Finding Path Minima in Incremental Unrooted Trees*

Haim Kaplan † Nira Shafrir *

February 25, 2008

Abstract

Consider a dynamic forest of unrooted trees over a set of n vertices which we update by *link* operations: Each link operation adds a new edge adjacent to vertices in two different trees. Every edge in the forest has a weight associated with it, and at any time we want to be able to answer a *path-min* query which returns that edge of minimum weight along the path between two given vertices.

For the case where the weights are integers we give an algorithm that performs $n-1$ link operations and m pathmin queries in $O(n+m\alpha(m,n))$ time. This extends well known results of Tarjan [11] and Yao [12] to a more general dynamic setting at the cost of restricting the weights to be integers. Using our data structure we get an optimal data structure for a restricted version of the mergeable trees problem [9].

We also suggest a simpler data structures for the case where trees are rooted and the link operation always adds an edge between the root of one tree and an arbitrary vertex of another tree.

*This work is partially supported by United States - Israel Binational Science Foundation, project number 2006204.

†School of Computer Science, Tel Aviv University, Tel Aviv 69978, Tel Aviv, Israel. {haimk,shafrirn}@post.tau.ac.il

1 Introduction

The *incremental path minima problem in rooted trees* is defined as follows. Let F be a forest of rooted trees with n vertices. Each edge e has an integer weight $w(e)$. We have to support the following operations on the forest.

- make-tree(v): Create a new tree consisting of the singleton node v.
- link(u, v, c). We assume that $u \in T^1$, $v \in T^2$, v is the root of T^2, and $T^1 \neq T^2$. Replace the trees T^1 and T^2, by the tree that is created by adding the edge $e = (u, v)$ with $w(e) = c$.
- path-min(u, v): If u and v belong to the same tree, return the edge of minimum weight on the unique path between u and v. Otherwise, return null.

We give an algorithm that supports $n - 1$ link operations and m path-min queries in $O(n + m\alpha(m, n))$ time where $\alpha(m, n)$ is the inverse of Ackermann's function. Alstrup and Holm [2] claimed this result without describing the data structure.

The *incremental path minima problem in unrooted trees* is defined analogously as follows. Let F be a forest of unrooted trees. Each edge e has an integer weight $w(e)$. The operations make-tree and path-min are defined as for rooted trees. We change the link(u, v, c) operation and remove the requirement that u is a root. That is we only require that $u \in T^1$, $v \in T^2$, and $T^1 \neq T^2$. The operation replaces the trees T^1 and T^2 by the tree that is created by adding the edge $e = (u, v)$ with $w(e) = c$. This data type is more general in the sense that it allows link between any two vertices that are not in the same tree. Our main result is a data structure with the same time bounds for this unrooted version of the problem.

Our data structures are based on two main components. The first component is *incremental trees*. Incremental trees support add-leaf and add-root operations and path-min queries in $O(1)$ time. They are based on a similar structure of Alstrup and Holm [2] for the level ancestor problem. We restrict the weights to be integers so we can use q-heaps [5] to construct incremental trees. We assume the RAM model of computation with word size b so that a weight of an edge fits into a single word. We also make sure that $b \geq \log n$ where n is the number of vertices in our forest.

The second component is a recursive decomposition of trees suggested by Gabow [7]. Gabow used this scheme to answer m nearest common ancestor (nca) queries on rooted trees while allowing links in $O(n+m\alpha(m, n))$ time. This recursive structure supports all links in $O(n+m)$ time and each query in $O(\alpha(m, n))$ time. Gabow also used a similar technique to solve the list splitting problem [6]. A similar recursion is also used in the union-find data structure of [10].

The recursive scheme at high level is as follows. A tree T is partitioned into clusters each of which is a subtree of T. Each cluster is represented as an incremental tree. We then contract each cluster and represent the resulting tree recursively. The depth of the recursion is $O(\alpha(m, n))$.

These two components alone are not sufficient. Even for the rooted problem subtle issues arise as of how to organize the information so that we only spend a constant time per level of the recursion when we answer a query. In the rooted version of the problem there is a natural root for each cluster and we maintain information on paths to this root. When such a root does not exist it is not clear anymore on which path to maintain information. To fully grasp our new ideas one has to go into a quite deep technical discussion after getting familiar with the two basic components we mentioned above. To help the reader we try to expose some of these ideas through a somewhat less formal discussion in Section 2.

Our application for the incremental path minima problem in unrooted trees is an optimal algorithm for a restricted version of the mergeable trees problem [9]. The mergeable trees problem is

defined as follows. Let F be a forest of rooted trees. Each node v has a unique weight $w(v)$. Each $S \in F$ is a heap ordered tree so that $w(v) \geq w(p(v))$. The data structure supports the following operations.

• merge(u, v). Let $u \in S^1$, $v \in S^2$, (S^1 may be equal to S^2.) Create a new tree S in which the path from u to the root of S^1 is merged with the path of v to the root of S^2, in a way that preserves heap order. Specifically, let $u = u_1, \cdots, u_r$ be the nodes on the path from u to the root of S^1, and let $v = v_1, \cdots, v_k$ be the nodes on the path from v to the root of S^2. In the new tree S, the nodes $u_1, \cdots, u_r, v_1, \cdots, v_k$ are on the same path sorted by their weights, such that the node with the smallest weight is the root, the second node in the sorted list is its child and so on. In case $S^1 \neq S^2$ we call the merge an *external-merge* and otherwise we call it an *internal-merge*.

• nca(u, v): If u and v belong to the same tree return the nearest common ancestor of u and v. Otherwise, returns null.

Let n be the number of nodes in the forest. Georgiadis *et al* [9] gave an algorithm with amortized time of $O(\log^2 n)$ per operation that also supports cut operation and parent operations, where cut(v) removes the edge between v and its parent splitting the tree into two trees, and parent(v) returns the parent of v in T. They also gave an algorithm that supports all operations except cut in $O(\log n)$ amortized time.

We define the *path minima problem in unrooted trees* to be the data structure that supports the operations of the incremental path minima problem in unrooted trees and in addition supports the operation cut(e) operation, that removes the edge e from the tree splitting it into two trees.

Recently, Georgiadis *et al* [8] reduced the mergeable tree problem without cuts to the path minima problem in unrooted trees.[1] They maintained a forest F' of trees. Each tree $T \in F$ is represented by a tree $T' \in F'$ containing the same vertices. An external-merge(u, v) is implemented by performing link(u, v). An internal-merge(u, v) is implemented as follows. Let $x =$pathmin(u, v). Let q be the vertex preceding x on the path between u and v. We perform cut$((q, x))$ and link(u, v).

Georgiadis *et al* used this reduction to get a simpler implementation of mergeable trees that supports merges and nca queries in $O(\log n)$ worst-case or amortized time. If we do not allow internal merges, then using this reduction and our data structure for the *incremental* path minima problem in unrooted trees, we get a data structure that supports $n - 1$ external merges and m nca queries (where the weights are integers), in $O(n + m\alpha(m, n))$ time. This is particularly interesting since it matches the lower bound for the problem, see [9].

Related results. Yao [12] and Alon and Schieber [1] (See also [3]) solved the following static problem. Let T be a tree with n nodes each associated with an element of a semigroup $(S, \circ)$. We want to preprocess the tree to answer queries of the form: Given two vertices u and v what is the product ($\circ$) of the element of S associated with the vertices on the path from u to v. Yao and Alon and Schieber show how to preprocess the tree in linear time so that we can answer each query in $O(\alpha(n))$ (where $\alpha(n)$ is yet another version of an inverse to Ackermann's function). As a special case we can use their data structure for a static version of our problem in which there is a single tree given in advance which we want to preprocess for path-min queries.

Tarjan [11] in his seminal paper used path compression to solve a restricted version of the problem on rooted trees. In Tarjan's version link(v,w,c) is defined only if both v and w are roots of their trees, and path-min queries are restricted to paths from a given vertex to the root. Tarjan also considered an arbitrary semigroup. This special case has numerous applications, one of which is to verify that a given tree is a minimum spanning tree.

[1] In their version of the problem, the weights are associated with vertices instead of edges, and the query returns the vertex of minimum weight instead of the edge of minimum weight. Also, the weights are not necessarily integers.

Both algorithms mentioned above, when applied to computing path minima queries, work in the comparison model and need not assume that keys are integers.

The outline of the rest of the paper. In Section 2 we describe our key ideas at a high level focusing on the difference between the rooted and the unrooted problems. In Section 3 we give a description of the incremental trees used to perform add-leaf and add-root and min query in $O(1)$ time. In Section 4 we give the data structure for incremental path minima in rooted trees problem that performs $n-1$ link and m pathmin queries in $O(n+m)\alpha(m,n))$. In Appendix A, we show how to change the data structure for rooted trees so it performs $n-1$ link and m pathmin queries in $O(n+m\alpha(m,n))$. In Section 5 we give a simple but not optimal data structure for incremental path minima in unrooted trees problem that supports $n-1$ link m pathmin queries in $O(n+m\alpha^2(m,n))$ time. Last, in Section 6, we describe the data structure for incremental path minima in unrooted trees problem that supports $n-1$ link m pathmin queries in $O(n+m\alpha(m,n))$ time.

2 Highlights of the data structure

In this section we try to give a high level intuition of the differences between the rooted problem and the unrooted problem. This shows where the difficulties are, and the ideas that we introduce to cope with them. We focus on the query operation.

We avoid formal definitions at this point but recall that each of our trees is partitioned into clusters. Each cluster is a connected subtree which we represent as an incremental tree. The clusters are contracted, and the contracted tree is again partitioned into clusters that are represented as incremental trees, and so on. See Figure 1(A).

For a vertex $x \in T$, we denote by $C_k(x)$ the cluster of level k that contains it. Levels of clusters decrease with our recursion so the nodes of a cluster of level k are clusters of level $k+1$. In particular if $C_k(x)$ is not a top-level cluster then $C_{k+1}(x)$ is one of its nodes, and if C_k contains the vertices x and y of T then $C_k(x) = C_k(y) = C_k$.

Let (x,y), $x,y \in T$, be the edge of T that connects between the clusters $C_{k+1}(x)$ and $C_{k+1}(y)$ in a cluster C_k. What would be the weight of this edge when we consider it as an edge of C_k ? To define this weight we must root C_k as a subtree of T even if the original tree is unrooted. Then if say $C_{k+1}(x)$ is the parent of $C_{k+1}(y)$ the weight of (x,y) is the minimum weight of an edge on the path from y to the root of $C_{k+1}(x)$ that is induced by the orientation of C_k.

When T is rooted then the root of each cluster is naturally defined. See Figure 1(A). In unrooted trees we can designate an arbitrary vertex to be a root. But to maintain such a root we will have to change directions of many edges during a link. On the other hand we still want the clusters to be rooted so that we can define the weight of each edge in a cluster. A key idea is to exploit the freedom that we have, and allow each cluster to be independently oriented, see Figure 1(B).

The query pathmin(x,y) for both rooted and unrooted trees works roughly as follows. We find the highest level k such that $C_k(x) = C_k(y)$. Let $C_k = C_k(x) = C_k(y)$. (For an illustration see vertices x and y in Figure 1(A) and Figure 1(B). In these figures we assume that $k+2 = \ell$, and all the clusters of level $k+1$ in are in the same cluster of level k). By the definition of C_k we have that the clusters $C_{k+1}(x)$ and $C_{k+1}(y)$ are different nodes in C_k. Let C_{k+1} be the nearest common ancestor of $C_{k+1}(x)$ and $C_{k+1}(y)$ in C_k, ($C_{k+1} = C_{k+1}(a)$ in the Figures). For simplicity assume that $C_{k+1} \neq C_{k+1}(y)$, and $C_{k+1} \neq C_{k+1}(x)$.

Let (x_r, x''), $x_r, x'' \in T$, be the edge between $C_{k+1}(x)$ to its parent cluster in C_k where $x_r \in C_{k+1}(x)$. Similarly, let (y_r, y''), $y_r, y'' \in T$, be the edge between $C_{k+1}(y)$ to its parent cluster in C_k. Let $C_{k+1}(x_1)$ be the cluster that precedes C_{k+1} on the path from $C_{k+1}(x)$ to C_{k+1} and let $(x_1, \hat{x})$ be

the edge between the cluster $C_{k+1}(x_1)$ to C_{k+1}. Similarly, let $C_{k+1}(y_1)$ be the cluster that precedes C_{k+1} on the path from $C_{k+1}(y)$ to C_{k+1} and let $(y_1, \hat{y})$ be the edge between the cluster $C_{k+1}(y_1)$ to C_{k+1}. The query path consists of the following parts. (1) from x to x_r in $C_{k+1}(x)$; (2) from x_r to x_1; (3) the edge $(x_1, \hat{x})$; (4) from $\hat{x}$ to $\hat{y}$ in C_{k+1}; (5) The edge $(\hat{y}, y_1)$; (6) from y_1 to y_r; and (7) from y_r to y.

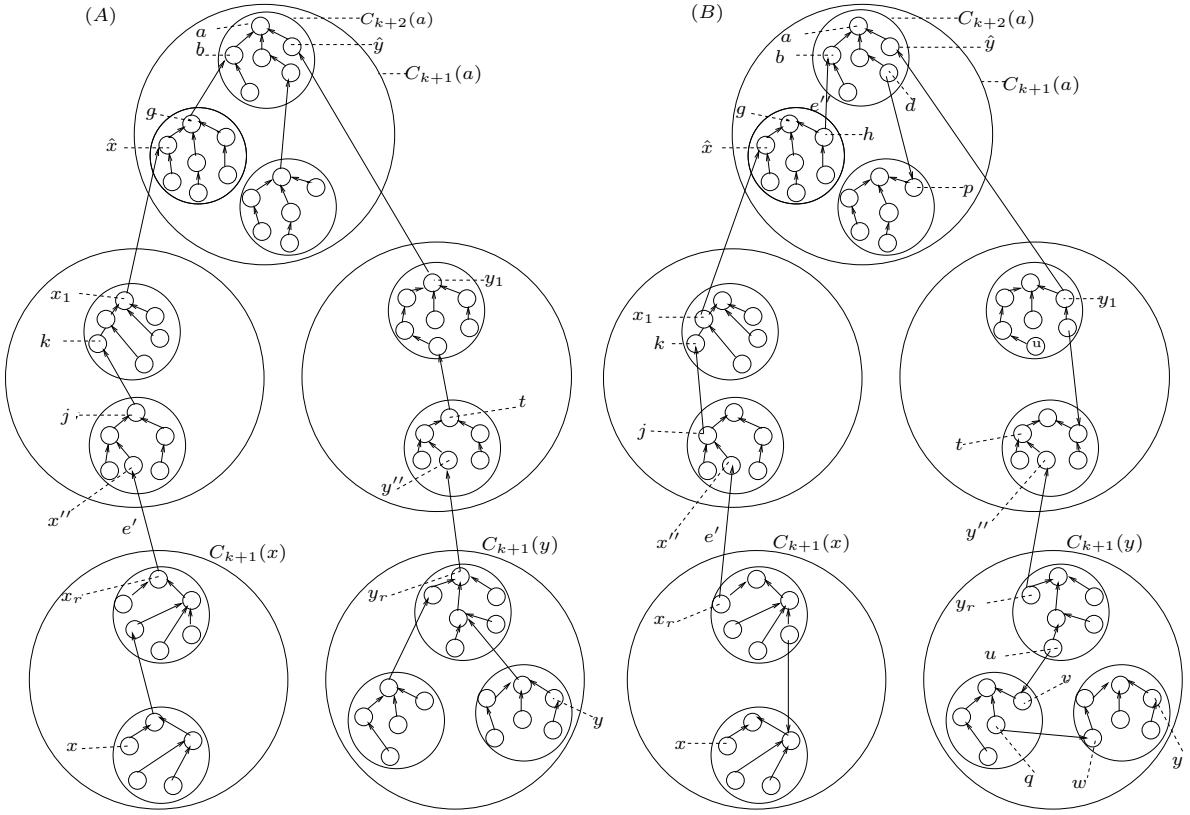

Figure 1: In both figures we assume here that $k+2 = \ell$, and that the small circles are the vertices of T. (A) The recursive structure for a rooted tree. Medium circles are clusters of T_{k+2}. Large circles are clusters of T_{k+1}. Edges of level $k+2$ are contained inside clusters of level $k+2$, (edges such as (a, b) $(\hat{x}, g)$ and so on). The edges (b, g) and (k, j) are of level $k+1$. The edges (x_r, x''), (y_r, y''), $(x_1, \hat{x})$ and $(y_1, \hat{y})$ are of level k. Let $e' = (x_r, x'')$. The weight of $e'(k)$ is the is the edge of minimum weight on the path from x_r to x_1. (B) The recursive structure in unrooted trees. Each cluster is oriented independently. We have that $r(C_{k+2}(a)) = t(C_{k+2}(a)) = a$, $r(C_{k+2}(g)) = g$, and $t(C_{k+2}(g)) = h$. Let $e' = (x_r, x'')$. The weight of $e'(k)$ is the minimum weight of an edge on the path from x_r to x_1. The weight of $e'(k+1)$ is the minimum weight of an edge on the path from x_r to j. Let $e'' = (h, b)$. The weight of $e''(k+1)$ is the minimum weight of an edge on the path from h to a in T.

We find the minimum on part (2) and part (6) by a query to the incremental tree of the cluster C_k. We find the minimum on part (4) recursively. Since $C_{k+1}(\hat{x}) = C_{k+1}(\hat{y}) = C_{k+1}$ the depth of this recursion is $O(\alpha(m, n))$.

The way we find the minimum on parts (1) and (7) is different in the rooted and the unrooted data structures. In the rooted structure we use the fact that x_r is the root vertex of the cluster

$C_{k+1}(x)$. This special case of finding the minimum from a vertex to the root of a cluster is easier and takes $O(\alpha(m,n))$ time.

If the trees are unrooted, (see Figure 1(B)), then x_r is not necessarily the root vertex of the cluster $C_{k+1}(x)$ and y_r is not necessarily the root vertex of the cluster $C_{k+1}(y)$. This makes the query slightly more difficult. We suggest two solutions. In the simple data structure of Section 5, we find the minimum on parts (1) and (7) again using recursion. This additional recursion degrades the running time of a query to $O(\alpha(m,n)^2)$.

In Section 6, we show how to reduce the query time in unrooted trees to $O(\alpha(m,n))$ by storing more information. The idea is as follows. Let y, and y_r be the vertices in the description of the query above such that $C_{k+1}(y) = C_{k+1}(y_r)$ and recall that we have to speed up parts (1) and (7) of the query.

For each cluster of level $k+2$ contained in $C_{k+1}(y)$, we store with the edge (y_r, y'') the edge of minimum weight on the path from y_r to this cluster. More generally, for each edge (u,v) such that $C_{k+1}(v) \neq C_{k+1}(u)$, we maintain the edge of minimum weight on the path from u to each cluster of level $k+2$ in $C_{k+1}(u)$. Using this information we find the edge of minimum weight on the path between y and y_r in $O(\alpha(m,n))$ time as follow. Assume that y and y_r are contained in different clusters of level $k+2$, (the case where both of them are contained in the same cluster is simpler). Let (w,q), be the edge that connects $C_{k+2}(y)$ to the next cluster on the path between $C_{k+2}(y)$ to $C_{k+2}(y_r)$ in $C_{k+1}(y)$, (See figure 1(B)). So we have that $C_{k+2}(w) = C_{k+2}(y) \neq C_{k+2}(q)$. The path from y_r to y is composed of the path from y_r to q, the edge (q,w) and the path from w to y. We find the edge of minimum weight on the path from y_r to q using the information we maintain for the edge (y_r, y''). We find the edge of minimum weight on the path from w to y in $C_{k+2}(y)$ by performing a recursive call on the cluster $C_{k+2}(y)$, that in turn will use the information maintain for the edge (w,q). This query costs $O(\alpha(m,n))$ time, and reduces the total query time to $O(\alpha(m,n))$.

