Near-optimal fully-dynamic graph connectivity

Mikkel Thorup
AT&T Labs—Research, Shannon Laboratory
180 Park Avenue, Florham Park, NJ 07932
mthorup@research.att.edu

ABSTRACT

In this paper we present near-optimal bounds for fully-dynamic graph connectivity which is the most basic non-trivial fully-dynamic graph problem. Connectivity queries are supported in $O(\log n/\log\log\log n)$ time while the updates are supported in $O(\log n(\log\log n)^3)$ expected amortized time. The previous best update time was $O((\log n)^2)$. Our new bound is only doubly-logarithmic factors from a general cell probe lower bound of $\Omega(\log n/\log\log n)$. Our algorithm runs on a pointer machine, and uses only standard AC^0 instructions.

In our developments we make some comparatively trivial observations improving some deterministic bounds. The space bound of the previous $O((\log n)^2)$ connectivity algorithm is improved from $O(m+n\log n)$ to $O(m)$. The previous time complexity of fully-dynamic 2-edge and biconnectivity is improved from $O((\log n)^4)$ to $O((\log n)^3\log\log n)$.

1. INTRODUCTION

In a fully-dynamic graph problem, we are considering a graph G over a fixed vertex set V, $|V|=n$. The graph G may be *updated* by insertions and deletions of edges. Unless otherwise stated, we assume that we start with an empty edge set. Further, the updates may be interspersed with *queries* concerning properties of G. By an *operation* we mean an update or a query. All operations are presented *online*, meaning that we have to respond without any knowledge of future operations.

In this paper, we study the most basic non-trivial fully-dynamic graph problem; namely that of fully-dynamic graph connectivity where the query Connected(v,w) should tell if the vertices v and w are connected. We present a near-optimal randomized and amortized algorithm for fully-dynamic graph connectivity, showing that the operation complexity of this problem is $\tilde{\Theta}(\log n)$ where $\tilde{\Theta}(f(n)) = f(n)(\log f(n))^{\pm\Theta(1)}$.

The connectivity problem reduces to the problem of maintaining a spanning forest (a spanning tree for each component) in that if we maintain *any* spanning forest F for G at cost $O(t(n)\log n)$ per update, then, using dynamic trees [8], we can answer connectivity queries in time $O(\log n/\log t(n))$. In this paper, we show how to maintain a spanning forest in $O(\log n(\log\log n)^3)$ time per update. Connectivity queries are then answered in time $O(\log n/\log\log\log n)$.

Our upper bounds are only doubly-logarithmic factors from a general cell probe lower bound of $\Omega(\log n/\log\log n)$ [3; 7]. This lower bound holds both for amortization and randomization, and it holds even if the graph is restricted to be a forest.

Previous work. In 1985 [2], Fredrickson introduced a data structure known as *topology trees* for the fully-dynamic minimum spanning tree problem with a worst case cost of $O(\sqrt{m})$ per update, permitting connectivity queries in time $O(\log n/\log(\sqrt{m}/\log n)) = O(1)$. In 1992, Epstein et. al. [1] improved the update time to $O(\sqrt{n})$ using the *sparsification technique*.

In 1995 [4], Henzinger and King used randomization to maintain some spanning forest in $O(\log^3 n)$ expected amortized time per update. Then connectivity queries are supported in $O(\log n/\log\log n)$ time. The update bound was further improved to $O(\log^2 n)$ in 1996 [5] by Henzinger and Thorup.

In 1998 Holm et al. [6] got rid of the randomization, providing a simple deterministic algorithm maintaining some spanning forest in $O(\log^2 n)$ amortized time per update and answering connectivity queries in $O(\log n/\log\log n)$ time.

For the incremental (no deletions) and decremental (no insertions) problems, the bounds are as follows. Incremental connectivity is the union-find problem, for which Tarjan has provided an $O(\alpha(m,n))$ bound amortized over m updates [9]. For decremental connectivity, Thorup has provided an $O(\log n)$ bound if we start with $\Omega(n\log^6 n)$ edges, and an $O(1)$ bound if we start with $\Omega(n^2)$ edges [10].

In this paper, we improve the amortized randomized update time for maintaining some spanning forest from $O((\log n)^2)$ to $O(\log n(\log\log n)^3)$ time. Connectivity queries are then answered in $O(\log n/\log\log\log n)$ time. Our upper bounds are only doubly-logarithmic factors logarithmic factors from an $\Omega(\log n/\log\log n)$ lower bound on fully-dynamic connectivity [3; 7].

Techniques. All the previous poly-logarithmic algorithms for fully-dynamic connectivity translate each update into $\Theta(\log n)$ standard operations on trees [4; 5; 6]. Each tree op-

STOC 2000 Portland Oregon USA

eration is supported in $\Theta(\log n)$, and hence these approaches have converged towards an $\Theta((\log n)^2)$ bound.
The tree operations involve connectivity queries, for which we have an $\Omega(\log n/\log\log n)$ lower bound in the cell probe model [3; 7]. Nevertheless, we will support almost all the tree operations in $O((\log\log n)^3)$ time!
We will take starting point in the connectivity algorithm of Holm et al. [6] and show how many tree operations can be organized and batched using a sampling lemma from [5]. This circumvents the $\Omega(\log n/\log\log n)$ lower bound that only holds for single on-line queries. We then make a careful analysis of the the dynamics of the abstract algorithm from [6], and construct some novel tailor made data structures to provide exponentially faster support for the tree operations. Some of our initial developments improve the space performance of the connectivity algorithm from [6] from $O(m + n\log n)$ to linear, yet preserving the deterministic $O((\log n)^2)$ amortized time bound. Also, we get some simple improvements of the deterministic time bounds for 2-edge and biconnectivity from $O((\log n)^4)$ to $O((\log n)^3\log\log n)$.
In our implementation we make the standard assumption that each word of the computer contains at least $\log n$ bits. Our implementation will be done on a pointer machine, meaning that we will not do any address arithmetic.

