(\log^\varepsilon n)$ with the maximum limit $3\log^\varepsilon n$, for some constant ε, where $0 < \varepsilon < 1$, using the micro-macro decomposition (Section 1.2). In our data structure, we do not use macro trees. Each micro tree contracts to a super-node, and a new tree T' is built containing these super-nodes. If there is an edge in T between two micro trees, then there is an edge in T' between their corresponding super-nodes. The weight of this edge is the minimum weight along the path between the root of the child micro tree and the root of the parent micro tree. We binarize T', and then we recursively decompose it into micro trees of the same size $O(\log^\varepsilon n)$.

Consider a path minima query between the nodes u and v, where u is an ancestor of v. If u and v do not lie within the same micro tree, the query is divided into at most four subqueries of three different types: (1) an edge that is between two micro trees; (2) a query that is within a micro tree; and (3) a query that is between the root of the micro tree containing v and a boundary node of the micro tree containing u. Queries of type 1 are trivial, since we store the edge-weights in all the levels of the decomposition. To support queries of type 2 efficiently, we precompute the answer of all possible queries within all possible micro trees. Queries of type 3 are divided into subqueries recursively. There are at most one subquery of type 3 at each level, and thus the overall query time is determined by the number of levels.

Updating edge-weights and insertions are performed in the appropriate micro tree in the first level, and if it is required we propagate them to the next levels recursively. To support updates within each micro tree, we precompute the result of all possible updates within all possible micro trees. We maintain the edge-weights of each micro tree in sorted order in a balanced binary search tree that supports insertions and deletions of new edge-weights in $O(\log(\log^{\varepsilon} n))$ time. Additionally, we assign a local ID to each node within a micro tree, which is the insertion time of the node.

We set the size of each micro tree in all the levels to $O(\log^{\varepsilon} n)$, thus the number of levels is $O(\log n / \log\log n)$.

Data structure. Let T_0 denote an input tree, and for $i \geq 1$, let T_i be the tree that is built of super-nodes (to which micro trees contract) in the level i of the decomposition. The data structure consists of the following parts:

– We explicitly store T_i in all the levels of the decomposition, including T_0.
– For each node in T_i, we store a pointer to the micro tree in T_i that contains the node. We also store the local ID of the node.
– 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 μ, we store a balanced binary search tree containing all the edge-weights of μ. This allows us to find the rank of a new edge-weight within μ in $O(\log(\log^{\varepsilon} n))$ time.
– For each micro tree μ in T_i, we store an array of pointers that point to the original nodes in T_i given the local IDs.

Precomputed tables. We precompute and store in a table all possible results of performing each of the following operations within all possible micro trees: pathmin, update, insert, insert-leaf, contract, LCA, root and child-ancestor. For a micro tree μ, root returns the local ID of the root of μ, LCA returns the local ID of the LCA 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). Each precomputed table is indexed by the tuples $(s_\mu, p_\mu, r_\mu, |\mu|)$ and the arguments of the corresponding operation. To perform update, insert, and insert-leaf within μ, we find the rank of the new edge-weight among the existing 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. Using appropriate tables, we can achieve the following lemma.

Lemma 2. *Within a micro tree of size $O(\log^{\varepsilon} n)$, 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.*

Proof. Let μ be the micro tree. In the table used to perform pathmin, each entry 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 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.

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

In the table used for insert-leaf, each entry is the tuple $(s_\mu, p_\mu, r_\mu, |\mu|)$ which represents 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 $[1 \cdots |\mu|]$ which represents the node that a new leaf is adjacent to, and (iii) the rank of the new edge-weight. Therefore, the table can be stored in $O(2^{|\mu|} \cdot |\mu|! \cdot |\mu|^4 \cdot \log |\mu|) = o(n)$ bits.

The sizes of the other tables used for LCA, root, child-ancestor, moving a subtree, insert, and contract are 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.

To compute the rank of the new edge-weight, which is part of the index to the tables for updates, we search and then update the balanced binary search tree of μ in $O(\log \log n)$ time. □

Query time. Each path minima query is divided into subqueries, and at most one subquery is divided recursively into subqueries. The division continues for $O(\log n / \log \log n)$ levels, and at each level all subqueries (of types 1 and 2) are supported in constant time. Therefore, the query time is $O(\log n / \log \log n)$.

Update. We perform 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 2. 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 check-update(μ) 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 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 2. (2) If $|\mu|$ exceeds the maximum limit $3\log^{\varepsilon} n$, then we split μ, in linear time, into $k \leq 4$ new micro trees, each of size at most $2\log^{\varepsilon} n + 1$ such that each new micro tree has at most two boundary nodes including the old boundary nodes of μ. 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 check-update(μ') 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 check-update(μ) to recursively update the edge-weights in T_1 between μ and its child micro trees, which takes $O(\log n)$ time.