To save this additional information we change the recursive structure of Gabow. In the recursive structure of Gabow each cluster of level k contained $\Omega(f(k))$ clusters of level $k+1$, where $f(k)$ is some function that ensures that the depth of the recursion is $O(\alpha(m,n))$. We also restrict the size of a cluster from above. To support this additional restriction we had to change the implementation of both link and query.

3 Incremental trees and partial incremental trees

Our building blocks are *incremental trees*. An incremental tree is a data structure to maintain a rooted tree T, with an integer weight on each vertex, such that the following operations are supported in $O(1)$ time.

add-leaf(v,w,c): Add a new leaf v with parent w to T. The weight of the edge (v,w) is c.

add-root(v,c): Add a new root v to T. The old root of T, say r, becomes a child of v and the weight of the edge (r,v) is c.

min(v,w): Returns the edge of minimum weight on the path from v to w.

change-weight(v,c): v must be a leaf. Changes the weight of the edge between v and its parent to c.

Our incremental trees also support nearest common ancestor (nca) queries in constant time. If the data structure does not support the add root operation then we call it a *partial incremental tree*.

The implementation of partial incremental trees is based on a data structure for the level ancestor problem of Alstrup and Holm [2]. The data structure uses $O(n)$ space where n is the size of T. To implement nca queries, we use the data structure of Gabow [7] that supports nca queries and add-leaf operations in constant amortized time, or the data structure of Cole and Hariharan [4] that supports these operations in worst case constant time.

Alstrup and Holm first describe a data structure which they call the *macro algorithm* that requires $O(n \log n)$ space but already answers a query in $O(1)$ time. In their final structure which requires only $O(n)$ space, they apply the macro algorithm to a tree T^M, consisting of a particular subset of $O(n/\log n)$ nodes called the *macro nodes*. In addition they maintain a partition of T into *micro trees*, each of size $O(\log n)$, and a set of tables for each micro tree.

3.1 The macro algorithm

This algorithm supports min query in $O(1)$ time, add-leaf in $O(\log n)$ time, and requires $O(n \log n)$ space, where n is the size of the tree T.

We denote by $d(v)$ the depth of a vertex v. That is if v is the root then $d(v) = 0$. Otherwise, $d(v) = d(p(v)) + 1$. Let $s(v)$ be the size of the subtree rooted at v.

We define the *rank* of a vertex v, denoted by $r(v)$, to be the maximum integer i such that $s(v) \geq 2^i$ and $2^i | d(v)$, where $b|a$ means that there is an integer k such that $a = kb$. The rank of the root is $\lfloor \log n \rfloor$. It is easy to verify that the number of nodes with rank at least i is $O(\frac{n}{2^i})$.

For each vertex v which is not the root we save the following tables. For $1 \leq x \leq 2^{r(v)}$, levelanc1$[v][x] = y$, where y is the ancestor of v whose depth is $d(v) - x$, and levelanc2$[v][x]$ contains the edge of minimum weight on the path between v and levelanc1$[v][x]$. For $0 \leq i \leq \lfloor \log(d(v)) \rfloor$, jump1$[v][i]$ contains the deepest proper ancestor of v whose depth is divisible by 2^i, and jump2$[v][i]$ contains the edge of minimum weight on the path from v to jump1$[v][i]$. The space used to store all these tables is $O(n \log n)$.

We now describe the query $\min(u, v)$. Let $z = \text{nca}(u, v)$. We find the edge of minimum weight on the path between u and z, and the edge of minimum weight on the path between v and z and return the smallest among these two edges. We now show how to compute the edge of minimum weight between u and z. If $d(u) - d(z) \leq 1$, then the computation is trivial. Assume that $d(u) - d(z) > 1$. Let $i \geq 1$ be the largest such that $d(u) - d(z) \geq 2^i$. Let $d(z) \leq k < d(u)$ be the smallest such that $2^{i-1} | k$ and $k - d(z) \leq 2^{i-1}$, then we have that $d(u) - k \geq 2^{i-1}$. Let y be the ancestor of u at depth k. Then, $2^{i-1} | d(y)$. The vertex y has at least $d(u) - d(y) \geq 2^{i-1}$ descendants. So $r(y) \geq i - 1$. The algorithm uses jump1 to find y, and jump2 to find the edge of minimum weight on the path between u and y. Finally, since $r(y) \geq i - 1$, entry $d(y) - d(z)$ exists in levelanc2$[y]$ and contains the edge of minimum weight on the path between y and z. The algorithm is as follows. We initialize a to be u and min-edge to be a dummy edge whose weight is infinity.

```
while d(a) − d(z) > 2^{i−1} {
        Let f = jump2[a][i − 1]
        If w(f) ≤ w(min-edge)
            min-edge = f
        a = jump1[a][i − 1]
}
Return the edge of minimum weight among min-edge and levelanc2[a][d(a) − d(z)].
```

At the end of the loop $a = y$ and levelanc1$[a][d(a) - d(z)] = z$. It is easy to see that there are less than 4 iterations of the loop and hence the query time is $O(1)$.

We define an alternative query operation $\min'(u, d)$ that returns a pair (q, f), where q is the ancestor of u of depth d and f is the edge of minimum weight on the path between u and q. Our query algorithm can be easily adapted to answer this type of query as well.

We now describe the add-leaf operation. When we add a leaf v, then $r(v) = 0$ and the tables are empty. We update jump1$[v][i]$ and jump2$[v][i]$ for $0 \leq i \leq \lfloor \log(d(v)) \rfloor$, in $O(\lfloor \log(d(v)) \rfloor)$ time using the values of the corresponding tables of the parent of v. That is, if $2^i | d(p(v))$, then jump1$[v][i] = p(v)$ and jump2$[v][i] = (v, p(v))$. Otherwise, jump1$[v][i]$ = jump1$[p(v)][i]$, and jump2$[v][i]$ contains the edge of minimum weight among jump2$[p(v)][i]$ and $(v, p(v))$. Notice that jump1$[u][i]$ and jump2$[u][i]$ for $u \neq v$ do not change.

We may need to double the tables levelanc1$[u]$ levelanc2$[u]$ for ancestors u of v whose rank has increased due to the addition of v. Alstrup and Holm [2] showed that the only vertices whose rank may have increased are the vertices in jump1$[v][i]$, for $0 \leq i \leq \lfloor \log(d(v)) \rfloor$. Instead of doubling the tables of such vertices when their rank increases, each time we add a leaf v we add one entry to levelanc1$[u]$ and to levelanc2$[u]$, if $u =$ jump1$[v][i]$, and $|\text{levelanc1}[u]| < 2^i$. By the time rank of u increases levelanc1$[u]$ and levelanc2$[u]$ contain the correct number of entries. It is easy to see that the storage required is still $O(n \log n)$. Given levelanc1$[v][x]$, levelanc2$[v][x]$ we update levelanc1$[v][x+1]$, levelanc2$[v][x+1]$ in $O(1)$ time as follows. If $x = 0$, then levelanc1$[v][x+1] = p(v)$ and levelanc2$[v][x + 1] = (v, p(v))$. Otherwise, we set levelanc1$[v][x + 1] = p($levelanc1$[v][x])$. The value of levelanc2$[v][x+1]$ is the edge of minimum weight among levelanc2$[v][x]$ and the edge between levelanc1$[v][x]$ to its parent. Clearly it takes $O(\log n)$ time to add a leaf.

Our macro tree does not support change-weight operation. In the implementation of the final structure (see Section 3.3), we use a macro tree to represent a subtree of the original tree T. This subtree consists only of internal nodes of T (nodes of rank greater than 0), and since change-weight is defined only for leaves of T, we do not need to change the weight of a macro node.

3.2 The micro algorithm

The size of a micro tree is $O(\log \log n)$. The implementation of micro trees uses q-heaps [5]. For a given n, each q-heap maintains a set of size $O(\log^{1/4} n)$ using a pre-computed table of size $O(n)$ which is common for all heaps. A q-heap supports insertions and deletions of elements and rank queries in $O(1)$ time. A rank query with a value a to a q-heap returns the number of values in the heap that are is smaller than a.

Let T be a micro tree. We maintain a q-heap [5] for T. We also keep for each tree T an identifier $I(T)$. We assign to each vertex $v \in T$ a number $n(v)$ which is the number of vertices in T just after adding v into T. To simplify the notation we assume $n(v) = v$. The identifier $I(T)$ is the concatenation of the pairs $(p(v), \text{rank}(w((v, p(v)))))$ for all vertices v other than the root, in increasing order of their insertion times. Since each of the components of a pair is an integer between 1 to $\log \log n$, we use $2 \log \log \log n$ bits to code the value of each pair. The total space to code the values of all pairs of T is $2 \log \log n \log \log \log n < \frac{1}{2} \log n$. Thus $I(T)$ fits into a single word, and there are at most $\sqrt{n}$ possible values for $I(T)$. Notice that if $I(T_1) = I(T_2)$ then the $\min_{T_1}(a, b) = \min_{T_2}(a, b)$, and we say that T_1 is *equivalent* to T_2. By the observation above, there are at most $\sqrt{n}$ equivalence classes of trees. We can find a *canonical tree* of each class by looking at all possible values of $I(T)$.

In the preprocessing phase, we build all *canonical trees* and store for each tree T and each pair of nodes u and v in T, the edge of minimum weight on the path between u and v. We store this information in a table whose keys are the identifier of T, u and v. We also construct in the preprocessing phase for each canonical tree T' two tables A and B. In A we have an entry $A[v][i]$

for each $v \in T'$ and $1 \leq i \leq |T'|+1$. The value stored in $A[v][i]$ is the identifier of the new tree that we get if we add to T' a vertex u as a child of v with $\text{rank}((u,v)) = i$. The table B is stored in order to perform change-weight efficiently. We have an entry $B[v][i]$ for each leaf $v \in T'$, and $1 \leq i \leq |T'|$, that contains the identifier of the new tree that we get if we change the rank of $(v, p(v))$ to i and shift all other ranks accordingly. By the arguments above, the total space used to save this information is $o(n)$.

It is easy to see that given this information we can perform a query on a micro tree in $O(1)$ time by extracting this information from the tables of the tree whose identifier is $I(T)$.

We implement the operation add-leaf$(v, p(v), c)$ on a micro tree T as follows. Let $e = (v, p(v))$. We assign v the number $|T|+1$. We add $w(e)$ to the q-heap of T. We perform the query rank$(w(e))$ on the q-heap and get the number of edges in T whose weight is smaller than $w(e)$. This operation changed the ranks of edges in T thus $I(T)$ has changed. We use the value of $A[p(v)][rank(e)]$ of the table associated with $I(T)$ to get the new identifier of T.

We now describe the implementation of change-weight(v, c). Let v be a leaf. Let $e = (v, p(v))$. To update the micro tree, we simply delete $w(e)$ from the q-heap. We set $w(e) = c$ and insert c into the q-heap. We perform the query rank(c) on the q-heap and get the new rank of e, let i be the new rank of e. We update $I(T)$ using the value of $B[v][i]$ of the tree whose identifier is $I(T)$.

3.3 Combining the macro algorithm and the micro algorithm

We combine the macro algorithm and the micro algorithm to reduce the memory size and the running time of add leaf. Let $r_0 = \lfloor \log\log n - 1 \rfloor$. Let $M = 2^{r_0}$. Then, $\frac{1}{4}\log n < M \leq \frac{1}{2}\log n$. A macro node is a vertex $v \in T$ such that $r(v) \geq r_0$. There are at most $n/2^{r_0}$ macro nodes in T. It is easy to see that if $s(v), d(v) \geq M$, then there is a macro node among the first M ancestors to v. We save for each node v the value jumpM1$[v]$ that contains the first proper ancestor of v whose depth is divisible by M. We have that first proper ancestor of v which is a macro node is either jumpM1$[v]$ or jumpM1[jumpM1$[v]$]. We also define jumpM2$[v]$ to be the edge of minimum weight on the path between v and jumpM1$[v]$.

Let T^M be the macro tree whose nodes are all macro nodes of T. Then $|T^M| = O(n/\log n)$, and we can use the macro algorithm on it using $O(n)$ space. Let y be a node in T^M and let z be the parent of y in T^M. Let e be the edge of minimum weight on the path between y and z in T. Then, the weight of the edge (y, z) in T^M is the weight of e. We also save a pointer from (y, z) to e.

We also divide the nodes of T into small trees of size at most M. We will later describe how these small trees are maintained. The small trees use linear space and support all operations in $O(1)$ time. Let μ be such small tree. We denote by root(μ) the root of the tree μ. We have that if $|\mu| < M$ then all the descendants of root(μ) are contained in μ. Let $\mu(v)$ denote the small tree containing v. It follows that if $\mu(parent(root(\mu(v))))$ exists then $|\mu(parent(root(\mu(v))))| = M$. We keep for each small tree μ the tables levelancM1$[\mu]$ and levelancM2$[\mu]$, of size $|\mu|$. The value of levelancM1$[\mu][i]$, $1 \leq i \leq |\mu|$ is the ancestor of root(μ) of depth $d(\text{root}(\mu)) - i$. The value of levelancM2$[\mu][i]$, is the edge of minimum weight on the path from $root(\mu)$ to levelancM1$[\mu][i]$. It is easy to see that the total space used to store these tables is $O(n)$.

We now describe the implementation of a query. We assume that we can perform min queries on the macro trees and on the small trees in $O(1)$ time. Assume we want to find the edge of minimum weight on the path between u and v. By the description of the macro algorithm, using the nca query we can reduce this problem to the problem of finding the edge of minimum weight on the path between u and z where z is an ancestor of u.

Notice that the distance in the original tree T between any macro node and its parent is ex-

actly M. Let x be the first proper ancestor of u which is a macro node, then $x = \text{jumpM1}[u]$ or $x = \text{jumpM1}[\text{jumpM1}[u]]$. Let y be the last macro node on the path between u to z. If $d(x) - d(z) < 0$, then there are no macro nodes on the path from u to z and we set $y = u$. Otherwise, we find the edge f of minimum weight between u and x in $O(1)$ time using the values of jump2M$[u]$ and possibly of jumpM2[jumpM1$[u]$]. The depth of y in T^M is $d = \lfloor \frac{d(x)-d(z)}{M} \rfloor$. If $d > 0$ then $y \neq x$. We use the query min'(x, d) of the macro tree to find the pair (f', y). The edge f' is the edge in T^M of minimum weight on the path from x to y. We extract from f' the edge e' in T of minimum weight on the path from x to y. We set f to be the edge of minimum weight among f and e'.

We now we need to find the edge of minimum weight on the path from y to z. We have that $d(y) - d(z) \leq M$. We do this as follows.

1. Let $\mu(y)$ be the small tree containing y. If $d(root(\mu(y))) \leq d(z)$, then z is contained in $\mu(y)$ and we return the edge of minimum weight among f and edge returned by the query min(y, z) on the small tree $\mu(y)$.

2. Otherwise, z is not in $\mu(y)$. Let $q = p(root(\mu(y)))$. We update f to be the edge of minimum weight among f and the edge returned by the query min$(y, root(\mu(y)))$ on the small tree $\mu(y)$ and the edge $(root(\mu(y)), q)$. Let $\mu(q)$ be the small tree containing q. If $d(root(\mu(q))) \leq d(z)$, then z is contained in $\mu(q)$ and we return the edge of minimum weight among f and the edge returned by the query min(q, z) on the small tree $\mu(q)$.

3. Otherwise, z is not in $\mu(q)$. Update f to be the edge of minimum weight among f and the edge returned by the query min$(q, root(\mu(q)))$ on the small tree $\mu(q)$. By the observation above, $|\mu(q)| = M$. We also have that $d(q) - d(z) \leq M$. Thus entry $d(q) - d(z)$ exists in levelancM2$[\mu(q)]$. We return the edge of minimum weight among f and levelancM2$[\mu(q)][d(q) - d(z)]$.

We now describe the implementation of add-leaf(v, w, c). If $|\mu(w)| = M$, v becomes the root of a new small tree and we set levelancM1$[\mu(v)][1] = w$, levelancM2$[\mu(v)][1] = (v, w)$. Otherwise, we add v to the small tree $\mu(w)$ in $O(1)$ time. We also add entry $|\mu(v)| = j$ to levelancM1 and to levelancM2 as follows. Let levelancM1$[j-1] = z$. We set levelancM1$[j] = p(z)$. We set levelancM2$[j]$ to be the edge of minimum weight among levelancM2$[j-1]$ and $(z, p(z))$.

We update jumpM1$[v]$ and jumpM2$[v]$ in $O(1)$ time as follows. If $M | d(p(v))$, then jumpM1$[v] = p(v)$, and jumpM2$[v] = (v, p(v))$. Otherwise, jumpM1$[v]$ = jumpM1$[p(v)]$, and we set jumpM2$[v]$ to be the edge minimum weight among jumpM2$[p(v)]$ and $(v, p(v))$. We may need to add $y = \text{jumpM1}[v]$ to the macro tree T^M, if $s(y)$ has increased to 2^M. We can do it in $O(\log n)$ time as described in the implementation of add-leaf of the macro tree. Since $|T^M| \leq n/\log n$, this yields an amortized cost of $O(1)$ time per such an insert. Alstrup and Holm showed that by doing in each add-leaf a constant number of operations we can actually do it $O(1)$ worst case time.

We now describe the implementation of change-weight(v, c). Let v be a leaf of T, then since $r(v) = 0$, v is not a macro node. We update jumpM2$[v]$ as in the previous operation. If v is the root of small tree μ, then $|\mu| = 1$ and we do nothing. Otherwise, We call change-weight(v, c) on the small tree containing v, (see below).

We now describe the implementation of the small trees. Alstrup and Holm [2], implemented in their level ancestor algorithm, each small tree as a micro tree. Their micro trees were of size $O(\log n)$. We however, can not use micro trees of size $O(\log n)$ since our micro trees use the q-heaps [5] data structure. The size of the q-heap used for a micro tree T' is $|T'|$. Given a set of n elements, the

maximum allowed size for a q-heap is $O(\log^{1/4} n)$. Thus our micro trees have to be no larger than $O(\log^{1/4} n)$.

We add another layer to the algorithm and implement a small tree using the macro algorithm and micro trees. Let r'_0 be $\lfloor \log\log\log n - 1 \rfloor$. Let $m = 2^{r'_0}$. Then, $\frac{1}{4}\log\log n < m \leq \frac{1}{2}\log\log n$. A macro node of a small tree T' is a vertex $v \in T'$ such that $r(v) \geq r'_0$. There are at most $|T'|/2^{r'_0}$ macro nodes in T'. Let T^m be the macro tree whose nodes are all macro nodes of T'. Then $|T^m| = O(|T'|/\log\log n)$ and the space required for T^m is $O(|T'|)$. We also divide the nodes of T' into trees of size at most $m < \log\log n$ implemented as micro trees. It is easy to see the with this implementation we support all operations in $O(1)$ time and use $O(n)$ space. Specifically, change-weight(v, c) is supported by updating jumpm2$[v]$ in $O(1)$ time as described above, and by calling change-weight(v, c) on the micro tree containing v.

3.4 The implementation of add-root

Last, we describe how to implement incremental trees, that is how to support also the add-root operation in constant time. Each incremental tree is built of a path of nodes which we call the *root path*. We implement the root path as a partial incremental tree. Each node v on the root path is the root of a partial incremental tree. An add-root operation on an incremental tree is implemented by adding a leaf to the root path thereby extending the path by another node. The new node is the root of the incremental tree. Other operations are implemented by constant number of operations on the appropriate partial incremental trees.