2. CONNECTIVITY A LA HOLM ET AL.

In this preliminary section, we present our starting point; namely a slightly modified version of the fully-dynamic connectivity algorithm from [6] for maintaining a spanning forest F of a graph G. The edges in F will be referred to as *tree edges*. Internally, the algorithm associates with each edge e a level $\ell(e) \leq \ell_{\max} = \lfloor \log_2 n \rfloor$. For each i, G_i denotes the subgraph of G induced by edges of level at least i, and F_i denotes $F \cap G_i$. The following invariants are maintained.

(i) F is a maximum (w.r.t. ℓ) spanning forest of G, that is, for each i, F_i is a spanning forest of G_i.

(ii) The maximal number of vertices in component of G_i, or F_i, is $\lfloor n/2^i \rfloor$. Thus, the maximal relevant level is $\ell_{\max}$.

Initially, all edges have level 0, and hence both invariants are satisfied. We are going to present an amortization argument based on increasing the levels of edges. The level of an edge is only going to be decreased when it is being deleted, so until then, we can have at most $\ell_{\max}$ increases per edge. Intuitively speaking, when the level of a non-tree edge is increased, it is because we have discovered that its end points are close enough in F to fit in a smaller tree on a higher level. Concerning tree edges, note that increasing their level cannot violate (i), but it may violate (ii).
We are now ready for a high-level description of insert and delete.

Insert(e) The new edge is given level 0. If the end-points were not connected in $F = F_0$, e is added to F_0. Clearly, neither (i) nor (ii) is violated.

Delete(e) If e is not a tree edge, it is simply deleted. If e is a tree edge, we call Replace(e) below.

Replace((v, w)) Set $i = \ell(v, w)$. By (i) a *replacement edge*, reconnecting F has to be on level at most i, and further, to preserve (i), we should seek a replacement edge on the hightest possible level.

Let T_v and T_w be the trees in $F_i \setminus \{(v, w)\}$ containing v and w, respectively. We are looking for a level i non-tree edge f incident to both T_v and T_w.

Assume, without loss of generality, that $|T_v| \leq |T_w|$. By (ii), the tree in F_i containing (v, w) has at at most $\lfloor n/2^i \rfloor$ vertices, so T_v has at most $\lfloor n/2^{i+1} \rfloor$ vertices. Hence, preserving our invariants, we can take all level i tree edges in T_v and increase their level to $i + 1$, so as to make T_v a tree in F_{i+1}.

Now level i non-tree edges incident to T_v are visited one by one until either a replacement edge is found, or all edges have been considered. Let f be an edge visited during the search.

If f has both end-points in T_v, we may increase its level to $i + 1$ without violating (i). This increase pays for considering f.

Otherwise f is the desired replacement edge connecting T_v and T_w. Since e and f are on the same level, we can replace e by f in F without violating (i) or (ii). Afterwards, e is non-tree edge that is removed.

Suppose we do not find a level i replacement edge. If $i = 0$, we conclude there is no replacement edge, so we finish by removing e. If $i > 0$, we decrease the level of (v, w) without violating (i), and then call Replace((v, w)) recursively.

Efficient implementations of the above high-level algorithm hangs on amortizing costs over edge level changes, of which we have at most $2\ell_{\max} = O(\log n)$ per edge inserted.

2.1 Implementation

In [6], for each level i, Holm et al. maintained the forest F_i together with all level i-tree non-tree edges incident to it, as well as the sizes of each tree in F_i. Using the ET-data structure from [4], the following operations were supported in $O(\log n)$ time: checking if the vertices v and w are connected in F_i, inserting or deleting a tree edge from F_i, adding or removing a level i non-tree edge at each of its end points in F_i, finding the size of the tree in F_i containing a vertex v, and, finally, checking if there is a level i tree/non-tree edge connected to a given node in F_i, and returning one such edge if any.
With the above representation, it is straightforward to implement the high-level algorithm in $O(\log n)$ time per edge level change plus $O((\log n)^2)$ time per edge insertion or deletion, hence in $O((\log n)^2)$ amortized time per update. This implementation uses $O(m)$ space for the non-tree edges and $O(n)$ space for each forest F_i, hence in $O(m + n\log n)$ total space.
As a side-effect of our developments, we are actually going to make the space bound linear. To see that this is non-trivial with the above approach, note that when an edge (v, w) to be deleted, for each F_i we need to know the sizes of the two subtrees T_v and T_w resulting from the deletion. It is the maintainance of these sizes that is the bottle-neck. If we ignore the sizes, it is trivial to collapse all the F_i in a single dynamic tree [8].

3. TOWARDS NEAR-OPTIMAL BOUNDS

In this paper, we are going to make a near-optimal implementation of the high-level algorithm from the previous section, spending $O(\log n(\log\log n)^3)$ expected amortized time

per edge insertion or deletion. First we consider the costs associated with testing if level i non-tree edges incident to T_v reconnects to T_w. During a deletion, we are looking for at most one reconnecting replacement edge, and this edge we are happy to pay $O(\log n(\log\log n)^3)$ for. In general, however, each non-tree edge may be tested and found not to reconnect $\log_2 n$ times, once for each level, so we can only afford to spend $O((\log\log n)^3)$ per failed test. This seemingly contradicts the cell probe lower bound of $\Omega(\log n/\log\log n)$ for maintaining connectivity in a dynamic forest [3; 7]. The lower bound, is, however, for arbitrary vertex pairs, and what we will do is to show

Test-all We can find all level i non-tree edges incident to T_v in $O((\log\log n)^3)$ time per edge. The edges are found via their end-points in T_v, which we mark. Then we can test if (x, y) is a replacement edge simply by checking that it has only one marked end-point.

