

Dynamic Graph Connectivity

... as part of the ‘Seminar on Advanced and Mobile Internet Technology’
<https://www.comsys.rwth-aachen.de/teaching/ws-1415/seminars-on-advanced-and-mobile-internet-technology/>

Angelika Schwarz

ABSTRACT

This paper introduces the fully-dynamic graph connectivity problem and some of its applications. Hereby an undirected graph is meant that consists of n fixed vertices and has an initially empty edge set. Then the graph is subject to a sequence of on-line updates, namely edge insertions or deletions. The goal is to support queries of the form “are the vertices v and w in the same connected component?”, which may be asked at any time and always refer to the current graph properties.

This paper presents a deterministic algorithm that answers connectivity queries in $\mathcal{O}((\log n)/(\log \log n))$ time. To keep track of the connectivity status, a hierarchically decomposed spanning forest is maintained. This data structure adapts to modifications of the underlying graph in form of edge insertions or deletions in $\mathcal{O}(\log^2 n)$ amortised time.

1. INTRODUCTION

The connectivity problem addresses the issue of identifying connected components of an undirected graph $G = (V, E)$. Throughout this paper, G is considered to consist of a fixed vertex set V with $|V| = n$. For the purpose of retrieving the connectivity status, queries of the form $\text{connected}(v, w)$ may be asked to check if there is a path between the vertices v and w .

The connectivity problem is called *fully-dynamic* if, in addition to connectivity queries, the following operations are allowed.

- insert ($e = \{v, w\}$). Add the undirected edge $\{v, w\}$ to the edge set E .
- delete ($e = \{v, w\}$). Remove the undirected edge $\{v, w\}$ from the edge set E .

In this paper, an *update* denotes an edge insertion or an edge deletion. An *operation* means an update or a connectivity query.

Throughout the paper, the edge set E of the graph G is assumed to be empty at the beginning. Then G is subject to an arbitrary sequence of updates of the edge set and connectivity queries. This sequence of operations is presented on-line, meaning that each update or query is processed without any knowledge of future updates and queries. Hence, connectivity queries always refer to the current connectivity status of the graph.

The goal is to maintain a dynamic data structure that allows answering connectivity queries efficiently rather than recomputing the connectivity status from scratch each time. Hence, the dynamic data structure must adapt to modifications of the underlying graph efficiently. The key idea to solve the graph connectivity problem is to maintain a spanning forest. As the graph might consist of several connected components, a spanning forest stores a spanning tree for each connected component. In this paper, the term “spanning forest” implicitly stands for a *maximal* spanning forest, i.e. a maximal cycle-free set of edges. Applying the idea of maintaining a spanning forest, Section 3 focuses on a dynamic graph algorithm which supports queries in $\mathcal{O}((\log n)/(\log \log n))$ time and updates in $\mathcal{O}(\log^2 n)$ amortised time.

The definition of the problem and the dynamic graph algorithm, as discussed in this paper, are based on work by Holm, de Lichtenberg and Thorup [9, 12].

Several applications require at some point to determine if two vertices are connected or, in more general terms, if the graph as a whole is connected. This should motivate the above-mentioned dynamic connectivity algorithm. Applications that involve solving the dynamic connectivity problem embrace:

- **Digital Image Processing.** Image processing can require identifying connected or disconnected components of a bitmap image, where a component encompasses pixels of the same colour that are vertically or horizontally adjacent. In terms of dynamic connectivity, pixels may underlie colour changes. Hence, the goal is to keep track of connected components, which is necessary for e.g. video games (cf. [3]).
- **Geographic Information Systems.** Aggregating geographical data with infrastructure information provides the basis for location-based services, interactive maps or navigational systems. Infrastructure networks are dynamic; as a result, connectivity information must be updated regularly. Geographic information systems usually involve the minimisation of costs such as transportation costs (cf. [1]). This augments the dynamic connectivity problem with weights attached to the edges. Consequently, the goal is to maintain a *minimum* spanning forest with respect to costs rather than an arbitrary spanning forest.