4 A data structure for rooted trees

We use the following definition of Ackermann's function

$$\begin{array}{rcll} A(i,1) & = & 2 & i \geq 1 \\ A(1,j) & = & 2^j & j \geq 1 \\ A(i,j) & = & A(i-1, A(i,j-1)) & i,j \geq 2 \end{array}$$

and the inverse functions

$$\begin{array}{rcl} a(i,n) & = & \min\{j \mid A(i,j) \geq n\} \\ \alpha(m,n) & = & \min\{i \mid A(i, \lceil m/n \rceil) \geq n\} \quad m, n \geq 1\ . \end{array}$$

Assume for now that we know the number of operations m ahead of time[2] and let $\ell = \alpha(m,n)$. We denote a tree in our forest of rooted trees by T. We denote by $p(v)$ the parent of a node v, and by $|T|$ the number of vertices in T.

Our forest is represented using a recursive family of data structures. At the top level each tree T in the forest is a member of the data structure D_ℓ. Each tree in D_ℓ is classified to a *universe*. There are $a(\ell, n)$ universes $0, \cdots, a(\ell,n) - 1$ in D_ℓ. The size of the tree T determines its universe as follows. If $|T| < 4$ then T is in universe 0. Otherwise, if $2A(\ell, i) \leq |T| < 2A(\ell, i+1)$ then T is in universe i. (Note that $2A(\ell, 1) = 4$.)

Let T be in universe $i > 0$. The vertices of T are partitioned into *clusters*. Each cluster is a subtree of T that contains at least $2A(\ell, i)$ vertices. Let T' be the tree obtained from T by contracting each cluster into a single node. The tree T' is represented using the data structure $D_{\ell-1}$ which is

[2] If m is not known ahead of time then we can globally rebuild the structure when $m = 2n$, and subsequently every time m is doubled. This does not affect the time bounds.

defined analogously. The last data structure in this recurrence is D_1. In D_1 each tree consists of a single cluster (so the tree with this cluster contracted is a singleton which is not represented using a recursive structure).

In the data structure D_j we have $a(j,n)$ universes $0, \cdots, a(j,n)-1$. Let H be a tree in D_j. If $|H| < 4$ then H is in universe 0 and otherwise if $2A(j,i) \leq |H| < 2A(j,i+1)$ then H is in universe i. If H is in universe $i > 0$ then the vertices of H are partitioned into *clusters*. Each cluster is a subtree of H that contains at least $2A(j,i)$ vertices. The tree H' obtained by contracting each cluster, is represented using the data structure D_{j-1}.

Consider a tree T in our forest. The tree T is a member of D_ℓ. If $|T| \geq 4$ then a tree T', obtained from T by contracting its clusters, is in $D_{\ell-1}$. Similarly, if $|T'| \geq 4$ then a tree T'' obtained by contracting clusters of T', is a member of $D_{\ell-2}$ and so on. We can also think of T'' as obtained from T by contracting even larger subtrees (each such subtree is a cluster of T' which is a cluster of clusters of T). When thinking of T as a member of D_ℓ we denote it by T_ℓ. We denote by $T_{\ell-1}$, the tree T' corresponding to T in $D_{\ell-1}$. In general we denote by T_j the tree corresponding to T in D_j. For $j < \ell$, the tree T_j contains all the edges of T that connect two clusters of T_{j+1}. See Figure 1(A).

We also use the following definitions. Let v be a vertex in T. We define $C_{\ell+1}(v) = v$. We denote by $C_\ell(v)$ the cluster in T_ℓ that contains v. The cluster $C_\ell(v)$ is a subtree of T. We define recursively $C_j(v)$ for $j < \ell$ to be the cluster of T_j that contains $C_{j+1}(v)$.

One can think of $C_j(v)$ as a subtree of T by substituting the subtrees of T corresponding to all clusters $C_{j+1}(w)$ contained in $C_j(v)$. We also refer to $C_j(v)$ as a node of T_{j-1}.

For $j \leq \ell+1$, we define $r(C_j(v))$ to be the root of the subtree of T that $C_j(v)$ represents. Let $e = (x,y) \in T = T_\ell$ and let j be the smallest level such that $C_{j+1}(x) \neq C_{j+1}(y)$. We define the *level* of e to be j and denote it by $\text{level}(e) = j$. A copy of the edge e appears in each T_i, for $j \leq i \leq \ell$. We denote by $e(i)$ the copy of e that appears in T_i. The edge e is contracted into the cluster $C_j(x) = C_j(y)$ in T_j, and therefore does not exist in T_i, for any $i < j$. We sometimes use e when in fact we refer to $e(j)$; the context will make clear which edge we refer to.

The next lemma proves that for a tree T, $\sum_{j=1}^{\ell} |T_j| = O(|T|)$.

Lemma 4.1 *Let $|T| = k$. $\sum_{j=1}^{\ell} |T_j| = O(k)$.*

Proof: The claim follows since by definition, $|T_j| < \frac{1}{2}|T_{j+1}|$, for any $j < \ell$, and $|T_\ell| = k$ □

For each tree T in our forest and for every level j such that T_j exists, we maintain T_j. In addition, the node representing the cluster C_j has a pointer to $r(C_j)$, a pointer to the cluster of level $j+1$ which is the root of C_j, and a pointer to the cluster of level $j-1$ containing C_j, if it exists. The edge representing a copy $e(j)$ of the edge e, has a pointer to e.

Let $e = (u,v) \in T$ such that $\text{level}(e) = j$ and assume $v = p(u)$. We store with e a list $L(e)$ of size at most $\ell - j + 1$. Each element in this list is a pair that contains a level i, and the edge of minimum weight on the path from u to $r(C_{i+1}(v))$. The pairs are sorted by decreasing levels. The first element of $L(e)$ is the pair (ℓ, e), and for level $j \leq k < \ell$ we have a pair in $L(e)$ if $r(C_{k+1}(v)) \neq r(C_{k+2}(v))$. We represent $L(e)$ as a doubly linked list with pointers to its first and last entry.

Consider a copy $e(k)$ of e in T_k for some $j \leq k \leq \ell$. Let k' be the smallest level which is not smaller than k for which there is a pair (k', f) in $L(e)$. The *weight* of $e(k)$, denoted by $w(e(k))$, is equal to the weight of f. Lemma 4.1 implies that the total space used to store the lists $L(e)$ is linear. Figure 1(A) illustrates this recursive structure.

Let C_j, $j \leq \ell$, be a cluster of T_j. We represent C_j as an incremental tree whose nodes are clusters of T_{j+1}. We keep a pointer from C_j to the incremental tree that represents it. We also sometimes refer to C_j as the incremental tree itself. Let $e = (u,v)$ be an edge of level j. Then, $e(j)$ is contained

in a cluster $C_j(u) = C_j(v)$, and in the incremental tree representing it. The weight of $e(j)$ in this incremental tree is $w(e(j))$. If T is in universe 0, ($|T| < 4$), we think of T as a single cluster C and represent C as an incremental tree.

4.1 The implementation of a path-min query

Assume that x and y are two vertices in the same tree T and let P_{xy} be the path from x to y in T. We want to find the edge of minimum weight on P_{xy}. Let k be the largest level for which $C_k(y) = C_k(x)$[3]. We define a recursive procedure $pathmin_k(x, y, e_x, e_y)$, where either e_x or e_y may be null. If $e_x = (x', x)$ is not null, then e_x is an edge incident to x which is of level $< k$ and we have a pointer to the pair in $L(e_x)$ associated with level k. Moreover, x is the parent of x'. Similarly, if $e_y = (y', y)$ is not null, then e_y is an edge incident to y which is of level $< k$ and we have a pointer to the pair in $L(e_y)$ associated with level k and y is the parent of y'. If e_x is not null, let $a = x'$ otherwise let $a = x$. Similarly, if e_y is not null, let $b = y'$, otherwise let $b = y$. The procedure $pathmin_k(x, y, e_x, e_y)$ finds the edge of minimum weight on P_{ab}.

We first assume that both $e_x = (x', x)$ and $e_y = (y', y)$ exist. We will relax these assumption later. The procedure $pathmin_k(x, y, e_x, e_y)$ works as follows. The base case is when $k = \ell$. In this case we perform $\min_{C_\ell(x)}(x, y)$ and get the edge f of minimum weight on P_{xy}. We return the edge of minimum weight among $\{e_x, e_y, f\}$. So assume now that $k < \ell$. If $C_{k+1}(x) = C_{k+1}(y)$ then we advance in $L(e_x)$ and in $L(e_y)$ at most one step to the pair associated with level $k+1$, and return the answer of $pathmin_{k+1}(x, y, e_x, e_y)$. If $C_{k+1}(x) \neq C_{k+1}(y)$ then we find $C_{k+1}(z) = nca_{C_k(x)}(C_{k+1}(x), C_{k+1}(y))$ and perform one of the following cases.

Case 1: $C_{k+1}(z) \neq C_{k+1}(x)$ **and** $C_{k+1}(z) \neq C_{k+1}(y)$**.** Let $C_{k+1}(x_1)$ be the node in $C_k(x)$ on the path from $C_{k+1}(x)$ to $C_{k+1}(z)$ that precedes $C_{k+1}(z)$, and assume that $x_1 = r(C_{k+1}(x_1))$. Let $C_{k+1}(y_1)$ be the node in $C_k(x)$ on the path from $C_{k+1}(y)$ to $C_{k+1}(z)$ that precedes $C_{k+1}(z)$, and assume that $y_1 = r(C_{k+1}(y_1))$. Let $\hat{x}$ be the parent of x_1 in T. Note that $\hat{x} \in C_{k+1}(z)$ and the edge $(x_1, \hat{x})$ is of level k. Similarly, let $\hat{y}$ be the parent of y_1 in T. Note that $\hat{y} \in C_{k+1}(z)$ and the edge $(y_1, \hat{y})$ is of level k. The path $P_{x'y'}$ splits into five disjoint parts: (a) from x' to $r(C_{k+1}(x))$, (b) from $r(C_{k+1}(x))$, to x_1, (c) from x_1 to y_1, (d) from y_1 to $r(C_{k+1}(y))$, (e) from $r(C_{k+1}(y))$ to y'.

The minimum edge on part (a) is stored with the pair of $L(e_x)$ associated with level k. We find the minimum edge on part (b) by a $\min_{C_k(x)}(C_{k+1}(x), C_{k+1}(x_1))$ query. We find the minimum edges on parts (d) and (e) symmetrically. Finally to find the minimum edge on part (c) we recursively perform $pathmin_{k+1}(\hat{x}, \hat{y}, (x_1, \hat{x}), (y_1, \hat{y}))$. Since $(x_1, \hat{x})$ is of level k the last or the next to last pair in $L((x_1, \hat{x}))$ is the pair associated with level $k+1$ so we can access it in $O(1)$ time. The situation with respect to $(y_1, \hat{y})$ is analogous. We return the edge of smallest weight among the minimum weight edges in each of the five parts.

Case 2: $C_{k+1}(z) = C_{k+1}(x)$ **and** $C_{k+1}(z) \neq C_{k+1}(y)$**.** Let $C_{k+1}(y_1)$ be the node in $C_k(x)$ on the path from $C_{k+1}(y)$ to $C_{k+1}(z)$ that precedes $C_{k+1}(z)$, and assume that $y_1 = r(C_{k+1}(y_1))$. Let $\hat{y} = p(y_1)$. Here we split $P_{x'y'}$ into three parts: (a) from x' to y_1, (b) from y_1 to $r(C_{k+1}(y))$, (c) from $r(C_{k+1}(y))$ to y'.

We find the minimum edge on parts (b) and (c) as we did in the previous case. We advance at most one step in $L(e_x)$ to the pair associated with level $k+1$ and we find the minimum edge on part (a) by performing $pathmin_{k+1}(x, \hat{y}, e_x, (y_1, \hat{y}))$. As in the previous case the pair associated with

[3]The level k always exists, since there exists a level j such that T_j is in universe 0. The tree T_j consists of a single cluster $C_j = C_j(x) = C_j(y)$

level $k+1$ in $L((y_1, \hat{y}))$ is either the last or the next to last. We return the edge of smallest weight among the minimum weight edges in each of the three parts.

Case 3: $C_{k+1}(z) = C_{k+1}(y)$, and $C_{k+1}(z) \neq C_{k+1}(x)$. This case is symmetric to the previous case.

If e_x is null and we perform Case 1 or Case 3, then we compute the minimum on the path from x to $r(C_{k+1}(x))$ by another procedure which we call *min-root*.

The procedure *min-root*(x, k) finds the edge of minimum weight on the path between a vertex x and $r(C_k(x))$ as follows. Let $b = r(C_k(x))$. If $k = \ell$ return the result of the query $\min_{C_k(x)}(x, b)$. If $C_{k+1}(x) = C_{k+1}(b)$ perform *min-root*$(x, k+1)$. Otherwise, $C_{k+1}(x) \neq C_{k+1}(b)$, and we perform $\min_{C_k(x)}(C_{k+1}(x), C_{k+1}(b))$, and get in $O(1)$ time the edge f_1 of minimum weight on the path between $r(C_{k+1}(x))$ to b. We also perform *min-root*$(x, k+1)$ to find the edge f_2 of minimum weight on the path between x and $r(C_{k+1}(x))$. We return the edge of smaller weight among f_1 and f_2.

We answer the *pathmin* query by finding the maximum level k such that $C_k(x) = C_k(y)$ and calling $pathmin_k(x, y, null, null)$.

It is clear that *min-root* runs in $O(\ell - k)$ time. Notice that if e_x is not null when we call $pathmin_k(x, y, e_x, e_y)$, then it would not be null also in the recursive call invoked by this *pathmin*. It follows that while performing $pathmin_k(x, y, null, null)$, we call twice to *min-root* with level at least $k+1$, and other than these two calls it takes $O(1)$ time per level between k and ℓ. So *pathmin* at level k also takes $O(\ell - k)$ time.

4.2 Link of rooted trees

Let x be a vertex in a tree T^1_ℓ and let y be the root of T^2_ℓ. The operation $\text{link}_\ell(x, y)$ combines T^1_ℓ and T^2_ℓ to a new tree T_ℓ by adding the edge $e = (x, y)$ and making x the parent of y. When combining T^1_ℓ and T^2_ℓ we also have to combine $T^1_{\ell-1}$ and $T^2_{\ell-1}$ etc. Therefore the implementation of link is recursive: We define the recursive operation $\text{link}_j(C_{j+1}(x), C_{j+1}(y))$ where $C_{j+1}(x)$ is a node in T^1_j and $C_{j+1}(y)$ is the root of T^2_j. The operation $\text{link}_j(C_{j+1}(x), C_{j+1}(y))$ combines T^1_j and T^2_j by making $C_{j+1}(x)$ the parent of $C_{j+1}(y)$ in the resulting tree T_j.

Let q_1 be the universe of T^1_j and let q_2 be the universe of T^2_j. To perform $\text{link}_j(C_{j+1}(x), C_{j+1}(y))$ we perform the appropriate of the following four cases.

Case 1: $|T^1_j| + |T^2_j| \geq 2A(j, \max\{q_1, q_2\} + 1)$. Create a new tree T_j in universe $\max\{q_1, q_2\} + 1$ containing a single, initially empty, cluster C. Traverse T^1_j top down and insert all nodes of T^1_j to C by performing add-leaf operations. Then insert $C_{j+1}(y)$ as a child of $C_{j+1}(x)$ into C by another add-leaf operation which adds $e(j)$ to the new cluster. Finally insert all the nodes of T^2_j into C, top-down by performing add-leaf operations.

For the add-leaf operation that inserts $C_{j+1}(y)$ as a child of $C_{j+1}(x)$ we have to compute the value of $e(j)$. If $j = \ell$ we create the list $L(e)$ and add to it the pair (ℓ, e), and the weight of $e(j)$ in C is the weight of e. Assume that $j < \ell$. We compute the value associated with $e(j)$ as follows. Let $x' = r(C_{j+1}(x))$. If $x' = r(C_{j+2}(x))$ then we are done since the value of $e(j)$ is the same as the value of $e(j+1)$. Otherwise, we add a pair with level j to $L(e)$. The value of $e(j)$ is the edge of minimum weight on the path between y and x'. We find this value as follows.

Since $x' \neq r(C_{j+2}(x))$ it follows that $C_{j+2}(x) \neq C_{j+2}(x')$. Let f_1 be the edge of T of minimum weight on the path between y to $r(C_{j+2}(x))$. Let f_2 be the edge of T of minimum weight on the path between $r(C_{k+2}(x))$ to x'. The smaller among the weights of f_1 and f_2 is the weight of $e(j)$ and the corresponding edge should be in the new pair of level j added to $L(e)$. We find f_1 in the

last pair currently in $L(e)$. We find f_2 by performing $\min(C_{j+2}(x), C_{j+2}(x'))$ query in $C_{j+1}(x)$. It follows that the new entry in $L(e)$ is computed in $O(1)$ time.

Each edge f such that $f(j)$ is either in T_j^1 or T_j^2 becomes an edge of T_j. The weight of $f(j)$ does not change and the level of f becomes j. We discard any elements of $L(f)$ that refer to levels $< j$.

Case 2: $q_1 > q_2$**.** We traverse the clusters of T_j^2 top down starting from $C_{j+1}(y)$ inserting them one by one to $C_j(x)$ by performing add-leaf operations to $C_j(x)$.

We start by inserting $C_{j+1}(y)$ as a child of $C_{j+1}(x)$. The edge connecting these two clusters is $e(j)$. We compute its weight and update $L(e)$ exactly as in Case 1.

The partition of T_j^2 into clusters of levels $\leq j$ is discarded. Each edge $f(j)$ of T_j^2 becomes an edge of T_j. The weight of $f(j)$ does not change and the level of f becomes j. We discard any elements of $L(f)$ that refer to levels $< j$.

Case 3: $q_1 < q_2$**.** We would like to add all nodes in T_j^1 to the cluster $C_j(y)$. We can add the nodes along the path from $C_{j+1}(x)$ to the root of T_j^1 by performing an add-root operation on each of these nodes bottom-up. Then we can add all other nodes of T_j^1 by performing add-leaf operations. We need to be careful however since an add-root operation to $C_j(y)$ changes the root of $C_j(y)$ and thereby may change the weight (and the corresponding edge) of any edge $e'(j-1)$ incident to $C_j(y)$, see Figure 2. This requires modifications to $L(e')$ and the incremental trees containing these edges. Similarly, for any level $k < j$ the root of $C_k(y)$ changes and thereby the weight (and the corresponding edge) of any edge $e'(k-1)$ incident to $C_k(y)$.

To define the actions which we take in this case we distinguish the following kind of cluster: Let C_k be a cluster of level k, and let C_{k+1} be the cluster of level $k+1$ which is the root of C_k. The cluster C_k is called an *unary root cluster*, if C_{k+1} is the last node on the root path (See Section 3.4), of the incremental tree representing C_k, and the partial incremental tree rooted by C_{k+1} does not contain any other vertex except C_{k+1}. See Figure 3.

Subcase 3.1: For every $q < j$ **the cluster** $C_q(y)$ **is an unary root cluster.** We compute the value of $e(j)$ as in Case 1 and update $L(e)$ if this value is different from the value of $e(j+1)$. Let f_1 denote the edge which determines the weight of $e(j)$. Recall that f_1 is the edge of minimum weight on the path from y to $r(C_{j+1}(x))$.