The most immediate problem now is that we want to amortize over level increases, and we can only move non-tree edges up that do not reconnect to T_w. Thus, we are fine if, say, 1/2 of the tested edges do not reconnect, but if most edges reconnect, we are doing a lot of testing that is not paid for. To check if there are too many replacement edges, we will support sampling:

Sample-and-test In $O(\log n(\log\log n)^3)$ time, we can almost uniformly sample a level i non-tree edge incident to T_v and check if it reconnects to T_w.

We are now ready to describe the new testing procedure:

- If T_v has $O(\log n)$ incident level i non-tree edges, we use test-all. If a replacement edge is found, it pays the cost of $O(\log n(\log\log n)^3)$; otherwise no edges reconnect, and they all get their levels increased, at a testing cost of $O((\log\log n)^3)$ per edge.

- If T_v has $\Omega(\log n)$ incident level i non-tree edges, we use Sample-and-test an expected constant number of times, either finding a replacement edge paying for the samples, or concluding with probability $O(1/n)$ that at least half the edges do not reconnect. In the latter case, we can apply Test-all at an expected cost of $O((\log\log n)^3)$ per non-connecting edge.

The above type of sampling procedure may look impossible, but as shown in [5, Sampling Lemma] it is possible. The subtle point is that we are making an *expected* constant number of samples. Often we will make a large number of samples but then, with correspondingly high probability, we will end up claiming correctly that at least half the edges are non-connecting.

Before showing how to make the level i non-tree edges incident to T_v readily available, we need to go through several developments.

4. A NEW STRUCTURAL VIEW

For our implementations, we need a different view of the Holm et al.'s algorithm from §2. As a side-effect we will get a quite different implementation using only linear space. The algorithm will still be deterministic and achieve the same time bounds. A main point in our implementation is an observation that we do not need to support general deletions of tree edges from the F_i. Rather than using general data structures, we will develop simpler tailor made data structures for implementing the connectivity algorithm. In this section, we will not yet worry about randomized sampling.

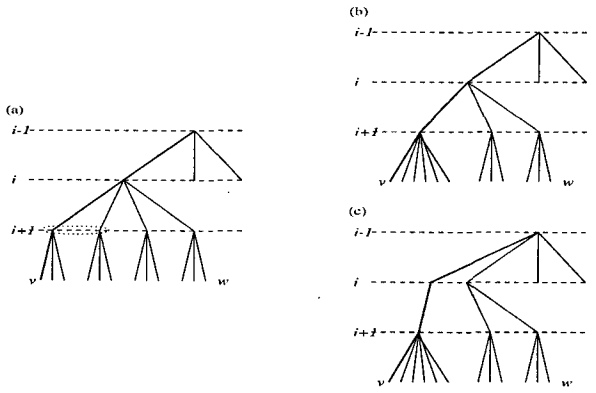

Figure 1: The effect on the structural forest of a replace on level i: Starting from Figure (a), two level $i+1$ components are to be merged into the new level $i+1$ component T_v containing v. If we find a replacement edge, we go to (b); otherwise we go to (c) and recurse on level $i-1$.

4.1 The structural forest

We are going to maintain a rooted forest C, called the *structural forest*, over the nodes of G. The leaves of C are the vertices G. All leaves have depth $\ell_{\max}$. The *level* of a node is its depth, and the nodes at level i represent the components in G_i. The level i ancestor of the vertex v is denoted v^i. For a linear space implementation, nodes of C with only one child should be suppressed. For each node $a \in C$, we maintain the *size* $n(a)$ which is the number of leaves descending from a. These leaves are the vertices in the component represented by a.

In the following, for clarity, vertices and edges always belong to G while nodes and arcs always belong to C (or C^L defined in the next subsection).

Note that C is independent of the choice of our ℓ-maximal spanning forest F. Since F_i is not represented, this seemingly gives the problem that we cannot delete an arbitrary edge (v, w) from F_i and isolate T_v and T_w as in the connectivity algorithm of Holm et al. However, this turns out not to be a problem if we do things in a different order.

We will now discuss how the structural forest changes as edges are inserted in and deleted from G. While physically changing C, v^i always denotes the ancestor of v which is $\ell_{\max} - i$ steps up from v.

We need to define merging of nodes in C. Two nodes a and b can be *merged* if either a and b are siblings, or a and b are both roots. By merging a and b in a, we mean that we move all the children of b to a and delete the node b.

Inserting and deleting non-tree edge does not affect C. Inserting a tree edge (v, w) amounts to merging v^0 and w^0 in v^0, setting $n(v^0) := n(v^0) + n(w^0)$.

When deleting a tree edge (v, w) on level i, we search a replacement edge on level i. The effect on C is illustrated in Figure 1. The first thing that happens is that we increase the levels of the level i tree edges in T_v from i to $i+1$. Let (x, y) be such a tree edge. In C this corresponds to merging x^{i+1}

and y^{i+1}, setting $n(x^{i+1}) := n(x^{i+1}) + n(y^{i+1})$. When done, T_v is represented by v^{i+1}. Afterwards, we may increase the levels of some non-tree edges, but this does not affect C.

Now, if a replacement edge f is found on level i, then replacing e by f in F, does not affect C, and afterwards we can just delete e as a non-tree edge. Otherwise, if $i < 0$, we logically increase the level of (v, w) to $i+1$ by removing v^{i+1} from the children list of $v^i = w^i$, create a new node v^i with v^{i+1} as single child, and make v^i a new child of w^{i-1}. Then we set $n(v^i) := n(v^{i+1})$ and $n(w^i) := n(w^i) - n(v^i)$. Note that $n(w^{i-1})$ is not affected. If $i = 0$, we do the same except that we do not make v^0 a child of a non-existing w^{-1}.