To perform 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 $O(n/\log^{\varepsilon} n)$ micro trees (since any created micro tree needs at least $\log^{\varepsilon} n$ node insertions before it splits again). Since the number of nodes in $T_0, T_1, \ldots, T_\ell$ is geometrically decreasing, the total number of micro tree splits is $O(n/\log^{\varepsilon} n)$. Because each micro tree split takes $O(\log^{\varepsilon} n)$ time, the amortized time per insertion is $O(1)$ for handling micro tree 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-based 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.*

2.2 RAM structure

In this section, we improve the update time of the the structure of Theorem 1 to $O(\log n/\log\log n)$ in the RAM model. The bottleneck in our comparison-based data structure is that we maintain a balanced binary search tree for the edge-weights of each micro tree to find the rank of new edge-weights in $O(\log\log n)$ time. We improve this by using a Q-heap structure [12] to maintain the edge-weights of each micro tree to find the rank of new edge-weights under insertions and deletions in $O(1)$ time with linear space and preprocessing time. The following theorem states our result.

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.*

3 Data structures for dynamic leaves

In this section, we first present a semigroup structure that supports path minima queries, and leaf insertions/deletions but no updates to edge-weights. We then describe a RAM structure supporting the same operations.

3.1 Optimal semigroup structure

Alon and Schieber [1] presented two static data structures to support path minima queries in the semigroup model. We observe that their structures can be made dynamic. The following theorems summarize our result. To prove this we need an additional restricted operation insert(e,v,w), where w is larger than the weight of e.

Theorem 3. *There exists a semigroup data structure of size $O(n\alpha_k(n))$ to maintain a tree containing n nodes, that supports path minima queries with at most $7k-4$ semigroup operations and leaf insertions/deletions in $O(\alpha_k(n))$ amortized time, for a parameter k, where $1 \le k \le \alpha(n)$.*

By substituting k with $\alpha(n)$, we obtain the following.

Corollary 1. *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.*

3.2 RAM structure

We also present a RAM structure that supports path minima queries and leaf insertions/deletions. This structure does not give a new result (due to [2, 15]) but is a another approach to solve the problem.

We decompose a tree into micro trees of size $O(\log n)$ and each micro tree into micro-micro trees of size $O(\log\log n)$ using the micro-macro decomposition (see Section 1.3). Decomposition of the tree into micro trees generates a macro-macro tree of size $O(n/\log n)$, and decomposition of each micro tree into micro-micro trees generates $O(n/\log n)$ macro trees, each of size $O(\log n/\log\log n)$.

The operations within each micro-micro tree is supported using precomputed tables and Q-heaps [12]. We do not store any representation for the micro trees. We represent the macro-macro tree and each macro tree with a Cartesian tree.

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

Lemma 3. *When a micro tree splits, we can insert the new boundary nodes by performing* insert-leaf *using the Cartesian tree of the corresponding macro tree.*

We represent each Cartesian tree using Lemma 1. Thus, Lemma 3 allows us to achieve the following.

Theorem 4. *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.*

4 Lower bounds

We consider the path minima problem with the update operations link and cut. Let t_q denote the query time, and t_u denote the maximum of the running time of link, and cut. 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 $t_u = \Omega(\log^{1+\varepsilon} n)$ 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_u to blow-up to $(\log n)^{\Omega(\log\log n)}$.

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 to maintain a forest of undirected trees under the 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. Pătraşcu and Demaine [20] 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 by putting 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 $t_q \log \frac{t_q+t_u}{t_q} = \Omega(\log n)$. From this, we e.g., conclude that if $t_q = O(\log n/ \log\log n)$, then $t_u = \Omega(\log^{1+\varepsilon} n)$, for some $\varepsilon > 0$. We can also show that if $t_u = O(t_q)$, then $t_q = \Omega(\log n)$.

The boolean union-find problem is to maintain a collection of disjoint sets under the 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. [16] 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 trivially 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 $t_q = \Omega(\frac{\log n}{\log(t_q+t_u)})$. 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_u blows-up to $(\log n)^{\Omega(\log\log n)}$.

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. *Algorithmica, Special issue on ESA 2010*, 2011. In press.
8. B. Chazelle. Computing on a free tree via complexity-preserving mappings. *Algorithmica*, 2:337–361, 1987.
9. R. Cole and R. Hariharan. Dynamic lca queries on trees. *SIAM Journal on Computing*, 34(4):894–923, 2005.
10. 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.

11. G. N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 14(4):781–798, 1985.
12. 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.
13. H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proc. 16th annual ACM symposium on Theory of computing*, pages 135–143. ACM Press, 1984.
14. 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.
15. 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.
16. 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.
17. V. King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18(2):263–270, 1997.
18. D. M. Neto. *Efficient cluster compensation for lin-kernighan heuristics*. PhD thesis, University of Toronto, Toronto, Ontario, Canada, Canada, 1999.
19. G. Nivasch. Inverse ackermann without pain. http://www.yucs.org/ gnivasch/alpha/, 2009.
20. M. Pătraşcu and E. D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing*, 35(4):932–963, 2006.
21. S. Pettie. An inverse-ackermann type lower bound for online minimum spanning tree verification. *Combinatorica*, 26(2):207–230, 2006.
22. D. Sleator and R. Endre Tarjan. A data structure for dynamic trees. *Journal of computer and system sciences*, 26(3):362–391, 1983.
23. R. E. Tarjan. Applications of path compression on balanced trees. *J. ACM*, 26(4):690–715, 1979.
24. J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.
25. 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.