We use the weight of f_1 to insert the cluster $C_{j+1}(x)$ (and the edge $e(j)$) into $C_j(y)$ by an add root operation.

For every $q < j$ we update $L(f)$ for the edge $f = (z, w)$ such that $\text{level}(f) = q$, and $f(q)$ is the single edge in the incremental tree $C_q(y)$, that is incident to $C_{q+1}(y)$ $(= C_{q+1}(z))$ in $C_q(y)$ $(= C_q(w))$ as follows. Let f_2 be the edge in the last pair of $L(f)$ and let t be the level of the last pair in $L(f)$. By definition f_2 is the edge of minimum weight along the path from w to y in T. If $t \geq j$ then we add a pair to $L(f)$. This pair contains the edge of minimum weight among f_1 and f_2, and its level is $j-1$. If $t < j$, then we delete the last pair of $L(f)$ and insert instead a pair with the edge of minimum weight among f_1 and f_2 and level t.

Subcase 3.2: For some $k' < j$ **the cluster** $C_{k'}(y)$ **is not an unary root cluster.** Let $k < j$ be smallest level such that the cluster $C_k(y)$ is not an unary root cluster.

We compute the edge f_1 whose weight is the weight of $e(j)$ as in Case 1 and update $L(e)$ if this value is different from the value of $e(j+1)$.

Now instead of inserting $C_{j+1}(x)$ into $C_j(y)$ we create a new cluster at level j consisting only of the single node $C_{j+1}(x)$. This cluster becomes $C_j(x)$. Similarly, if $k < j-1$ then for each level q, $k < q < j$ we create a new singleton cluster, $C_q(x)$ at level q containing $C_{q+1}(x)$. Finally we insert the cluster $C_{k+1}(x)$ into $C_k(y)$ by performing an add-root operation. The level of e becomes k (rather than j in Subcase 3.1). However since $r(C_q(x))$ is the same for any $k \leq q \leq j$ then we do

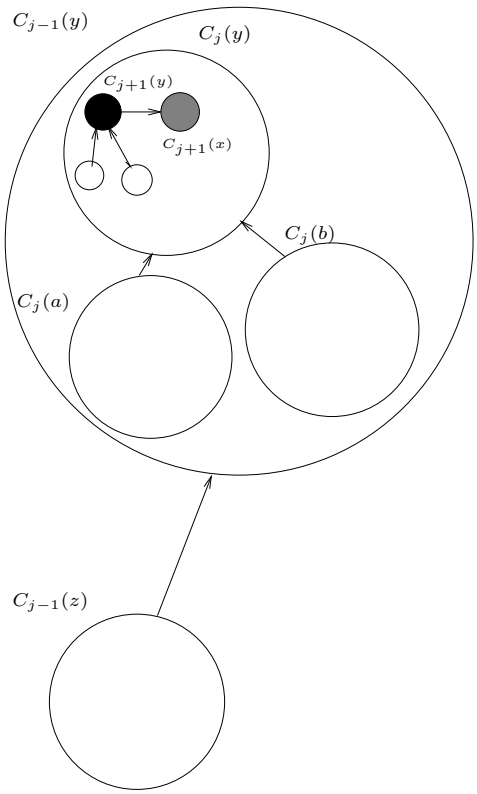

Figure 2: If we add $C_{j+1}(x)$ as the root cluster of $C_j(y)$, then we will have to update the weight of all edges that are incident to $C_k(y), k \leq j$. In this example, the weight of edge going out from $C_j(a)$ to $C_j(y)$, the weight of the edge going out from $C_j(b)$ to $C_j(y)$ and weight of the edge going from $C_{j-1}(z)$ to $C_{j-1}(y)$ will change. Therefore we distinguish between unary root clusters and non unary root clusters in Case 3.

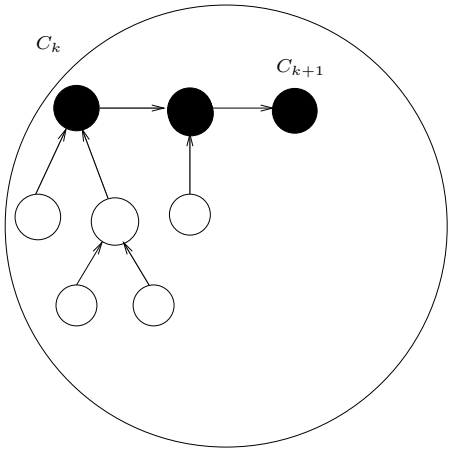

Figure 3: C_k is an unary root cluster. If we perform add-root in $T_j, j \geq k+1$, we only have to update the weight of one edge of level k, the edge incident to C_{k+1}.

not need to further modify $L(e)$. The edge $e(k)$ is inserted into $C_k(y)$, and has the same weight as of the edge f_1.

For every $q \leq k-1$ we update $L(f)$ for the edge $f = (z, w)$ such that $f(q)$ is the single edge in $C_q(y)$, incident to $C_{q+1}(y)$. We perform these updates as in Subcase 3.1.

In both Subcase 3.1 and Subcase 3.2 after combining $C_{j+1}(x)$ and T_j^2 we have to combine the rest of T_j^1 with T_j^2. Let $C_j(x)$ be the cluster at the root of T_j^2. (This cluster is either $C_j(y)$ with $C_{j+1}(x)$ as the root if Subcase 3.1 has been performed, or a cluster containing a single node $C_{j+1}(x)$ if Subcase 3.2 has been performed.) We traverse the path from $C_{j+1}(x)$ to the root of T_j^1. Let $C_{j+1}(z)$ be a node on this path and let f be the edge connecting it to its predecessor on this path. We insert $C_{j+1}(z)$ and f to $C_j(x)$ by an add-root operation. After the add-root operation the edge f is of level j. So we delete items from $L(f)$ associated with levels smaller than j.

Following these operations, for all $q < j$, $C_q(x)$ is an unary root cluster. For every $q < j$ we now update $L(f)$ for the edge $f = (z, w)$ such that $f(q)$ is the single edge in $C_q(x)$ incident to $C_{q+1}(x) = C_{q+1}(z)$. This is done in a way similar to the way we updated the lists of these edges in Subcase 3.1. Let f' be the edge of minimum weight on the path from $r(C_{j+1}(x))$ to the root node of T^1. We compute f' while doing the add root operations in the previous paragraph: it is the edge of smallest weight among the edges that determine the weight of the edges $e'(j)$ that are inserted into $C_j(x)$ by the add root operations. Let f_2 be the edge in the last pair of $L(f)$ and let t be the level of the last pair in $L(f)$. Then, f_2 contains the edge of minimum weight on the path between w and $r(C_{j+1}(x))$. We update the last pair of $L(f)$ as follows. If $t \geq j$ then we add a pair to $L(f)$. This pair contains the edge of minimum weight among f' and f_2, and its level is $j-1$. If $t < j$, then we delete the last pair of $L(f)$ and insert instead a pair with the edge of minimum weight among f' and f_2 and level t.

Finally, we insert all other clusters in T_j^1 to $C_j(x)$. We insert these clusters top-down starting from the clusters that were hanging off of the path from $C_{j+1}(x)$ to the root of T_j^1. Let $C_{j+1}(z)$ be such a cluster and let f be the edge connecting it to its parent in T_j^1 we insert $C_{j+1}(z)$ and f into $C_j(x)$ by an add-leaf operation. We add the leaf as a child of the old parent of $C_{j+1}(z)$ in T_j^1. The edge f after the link is of level j so we delete elements from $L(f)$ associated with levels smaller than j.

Let $C_q(x)$, for $q < j$ be an unary root cluster. Let the edge $f = (z, w)$ be such that $f(q)$ is the single edge in $C_q(x)$ incident to $C_{q+1}(x) = C_{q+1}(z)$. If the weight of $f(q)$ changes, then in addition to updating $L(f)$ we also have to change the weight associated with $f(q)$ in the incremental tree $C_q(x)$. The cluster $C_q(x)$ is an unary root cluster so the node $C_{q+1}(x)$ is a leaf in the partially incremental tree that represents the root path of $C_q(x)$, and $f(q)$ is the edge incident to this leaf. We update the weight of $f(q)$ as specified in Section 3 where we showed that we can update the weight of the edge incident to a leaf in a partially incremental tree in $O(1)$ time. It is easy to see that we update the weight of such edge $f(q)$ at most twice per level q during the link operation. The first update is after we perform add root to $C_{j+1}(x)$, and the second update is after we perform add root to all other cluster nodes of T_{j+1}^1 on the path from $C_{j+1}(x)$ to the root of T_{j+1}^1. It follows that these updates add only a constant factor to our running time.

Case 4: $q_1 = q_2$**.** Create T_j by adding $e(j)$ and making $C_{j+1}(y)$ a child of $C_{j+1}(x)$. Recursively perform $\text{link}_{j-1}(C_j(x), C_j(y))$ combining T_{j-1}^1 and T_{j-1}^2.

If the value of $e(j)$ is different than the value of $e(j+1)$ then we have to add an item to $L(e)$. We compute the value associated with $e(j)$ as in Case 1.

To establish the correctness of our implementation of link we claim that Case 4 cannot occur when linking two trees of level 1 (i.e. when $j = 1$). Let q_1 be the universe of T_1^1, and let q_2 be the

universe of T_1^2. By the definition of the link if $q_1 \neq q_2$ we perform either Case 2 or Case 3. If $q_1 = q_2$ then $|T_1^1|, |T_1^2| \geq A(1, q_1) = 2^{q_1}$. So $|T_1^1| + |T_1^2| \geq 2^{q_1+1} = 2A(1, \max\{q_1, q_2\} + 1)$ and we perform Case 1.

Case 3 of our implementation of link creates small clusters, therefore it is not obvious why Lemma 4.1 still holds. So we classify clusters into two kinds. For each $v \in T$, $C_{\ell+1}(v) = v$ is considered a *good* cluster. Let T_k be in universe $i > 0$. A cluster $C_k(v) \in T_k$ is considered *good* if it contains at least $2A(k, i)$ good clusters of level $k + 1$. The cluster $C_k(v)$ is called *bad* otherwise. We change the definition of $|T_j|$ to be the number of vertices in T_j that correspond to good clusters of level $j + 1$. Then Lemma 4.1 holds with respect to this new definition of the size of a tree. We will show that the number of bad clusters is not larger than the number of good clusters.

For our analysis to work we change the definition of a *universe* and say that T_j is in universe i if the number of *good* clusters of level $j + 1$ that are contained in T_j is in $[2A(k, i), 2A(k, i + 1))$. In Case 1 of the implementation of link we compute $|T_j^1|$ and $|T_j^2|$ according to the new definition of $|T_j|$.

4.3 Analysis

The next lemma proves that in each tree T_k the number of bad clusters of level k is smaller than the number of good clusters of level k.

Lemma 4.2 *Let T be a tree. For each k such that T_{k-1} exists, the number of good clusters of level k is greater than the number of bad clusters of level k.*

Proof: Let T be a tree in the forest and let u be the root of T. Let g_k be the number of good clusters of level k, and let b_k be the number of bad clusters of level k in T. We prove that one of the following holds for each tree T_k.

1. The size of T_k is smaller than 4 (that is T_k is in universe 0), so $b_k = g_k = 0$.
2. If for each $i < k$ such that $C_i(u)$ exists, $C_i(u)$ is an unary root cluster, then $b_k + 1 \leq g_k$. Otherwise, if there exists $i < k$ such that $C_i(u)$ is not an unary root cluster, then $b_k + 2 \leq g_k$.

We prove our claim by induction on the sequence of link operations. The claim trivially holds if $|T| < 4$ since in this case $b_\ell = g_\ell = 0$.

Consider a link between T^1 and T^2 that creates the tree T by adding the edge (x, y) with $x \in T^1$ and $y \in T^2$. Let g_k^1 be the number of good clusters of level k of T^1, and let b_k^1 be the number of bad clusters of level k of T^1. Similarly, let g_k^2 be the number of good clusters of level k of T^2, and let b_k^2 be the number of bad clusters of level k of T^2. By the induction hypothesis $b_k^1 < g_k^1$ and $b_k^2 < g_k^2$. Let g_k and b_k be the number of good and bad clusters of level k, respectively, in the resulting tree T, and let u be the root of T.

Assume that for each $k > j$ Case 4 of $\text{link}_k(C_{k+1}(x), C_{k+1}(y))$ was performed, and for $k = j$ one of the other cases was performed. Let T_j^1 be in universe q_1 and T_j^2 be in universe q_2.

Suppose that Case 1 of the link was performed by $\text{link}_j(C_{j+1}(x), C_{j+1}(y))$ so $|T_j^1| + |T_j^2| \geq 2A(j, max\{q_1, q_2\} + 1)$. Then we create a new tree T_j in universe $\max\{q_1, q_2\} + 1$ containing a single cluster C and we insert all nodes of T_j^1 and of T_j^2 into C. The cluster C is good since it contains at least $2A(j, max\{q_1, q_2\} + 1)$ good clusters of level $j + 1$. The tree T has no bad clusters of level j, so $b_j + 1 = g_j$ as required. Since for $i < j$, $C_i(u)$ does not exist the claim follows for $i \leq j$. For each $j < k \leq \ell$, $g_k = g_k^1 + g_k^2$ and $b_k = b_k^1 + b_k^2$. Thus we get that $b_k + 2 \leq g_k$, and the claim follows.

Suppose that Case 2 of the link was performed by $\text{link}_j(C_{j+1}(x), C_{j+1}(y))$. Then, as in the previous case for each $j < k \leq \ell$, $g_k = g_k^1 + g_k^2$ and $b_k = b_k^1 + b_k^2$ and we get that $b_k + 2 \leq g_k$. Consider now level $k = j$. In this level we inserted all cluster nodes of level $j+1$ of T^2 into $C_j(x)$, since we did not decrease the number of good cluster nodes of level $j+1$ that are contained in $C_j(x)$, $C_j(x)$ can not change from a good cluster to a bad cluster. All other cluster nodes of level j of T are the same as in T^1. Thus we get that $g_j \geq g_j^1$ and $b_j \leq b_j^1$. A similar argument shows that for each $k < j$ we have that $g_k \geq g_k^1$ and $b_k \leq b_k^1$. Since the induction hypothesis was true for clusters of level $k \leq j$ of T^1, it also holds for clusters of level $k \leq j$ of T.

Suppose that Case 3 of the link was performed by $\text{link}_j(C_{j+1}(x), C_{j+1}(y))$. Then, as in the previous case for each $j < k \leq \ell$, $g_k = g_k^1 + g_k^2$ and $b_k = b_k^1 + b_k^2$ and we get that $b_k + 2 \leq g_k$.

Suppose we performed Subcase 3.1. Then for each $q < j$, $C_q(y)$ is an unary root cluster of T^2, and by the induction hypothesis $b_k^2 + 1 \leq g_k^2$, for $k \leq j$. As in the previous case, we have that for each $k \leq j$, $g_k \geq g_k^2$ and $b_k \leq b_k^2$. Thus by the induction hypothesis we get that $b_k^2 + 1 \leq g_k^2$, for $k \leq j$. The structure of T_q for $q < j$ is the same as the structure of T_q^2, thus we have that $C_q(y)$, for $q < j$ is an unary root cluster of T and the claim follows for $k \leq j$ as well.

Suppose we performed Subcase 3.2. Let $q < j$ be the smallest level such that $C_q(y)$ is not an unary root cluster. By the induction hypothesis, we have that for each $q < k \leq \ell$, $b_k^2 + 2 \leq g_k^2$. For each $q < k \leq j$, we create one new bad cluster in T_k. Thus we get that $b_k \leq b_k^2 + 1$, $g_k \geq g_k^2$, and so $b_k + 1 \leq g_k$, for $q < k \leq j$. The induction hypothesis holds, since following the link, we have that for each $k' < j$, $C_{k'}(u) = C_{k'}(x)$ is an unary root cluster of T.

For $k < q$, $C_k(y)$ is an unary root cluster of T^2, so by the induction hypothesis $b_k^2 + 1 \leq g_k^2$ for $k \leq q$. In T, for $k \leq q$, we have that, $b_k \leq b_k^2$, and $g_k \geq g_k^2$, and so $b_k + 1 \leq g_k$. The induction hypothesis holds since for each $k \leq q$, $C_k(u) = C_k(x)$ is an unary root cluster of T. □

We now show that the total cost of all link operations is $O(n(a(\ell, n) + \ell)) = O(n\ell + m)$. Consider first a single link operation that adds the edge (x, y). Let j be the smallest level for which we perform $\text{link}_j(C_{j+1}(x), C_{j+1}(y))$. Each of the calls $\text{link}_k(C_{j+1}(x), C_{j+1}(y))$ for $k > j$ performs Case 4 and therefore takes $O(1)$ time. Consider the time it takes to perform $\text{link}_j(C_{j+1}(x), C_{j+1}(y))$ excluding the add-root and add-leaf operations. In case $\text{link}_j(C_{j+1}(x), C_{j+1}(y))$ performs Case 1 or Case 2 then it clearly takes $O(1)$ time. In case $\text{link}_j(C_{j+1}(x), C_{j+1}(y))$ performs Case 3 then it has to update the weight of up to j edges incident to roots of unary root clusters. Since it takes $O(1)$ time to update each such edge $\text{link}_j(C_{j+1}(x), C_{j+1}(y))$ takes $O(j)$ time. Summing up we obtain that the cost of the link is $O(\ell)$.

We now show that the total cost of all add-root and add-leaf operations performed by the link operations is $O(na(\ell, n))$. Since each add-root and add-leaf moves a cluster into higher universe, we need to bound the number of times a cluster can move from a tree in one universe to a tree in a higher universe. For each level j we bound the number of times clusters of level j can change universe. For $j = \ell + 1$, this number is $a(\ell, n) = \lceil m/n \rceil$. Thus the cost of moving the nodes of level $\ell + 1$ into higher universe over all link operations is $O(na(\ell, n)) = O(m + n)$.

For level $j \leq \ell$ we have the following Lemma that was proven by Gabow [7].

Lemma 4.3 *Let T be a tree in our forest and let j be a level such that $j < \ell$. Then the number of universes that T_j can be in, provided that T_{j+1} is in universe i, is $A(j+1, i)$.*

Proof: By definition $|T_{j+1}| < 2A(j+1, i+1)$. (Recall that $|T_{j+1}|$ is the number of nodes corresponding to good level-$j+2$ clusters in T_{j+1}.) Since each good cluster of level $j+1$ of T contains at least $2A(j+1, i)$ good clusters of level $j+2$ we obtain that $|T_j| \leq 2A(j+1, i+1)/2A(j+1, i)$. Therefore

T_j must be in one of $a(j, 2A(j+1,i+1)/2A(j+1,i))$ universes. The lemma follows from simple properties of Ackermann's function which imply that

$$a(j, 2A(j+1,i+1)/2A(j+1,i)) \leq a(j, A(j+1,i+1)) = a(j, A(j, A(j+1,i))) = A(j+1,i).$$

□

We now count the number of times the universe of the tree T_j containing a good cluster C_{j+1} changes, summed over all levels and universes. By Lemma 4.2 this sum bounds the total number of times the universe of the tree containing any cluster (good or bad) changes. In the following discussion up to Lemma 4.4 we consider only good nodes.

If $j = \ell$ then $C_{\ell+1}$ is some real vertex v. Clearly the universe of the tree containing v may change at most $a(\ell, n) = m/n + 1$ times. If we sum over all n nodes we get that the total number of times the universe of the tree containing a vertex v changes is $m + n$.