In our later implementations it is convenient if no child can change its size, so when we want to change the size of a child, first we remove it from its parent, then we change the size, and afterwards we reinsert it with its new size. Clearly the number of structural changes remain bounded in the number of edge level changes. Summing up,

LEMMA 1. *Each edge level change gives rise to $O(1)$ changes in C where each change is a creation of a new node, adding or removing a child from a node, or merging two nodes. While the size of a node changes, it is not a child of any node.*

4.2 Local search trees

In order to guide searches for level i tree and non-tree edges we will expand C to a binary tree C^L of height $O(\log n)$.

For each node a in C, we maintain a binary tree $L(a)$ over the children of a in C. We will refer to $L(a)$ as *local tree* at a. The tree $L(a)$ will be weight balanced in the sense that for each child b of a, the depth in $L(a)$ of b is $O(1 + \log(n(a)/n(b)))$. By C^L we refer to the binary tree resulting from C when for each a, we put $L(a)$ between a and its children in C. The weight balancing implies that the depth of any leaf in C^L is $O(\log n)$. As an immediate result, we can get from v to any v^i in $O(\log n)$ time.

We can now in $O(\log n)$ time test if an inserted edge (v, w) should be a tree edge by testing if $v^0 \neq w^0$. Similarly, when recovering on level i after having collected T_v under v^{i+1}, we can test if a non-tree edge (x, y) is a replacement edge by testing if $x^{i+1} \neq y^{i+1}$.

For each node a in C^L, we will maintain two bitmaps tree(a) and non-tree(a) that for each i tell whether there is a descending leaf with an incident tree or non-tree edge.

Finally, for each vertex v of G, we have the incident edges grouped depending on their level and on whether they are tree or non-tree edges. Since there are at most $2 \log n$ groups, a standard search tree brings us down to a particular group in $O(\log \log n)$ time.

Now, from any node a in C, we can find a level i tree/non-tree edge incident to a leaf descending from a, if any, in time $O(\log n)$. Also, we can change the level or tree/non-tree status of any edge in time $O(\log n)$.

4.3 Identifying the smaller tree

When looking for a replacement of (v, w) we need to find the smaller of the two level i trees resulting from cutting (v, w). We show how to find the size of T_v in time $O(\log n)$ times the number of level i tree edges in T_v. Applying this procedure in parallel to T_w, stopping as soon as we have found one of the sizes, we spend time $O(\log n)$ times the number of level i tree edges in the smaller tree, and hence $O(\log n)$ time per level increase. The other size is found in constant time by subtracting the size found from $n(w^i)$. The following procedure is used:

Size$((x, y), i)$ where (x, y) is a level i edge, finds the number of nodes connected to y in $F_i \setminus (x, y)$.

First we move up to y^{i+1} in C^L in $O(\log n)$ time, and set $s := n(y^{i+1})$. Then we search down from y^{i+1} using the bitmaps to find all level i tree edges incident to a descendant x' of y^{i+1} in $O(\log n)$ time per edge. For each such edge $(x', y') \neq (x, y)$, we add Size($(x', y'), C$) to s. Finally, we return s.

A call Size($(w, v), i$) finds the size of T_v in $O(\log n)$ per level i-tree edge in T_v, as desired. It traverses all level i-tree edges in T_v. By recording them, we are ready to increase their level if T_v turns out to be smaller than T_w. Increasing the level of such an edge (x, y) is trivially done in $O(\log n)$ time, and thereby we also identify x^{i+1} and y^{i+1} to be merged. Hence, all that remains is to implement the structural changes.

4.4 Tailor made weight balanced structure

We are now going to tell how we maintain the local trees as we remove and insert children and merge nodes. Each child b of a node a is given a rank rank(b) = log($n(b)$). A local tree $B(a)$ is built bottom-up in the following greedy fashion. We start with each child of a being its own local root. Then, while there are two local roots with the same rank r, we *pair* them, creating a new new root above them with rank $r + 1$. At the end, we have at most $\log n$ local roots over what we call *local rank trees*. Finally we make a path P down from a where the local rank roots branch off in order of decreasing ranks. Here by branching off we mean that each node $x \in P$, but the last, has two children; namely its successor in P and a rank root. The last node in P has two rank roots as children. This completes the description of the syntax of a local tree. A straightforward analysis shows that the depth in $L(a)$ of a structural child b of a is at most $\log(n(a)/n(b))+1$. Recall from Lemma 1 that a structural child never changes size, so it is only structural changes that can violate the organization of the rank trees.

Merging of a and b is done as follows. First we strip off the connecting paths, leaving us with the at most $2 \log n$ rank roots in two sorted lists. These two lists are merged in constant time per rank root. Now we go through the merged list backwards, pairing neighboring rank roots with common ranks, inserting the new root at most two ahead in the sorted list. Finally, we make a new connecting path. All the above is done in time $O(\log n)$.

Adding a structural child is like the trivial case of merging with a single rank tree consisting of a single node, and is hence also done in $O(\log n)$ time.

Removing a child b from a is very similar to merging. First we remove the connecting path from a. This leaves us with a sorted list of rank trees, from which we pull out the rank tree R_b containing b. We now remove from R_b the path from b to the rank root. This leaves us with a sorted list of rank subtrees of R_b that we can now merge with the remaining rank trees as described above in time $O(\log n)$.