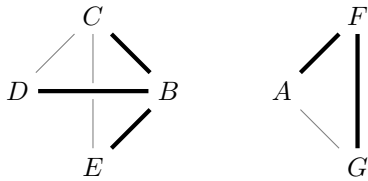


Figure 1: A graph that consists of two connected components. Thick edges mark tree edges that belong to one out of many possible spanning forests.

- **Topology Control.** In mobile wireless ad hoc networks (MANETs), mobile nodes are assigned a transmission power level so that, on the one hand, the desired network topology is achieved, and, on the other hand, the energy consumption for each single node is minimised. As the nodes are subject to movement, the graph is dynamic and power levels have to be adapted constantly. In this context, the dynamic connectivity algorithm is a means of keeping track of the connected components (cf. [13]).

The next section elaborates the idea of dynamically maintaining a spanning forest in order to minimise the cost of connectivity queries. Section 3 refines this idea and eventually implements the algorithm announced above. Section 4 outlines briefly how randomisation allows updates to be processed even faster. Section 5 classifies the presented algorithms with regard to other results on the fully-dynamic connectivity problem. Finally, Section 6 concludes this paper with a summary.

2. A DYNAMIC DATA STRUCTURE

Instead of performing DFS or BFS for every query, it is cheaper to maintain connectivity information, which is updated whenever edges are deleted or inserted. The high-level idea is therefore to maintain a spanning forest, meaning that for every connected component of the graph a spanning tree is maintained. Section 2.1 deals with the problem of adapting the spanning forest to changes of the underlying graph. This, in turn, demands a flexible data structure such as an Euler Tour tree, which is introduced in Section 2.2.

2.1 Spanning Forest

A spanning forest is a collection of spanning trees for each connected component. Figure 1 gives an example for a spanning forest. Whenever the underlying graph is updated, also the spanning forest may have to be updated.

If an edge is inserted into the underlying graph, the following two cases can happen. First, the new edge can connect two previously isolated components. Then the union of the two spanning trees and the newly inserted edge form a new spanning tree. Therefore, a suitable data structure that stores the spanning forest must support a link operation. Second, the new edge connects two vertices that already belonged to the same connected component before the insertion of the new edge. In this case, the new edge does not have any effect on the spanning forest.

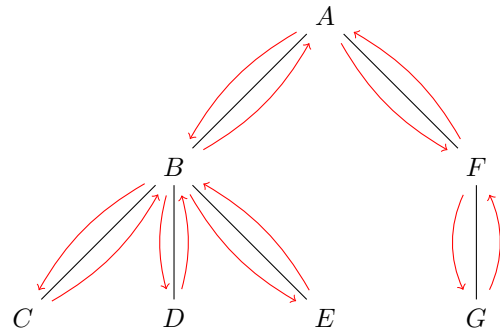


Figure 2: A spanning tree augmented with an Euler tour. One possible ET sequence then is A B C B D B E B A F G F A.

If an edge is removed from the graph, there are again two possibilities. If the edge is not part of the spanning forest, it can simply be deleted without affecting the spanning forest. If the edge is a tree edge, however, its deletion results in splitting the spanning tree into two components. For that reason, a suitable data structure has to allow for efficient cut operations. After the split the underlying graph might still be connected, though. Therefore, a replacement edge that reconnects the parts of the spanning tree has to be searched for.

2.2 Euler Tour Trees

As indicated in the previous section, the goal is to maintain the spanning trees of the connected components in the spanning forest. As such a spanning tree per se features only little structural information, it is difficult to realise efficient link and cut operations when it comes to adapting the spanning forest. For this reason, every spanning tree of the spanning forest is stored in an Euler Tour (ET) tree. An ET tree is especially suited for the representation of dynamic trees because it cumulates information about subtrees and thereby supports in particular the desired link and cut operations efficiently.