Now consider a cluster C_ℓ in a tree of level ℓ. The cluster C_ℓ is created when some link operation at level ℓ performs Case 1. Let i be the universe of the tree of level ℓ containing C_ℓ. Although the tree of level ℓ containing C_ℓ may change and the cluster C_ℓ may grow, the universe of the tree of level ℓ containing C_ℓ is always i. So we can define the *universe* of a cluster of level ℓ to be the universe of the tree containing it. The cluster C_ℓ exists until some link operation of level ℓ decides to insert the vertices of the tree containing C_ℓ into some cluster of another tree of a larger universe in level ℓ. Let S_i be the set of clusters of level ℓ and of universe i in trees of level ℓ. Since each level-ℓ cluster of universe i contains at least $2A(\ell, i)$ nodes, and a node belongs to at most one such cluster throughout the process, then $|S_i| \leq n/2A(\ell, i)$. By Lemma 4.3 the tree of level $\ell - 1$ containing a node which corresponds to a cluster of S_i could be in one of $A(\ell, i)$ universes. So we get that the total number of times the universe of a tree containing a node corresponding to a cluster of S_i changes is at most $n/2$. If we sum this over all $0 \leq i \leq a(\ell, n) = m/n$ we get that the total number of times a cluster of level ℓ changes its universe is $(m + n)/2$.

Consider now a cluster C_{j+1} of level $j + 1 \leq \ell$ and universe i_{j+1}. For $k \geq j + 1$, we define the *universe of level k of C_{j+1}* to be the universe i_k of the tree T_k which gets contracted to the tree T_j in which C_{j+1} is a cluster. Note that throughout the lifetime of C_{j+1}, T_k may change by links, but i_k can not change. For a sequence of universes $i_{j+1}, \ldots, i_\ell$ of levels $j + 1, \ldots, \ell$ we define $S_{i_{j+1},\ldots,i_\ell}$ to be the set of clusters of level $j + 1$ of universe i_k of level k for every $j + 1 \leq k \leq \ell$. Since each real vertex belongs to at most one cluster corresponding to a node of $S_{i_{j+1},\ldots,i_\ell}$ it follows that $|S_{i_{j+1},\ldots,i_\ell}| \leq \frac{n}{2A(\ell,i_\ell)\cdots 2A(j+1,i_{j+1})}$. By Lemma 4.3 the level-j tree containing each cluster of $S_{i_{j+1},\ldots,i_\ell}$ can be in one of $A(j+1, i_{j+1})$ universes. Therefore the universe of a tree containing a node in $S_{i_{j+1},\ldots,i_\ell}$ changes at most $\frac{n}{2A(\ell,i_\ell)\cdots 2A(j+2,i_{j+2})*2}$ times. To sum over all sets $S_{i_{j+1},\ldots,i_\ell}$ note that by Lemma 4.3 the number of possible values i_k can have given i_{k+1} is $A(k+1, i_{k+1})$ for every $j + 1 \leq k < \ell$. Thus, if we also recall that the number of possible values for i_ℓ is $\frac{m}{n} + 1$ we obtain that the number of times a tree containing any cluster of level $j + 1$ can change its universe is $(m + n)/2^{\ell-j}$. The following lemma summarizes what we have just proved.

Lemma 4.4 *The number of times a cluster of level j moves from a tree in one universe to a tree in a higher universe is $O((m + n)/2^{\ell-j})$.*

By Lemma 4.4 if we sum the number of nodes which move from a tree of low universe to a tree of higher universe over all levels we get that the total number of nodes which move from a tree of low universe to a tree of higher universe is $O(m + n)$. It follows from this that the total cost of all link operations is $O(m + n\alpha(m, n))$. Combining this with the analysis of a query in Section 4.1 we get following theorem.

Theorem 4.1 *The data structure presented in this section performs at most n link operations and m path-min queries in $O((m+n)\alpha(m,n))$ time.*

5 An $O(n + m\alpha(m,n)^2)$ data structure for unrooted trees

We use the recursive decomposition of Gabow [7] as in the previous algorithm together with incremental trees. Here the incremental trees do not need to support the add-root operation. Each tree is partitioned into clusters as before and we keep the notation where $C_j(v)$ is the cluster which contains v in T_j. The tree with its clusters contracted is represented recursively. Each cluster C_k of a tree T_k is rooted at a certain node C_{k+1} (which is a cluster of level $k+1$). The cluster C_{k+1} in turn is rooted at some cluster C_{k+2} etc. If we unravel this recursion all the way to its bottom we obtain a vertex v of T_ℓ which is the root of the cluster C_k when thinking of it as a subtree of T. We denote this vertex by $r(C_k)$. The cluster C_k as a rooted subtree of T_k is represented by an incremental tree.

Let C_{j+1} be a node of the cluster C_j. Assume C_{j+1} is not the root of C_j and let C'_{j+1} be the parent of C_{j+1} in C_j. Let $e=(v,w)$ be the edge (of level j) such that $e(j)$ connects C_{j+1} to C'_{j+1} in C_j, so $C_{j+1}=C_{j+1}(v)$ and $C'_{j+1}=C_{j+1}(w)$. We define the j-root of C_{j+1} to be the vertex v, and denote it by $t(C_{j+1})$. This is the "root" of C_{j+1} when considering it as a subtree of C_j. Notice that $t(C_{j+1})$ need not be equal to $r(C_{j+1})$. If C_{j+1} is the root of C_j then we define $t(C_{j+1})$ to be $r(C_{j+1})$.

As in Section 3 a node representing the cluster C_j has a pointer to the root of its subtree in T, to the cluster C_{j+1} which is the root of C_j and to the cluster C_{j-1} containing C_j if such a cluster exists. Let $e=(u,v)\in T$ be an edge of level j such that $C_{j+1}(v)$ is the parent of $C_{j+1}(u)$ in the incremental tree $C_j(v)$ $(=C_j(u))$. We store with e two lists $L_e(v)$ and $L_e(u)$. The list $L_e(v)$ is of length exactly $\ell-j+1$. Entry ℓ of $L_e(v)$ contains e. For level $j\le k<\ell$ entry k of $L_e(v)$ contains the edge f of minimum weight on the path from u to $t(C_{k+1}(v))$ in T. The *weight* of $e(k)$, $w(e(k))$, is equal to the weight of f. If level$(e)=j$ then $e(j)$ is contained in a cluster C_j. The weight $w(e(j))$ is also maintained by the incremental tree representing C_j. Similarly, entry ℓ of $L_e(u)$ contains e, and for level $j\le k<\ell$, entry k contains the edge of minimum weight on the path from v to $t(C_{k+1}(u))$ in T. Notice that since $C_{j+1}(v)$ is the parent of $C_{j+1}(u)$, then $u=t(C_{j+1}(u))$ and e is the value of entry j in $L_e(u)$. Each list is represented as a doubly linked list with pointers to its first and last entry. Lemma 4.1 implies that the total space used to store these lists is linear.

Figure 1(B) illustrates the data structure, and the weights of the edges in different levels.

5.1 The implementation of a path-min query

Our query algorithm is similar to the query algorithm of Section 4.1. Let P_{xy} be the path from x to y. We define a recursive procedure $pathmin_k(x,y,e_x,e_y)$ as in Section 4.1. We first assume that both e_x and e_y exist. Let $e_x=(x',x)$ and let $e_y=(y',y)$.

The procedure $pathmin_k(x,y,e_x,e_y)$ finds the minimum on $P_{x'y'}$ under the assumptions that: (1) $C_k(y)=C_k(x)$, (2) the edge e_x is of level $<k$ (that is $x'\notin C_k(x)$), and we have a pointer to the entry in $L_{e_x}(x)$ associated with level k, and (3) the edge e_y is of level $<k$ (that is $y'\notin C_k(y)$), and we have a pointer to the entry associated with level k in $L_{e_y}(y)$.

The implementation of $pathmin_k(x,y,e_x,e_y)$ is the same as described in Section 4.1. The main difference of our algorithm here and the algorithm of Section 4.1 is when e_x or e_y are not available.

Let $d=t(C_{k+1}(x))$. In Section 4.1 we used the min-root algorithm to find the edge of minimum weight on the path from x to d. Here we cannot use the min-root procedure since $t(C_{k+1}(x))$ may not be the root of $C_{k+1}(x)$ (recall that here each cluster is rooted independently of the higher level clusters containing it).

Instead, if e_x does not exist, we find the edge of minimum weight on the path from x to d as follows. There exists an edge $e' = (d, q)$ of level k, between $C_{k+1}(x)$ $(= C_{k+1}(d))$ and $C_{k+1}(q)$ in $C_k(x)$, where q in on P_{xy}. We locate the entry in $L_{e'}(d)$ associated with level $k+1$ and recursively perform $pathmin_{k+1}(x, d, null, e')$.[4] If e_y does not exist and we have to find the minimum from y to $t(C_{k+1}(y))$, then we do it similarly.

The complexity of $pathmin_k$ is dominated by the number of recursive calls to *pathmin*. To bound this number observe that: (1) Each call to *pathmin* in which e_x (e_y) does not exist makes at most a single call to *pathmin* in which both edges exist and a single call to *pathmin* in which e_x (e_y) does not exist. (2) A call to *pathmin* in which both e_x and e_y exist makes at most a single recursive call to *pathmin* in which e_x and e_y exist.

¿From the first observation follows that there are $O(\ell - k)$ recursive calls to *pathmin* in which e_x does not exist, and at most $O(\ell - k)$ recursive calls to *pathmin* in which e_y does not exist. This together with the second observation imply that the total number of recursive calls initiated by $pathmin_k$ is $O((\ell - k)^2)$.

5.2 The implementation of link

Let x be a vertex in a tree T^1_ℓ and let y be a vertex of T^2_ℓ. The operation $\text{link}_\ell(x, y)$ combines T^1_ℓ and T^2_ℓ by adding the edge $e = (x, y)$. When combining T^1_ℓ and T^2_ℓ we also have to combine $T^1_{\ell-1}$ and $T^2_{\ell-1}$ etc. Therefore the implementation of link is recursive. We define the recursive operation $\text{link}_j(C_{j+1}(x), C_{j+1}(y))$ where $C_{j+1}(x)$ is a node in T^1_j and $C_{j+1}(y)$ is a node in T^2_j. The operation $\text{link}_j(C_{j+1}(x), C_{j+1}(y))$ combines T^1_j and T^2_j by adding the edge (x, y) to the resulting tree T_j. Let q_1 be the universe of T^1_j and let q_2 be the universe of T^2_j. To perform $\text{link}_j(C_{j+1}(x), C_{j+1}(y))$ we perform the appropriate of the following four cases.

Case 1: $|T^1_j| + |T^2_j| \geq 2A(j, \max\{q_1, q_2\} + 1)$**.** Create a new tree T_j in universe $\max\{q_1, q_2\} + 1$ containing a single, initially empty, cluster C. Traverse T^1_j top down and insert all nodes of T^1_j to C by performing add-leaf operations. Then insert $C_{j+1}(y)$ as a child of $C_{j+1}(x)$ into C by another add-leaf operation which adds $e(j)$ to the new cluster. Finally insert all the nodes of T^2_j into C, top-down by performing add-leaf operations.

To insert $C_{j+1}(y)$ as a child of $C_{j+1}(x)$ we have to compute the edges of $L_e(x)$ and $L_e(y)$ associated with level j. The weight of the edge associated with level j in $L_e(x)$ is the weight of $e(j)$ in the incremental tree representing the new cluster.

Notice that the edge of level j in $L_e(y)$ is e. We now show how to update the edge of level j of $L_e(x)$. If $j = \ell$, then the edge of level ℓ of $L_e(x)$ is e. Otherwise, if $j < \ell$, we have to find the edge of minimum weight in T on the path from y to $t(C_{j+1}(x))$. Let $b = t(C_{j+1}(x))$. We do that by a $pathmin_{j+1}(x, b, null, null)$ query in T^1. Let f be the edge returned by this query. The edge of level j of $L_e(x)$ is the edge of minimum weight among (x, y) and f.

Each edge $f = (v, w)$ such that $f(j)$ is either in T^1_j or in T^2_j becomes an edge of T_j. The level of f becomes j and we discard any element of $L_f(v)$ and $L_f(w)$ of level $< j$.

Let $f = (v, w)$ be an edge such that $f(j) \in T^2_j$. We have to update the edges of level j of $L_f(v)$ and $L_f(w)$ since $t(C_{j+1}(v))$ and $t(C_{j+1}(w))$ may have changed in C. Assume $C_{j+1}(v)$ is the parent of $C_{j+1}(w)$ in C. Then clearly $w = t(C_{j+1}(w))$ and therefore the edge of level j of $L_f(w)$ is f. Let $b = t(C_{j+1}(v))$. The edge of level j of $L_f(v)$ should be the edge of minimum weight on the path from

[4] Notice that $pathmin_{k+1}(x, d, null, e')$ actually finds the edge of minimum weight on P_{xq}. The result remains correct since $q \in P_{xy}$.

w to b in $C_{j+1}(v)$. We find this edge by performing $pathmin_{j+1}(w, b, null, null)$ on T^2. We perform these updates to all edges of T_j^2 while traversing it top-down.

Case 2: $q_1 > q_2$**.** We traverse the clusters of T_j^2 top down starting from $C_{j+1}(y)$ inserting them one by one into $C_j(x)$ by performing add-leaf operations to $C_j(x)$. We start by inserting $C_{j+1}(y)$ as a child of $C_{j+1}(x)$. The edge connecting these two clusters is $e(j)$. We compute its weight and update $L_e(x)$ and $L_e(y)$ exactly as in Case 1.

The partition of T_j^2 into clusters of levels $\leq j$ is discarded. Let $f = (v, w)$ be an edge such that $f(j) \in T_j^2$. The level of f becomes j. We discard any elements of $L_f(v)$ and $L_f(w)$ of levels smaller than j, and update the edge of level j in these lists exactly as in case 1.

Case 3: $q_1 < q_2$**.** This case is analogous to Case 2, with the roles of T_j^1 and T_j^2 switched.

Case 4: $q_1 = q_2$**.** Create T_j by adding $e(j)$. Recursively perform $\text{link}_{j-1}(C_j(x), C_j(y))$ combining T_{j-1}^1 and T_{j-1}^2. We update the edges of level j of $L_e(x)$ and $L_e(y)$ as in Case 1.

5.3 Analysis

We now show that the total cost of all link operations is $O(n + m)$. Consider a link operation in which we perform the non-recursive case (Case 1, 2, or 3) at level j, and for each level greater than j we perform Case 4. We call such a link a *level-j* link.

At each level $j \leq k \leq \ell$, we perform a query to discover the values of $L_e(x)$ and $L_e(y)$ associated with level k. Each such query takes $O((\ell - k)^2)$ time so the total cost of updating the lists $L_e(x)$ and $L_e(y)$ is $O((\ell - j)^3)$.

It is easy to verify that a level-j cluster contains at least $2^{\ell-j}$ vertices of T, (in fact it contains a lot more). Therefore after the level-j link of T^1 and T^2, the resulting tree T satisfies $|T| \geq \max\{|T^1|, |T^2|\} + 2^{\ell-j}$. It follows that there could be at most $n/2^{\ell-j}$ links at level j and the total time it takes to update the lists $L_e(x)$ and $L_e(y)$ over all links is $O((n/2^{\ell-j})(\ell - j)^3) = O(n)$.

Assume now that at level j we performed Case 1 or Case 2. In these cases, the link at level j also updates the elements of level j of $L_f(v)$ and $L_f(w)$ for each edge $f = (v, w)$ such that $f(j) \in T_j^2$, using a query. Each such query takes $O(\ell - j)^2$ time. Since the universe of each node in T_j^2 increases, the number of queries is proportional to the number of nodes whose universe increases. By Lemma 4.4, over the entire process only $O((m + n)/2^{\ell-j})$ nodes of level j increase their universe. So the total cost of moving level-j nodes from cluster to another cluster is $O(((m+n)/2^{\ell-j})(\ell - j)^2)$. If we sum over j we get that the total cost of moving nodes between clusters is $O(m+n)$. The analysis of Case 3, is identical to the analysis of case 2, with the roles of T_j^1 and T_j^2 switched.

6 Improving the query time to $O(\alpha(m, n))$

A *pathmin* query takes $O(\ell^2)$ time in the data structures of Section 5 because we recursively find the edge of minimum weight between x and $t(C_{j+1}(x))$ or between y and $t(C_{j+1}(y))$ for every level j. This takes $O(\ell - j)$ time since we may have two recursive calls to $pathmin_{j+1}$ query, one call with two edges, and one call with only one edge. To reduce the query time to $O(\ell)$ we change this query algorithm to use more information which we maintain with the edges. With this additional information we can answer a pathmin query with two edges in $O(1)$ time, and the total query time goes down to $O(\ell)$.

We first describe the data structure, and then, before going into the details of the operations, we show how we can answer a pathmin query with two edges in $O(1)$ time. Each cluster of T_k is

either a *proper* cluster or an *improper* cluster. When we construct the clusters of T_k, every improper cluster must be a leaf (a node of degree one). To obtain T_{k-1}, we prune the improper clusters after contracting the clusters of T_k. We also maintain the following. Let C_k be an improper cluster of T_k, and let $e = (a, b)$ be the edge such that $e(k)$ is incident to C_k, and $C_k = C_k(a)$. Then, $C_k(b)$ must be proper. Let C_k be an improper cluster as above. We redefine $t(C_{k+1}(a))$ to be a rather than $r(C_{k+1}(a))$ as in Section 5. We also define $t(C_k(a))$ to be a. We also define all nodes (which are clusters of level $\ell + 1$) to be *proper* clusters. Note that $|T_{k-1}|$ is equal to the number of proper clusters of T_k.

Let $e = (x, y) \in T = T_\ell$. We define the *level* of an edge e as follows. Let j the smallest level such that e appears in T_j. Then we define the *level* of e to be j and denote it by level$(e) = j$. A copy of the edge e appears in each T_i, for $j \leq i \leq \ell$. We denote by $e(i)$ the copy of e that appears in T_i. Let $e = (x, y) \in T$ such that level$(e) = j$. The edge $e(j)$ is either contained in $C_j(x) = C_j(y)$, and adjacent to $C_{j+1}(x)$ and $C_{j+1}(y)$, or $e(j)$ connects the improper cluster $C_j(x)$ to a proper cluster $C_j(y)$, and $x = t(C_{j+1}(x))$. In anycase, $e(j)$ is adjacent to two proper clusters of level $j+1$, $C_{j+1}(x)$ and $C_{j+1}(y)$. Assume that $e(j)$ is an edge of the first type, that is $e(j)$ is contained in the incremental tree $C_j(x) = C_j(y)$. Let $C_{j+1}(y)$ be the parent of $C_{j+1}(x)$ in $C_j(x)$. As in Section 5 we define the weight of $e(j)$ in the incremental tree $C_j(x)$ to be the minimum weight of a vertex on the path from x to $t(C_{j+1}(y))$. Notice that unlike in Section 5, here we need the weight to be defined only for $e(j)$, where $j =$ level(e).

Let $e = (a, b)$. For each level$(e) \leq k < \ell$ we maintain the following information associated with $e(k)$, which we denote by $Inf(e(k))$. Let $C_{k+2} \neq C_{k+2}(a)$ be a node in the incremental tree $C_{k+1}(a)$. Let $C_{k+2}(a) = C^1_{k+2}, \cdots C^r_{k+2} = C_{k+2}$ be the path in $C_{k+1}(a)$ from $C_{k+2}(a)$ to C_{k+2}. Let $(d, c) \in T$ be the edge in $C_{k+1}(a)$ between C^{r-1}_{k+2} to C_{k+2}, where $c \in C_{k+1}$. We save for C_{k+2} the edge in T of minimum weight on the path between a to d. We save similar information with respect to the nodes in $C_{k+1}(b)$.