Concerning our tree and non-tree bitmaps, we need to update them at the $O(\log n)$ local nodes affected by the structural change. The updates are done bottom-up. Each update is a constant time bitwise 'or' of the bitmaps at the two children in C^L. Also, we need to update the bitmaps at the

$O(\log n)$ nodes above the structural parent of an added or deleted node.
All parts of our implementation spend $O(\log n)$ time per edge level change, of which we have $O(\log n)$ per edge, so the total cost per edge is $O((\log n)^2)$. Further, we get linear space if we suppress all nodes in C with one child.

PROPOSITION 2. *We can maintain a spanning forest of a fully-dynamic graph over n nodes in $O(\log^2 n)$ amortized time per update using linear space.*

In the following sections, we will now show how to improve almost all the $O(\log n)$ costs from this section to $O((\log \log n)^2)$.

5. LAZY LOCAL TREES

In this section, we are going to present a more lazy version of the local tree, where the vast majority of the changes are confined to poly-logarithmically sized subtrees with operation time $O(\log \log n)$.
Let B be the set of children of a in C. As in §4.4, a local tree is a binary tree with leaf set B and root a, and C^L is the result of replacing for each $a \in C$, the children pointers with the local tree $L(a)$.
The children in B are divided into groups of size at most $2(\log n)^\alpha$, where $\alpha > 2$ is a constant to be determined later. Over each of these groups, we have a standard search tree with operation time $O(\log(2(\log n)^\alpha)) = O(\log \log n)$. One of the small search trees is called *the buffer tree*, while the other are called *bottom trees*. A root of a bottom tree has a rank that is the maximal rank of a descending $b \in B$, where, as in in §4.4, the rank of b is $\text{rank}(b) = \lfloor \log n(b) \rfloor$. All the ranked roots of the bottom trees are now collected in rank trees exactly as described in §4.4, that is, two roots of the same rank r get paired under a new root with rank $r+1$. At the end we have $\leq \log n$ different rank roots, and these together with the root of the buffer tree form the leaves of a standard search tree with root a. We call this last search tree the *top tree*, and like the bottom and buffer trees, the top tree has operation cost $O(\log \log n)$.

LEMMA 3. *The height of C^L is $O(\log n \log \log n)$.*

PROOF. Let b be a structural child of a in C. Then the top, bottom, and buffer trees contributes $O(\log \log n)$ to the depth of b in $L(a)$, while the rank tree contributes at most $\log(n(a)/n(b))$. □

For each nodes in a top tree, we will have a bitmap telling the ranks of rank roots below it. When merging a with b in a, we make a bitwise 'and' of the bitmaps at their top roots to see which ranks they have in common. The corresponding rank roots are identified and paired up in $O(\log \log n)$ time per pairing. Also, we merge the buffer trees. If the resulting buffer tree gets more than $(\log n)^\alpha$ leaves, we turn it into a new bottom tree, leaving an empty buffer tree. The root of the new bottom tree forms a trivial rank tree that we may have to pair up.
When adding a structural child, we just put it under the buffer tree. As under merge, we turn the buffer tree into a bottom tree if it gets more than $(\log n)^\alpha$ leaves.
If we delete a structural child which is not of maximal rank in its bottom tree, this does not affect the rank trees, so we only pay a purely local cost of $O(\log \log n)$ for re-balancing its bottom tree. However, if the child deletion does decrease the rank of the root a of its bottom tree, we delete all rank nodes above a. The deletion is done top down so that any rank node deleted is a rank root. Now the $O(\log n)$ previous siblings of ancestors of a are new rank roots. As under merge, we may have to pair these new rank roots with other rank roots.

LEMMA 4. *Each change in C gives rise to $O(\log \log n)$ changes in the top, bottom, and buffer trees. Moreover, each change in the rank trees is amortized over $(\log n)^{\alpha-2}$ changes in C.*

PROOF. The immediate cost of each change in C is $O(\log \log n)$ changes in the top, bottom, and buffer trees. All other changes in the top, bottom, and buffer trees are amortized over changes in the rank trees.
From an amortization perspective, we can assume that the graphs ends empty, which also means that all rank nodes end up being deleted. Thus creations of rank nodes can be amortized over deletions of rank nodes.
Deletions of rank nodes only happen when a bottom tree root decreases its rank. For each such decrease, we loose at most $\log n$ rank roots. However, the maximal rank is $\log n$, bounding the number of times a bottom tree root can decrease its rank. Thus, each bottom tree can give rise to $(\log n)^2$ rank root deletions. However, a bottom tree is started with at least $(\log n)^\alpha$ leaves, all of which will be deleted eventually. Hence, we conclude that each rank root deletion can be amortized over $(\log n)^{\alpha-2}$ structural child deletions. □

6. LEVEL INDUCED FORESTS

We are now going to address the problem of identifying incident tree edges as in §4.3 in $O((\log \log n)^2)$ time per edge found.
Abstractly, for each level i, we will maintain a *i-induced forest* S_i. The i-induced leaves are the vertices of G with an incident level i tree edge. The i-induced roots are the level $i+1$ nodes $a \in C$ with a descending i-induced leaf. Finally, the i-induced branch node is a node $a \in C^L$ below or equal to an i-induced root with i-induced leaves descending from two different children. The i-induced parent of an i-induced node is its nearest i-induced ancestor. Since C^L is binary, so is S_i.
Our goal is to get between neighboring i-induced nodes in $O((\log \log n)^2)$ time. Then, given a pointer to the i-induced leaf y, we can find all i-induced nodes connected to y in S_i in $O((\log \log n)^2)$ per node, hence in $O((\log \log n)^2)$ time per i-induced leaf. This includes the i-induced root y^{i+1} and the i-induced leaves, which are exactly the vertices connected to y in F_{i+1} with an incident level i tree edge.