2.2.1 Construction

In order to construct an ET tree, the original tree is rooted at an arbitrary vertex. Furthermore, its (undirected) edges are substituted by two anti-parallel directed arcs. The arcs are then traversed such that the tour begins at the root, visits each arc exactly once, and eventually ends at the root. The order in which the vertices are visited yields the ET sequence. During the traversal a vertex can be visited several times and therefore can have several occurrences in the ET sequence. The ET sequence is then stored in a dynamic, balanced search tree, the ET tree. Figure 2 illustrates how to obtain an ET sequence from a tree. Note that the length of any ET sequence is $2n - 1$ and therefore linear in the number of vertices.

Henzinger and King [7] introduce ET trees as dynamic balanced search trees with branching factor b . Thus, the height of an ET tree is $\mathcal{O}(\log_b n)$. For the purpose of solving the dynamic connectivity problem, Holm, de Lichtenberg and Thorup [9] employ two types of ET trees. On the one hand, they set $b = 2$ and thereby store the ET se-

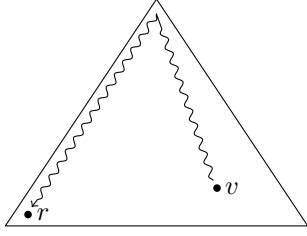


Figure 3: Finding the root r of a spanning tree stored in an ET tree.

quence in a balanced binary search tree. An AVL tree is for instance a suitable data structure for storing the ET sequence. On the other hand, the branching factor is adapted to $b = \Theta(\log n)$. Thus, the height of the search tree reduces to $\mathcal{O}(\log_{\log n} n) = \mathcal{O}((\log n)/(\log \log n))$. A B-tree is a suitable way to represent a $\Theta(\log n)$ -ary balanced search tree.

For the purpose of this paper, the two types of balanced search trees are summed up under the term ET tree if the branching factor is negligible. If the branching factor is relevant, a case distinction is made.

As an ET tree does not preserve the structure of the original (spanning) tree, degenerations like for instance path structures do not affect the ET tree structure. In particular, an ET tree's height remains unaffected.

2.2.2 Implementation

Section 2.1 has explained when changes to the underlying graph trigger a link or a cut operation on the spanning forest. Having defined the structure of an ET tree, these operations can now be implemented. Moreover, the operation $\text{findroot}(v)$ is implemented in order to prepare efficient connectivity queries, which is going to be tackled in Section 3. Note that at this stage the cut operation does not solve the problem of finding a replacement edge. Based on [2, 7, 9], the ET tree operations are implemented as follows.

- **findroot(v)**. In an ET tree, the root of the original tree is the first vertex in any ET sequence and therefore stored at the very bottom in the leftmost path of the tree. Figure 3 demonstrates how the root can be identified for a vertex v in an ET tree. A query $\text{findroot}(v)$ therefore costs at most $2 \cdot |\text{height}|$.
If the ET tree equals a balanced binary search tree, the costs for a query are $\mathcal{O}(\log n)$. For a $\Theta(\log n)$ -ary search tree, the time spent on a query reduces to $\mathcal{O}((\log n)/(\log \log n))$. This is the case because the $\Theta(\log n)$ entries stored per node in the search tree are irrelevant for finding the root.
- **cut(w)**. When the subtree rooted at w is removed from the rest of the tree, there is an edge $\{v, w\}$ that connects w 's subtree with the rest. Figure 4 illustrates the situation. In the corresponding ET sequence, w 's subtree is a contiguous subsequence, which is denoted by $ET(T_w)$. $ET(T_w)$ is framed by two occurrences of v . Removing $ET(T_w)$ and one of the two occurrences of v yields the new ET sequence. After the cut, rebalancing may be necessary.