To store this information while using linear space, we need to impose an upper bound on the maximum size of a cluster. On the other hand, we still have to ensure that each tree T_j does not contain too many clusters of level $j + 1$. Therefore we maintain the following invariants. Let $j \leq \ell$, and let T_j be in universe i.

1. Each cluster C_j contains up to $6A(j, i)$ nodes which correspond to clusters of level $j + 1$. In Section 5 it contained $\geq 2A(j, i)$ nodes. Here, we don't have such an explicit requirement, but we have Invariant (2) instead.

2. By definition, the size of T_{j-1} is the number of proper clusters of level j. We maintain the invariant that $|T_{j-1}| \leq |T_j|/2A(j, i)$.

For an edge $e = (a, b)$ of level j we have an array of size $\ell - j$ indexed by the integers $\ell - 1, \cdots, j$. Entry k of the array contains $Inf(e(k))$. Let T_{k+1} be in universe h. We store the information of $Inf(e(k))$ associated with nodes of $C_{k+1}(a)$, in an array of size $6A(k + 1, h)$. By Invariant (1), $|C_{k+1}(a)| \leq 6A(k + 1, h)$ so an array of this size is sufficient to store all the information regarding the nodes in $C_{k+1}(a)$. Entry i in that array corresponds to node numbered i in $C_{k+1}(a)$. We use a similar array to store the information regarding the nodes of level $k+2$ that are contained in $C_{k+1}(b)$.

Let $e = (x_1, \hat{x})$ be the edge in Figure 1(B). Assume that $C_{k+2}(a), C_{k+2}(g)$ and $C_{k+2}(p)$ are numbered 1, 2, and 3 respectively in $C_{k+1}(a)$. Then, $Inf(e(k))$ associated with $\hat{x}$ is as follows. $Inf(e(k))[1]$ contains the edge of minimum weight on the path from $\hat{x}$ to h. $Inf(e(k))[2]$ is empty, and $Inf(e(k))[3]$ contains the edge of minimum weight on the path from $\hat{x}$ to d.

We now sketch we can answer a pathmin query with two edges in $O(1)$ time. Consider $pathmin_k(x, y, e_x, e_y)$, and assume that both e_x and e_y, exist, and that $C_k(x) = C_k(y)$ and that level(e_x), level$(e_y) < k$.

Using bit operations and some extra information that we save for each edge, we find in $O(1)$ time the smallest level $j > k$, such that $C_j(x) \neq C_j(y)$. Notice that level(e_x) and level$(e_y) \leq j-2$. We now show how to find the edge of minimum weight on the path from x to y in $C_{j-1}(x)$. Let the path from $C_j(x)$ to $C_j(y)$ consists of the nodes $C_j(x) = C_j^1, C_j^2, \cdots C_j^r = C_j(y)$. Let the edge adjacent to C_j^{r-1} and $C_j(y)$ be (x_{r-1}, x_r) (note that x_r may be different from y). We use $Inf(e_x(j-2))$ to extract the edge of minimum weight on the path between x and x_{r-1}. Let (x_1, x_2) be the edge adjacent to $C_j(x)$ and C_j^2 (note that x_1 may be different from x). We use $Inf(e_y(j-2))$ to extract the edge of minimum weight on the path between y and x_2. It is easy to see that the edge of minimum weight on the path from x to y is the edge of minimum weight among these two edges and the edge (x_{r-1}, x_r).

We will later show how to maintain Invariants (1) and (2) while performing links and queries. Assume that these invariants hold. We prove in the next Lemma that the space used to store this information remains linear.

Lemma 6.1 *The total space used to save $Inf(e(k))$ for all edges $e(k)$ in trees T_k and over all k is $O(n)$.*

Proof: Let $e(k) = (a, b)$ be an edge in T_k such that level$(e) \leq k < \ell$, adjacent to $C_{k+1}(a)$ and $C_{k+1}(b)$. Assume that T_{k+1} is in universe i. We save for $e(k)$ two arrays each of size $6A(k+1, i)$. Let $2A(k+1, i) \leq x < 2A(k+1, i+1)$ be the number of nodes in T_{k+1}. By Invariant (2), there are at most $x/2A(k+1, i)$ edges in T_k. So we get that the total space used by edges at level k is $O(x)$. Since the size of the trees T_k decreases by at least a factor of 2, we get that the total amount of information saved is $O(n)$. □

Notice that by the construction, all the clusters of levels h, where $h \geq k+1$ that are contained in C_k are proper clusters of level h, and this holds even if C_k is an improper tree of level k.

The outline of this Section is as follows. In Section 6.1 and in Section 6.2 we describe the implementation of the query. In Section 6.3 we give the implementation of the link operation, and finally in Section 6.4 we give the analysis of the data structure.

6.1 Finding the smallest uncommon cluster

Given two edges $e = (x, a)$, and $e' = (y, d)$, $x \neq y$, of level $< k$ such that $C_k(x) = C_k(y)$. To make our query work in $O(\ell)$ time, we want to find the smallest level $j > k$, such that $C_j(x) \neq C_j(y)$ in $O(1)$ time.

To do this efficiently, we assume that the word size is $O(\log n)$, and that we can do bit operations such as AND, NOT, OR, and XOR, on words of size $O(\log n)$ in $O(1)$ time.

We also maintain the following additional information.

- For an edge $e = (a, b)$ of level j, we maintain an extendable array $A_r(e)$ of size $\ell - j + 1$, indexed by $j, \cdots, \ell$. Entry k of $A_r(e)$, $j \leq k \leq \ell$, corresponds to level k, and contains a pointer to the node $C_{k+1}(a)$ and a pointer to the node $C_{k+1}(b)$.
- We store a static table msb, such that $msb[i]$ contains the index of the most significant bit in binary representation of i for $1 \leq i \leq n$.
- For each cluster C_k we number the nodes of level $k+1$ contained in C_k in ascending order starting from 1. So each cluster $C_{k+1} \in C_k$ stores a number in the range $\{0, \cdots, |C_k| - 1\}$ which we call the *id* of C_{k+1}.

Let T be such that $T_\ell, \cdots, T_k$, $k \geq 1$ exist, and T_i is in universe j_i. We define start$_\ell(T) = 0$, end$_\ell(T) = \lceil \log(6A(\ell, j_\ell)) \rceil$, and range$_\ell(T) = [\text{start}_\ell(T), \text{end}_\ell(T)]$. For $k < h < \ell$, we define start$_h(T) = \text{end}_{h+1}(T) + 1$, end$_h(T) = \text{end}_{h+1}(T) + 1 + \lceil \log(6A(h, j_h)) \rceil$, and range$_h(T) = [\text{start}_h(T) \cdots, \text{end}_h(T)]$.

• We store for each edge $e = (a, b)$ such that level$(e) = k$ an integer $id(e, a)$ that for every $k \leq h \leq \ell$ contains the id of $C_{h+1}(a)$ within $C_h(a)$ in the bits whose indices are in range$_h(T)$. We store an integer $id(e, b)$ that is defined analogously with respect to b.
• We store for T an extendable array $L(T)$ of size $\sum_{h=\ell}^{k}(\lceil \log(6A(h, j_h)) \rceil)$ such that for every $x \in$ range$_h(T)$, $L(T)[x] = h$.

The next lemma shows that $id(e, a)$ can fit into constant number of words. We therefore assume below that it fits into a single computer word.

Lemma 6.2 *Let $e = (a, b)$ be an edge in T. Then the length of $id(e, a)$ is $O(\log n)$ bits.*

Proof: Let $\ell, \cdots, k$ be the such that $T_\ell, \cdots, T_k$ exist. Let T_i be in universe j_i. By Invariant (2) $1 \leq |T_k| \leq \frac{|T|}{2A(\ell, j_\ell) \cdot 2A(\ell-1, j_{\ell-1}) \cdots 2A(k+1, j_{k+1})}$. So we get that $2A(\ell, j_\ell) \cdot 2A(\ell-1, j_{\ell-1}) \cdots 2A(k+1, j_{k+1}) \leq |T|$. By taking logarithms of both size we see that

$$\log(2A(\ell, j_\ell)) + \log(2A(\ell - 1, j_{\ell-1})) + \cdots + \log(2A(k + 1, j_{k+1})) \leq \log |T| \leq \log n \ . \qquad (1)$$

The space used to store $id(e, a)$ is $\lceil \log(6A(\ell, j_\ell)) \rceil + \lceil \log(6A(\ell-1, j_{\ell-1})) \rceil + \cdots + \lceil \log 6A(k+1, j_{k+1}) \rceil$ which since $\ell << \log n$ is $O(\log n)$ by Equation (1). □

Given two edges of level $< k$, $e = (x, a)$, and $e' = (y, d)$, $x \neq y$, in a tree T such that $C_k(x) = C_k(y)$, we now show how to find the smallest level $j > k$, such that $C_j(x) \neq C_j(y)$ in $O(1)$ time. We copy $id(e, x)$ into a variable X and copy $id(e', y)$ into a variable Y. Using the appropriate bit operations we zero the bits whose indices are in start$_{k-1}(T) \cdots \log n$. So now both X and Y contain information on levels $k, \cdots, \ell$. Notice that if $j > k + 1$ then since $C_i(x) = C_i(y)$ for $k + 1 \leq i < j$, we have that $X[r] = Y[r]$, for each $r \in start_j(T) \cdots end_k(T)$. But since $C_j(x) \neq C_j(y)$, there exists an index r in range$_{j-1}(T)$ such that $X[r] \neq Y[r]$.

Let $W = X \ xor \ Y$. By the observations above, the most significant bit of W which is 1 is in range$_{j-1}(T)$. We now use the pre-computed table msb to get $z = msb[W]$. We have that start$_{j-1}(T) \leq z \leq$ end$_{j-1}(T)$ so $L(T)[z] = j - 1$. We discovered that the id of $C_j(x)$ is different from the id of $C_j(y)$ in $C_{j-1}(x) = C_{j-1}(y)$, so $C_j(x) \neq C_j(y)$. Now, using the arrays $A_r(e_x)$, and $A_r(e_y)$ we find the nodes $C_j(x)$, and $C_j(y)$.

6.2 The implementation of path-min query

If x is contained in an improper cluster $C_j(x)$, then x is not contained in any cluster of level $j-1$, so it is possible that there is no k such that $C_k(x) = C_k(y)$. This fact somewhat complicates our query algorithm. We first assume that there exists an index k such that $C_k(x) = C_k(y)$. We show that in this case we can find the edge of minimum weight on P_{xy} in $O(\ell - j)$ time.

We follow the same approach as in Section 5.1 and describe the changes required in the procedure $pathmin_k(x, y, e_x, e_y)$. Recall that this procedure assumes that $C_k(y) = C_k(x)$. The edge e_x if exists is an edge incident to x which is of level $< k$. Similarly, the edge e_y if exists is an edge incident to y which is of level $< k$. The $pathmin_k$ procedure returns the edge of minimum weight on P_{xy}. In the case where $k = \ell$ we perform $\min_{C_\ell(x)}(x, y)$ and return the answer of this query.

Consider the case where $k < \ell$ and both edges e_x and e_y exist. We use the algorithm of Section 6.1 to find the smallest level $j > k$ such that $C_j(x) \neq C_j(y)$ in $O(1)$ time. Notice that level$(e_x) \leq j - 2$ and level$(e_y) < j - 2$. The procedure in Section 6.1 returns the level j and pointers to $C_j(x)$ and to $C_j(y)$. We now show how to find the edge of minimum weight on the path from x to y in $C_{j-1}(x)$. Let the path between $C_j(x)$ to $C_j(y)$ consists of the nodes $C_j(x) = C_j^1, C_j^2, \cdots C_j^r = C_j(y)$. Let the edge adjacent to C_j^{r-1} and $C_j(y)$ be (x_{r-1}, x_r) (note that x_r may be different from y). We use

$Inf(e_x(j-2))$ to extract the edge of minimum weight on the path between x and x_{r-1}. Let (x_1, x_2) be the edge adjacent to $C_j(x)$ and C_j^2 (note that x_1 may be different from x). We use $Inf(e_y(j-2))$ to extract the edge of minimum weight on the path between y and x_2. It is easy to see that the edge of minimum weight on P_{xy} is the edge of minimum weight among these two edges and the edge (x_{r-1}, x_r)[5]. It is easy to see that in this case where both e_x and e_y exist $pathmin_k$ takes $O(1)$ time.

Assume now that $k < \ell$ and only e_x exists. The case where only e_y exists is symmetric. If $C_{k+1}(x) = C_{k+1}(y)$ then we call $pathmin_{k+1}(x, y, e_x, null)$. Otherwise, let $C_{k+1}(x) = C_{k+1}^1, C_{k+1}^2, \cdots, C_{k+1}^r = C_{k+1}(y)$, be the path from $C_{k+1}(x)$ to $C_{k+1}(y)$ in $C_k(x)$. Let (x_{r-1}, x_r) be the edge adjacent to C_{k+1}^{r-1} and $C_{k+1}(y)$. Recall that $Inf(e_x(k-1))$ contains the edge f of minimum weight on the path between x to x_{r-1}. We recursively call $pathmin_{k+1}(x_r, y, (x_{r-1}, x_r), null)$. We return the edge of minimum weight among f, (x_{r-1}, x_r) and the edge returned by the recursive call. It is easy to see that this case of $pathmin_k$ takes $O(\ell - k)$ time.

As in Section 5.1 we start the query by finding the largest level k such that $C_k(x) = C_k(y)$. Since k is maximal, we have that $C_{k+1}(x) \neq C_{k+1}(y)$. Initially, we do not have the edges e_x and e_y. We start by calling to $pathmin_k(x, y, null, null)$. The procedure $pathmin_k(x, y, null, null)$ works exactly as in Section 5.1. That is, it consists of a constant number of min queries on the incremental tree associated with the cluster $C_k(x)$, at most one recursive call to *pathmin* in which both edges exist and at most two recursive calls to the procedure *pathmin* in which only one edge exists. It is easy to see that the running time of the algorithm is $O(\ell - k)$.

We now give the general algorithm. The main idea is that each time we reach an improper cluster $C_j(x)$ such that $C_j(x) \neq C_j(y)$, we compute the edge of minimum weight on the path between x and $t(C_j(x))$. Let $w = t(C_j(x))$. The cluster $C_j(x)$ is improper so there is an edge between w and z such that $C_j(z)$ is proper. We then continue the query with z instead of x. If for some $k < j$, $C_k(z)$ is improper and we need to compute the edge of minimum weight on the path between z and the root of $C_k(z)$, we can do it in $O(j - k)$ time using the information stored for the edge (w, z). We now present the details.

We modify $pathmin_j$ to handle improper clusters. To distinguish this algorithm from the previous one we denote it $\overline{pathmin}_j$. It has the same parameters as $pathmin_j$. When we invoke $\overline{pathmin}_j(x, y, e_x, e_y)$ then $C_{j+1}(x)$ and $C_{j+1}(y)$ are proper clusters. The edge e_x if exists is incident to x of level j_x such that $C_{j_x}(x)$ is proper. Note that unlike the implementation of *pathmin* here $j_x > j$. A similar invariant holds for e_y. $\overline{pathmin}_j(x, y, e_x, e_y)$ applies one of the following cases. At the top level, to answer a query we invoke $\overline{pathmin}_\ell(x, y, null, null)$.

1. $C_j(x) \neq C_j(y)$ and both $C_j(x)$ and $C_j(y)$ are proper. We call $\overline{pathmin}_{j-1}(x, y, e_x, e_y)$ and return its result.

2. $C_j(x) = C_j(y)$. We perform $pathmin_j(x, y, null, null)$ (in $O(\ell - j + 1)$ time) and return the result.

3. $C_j(x) \neq C_j(y)$ and $C_j(x)$ is improper and e_x is null. Let $a = t(C_j(x))$ and let (a, z) be the edge adjacent to $C_j(x)$ and the proper cluster $C_j(z)$. We perform $pathmin_j(x, a, null, null)$ query as described above to find the edge f of minimum weight between a and x in $O(\ell - j)$ time. We recursively perform $\overline{pathmin}_j(z, y, (a, z), null)$ to find the edge f' of minimum weight on the path from z to y. We return the edge of minimum weight among f, f', and (a, z).

[5]We need to consider the weight of (x_{r-1}, x_r) for the case where the path between $C_j(x)$ to $C_j(y)$ consists only of the nodes $C_j(x)$ and $C_j(y)$

4. $C_j(x) \neq C_j(y)$ and $C_j(x)$ is improper and e_x is not null. Let $j' = \text{level}(e_x)$ and notice that since $C_{j'}(x)$ is proper then $j' \geq j+1$. Let $a = t(C_j(x))$ and let (a, z) be the edge that connects $C_j(x)$ to the proper cluster $C_j(z)$.

   We first find the edge f of minimum weight on the path between x and a in $O(j' - j)$ time as described below. Then we recursively find the edge f' of minimum weight between z and y by recursively invoking $\overline{pathmin}_j(z, y, (a, z), null)$. We return the minimum among f, f' and (a, z).

   We find f as follows. If $C_{j+1}(x) = C_{j+1}(a)$ we set $b = a$ and $e = (a, z)$. Otherwise, $C_{j+1}(x) \neq C_{j+1}(a)$, we perform $\min_{C_j(x)}(C_{j+1}(x), C_{j+1}(a))$, and get the edge g of minimum weight between a and $t(C_{k+1}(x))$ We set $b = t(C_{j+1}(x))$. Let (b, q) be the edge incident to $C_{j+1}(x)$ on P_{ba}. We set $e = (b, q)$. We maintain min to be the minimum edge on the part of the path between x and a that we have already considered. So now we set min to be g and continue to find the minimum on the path between x and b.

   We do it by calling $pathmin_{j+1}(x, b, null, e)$ with the following modifications. Recall that $pathmin_{j+1}(x, b, null, e)$ performs $pathmin_{j+2}(x, b, null, e)$ if $C_{j+2}(x) = C_{j+2}(b)$. If $C_{j+2}(x) \neq C_{j+2}(b)$, then $pathmin_{j+1}(x, b, null, e)$ finds an edge f of minimum weight between b and some vertex b' where $C_{j+2}(b') = C_{j+2}(x)$, such that there is an edge e' of level $j+1$ incident to b', and recursively performs $pathmin_{j+2}(x, b', null, e')$. The first modification is that before each recursive call we update min to be f if the weight of f is smaller than the weight of min.

   The second modification is that we stop the recursion after $O(j' - j)$ steps. At this point, we are left to find the edge of minimum weight between x and some vertex b'' in $C_{j'+1}(x) = C_{j'+1}(b'')$. If $j' = \ell$, then $x = b''$ and we are done. Otherwise, we have an edge e'' of level $\leq j'$ adjacent to b'', and the edge e_x of level j' adjacent to x. Also min at this point is the edge of minimum weight between b and b''. We perform $pathmin_{j'+1}(x, b'', e_x, e'')$, and get the edge $\hat{e}$ minimum weight between x and b'' in $O(1)$ time. (Recall that the procedure $pathmin$ costs $O(1)$ if both edges exist.) We set f to the edge minimum weight among min and $\hat{e}$.

5. $C_j(y) \neq C_j(x)$ and $C_j(y)$ is improper and e_y is null. This case is similar to Case 3.

6. $C_j(y) \neq C_j(x)$ and $C_j(y)$ is improper and e_y is not null. This case is similar to Case 4.

Clearly, for each j there are at most two steps that do not decrease j, in which the one of $C_j(x)$ and $C_j(y)$ is improper, after these steps we have that both $C_j(x)$ and $C_j(y)$ are proper and we either perform step (2) which terminates the process, or step (1) that decreases j by one. Thus $\overline{pathmin}_\ell$ performs $O(\ell)$ recursive calls.