7. SHORT CUTTING

An *i-induced trace path* is a path in C^L from a child in C^L of an i-node to its nearest descending i-induced node. Our goal is to make it quick to move between end-points of trace paths. To this end, we are going to maintain a system of short cuts.
First we associate a *power* $\wp(a) \leq 2 \log \log n$ to all nodes in C^L. Nodes interior to bottom, top, and buffer trees all get power 0. If $a \in C^L$ is a rank node with rank $i > 0$, $\wp(a)$ is the least significant bit of i, or equivalently, the maximal j

for which 2^j divides i. If a has rank 0, $\wp(a) = \log\log n$. Let $\wp_C = \log\log n + 1$. Finally, for $v^i \in C$, $\wp(a)$ is $\wp_C$ plus the least significant bit of i, or 0 if $i = 0$.
Suppose a is an ancestor of b and all nodes between a and b have power strictly smaller than $q = \min\{\wp(a), \wp(b)\}$. Then (a, b) is a *short cut of power q* unless one of a and b is in C and the other is a rank node. Note that the arcs in C^L are exactly the short cuts of power 0, and we refer to them as the *trivial short cuts*. If both end-points of a short cut are in C, we call it a *structural short cut.*
We say a short cut (a, b) *short cuts* a path P if a and b are in P. A short cut (a, b) is a *subcut* of a short cut (c, d) if (a, b) short cuts the path from a to b.

LEMMA 5. *If a is an ancestor of b, the maximal short cuts of the path from a to b in C form a path of length $O(\log\log n)$ from a to b.*

PROOF. The worst case is if the path moves from a top tree to a rank tree, to a bottom tree, to C, to a top tree, to a rank tree, to a bottom tree. In each of these trees, we follow $O(\log\log n)$ short cuts, and there are at most 5 trees. □

LEMMA 6. *The total number of short cuts is $O(|C^L|\log\log n)$.*

PROOF. For each power q, the short cuts form a forest over the nodes in C^L. □

An *i-induced short cut* is either a trivial short cut from an i-induced to its child in C^L, or a maximal short cut over some i-induced trace paths.
By a *level induced short cut* we generally refer to an i-induced short cut for some i. We will maintain all level short cuts, and for each level shortcut, we will have a bitmap telling us for which i it is an i-induced short cut.
Consider any node $a \in C$. For each i, a is in at most one i-induced trace path, so a can have at most one upwards and one downwards non-trivial i-induced shortcut. Also, since C^L is binary, so a has at most two downwards trivial short cuts. Hence there are at most $2\log n + 2$ level short cuts incident to a, so a standard search tree will allow us to find an upwards or downwards level i-induced short cut from a, if any, in $O(\log\log n)$ time. Combining this with Lemma 5, gives

LEMMA 7. *We can find the i-induced parent and i-induced children of any i-induced node in $O((\log\log n)^2)$ time.*

As discussed at the end of §6, Lemma 7 immediately tells us how to find the Size from §4.3 in $O((\log\log n)^2)$ time per level i-tree edge in T_v.
As the short cutting structure changes, we will often need to make searches within a trace path. To this end, we will maintain the *base cuts* which are all subcuts of level induced short cuts. In our accounting, we will only pay for creating base cuts, but not for deleting them.

8. INDUCED LEVEL INCREASES

Having identified the i-level tree edges of the smaller tree T_v, we want to increase their levels to $i + 1$. This leads to two subproblems: moving incidence information from S_i to S_{i+1}, and merging some level $i + 1$ nodes. In this section, we deal with the former problem.

We are given a node x^{i+1} on level $i+1$. Loosely speaking we want to take the i-induced tree T rooted in x^{i+1} and merge T in with S_{i+1}. For space reasons, we restrict ourselves to the simple, but illustrative, case where T has only one leaf z.
Our first step is to look for the level $i + 2$-node a which is the structural child of x^{i+1} above z. Since $i + 1$-induced roots are on level $i + 2$, our first natural step is to move the i-induced short cuts over T down to be maximal short cuts from a to z.
If the i-induced down-going short cut from x^{i+1} has power $\leq \wp_C$, we can simply follow the i-induced short cuts down from x^{i+1} until we hit the level $i + 2$-node a. All i-induced short cuts encountered above a are *i-canceled*, meaning that we unset the i-bit in the level short cut, possibly removing the level short cut if this was the last set bit.
Suppose instead that the i-induced down-going short cut (x^{i+1}, b) from x^{i+1} has power $> \wp_C$. Then b is a descendant of the desired level $i + 2$ node a. We then *i-push* (x^{i+1}, b), meaning that we replace it with its immediate subcuts. The immediate subcuts are found by first going to b and then follow up the base cuts of maximal power $< \wp(x^{i+1}, b)$ until we hit x^{i+1}. The base cuts followed are the immediate subcuts that we now make i-induced short cuts. Finally, we i-cancel (x^{i+1}, b). After at most $\log\log n$ pushes, we end up with a short cut (x^{i+1}, a) that we just i-cancel. It is quite easy to see that all the i-induced short cuts created, except for (x^{i+1}, a), are maximal short cuts of the path from a to z, as desired.
If a is not already an $i + 1$-induced root, we simply take all the i-induced short cuts below a and make them $i + 1$-induced, unsetting the i-bit and setting the $i + 1$-bit.
Suppose a is already an $i + 1$-induced root. We then want to find the last node b on the path from a to z with an $i+1$-induced descendant. Then b should be a new $i + 1$-induced branch node. The algorithm works recursively, assuming that a has both i- and $i + 1$-induced short cuts.
Since the leaf z is the only i-induced node below a, we know that the power of the downgoing i-induced short cut (a, a') is at least as big as that of the downgoing $i+1$-induced short cut (a, a'').