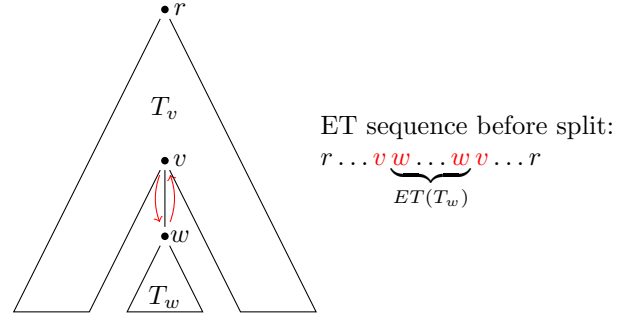


Figure 4: Deletion of the tree edge $\{v, w\}$ in the spanning tree amounts to splicing out the contiguous interval that corresponds to T_w 's ET sequence from the spanning tree's overall ET sequence.

For the time complexity, the decisive factor is the number of ET vertices that rebalancing can affect. In a binary search tree, this number is bound by $\mathcal{O}(\log n)$ ET vertices due to the tree height of $\mathcal{O}(\log n)$. Hence, a cut costs $\mathcal{O}(\log n)$. In a search tree with branching factor $\Theta(\log n)$, however, a cut can affect up to $\mathcal{O}((\log^2 n)/(\log \log n))$ vertices because on a path from a leaf to the root $\mathcal{O}((\log n)/(\log \log n))$ vertices may have to be rebalanced and each of the vertices visited has $\mathcal{O}(\log n)$ children.

- **link($\text{tree}_1, \text{tree}_2$)**. Two trees are linked if a new edge $e = \{v, w\}$ is inserted that connects two previously isolated trees. Let $ET(T_v)$ be the ET sequence of the tree that contains the vertex v . Furthermore, let $ET(T_w)$ be the ET sequence of w 's tree with w being the root¹. The ET sequence of the new linked tree can be constructed as follows, which is also illustrated in Figure 5. $ET(w)$ is inserted into $ET(v)$ immediately after v 's last occurrence. After $ET(w)$ another occurrence of v is inserted, which in turn is followed by the remainder of $ET(v)$. A link operation may cause the resulting tree to be unbalanced.
- **reroot(r, s)**. Changing the root of a tree from r to s requires a constant number of splits and concatenations on the ET sequence. Let o_s be any occurrence of s . Remove the last occurrence of r , which is the last entry in the ET sequence. Then the new ET sequence with s as root can be obtained by splicing out the first part of the ET sequence up to o_s and attaching it to the end. Finally, one occurrence of s is added at the very end of the ET sequence. An example for rerooting is illustrated in Figure 6.

¹Note that only T_w might have to be rerooted; changing the root of T_v to v is not necessary as done in [7].

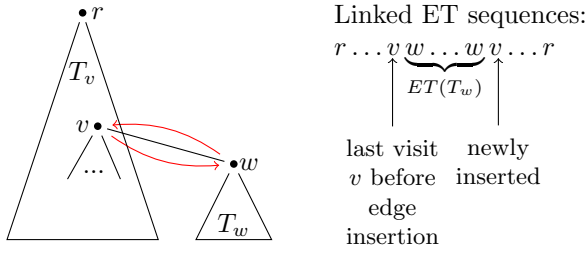


Figure 5: The insertion of the edge $\{v, w\}$ effects that w 's subtree becomes the last child of v .

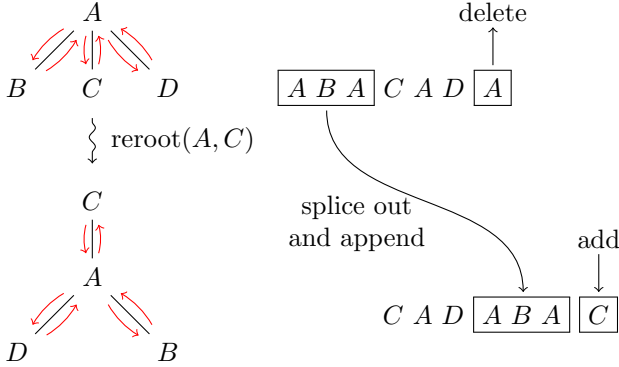


Figure 6: Changing the root from A to C in the original tree and its emulation by ET sequences.