Step 2 is performed only once and takes $O(\ell)$ time. Step 1 takes $O(1)$ time and is performed at most $O(\ell)$ times. Step 3 and Step 5 are performed at most once and take $O(\ell)$ time. The cost of step 4 for a certain level j is $O(j_x - j)$ where j_x is the level of the edge e_x. After Step 4 at level j, the new e_x is of level $< j$. Therefore the total time we spend executing Step 4 over all levels j is $O(\ell)$. Similarly, the total time we spend executing Step 6 is $O(\ell)$. We conclude that the overall running time of $\overline{pathmin}_\ell$ is $O(\ell)$.

6.3 The implementation of link

When we link T_j^1 to T_j^2 by adding the edge $e = (x, y)$ we may encounter a problem if either $C_j(x)$ or $C_j(y)$ is improper. In this case x or y do not exist in T_{j-1}^1 or T_{j-1}^2, respectively, so it is impossible simply to link T_{j-1}^1 and T_{j-1}^2 by adding (x, y).

To define the link operation and for its analysis we need the following definitions. Let C_k be a cluster of T_k, and assume that T_k is in universe $i > 0$. If $|C_k| < 2A(k,i)$, then C_k is a *small cluster*. If $|C_k| \geq 4A(k,i)$, then C_k is a *full cluster*. Otherwise, $2A(k,i) \leq |C_k| < 4A(k,i)$, and we say that C_k is a *medium cluster*. We will maintain the following additional invariants.

3. If C_k is an improper cluster, then C_k is a small cluster.

4. Let C_k be an improper cluster of level k. Let (w,z) be the single edge connecting it into a proper cluster C'_k of level k. Then, C'_k is a full cluster. To be precise, $|C'_k| = 6A(k,i)$.

A link between improper clusters may create a proper small cluster. To maintain Invariant 2, we will ensure that the number of proper small clusters of level k is always not larger than the number of full clusters of level k. We now describe the link operation.

As in Section 5 the link starts by performing $\text{link}_\ell(C_{\ell+1}(x), C_{\ell+1}(y))$. Notice that both $C_{\ell+1}(x)$, and $C_{\ell+1}(y)$ are proper clusters. In general before performing $\text{link}_k(C_{k+1}(x), C_{k+1}(y))$ both $C_{k+1}(x)$ and $C_{k+1}(y)$ would be proper clusters.

Let q_1 be the universe of T_k^1 and let q_2 be the universe of T_k^2. $\text{link}_k(C_{k+1}(x), C_{k+1}(y))$ performs the appropriate of the following four cases.

Case 1: $|T_k^1| + |T_k^2| \geq 2A(k, \max\{q_1,q_2\}+1)$**.** Create a new tree T_k in universe $\max\{q_1,q_2\}+1$ containing a single, initially empty, cluster C_k. Traverse T_k^1 top down and insert all nodes of T_k^1 to C_k by performing add-leaf operations. Then insert $C_{k+1}(y)$ as a child of $C_{k+1}(x)$ into C by another add-leaf operation which adds $e(k)$ to the new cluster. Finally insert all the nodes of T_k^2 into C, top-down by performing add-leaf operations. Since $|T_k^1| < 2A(k, q_1+1)$ and $|T_k^2| < 2A(k, q_2+1)$, we have that $|T_k| < 4A(k, \max\{q_1,q_2\}+1)$, so by inserting all nodes of T_k^1 and T_k^2 into a single cluster C we do not violate Invariant (1).

We update the information of the edge $e = (x,y)$. We compute the weight of $e(k)$ as in Case 1 of Section 5.2. If $k < \ell$, we also need to update $Inf(e(k))$. Let $C_{k+2} \neq C_{k+2}(x)$ be a node in the incremental tree $C_{k+1}(x)$. Let $C_{k+2}(x) = C_{k+2}^1, \cdots C_{k+2}^r = C_{k+2}$ be the path in $C_{k+1}(x)$ from $C_{k+2}(x)$ to C_{k+2}. Let $(d,c) \in T$ be the edge in $C_{k+1}(x)$ that is adjacent to C_{k+2}^{r-1} and C_{k+2} with $c \in C_{k+2}$. We store in $Inf(e(k))$ for $C_{k+2}(c)$ the edge in T of minimum weight on the path from x to d. We find the edge by performing pathmin query on T^1 in $O(\ell - k)$ time. We store similar information for the nodes contained in $C_{k+1}(y)$, which we compute by performing the appropriate queries on T^2. We add entry k to the array A_r of (x,y) that contains pointers to $C_{k+1}(x)$ and to $C_{k+1}(y)$. We also update the part of $id(e,a)$, and $id(e,a)$ that corresponds to level k.

The direction of the edges of T_k^2 may have changed (All edges are now directed towards $C_{k+1}(y)$), we update the weight of the edges in T_k^2 as in Case 1 of Section 5.2. For each edge $f = (a,b) \in T_k^2$, we update the part of $id(f,a)$, and $id(f,b)$ that corresponds to level k.

Each edge f such that $f(k)$ is either in T_k^1 or T_k^2 becomes an edge of T_k. The level of f becomes k. We discard any information of f that refers to levels $< k$. We set $L(T) = L(T^1)$. We remove from $L(T)$ all entries that refer to levels smaller than k. We update the entries of $L(T)$ that correspond to level k appropriately. We discard $L(T^2)$.

Case 2: $q_1 > q_2$**.** We use the following recursive *insert procedure*. Let $e' = (a,b)$. The procedure $insert_k(C_{k+1}(a), C_{k+1}(b), e')$ inserts the cluster $C_{k+1}(b)$ into $C_k(a)$ by adding the edge $e'(k)$. The procedure assumes that $Inf(e'(k))$ has already been updated.

Assume that T_k is in universe i. If $|T_k| = 2A(k,i+1) - 1$ then we increase the universe of T_k to $i+1$ and rebuild it as a single cluster containing $C_{k+1}(a)$ and $e'(k)$.

If $|C_k(a)| < 6A(k,i)$ we add $C_{k+1}(b)$ to $C_k(a)$ as a child of $C_{k+1}(a)$. We update the weight of $e'(k)$, $id(e',a)$, and $id(e',b)$ as in Case 1. If $C_k(a)$ is a proper cluster, then for each edge $f(k-1) = (u,v)$

such that $C_{k-1}(u) = C_{k-1}(a)$, we need to add to $Inf(f(k-1))$ the edge of minimum weight between u and a. We find it by performing $pathmin_k(u, a, null, null)$ in $O(\ell - k)$ time.

If $C_k(a)$ was an improper cluster and following the insertion $|C_k(a)| = 2A(k,i)$ then we make $C_k(a)$ proper as follows. Let $g = (w, z)$ be the edge adjacent to $C_j(a)$ and $C_j(z)$. We update $Inf(g(k-1))$ as described in Case 1, and perform $insert_{k-1}(C_k(z), C_k(a), g)$.

If $|C_k(a)| = 6A(k,i)$, we create a new cluster node $C_k(b)$. The cluster $C_k(b)$ is an improper cluster of level k. We add $C_{k+1}(b)$ into $C_k(b)$. In this case, b is the root of $C_k(b)$ and the edge (a, b) is the single edge connecting $C_k(b)$ into a proper cluster $C_k(a)$. It is easy to see that this *insert procedure* maintains Invariant (1), (3) and (4).

Now we show how to implement Case 2 using the insert procedure. We update $Inf(e(k))$ as in Case 1, and then insert $C_{k+1}(y)$ into T_k^1 by performing $insert_k(C_{k+1}(x), C_{k+1}(y), (x, y))$. We continue to insert all clusters of T_k^2 into $C_k(y)$ using the *insert* procedure. Notice that for all other edges $e' \neq (x, y)$ of T_k^2 that we insert to T_k^1 we don't need to recompute $Inf(e'(k))$.

The partition of T_k^2 into clusters of levels $\leq k$ is discarded. Each edge $f(j)$ of T_k^2 becomes an edge of T_k. We set $L(T) = L(T^1)$, and we discard $L(T^2)$.

Case 3: $q_1 < q_2$**.** This case is similar to Case 2.

Case 4: $q_1 = q_2$**.** We add the edge (x, y) into T_k as an edge that connects $C_{k+1}(x)$ with $C_{k+1}(y)$ and update $inf(e(k))$, $id(e, x)$ and $id(e, y)$ as described in Case 1. If both $C_k(x)$ and $C_k(y)$ are proper clusters of level k, we recursively call $\text{link}_{k-1}(C_k(x), C_k(y))$. If at least one of $C_k(x)$, $C_k(y)$ is an improper cluster of level k we perform one of the following cases. Let $i = q_1 = q_2$.

1. The cluster $C_k(x)$ is a proper cluster of T_k^1. The cluster $C_k(y)$ is an improper cluster of T_k^2, and $C_k(x)$ is not full (that is $C_k(x)$ is a small or medium cluster). Let $(w, z), \in T^2$ be the edge that connects $C_k(y)$ into the proper cluster $C_k(z)$. We insert all clusters of level $k+1$ of $C_k(y)$ into $C_k(x)$ using the *insert procedure* described in Case 2. Notice that the *insert procedure* will indeed insert all of these clusters into $C_k(x)$, since $|C_k(x)| < 4A(k,i)$, and by Invariant (3), $|C_k(y)| < 2A(k,i)$. Notice that w now belongs to T_k^1, and z belongs to T_k^2 and (w, z) is the only edge that connects between these two trees of level k, Also remember that $\text{level}((w, z)) = k$. We continue by recursively performing $\text{link}_{k-1}(C_k(w), C_k(z))$ to link T_{k-1}^1 and T_{k-1}^2. Since Case 1 of the link operation was not applicable, T_k^1 remains in universe i. The tree T_k^2 also remains in universe i, since by Invariant (4), $C_k(z)$ is a full cluster. Clearly this transformation does not affect $T_i, i > k$.

2. The cluster $C_k(x)$ is a proper cluster of T_k^1, the cluster $C_k(y)$ is an improper cluster of T_k^2, and $C_k(x)$ is a full cluster. Let the edge (w, z) be as defined in Case 1. We make $C_k(y)$ a proper cluster by adding the edge $f = (w, z)$ into T_{k-1}^2. That is, we update $Inf(f(k-1))$ as in Case 1, and call $insert_{k-1}(C_k(z), C_k(y), (w, z))$ to add $C_k(y)$ into $C_{k-1}(z)$ as a child of $C_k(z)$ in T_{k-1}^2. The cluster $C_k(y)$ becomes a small proper cluster. We recursively call $\text{link}_{k-1}(C_k(x), C_k(y))$.

3. The cluster $C_k(x)$ is an improper cluster of T_k^1, and the cluster $C_k(y)$ is an improper cluster of T_k^2. We make $C_k(x)$ proper by adding it into T_{k-1}^1 as described in the previous case. We add all the clusters contained in $C_k(y)$ into $C_k(x)$, using the *insert procedure* . Notice that the *insert procedure* will indeed insert all these clusters into $C_k(x)$, since prior to the operation, $|C_k(x)|, |C_k(y)| < 2A(k,i)$. Let $(w, z) \in C_k(y)$ be the edge that connected $C_k(y)$ into a proper cluster $C_k(z)$. Notice that now, $w \in T^1$ and $z \in T^2$, and (w, z) is the only edge of level k that connects T^1 to T^2. We continue linking T^1 and T^2 by performing $\text{link}_{k-1}(C_k(w), C_k(z))$. Using the same arguments as in Step (1), T_k^1 and T_k^2 both remain in universe i after these changes.

6.4 Correctness and analysis

We now show that the invariants hold after the link operation.

Lemma 6.3 *Let T be the result of performing link between T_1 and T_2. If both T^1 and T^2 satisfy the invariants the so does T.*

Proof: Invariants (1), (3) and (4), clearly hold by the definition of the link. Let $g(T_k)$ be the number of full clusters of T_k, and let $b(T_k)$ be the number of small proper clusters of T_k. To show that Invariant (2) holds we first prove that either $b(T_k) = g(T_k) = 0$ or $b(T_k) < g(T_k)$. The proof is by induction on the link operations. Let T_k^1 and T_k^2 be two trees that we link and assume that T_k^1 and T_k^2 satisfy the induction hypothesis.

In Cases 1, 2, and 3, we do not create any small proper cluster. So clearly if we perform one of these three cases then the induction statement holds for T_k. In Case 4 if both $C_k(x)$ and $C_k(y)$ are proper, or if we perform Subcase 1 then $b(T_k) = b(T_k^1) + b(T_k^2)$, and $g(T_k) = g(T_k^1) + g(T_k^2)$ so the induction statement holds for T_k.

Assume that we perform Subcase 2 of Case 4. Since $C_k(x)$ is a full cluster then by the induction hypothesis $g(T_k^1) > b(T_k^1)$. The cluster node $C_k(y)$ is an improper cluster of level k, thus it is adjacent to a proper cluster $C_k(z)$. By Invariant (4), $C_k(z)$ is a full cluster in T_k^2, and therefore by the induction hypothesis $g(T_k^2) > b(T_k^2)$. In this case we add $C_k(y)$ as a small cluster into T_k. Thus we get that $g(T_k) = g(T_k^1) + g(T_k^2)$, and $b(T_k) = b(T_k^1) + b(T_k^2) + 1$, and therefore the induction statement holds for T_k.

In Subcase 3 of Case 4 both $C_k(x)$ and $C_k(y)$ are improper. Therefore each of them is adjacent to a full cluster and by the induction hypothesis $g(T_k^2) > b(T_k^2)$ and $g(T_k^1) > b(T_k^1)$. Since we add at most one new small proper cluster to T_k the induction statement follows in this case as well.

By definition the number of nodes in T_{k-1} is equal to the number of proper clusters of T_k. Let p be the number of full clusters of T_k, let q be the number of small proper clusters of T_k, and let r be the number medium clusters in T_k. We proved that $p \geq q$. So it follows that $|T_k| \geq p4A(k,i) + q + r2A(k,i) \geq p2A(k,i) + q2A(k,i) + r2A(k,i)$, and therefore we get that $|T_{k-1}| = p + q + r \leq |T_k|/(2A(k,i))$, and Invariant (2) holds. □

We now show that the total cost of all link operations is $O(na(\ell, n)) = O(n + m)$. Suppose that we perform link between T^1 and T^2. Let k be the level such that Case 1, 2, or 3 is performed at level k. By the description of the link operation, for all levels j such that $j > k$, we performed the following operations. We added an edge $e(j)$ at level j. If at least one of the endpoints of $e(j)$ is in an improper cluster of level j we may have moved nodes from an improper cluster of level j to another cluster of level j in Case 4. Assume that T_j, for $j = 1, \cdots, \ell$ is in universe i_j.

When adding $e(j)$ we compute $Inf(e(j))$ which contains the edge of minimum weight on the path to $O(A(j+1, i_{j+1}))$ proper nodes of level $j+2$. It takes $O(A(j+1, i_{j+1})(\ell - j))$ time to compute these edges. Similarly, When we insert a leaf $C_{j+2}(b)$ into a proper cluster $C_{j+1}(a)$, by adding the edge (a, b) we update $Inf(e(j))$ for each edge $e(j) = (x, y)$, such that $C_{j+1}(x) = C_{j+1}(a)$, and $C_{j+1}(y) \neq C_{j+1}(a)$. We add to $Inf(e(j))$ the edge of minimum weight on the path from x to a in T. We find this edge by performing $pathmin(x, a, null, null)$ in $O(\ell - j)$ time.

Let $e = (a, b)$ be an edge in T_j adjacent to $C_{j+1}(a)$ and $C_{j+1}(b)$. By the observations above, if we charge e $O(\ell - j)$ time for each node that ever existed in $C_{j+1}(a)$ or in $C_{j+1}(b)$ then we cover the cost of computing $Inf(e(j))$. By Invariant (1), $|C_{j+1}(a)|, |C_{j+1}(b)| \leq 6A(j+1, i_{j+1})$ so we charge e by $O(A(j+1, i_{j+1})(\ell - j))$. We bound the total charges to edges of T_j, over all universes and over all link operations. Let $|T_{j+1}|$ be the number of nodes in T_{j+1}. By Invariant (2), the number of edges

in T_j is at most $|T_{j+1}|/2A(j+1, i_{j+1})$. Thus the total charge of the edges of T_j is

$$O(A(j+1, i_{j+1})(\ell - j)|T_{j+1}|/2A(j+1, i_{j+1})) = O(|T_{j+1}|(\ell - j)/2)$$

Since by Lemma 4.4, there are $O(\frac{n+m}{2^{\ell-j}})$ clusters of level $j+1$, the total cost of adding edges to level j is $O((\ell - j)\frac{m+n}{2^{\ell-j}})$. The cost for all levels is $O(m+n)$.[6]

Let $e = (a, b)$ be an edge in T_j adjacent to $C_{j+1}(a)$ and $C_{j+1}(b)$. Assume that $C_{j+1}(a)$ and $C_{j+1}(b)$ are both contained in the cluster $C_j(a)$, and that $C_{j+1}(a)$ is the parent of $C_{j+1}(b)$ in $C_j(a)$. When we insert $C_{j+1}(b)$ into $C_j(a)$ we compute the weight of the edge (a, b) in $C_j(a)$. This weight is the minimum weight of an on the path between b to $t(C_{j+1}(a))$, and it can be computed in $O(\ell - j)$ time by performing the pathmin query. If $C_j(a)$ is an improper cluster, then we may need recalculate the weight of $e(j)$, if we perform later on Subcase 1 or Subcase 3 of Case 4 of the link, and transfer all the clusters of $C_j(a)$ into a proper cluster of level j. Clearly, this adds a constant factor to the running time. By Lemma 4.4 there are $O((m+n)/2^{\ell-j})$ nodes at level j. So the total time required to compute the weights of all edges of level j is $O((m+n)(\ell - j)/2^{\ell-j})$, summing over all levels this is $O(m+n)$.

References

[1] Noga Alon and Baruch Schieber. Optimal preprocessing for answering on-line product queries.

[2] Stephen Alstrup and Jacob Holm. Improved algorithms for finding level ancestors in dynamic trees. In *ICALP '00: Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, pages 73–84, London, UK, 2000. Springer-Verlag.

[3] Bernard Chazelle and Burton Rosenberg. The complexity of computing partial sums off-line. *Int. J. Comput. Geometry Appl.*, 1(1):33–45, 1991.

[4] Richard Cole and Ramesh Hariharan. Dynamic lca queries on trees. *SIAM J. Comput.*, 34(4):894–923, 2005.

[5] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 48(3):533–551, 1994.

[6] Harold N. Gabow. A scaling algorithm for weighted matching on general graphs. In *FOCS*, pages 90–100, 1985.

[7] Harold N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 434–443, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[8] Loukas Georgiadis, Haim Kaplan, Nira Shafrir, Robert E. Tarjan, and Renato F. Werneck. Data structures for mergeable trees (unpublished).

[9] Loukas Georgiadis, Robert E. Tarjan, and Renato F. Werneck. Design of data structures for mergeable trees. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 394–403, New York, NY, USA, 2006. ACM Press.

[10] J. A. La Poutre;. New techniques for the union-find problem. In *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 54–63, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[11] Robert Endre Tarjan. Applications of path compression on balanced trees. *J. ACM*, 26(4):690–715, 1979.

[12] Andrew Chi-Chih Yao. Space-time tradeoff for answering range queries (extended abstract). In *STOC*, pages 128–136. ACM, 1982.

[6] Notice that if T_j changes universe and the universes of $T_{j+1}, \cdots T_\ell$ do not change there is no extra charge to the edges in T_j, since it does not affect $Inf(e(j))$.