LEMMA 8. *a' has $i+1$-induced descendants if and only if it has an incident $i + 1$-induced short cut.*

PROOF. By definition, we cannot have a short cut (c, d) with c a strictly between a and a' and d strictly below a'. □

Thus if a' has an incident $i+1$-induced short cut, we simply i-cancel (a, a') and repeat from a'.
Supose a' has no incident $i + 1$-induced short cut. If (a, a') and (a, a'') both are trivial, a is our new $i+1$-induced branch node b. Then we complete the merge by taking all i-induced short cuts below a and make them $i+1$-induced. Otherwise, if (a, a') and (a, a'') have the same power we push them both, and if (a, a') has the bigger power, we just push (a, a'). In either case we repeat from a after the push.

LEMMA 9. *When T has a single leaf, it takes $O((\log\log n)^2)$ time to merge T in with S_{i+1}.*

PROOF. Generally, it cost $O(\log\log n)$ time to manipulate a short cut. The essential observation is then that we have been following/creating/deleting short cut paths of lenght at most $\log\log n$ from x^{i+1} to the level $i + 2$ node a, from a to the new $i + 1$-branch node b, and from b to z. □

9. THE STRUCTURAL CHANGES

Pretending that there are no non-tree edges, we will now explain how to preserve our level induced short cuts during structural changes.

When removing or adding a structural child a of b, technically it is very convinient to require that a is a root in C^L with power 0. The later implies that there are no structural level induced short cuts from a. Similarly, when merging a node a with a node b, we require that both a and b are roots of C^L of power 0.

In the following, for $i \geq \ell(v,w)$, set u^i to be the current node $v^i = w^i$. That is, u^i will keep pointing to this node in C^L no matter the changes we make to C^L.

Now, for $i = 0, .., \ell(v,w)-1$, we push the power of u^i down to 0 and remove u^{i+1} from its structural parent u^i. Afterwards, for $i = \ell(v,w)$, we puch the power of u^i down to 0, remove v^{i+1} and w^{i+1} from their structure parent u^i, and finally, we push the power of v^{i+1} and w^{i+1} down to 0. After the above procedure, each u^i has lost exactly the structural children that are ancestors of v or w.

Now, inductively, when recover is called for $i = \ell(v,w), ..., 1$, our starting point will be that we have pointers to v^{i+1} and w^{i+1}, both of which are roots of power 0. Also, if $i > 0$, we have a pointer to the previous parent u^i of v^{i+1} and w^{i+1}. All nodes below level $i+1$ are assumed to have the powers specified in §7. Since S_i is rooted at level $i+1$ we can still move between i-induced nodes in $O((\log\log n)^2)$ time.

During the recover on level i, when merging level $i+1$ nodes, first we need to push their power, and remove them from their structural parent u^i. After the merges, v^{i+1} covers T_v. Then we pop the powers of w^{i+1} and v^{i+1} back up to the value specified in §7, and add w^{i+1} back as a child of u^i. If a replacement edge was found, we also add v^{i+1} back as a child of u^i, and then, for $i = \ell(v,w), ..., 1$, we push the power of u^i back up and add it back as a child of u^{i-1}.

If no replacement edge was found, v^i is created with power 0 and v^{i+1} as only child. Since w^{i+1} has been added back under u^i, $w^i = u^i$. If $i = 0$, we are done since nodes on level 0 have power 0. Otherwise, we can now call recover on level $i-1$ with pointers to v^i, w^i, and u^{i-1}.

9.1 Pushing and popping powers

When pushing the power of a node a to 0, a is always a root node. The pushing is done as follows. While there is a down-going level induced short cut of positive power, we pick a level induced short cut (a,b) of maximal power q. Then, for each immediate subcut (c,d) of (a,b), we set $\text{tree}(c,d) := \text{tree}(c,d) \vee \text{tree}(a,b)$. The immediate subcuts are each found in $O(\log\log n)$ time by going down to b and moving back up to a along base cuts of maximal power $< q$. Finally, the level induced short cut (a,b) is removed. Since (a,b) was of maximal power and a was a root, (a,b) is not a subcut of any other level short cut. Hence we remove (a,b) as a base cut. This actually pays for the push because we pay to create base cuts but not to removing them. The process is repeated until the maximal power of a downgoing level short cut is 0.

Popping is much more subtle. However, observe that all nodes popped are ancestors of v and w in the final tree. Ignoring the problem of identifying the short cuts, the lemma below states that the total number of level short cuts and base cuts incident to all these ancestors is sufficiently limited given that we have $O(\log n(\log\log n)^2)$ time available for an edge deletion.

LEMMA 10. *For any vertex v in G, the total number of level induced short cuts and base cuts incident to ancestors of v in C^L is $O(\log n \log\log n)$.*

PROOF. Let P be the path in C^L from v to the root. Then P has length $O(\log n \log\log n)$, so the total number of short cuts of P, including all level induced short cuts and base cuts, is $O(\log n \log\log n)$. The number of trivial short cuts leaving P is also bounded by the length, so it only remains to bound the number of non-trivial level induced short cuts and base cuts leaving P. Consider any level i, and let (a,b) be a non-trivial level i-induced short cut leaving P. The (a,b) short cuts an i-induced trace path. Let c be the last node from P in the path from a to b. Then c is the unique last node in P with a descending i-induced leaf. Since no node is contained in more than one i-induced trace path, we conclude that there is at most one i-induced short cut leaving P. Similarly, for each power $\leq \wp((a,b))$, there is exactly one subcut of (a,b) leaving P; namely the one passing c, so (a,b) has only $O(\log\log n)$ subcuts leaving P. □

9.2 Merging etc.

Merging and adding structural children is very simple. We know that the roots involved have power 0, so we only need to worry about short cuts within the local tree. Then the only non-trivial short cuts are over the rank trees. The important observation now is that a merge or child addition does not affect the subtree below any existing rank node. Thus, merging and child additions can only create problems when rank nodes are created or deleted. However, by Lemma 4, if we set $\alpha \geq 3$, we only delete or create one rank root node per $O(\log n)$ structural changes. This means that we have plenty of time to update the short cutting around the affected rank nodes.