Appendix

A Improving the running time the data structure of Section 4 from $O((n+m)\alpha(m,n))$ to $O(n+m\alpha(m,n))$

We now show how to improve the running time of the link operations from $O(n\alpha(m,n)+m)$ to $O(n+m)$. This will give us a total running time of $O(n+m\alpha(m,n))$ for all operations.

Let $x \in T^1, y \in T^2$. Suppose that $\text{link}_j(C_{j+1}(x), C_{j+1}(y))$ performs Case 3. Notice that in this case we may have to fix the weight of the edges in levels smaller than j. This adds $O(j)$ to the running time of a link. As shown in the analysis the total time it takes to perform all other operations is $O(n+m)$.

The following example illustrates that the updates in lower levels can increase the total running time of all link operations to $O(n\ell+m)$. Suppose our forests consists a tree T of size $\Theta(n)$ and of $\Theta(n)$ singleton trees $x_1, \cdots, x_k, k < |T|$. Let x_0 be the root of T. Suppose that T_ℓ is in universe $i > 1$, and that $2|T| \leq 2A(\ell, i+1)$. Assume that $T_\ell, \cdots, T_c$ exist where c is some constant, and that $C_j(x_0), j < \ell$ is an unary root cluster. Assume that we perform $\text{link}_\ell(C_{\ell+1}(x_1), C_{\ell+1}(x_0))$ between the singleton tree x_1 and x_0. Then, following the operation x_1 is the new root of T. In this case, the universe of T_ℓ doesn't change and case 3 of $\text{link}_\ell(C_{\ell+1}(x_1), C_{\ell+1}(x_0))$ is performed. Since $C_r(x_0), r = \ell-1, \cdots, k$ is an unary root cluster we have to update the weight of the single edge in $C_r(x_0)$ that enters $C_{r+1}(x_0)$, this adds $O(\ell)$ time to the cost of the link operation. We can repeat it and perform $\text{link}_\ell(C_{\ell+1}(x_{j'+1}), C_{\ell+1}(x_{j'})), 1 \leq j' < k$. Each such link updates the edges of lower levels at cost $O(\ell)$. since $k = \Theta(n)$, the total cost of all links is at least $O(n\ell)$.

We do the following modifications in the link and in the path-min operations. For a tree T with root u, we let $\kappa(T)$ be the smallest level q such that $C_q(u)$ is not an unary root cluster. If there is no such level we set $\kappa(T) = \ell+1$. It is easy to see that we can maintain this information while performing links, without increasing the asymptotic complexity of the link operation. The main modification is that in add root operation we postpone some of the updates. The result of this is that the weight of some of the edges may be incorrect.

We maintain the following invariants. Let u be the root of T.

1. For each level $k < \kappa(T)$, $C_k(u)$ is an unary root cluster. Let $e = (a,b)$, $C_{k+1}(a) = C_{k+1}(u), k < \kappa(T)$ be the single edge in $C_k(u)$, going into $C_{k+1}(u)$. Then, the weight of $e(j), j \geq k$ may be incorrect. The weight of all other edges is correct. Specifically, if $\text{level}(e) \geq \kappa(T)$, the weight of $e(k)$ for all $k \geq level(e)$ is correct.

2. Let $k < \ell - \sqrt{\ell}$. Let a be the parent of b in T. Let $e = (a,b)$ be an edge of level $j < k$, and assume that $C_{k+1}(a) \neq C_{k+1}(u)$, then the weight of $e(k'), k' \geq k$ is correct.

Invariant (1) implies the following two properties of our trees.

1. Let C_k be a cluster of level k such that $C_k \neq C_k(u)$. Let $e = (a,b)$ be an edge of level $j \geq k$, such that $C_k(a) = C_k(b) = C_k$, then the weight $e(k'), k' \geq j$ is correct. That is, if C_k is not the root cluster of T_k, then for all edges e that are contained in the subtree of T that C_k represents, the weight of $e(k'), k' \geq level(e) \geq k$ is correct.

2. Let $e = (a,b)$ be an edge of level k, such that $C_k(a) = C_k(b) = C_k(u)$, and $C_{k+1}(a), C_{k+1}(b) \neq C_{k+1}(u)$ then the weight $e(k'), k' \geq k$ is correct.

Invariant (1) implies that if $e = (a, b)$, $C_{k+1}(a) = C_{k+1}(u), k < \kappa(T)$ is the single edge in $C_k(u)$, going into $C_{k+1}(u)$, then the weight of $e(j), j \geq k$ may be incorrect. As described below in the implementation of the path-min query, this invariant is not enough to make the query work in $O(\ell)$ time. So we added invariant (2). Invariant (2) implies that if for some $k + 1 < j < \ell - \sqrt{\ell}$, $C_{j+1}(a) \neq C_{j+1}(u)$, then the weight of $e(j'), j' \geq j$ is correct.

The implementation of path-min. Assume that the invariants hold. We now describe the changes in the path-min query. Suppose we want to perform the query $\text{pathmin}_j(x, y, e_x, e_y)$, where $e_x = (x', x), e_y = (y', y)$. We find the largest level k such that $C_k(x) = C_k(y)$. Notice that when the recursive procedure *pathmin* performs a min query inside a cluster (steps 1(b), 1(d), 2(a)) it does not use the value of the last edge on the path form x to the root or on the path from y to the root. Thus, by properties (1) and (2) these queries use correct edge weights. However, when $pathmin_j(x, y, e_x, e_y)$ query uses an edge e_x of level $j' < j$ to find the edge of minimum weight on the path between a vertex x' and $r(C_{j+1}(x))$, the weight associated with $e_x(j)$ may be incorrect, since $e_x(j')$ may be the single edge entering the root cluster of $C_{j'}(x)$.

If $\text{nca}_{C_j}(C_{j+1}(x), C_{j+1}(y)) = C_{j+1}(x)$, then case 2 of the min query is performed and the pathmin procedure does not use the value associated with $e_j(x)$. So assume that $C_{j+1}(x) \neq \text{nca}_{C_j}(C_{j+1}(x), C_{j+1}(y))$, thus $C_{j+1}(x) \neq C_{j+1}(u)$. If $j < \ell - \sqrt{\ell}$, then by invariant (2) the weight of $e_x(j)$ is correct. Thus if $j < \ell - \sqrt{\ell}$ we use the weight of $e_x j)$ to find the edge of minimum weight on the path between x' and $r(C_{j+1}(x))$ in $O(1)$ time.

If $j \geq \ell - \sqrt{\ell}$, the weight of $e_x(j)$ may be incorrect. All the edges that are contained in the subtree represented by $C_{j+1}(x)$ are of level at least $j + 1$. Notice that since $C_{j+1}(x)$ is not the root cluster of $C_j(a)$, by property (1), the weight of all edges in $C_{j+1}(a)$ is correct. So we can use the min-root procedure to find the edge of minimum weight on the path between x and $r(C_{j+1}(x))$ in $O(\ell - j)$ time and return the edge f of minimum weight among $e_x a$ and f. Since $j \geq \ell - \sqrt{\ell}$ we will use this procedure at most $\sqrt{\ell}$ times in a total cost of $O(\ell)$ time.

Notice that the procedure min-root$(x, j + 1)$ is always performed on a cluster $C_{j+1}(x)$ such that $C_{j+1}(x)$ is not the root cluster of $C_j(x)$, thus by property (1), the weight of all of its edges is correct and min-root can be performed as before.

The implementation of link. We describe the changes required in the implementation of the link operation in Section 4.2. Consider a link operation that creates a tree T by adding an edge between a vertex y, which is the root of T^2 and a vertex $x \in T^1$. Let u be the root of T^1. For an edge $e \in T$, we define the list $L(e)$ and the weights of $e(k)$ for levels $k \geq j$ in the incremental trees containing them as the *information associated with* e.

Throughout the recursive links at decreasing levels we maintain the following invariant.

3. When we perform $\text{link}_j(C_{j+1}(x), C_{j+1}(y))$ then the information associated with each edge $e \in T$ of level $> j$ is correct.

Suppose that $\text{link}_j(C_{j+1}(x), C_{j+1}(y))$ performs Case 4. If $\kappa(T^1) \leq j$ then by Invariant (1), the information of edges $e \in T^1$ of level j is correct and we do not fix any edge of T^1. If $\kappa(T^1) > j$, then $C_j(u)$ is an unary root cluster. Let $e = (a, b)$ be the single edge in $C_j(u)$ that enters $C_{j+1}(a) = C_{j+1}(u)$. We fix the weight of $e(k)$ for all $k \geq j$ and the list $L(e)$, level by level, in order of decreasing levels. When we fix the weight of $e(k)$, $e \in T^1$ we assume that the weight of $e(i)$ for every $i > k$ is correct. We find the edge of minimum weight on the path between a and $r(C_{k+1}(a))$ in $O(1)$ time, in the same manner as we find it for the new edge (x, y) added by the link (see Section 4.2). This computation uses only the weight of $e(k + 1)$ and information of edges in $C_{k+1}(a)$ of level $\geq k + 1$.

We may need to fix an edge incident to an unary root of T_j^2 similarly. When we perform the recursive call $\text{link}_{j-1}(C_j(x), C_j(y))$ then Invariant 3 clearly holds.

Suppose that $\text{link}_j(C_{j+1}(x), C_{j+1}(y))$ performs Case 1. By Invariant 3 the information associated with all edges in T of level $> j$ is correct so we can fix the information associated with edges of level $\leq j$ in T^1 and T^2. (Note that these edges are of level exactly j after the link.) We fix these edges in order of decreasing levels as in Case 4. We set $\kappa(T)$ to be the smallest level j such that $C_j(u)$ is not an unary root cluster, or to $\ell + 1$ if there is no such level.

Suppose that $\text{link}_j(C_{j+1}(x), C_{j+1}(y))$ performs Case 2. In this case all edges of T^2 are of level $\geq j$ in T. By Invariant 3 the information associated with all edges in T of level $> j$ is correct so we can fix the information associated with every edge $e \in T_j^2$ such that $\text{level}(e) \leq j$ as in Case 4.

Since clusters of T_j^1 do not change their universe it may be too expensive to fix all edges of T_j^1. If $\kappa(T^1) \leq j$, then by Invariant (1), the information associated with edges $e \in T^1$ such that $\text{level}(e) = j$ is correct. So assume that $\kappa(T^1) > j$ and let $e = (a, b)$ be the single edge in $C_j(u)$ that enters $C_{j+1}(a) = C_{j+1}(u)$ in T_j^1. We fix the information associated with e as in Case 4. This update works correctly since by Invariant 3 the information associated with all edges in T^1 of level $> j$ is correct. We set $\kappa(T)$ to be $\kappa(T^1)$, unless $\kappa(T^1) > j$, $C_j(x) = C_j(u)$, and after the link $C_j(x)$ is not an unary root cluster. In the latter case we set $\kappa(T)$ to be j.

Suppose that $\text{link}_j(C_{j+1}(x), C_{j+1}(y))$ performs Case 3. Notice that all edges of T^1 are of level at least j in T. We fix the information of all edges $e \in T^1$ such that $\text{level}(e) \leq j$ as in the previous cases. As in Case 2, if $\kappa(T^2) > j$ we fix the information associated with the edge of level j entering the root of the unary root cluster $C_j(y)$. We now have that the information associated with all edges $e \in T$ of level $\geq j$ is correct.

We also make the following changes in the implementation of Case 3 as follows. If $\kappa(T^2) = k \geq j$, then we perform Case 3.1. We add $C_{j+1}(x)$ into $C_j(y)$ and update the weight of the new edge (x, y) associated with level j as described in Section 4.2. However, here, we do not update edges f such that $f(q)$ is the single edge entering the root of $C_q(y)$ for $q < j$.

If $\kappa(T^2) = k < j$, then we perform Case 3.2. As described in Section 4.2, for every $k < i \leq j$, we add a new bad cluster $C_i(x)$ into T_i^2 containing the singleton node $C_{i+1}(x)$. We also insert (via an add-root operation) $C_{k+1}(x)$ into $C_k(y)$ making $C_k(y)$ an unary root cluster. Here, for every $i < k$ we do not update edges f such that $f(q)$ is the single edge entering the root of $C_i(y)$.

In both cases after performing all add root and add-leaf operations as in Section 4.2, for each $i < j$, $C_i(x)$ is an unary root cluster. But unlike in the previous implementation, we do not update the weight of the single edge incident to the root of $C_i(x)$.

We set $\kappa(T) = j$ unless $\kappa(T^2)$ was larger than j and $C_j(y)$ is still an unary root cluster after the link. In the latter case we set $\kappa(T) = \kappa(T^2)$.

Last, to ensure that Invariant (2) holds, we fix the weight of the following edges as well. Let $i = \min\{j, \kappa(T^2)\}$. If $i \leq \ell - \sqrt{\ell}$, we fix all other edges $e \in T^2$ of level $< i$ incident to unary root clusters $C_i(y)$ of T^2.

We now analyze the running time of our modified implementation of link. Consider a link that performs Case 1, Case 2, or Case 3 at level j. This link performed Case 4 $\ell - j$ times, each time fixing a single edge of T^1 and a single edge of T^2. Each such fix takes $O(\ell - j)$ time (since we recover the weights associated with these edges in $O(\ell - j)$ levels, paying $O(1)$ time per level). So the total time spent fixing these edges while performing Case 4 is $O((\ell - j)^2)$. We charge this time to the link operation itself. Since we have at most $n/2^{\ell-j}$ links at level j the total charges to such links is $O((n/2^{\ell-j})(\ell - j)^2) = O(n)$.

When performing Case 1 we fix all edges of T_j^1 and T_j^2. Each such fix takes $O(\ell - j)$ time so the

total time to fix all these edges is $O((|T_j^1|+|T_j^2|)(\ell-j))$. We charge this time to the nodes of T_j^1 and T_j^2 which change their universe. By Lemma 4.4 the number of times a node of level j changes its universe is $O((m+n)/2^{\ell-j})$. Thus the total charges to these nodes is $O(((m+n)/2^{\ell-j})(\ell-j))$. If we sum this cost over all levels j, we get that the total charges for fixing edges in Case 1 is $O(n+m)$.

When performing Case 2 we fix the edges of T_j^2 each such fix takes $O(\ell-j)$ time. We charge this cost to the nodes of T_j^2 which increase their universe. As in the analysis of Case 1 the total charges are $O(m+n)$. We also fix at most one edge of T^1 of level j in $O(\ell-j)$ time. We charge this cost to the link operation itself and as in the analysis of Case 1 the total charges of all links made by Case 2 is $O(n)$.

When performing Case 3 we fix the edges of T_j^1 each such fix takes $O(\ell-j)$ time. We charge this cost to the nodes of T_j^1 which increase their universe. As in the analysis of Case 1 the total charges are $O(m+n)$. If we perform Case 3.2 then we spend a constant time at each level from j down to $k=\kappa(T^2)$ adding a singleton bad cluster and inserting a cluster into $C_k(y)$. This takes $O(j-k)$ time which we charge to the link operation which made $C_k(y)$ not an unary root cluster. (The level in which this link ended was k.) Since after the link $C_k(y)$ is an unary root cluster each link operation is charged once this way. The total charges to all links are $O(n)$.

Finally if $i=\min\{j,\kappa(T^2)\}$ is smaller than $\ell-\sqrt{\ell}$ we fix $O(\ell)$ additional edges. The time it takes to fix these edges is $O(\ell^2)$ which is $O((\ell-i)^4)$ since $i\leq\ell-\sqrt{\ell}$. If $i=\kappa(T^2)$, we charge this cost to the link operation at level i that made $C_i(y)$ not an unary root cluster. Otherwise, we charge this cost to the current link operation between x and y. Using similar arguments as in the analysis of the previous cases the total charge for all link operations is $O(n)$.

The next Lemma proves that T satisfies the invariants.

Lemma A.1 *Assume that T^1 and T^2 satisfy invariants (1),$\cdots$, (2), then the resulting tree T also satisfies invariants (1),$\cdots$, (2).*

Proof: Let j be the smallest level for which we perform $\text{link}_j(C_{j+1}(x),C_{j+1}(y))$. If case 1 of the link was performed by $\text{link}_j(C_{j+1}(x),C_{j+1}(y))$, then the weight of all edges is correct and the invariants hold.

If case 2 of the link was performed by $\text{link}_j(C_{j+1}(x),C_{j+1}(y))$, then the weight of all edges of level at least j is correct. We didn't perform add root and thus didn't add new edges whose weight is incorrect, so it is easy to see that in this case invariants (1) and (2) hold for T.

Assume that case 3 of the link was performed by $\text{link}_j(C_{j+1}(x),C_{j+1}(y))$. If $\min\{j,\kappa(T^2)\}\leq\ell-\sqrt{\ell}$ all edges of T have the correct weight and the invariants hold. Assume that $\min\{j,\kappa(T^2)\}>\ell-\sqrt{\ell}$. Clearly at the end of the operation $\kappa(T)\geq j$, and for all $k<j$, $C_k(u)$ is an unary root cluster. We have that the weight of all edges $e(k)\in T_k$ with $\text{level}(e)\geq j$ is correct. Specifically, the weight of all edges $e(k)\in T_k$ with $\text{level}(e)\geq\kappa(T)$ is correct. To show that invariant (1) holds for T, we have to show that for each $e=(a,b)\in T$, with $\text{level}(e)=k'<j\leq\kappa(T)$, if $e(k')$ is not the single edge in $C_{k'}(u)$ that enters $C_{k'+1}(u)$, then the weight of $e(k),k\geq k'$ is correct. Assume that $e=(a,b)$ is such an edge, then clearly, $\text{level}(e)=k'<j$ implies that e was an edge of T^2. If $C_{k'+1}(a),C_{k'+1}(b)\neq C_{k'+1}(y)$, then by invariant (1) the weight of $e(k),k\geq k'$ was correct in T^2. It is easy to see that the weight of $e(k)$ hasn't changed in T, and therefore the weight of $e(k)$ is correct in T as well. Assume w.l.o.g that $C_{k'+1}(a)=C_{k'+1}(y)$, then if $k'\geq\kappa(T^2)$, by invariant (1) the weight of $e(k),k\geq k$ is correct in T^2. It is easy to see that $C_{k'+1}(a)=C_{k'+1}(y)\neq C_{k'+1}(u)$, and thus the weight of $e(k)$ hasn't changed in T and therefore it is correct in T as well. Last, assume that $C_{k'+1}(a)=C_{k'+1}(y)$, and that $k'<\kappa(T^2)$, in this case, we have that $C_{k'+1}(a)=C_{k'+1}(y)=C_{k'+1}(u)$,

and $e(k')$ is the single edge in $C_{k'}(u)$ that enters $C_{k'+1}(u)$, in contradiction to our assumption. Thus invariant (1) holds after the link operation.

We now show that invariant (2) holds. Let $i < \ell - \sqrt{\ell}$. Let $e = (a, b) \in T$ be an edge of level $k' \leq i$, such that a is the parent of b in T, and assume that $C_{i+1}(a) \neq C_{i+1}(u)$, we need to show that the weight of $e(i'), i' \geq i$ in T is correct.

The inequalities $k' \leq i < \ell - \sqrt{\ell} < \min\{j, \kappa(T^2)\}$, imply that e was an edge in T^2. The above inequalities also imply that $i + 1 < \min\{j, \kappa(T^2)\}$, and $C_{i+1}(a) \neq C_{i+1}(u) = C_{i+1}(y)$. Thus, by invariant (2), the weight of $e(i'), i' \geq i$ was correct in T^2, and since it hasn't changed in T, the weight of $e(i')$ is correct in T as well. □