Child deletions are somewhat more tricky. For each i, we will assume a potential of 1 at every i-induced branch node within a rank tree, added when the branch node was created. To see that this is valid, note that when a node in a rank tree becomes i-induced it is either because of a level increase, or because the rank node is created as in Lemma 4. In the latter case, we need at most a potential of 1 for each i, which is is done by setting $\alpha \geq 3$.

Let b be a node to be deleted. If b is a leaf of a buffer tree, the deletion is trivial, so suppose b is a leaf of a bottom tree with root r, and let r^* be the root of the rank tree above r. When b is deleted, in $O((\log\log n)^2)$ time, we update the bitmaps up to r. Suppose this causes some bit $\text{tree}(b)[j]$ to be unset. That is, suppose that b was the only leaf in the bottom tree that had a j-induced descendant.

Suppose r was the only leaf in the rank tree with a j-induced descendants. This means that the path from r up to r^* is a j-induced trace path, so we can follow the path up using $O(\log\log n)$ level short cut of maximal power. Conversely, this means that we can identify all such j by following maximimal level short cuts up from r in $O((\log\log n)^2)$ time, and since the j share these level short cuts, we can cancel them all in $O((\log\log n)^2)$ time.

It remains to deal with the case where r is not the only leaf in the rank tree with a j-induced descendants. In this case, we just have to cancel the j-short cuts up to the first j-induced branch node c above r. Then c is no longer a j-branch node,

and hence we need to pop the j-short cuts incident to c so as to get a proper short cutting of the j-trace path now going though c. All this takes $O((\log\log n)^2)$ time, and is paid by the cancellation of c as a j-branch node.

10. THE NON-TREE EDGES

The main problem in dealing with the non-tree edges is that we want to provide a reasonably uniform sampling as described in §3. What we need for i-induced short cut is an *i-induced weight* telling how many level i tree edges are incident to vertices below it. At first this kills everything done so far because we have been able to store information for all levels in a single bitmap.

We now make the first simple observation, that it suffices with approximate counting. Generally counters will be added up along a path of lenght $O(\log n \log\log n)$ and we can allow each addition to make a mistake by a factor $(1+(\log n)^{-2})$. This means that it suffices to use floats with $O(\log\log n)$ bits of precission. This in turns means that we can store the $\log n$ weights in float maps distributed over $O(\log\log n)$ words. It is straightforward to, say, add two float maps in $O(\log\log n)$ time.

Our second much more challenging problem when dealing with the weights is that every time a new level i non-tree edge arrives or disappears, we have update the i-induced weight counters above it. For weights bigger than $(\log n)^3$, however, the approxmate counting means we are allowed an absolute error of $\log n$.

For i-induced weights smaller than $(\log n)^3$, we introduce a system of heavy paths. A child in S_i is heavy if its i-induced weight is more than twice that of its sibling. Heavy paths are now formed by the paths from parents to their heavy children, and for each heavy path we have an extra efficient short cutting system for the i-induced weights.

Now, when a level i non-tree edge arrives or disappears, for each end-point, we update the i-induced weights along $O(\log\log n)$ heavy paths, until we arrive at a big weight of size $(\log n)^3$. Spending $O((\log\log n)^2)$ time per heavy path, the update up to the big weight takes $O((\log\log n)^3)$ time. At the big weight we wait and accumulate $\log n$ changes before we update the i-induced weights higher up. The higher up i-induced weights can be updated in $O(\log n(\log\log n)^2)$ time, hence in $O((\log\log n)^2)$ time per edge level change. Thus, the total cost per edge level change is $O((\log\log n)^3)$. The integration of these sketchy ideas for weight maintainance with the structural changes is postponed to the journal version of this paper. The total cost per edge level change will be kept at $O((\log\log n)^3)$, implying a randomized fully dynamic connectivity algorithm maintaining a spanning forest in $O(\log n(\log\log n)^3)$ expected amortized time per update.

In the journal version, it will also be shown that the floating point weight idea improves the deterministic time complexity for the 2-edge and biconnectivity in [6] from $O((\log n)^4)$ to $O((\log n)^3 \log\log n)$.

11. REFERENCES

[1] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification — a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, 1997. See also FOCS'92.

[2] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comp.*, 14(4):781–798, 1985. See also STOC'83.

[3] M. Fredman and M. Henzinger. Lower bounds for fully dynamic connectivity problems in graphs. *Algorithmica*, 22(3):351–362, 1998.

[4] M. R. Henzinger and V. King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Proc. 27th STOC*, pages 519–527, 1995.

[5] M. R. Henzinger and M. Thorup. Sampling to provide or to bound: With applictions to fully dynamic graph algorithms. *Rand. Struct. Algor.*, 11:369–379, 1997. See also ICALP'96.

[6] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In *Proc. 30th STOC*, pages 79–89, 1998.

[7] P. B. Miltersen, S. Subramanian, J. S. Vitter, and R. Tamassia. Complexity models for incremental computation. *Theor. Comp. Sc.*, 130(1):203–236, 1994.

[8] D. Sleator and R. Tarjan. A data structure for dynamic trees. *J. Comp. Syst. Sc.*, 26(3):362–391, 1983. See also STOC'81.

[9] R. E. Tarjan. Efficiency of a good but not linear set union algorithms. *J. ACM*, 22:215–225, 1975.

[10] M. Thorup. Decremental dynamic connectivity. In *Proc. 8th SODA*, pages 305–313, 1997.