After appending the left part of the former tree to the end, the resulting tree may be unbalanced. This again affects $\mathcal{O}(\log n)$ vertices in a binary search tree and $\mathcal{O}((\log^2 n)/(\log \log n))$ vertices in a $\Theta(\log n)$ -ary search tree.

The choice of the branching factor of the ET trees turns out to be crucial for the trade-off between queries and updates on the original graph G . In order to answer connectivity queries, the ET tree operation findroot is going to be used. As already mentioned above, ET tree links and cuts may be triggered if G is updated. On the one hand, a binary ET tree yields that the operations findroot, link and cut uniformly take $\mathcal{O}(\log n)$ time. On the other hand, using a branching factor of $\Theta(\log n)$ speeds up the operation findroot, but it slows down links and cuts.

Reflecting on the trade-off should adumbrate why two types of ET trees are going to be employed in the algorithm that solves the dynamic connectivity problem. For fast queries, a $\Theta(\log n)$ -ary tree is the favourite option, but for updates a binary tree is superior. The next chapter scrutinises how the advantages of both types of trees can be combined.

3. DYNAMIC CONNECTIVITY

Relying on ET trees as data structure to maintain connectivity information, the goal now is to achieve an efficient implementation of the operations $\text{connected}(v, w)$, $\text{insert}(e = \{v, w\})$ and $\text{delete}(e = \{v, w\})$ on the graph G . The operations $\text{connected}(v, w)$ and $\text{insert}(e = \{v, w\})$ are easy to realise via ET tree operations. A deletion, however, turns out to be a delicate operation if a replacement edge has to

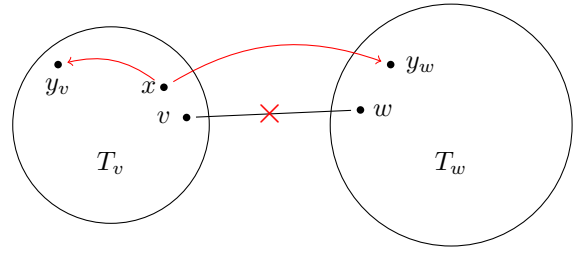


Figure 7: Deleting the tree edge $\{v, w\}$ yields two temporarily split subtrees T_v and T_w , respectively. The search for a replacement edge is conducted in the smaller subtree T_v .

be searched for. To support deletions efficiently, a hierarchical decomposition of the spanning forest, as presented in [9, 12], is conducted. The following sections derive the desired connectivity algorithm step-by-step.

3.1 Reduction of the Search Space

Reconsider that there are two possibilities if a graph edge $\{v, w\}$ is deleted. The graph edge is either a tree edge of the spanning forest or it is a non-tree edge. The latter case is easy to handle because the edge can simply be deleted without affecting the spanning forest. The former case, however, is difficult because a replacement edge has to be found, if one exists. Searching for a replacement edge in a naive way is expensive because potentially every vertex x in either of the two split components T_v or T_w has to be investigated. Figure 7 shows the situation after the deletion of the tree edge $\{v, w\}$. Assume w.l.o.g. x to be in T_v . Then every edge $\{x, y\}$ is considered: Either a replacement edge is found, i.e. $y \in T_w$, or, unluckily, an edge that stays within T_v is found, i.e. $y \in T_v$. Obviously, this kind of uninformed search, which could for instance be conducted with DFS, can be very expensive if many vertices and edges have to be considered, but none is a replacement edge.

In order to support deletions efficiently, a hierarchy of spanning forests is introduced. The high-level idea is that considering a graph edge that does not reconnect T_v and T_w has to charge against detecting a reconnecting edge. It is essential to note that this kind of consideration is only feasible because one is interested in an amortised time bound rather than a worst-case analysis. As the graph is subject to a *sequence* of updates, an amortised analysis counts the *average* time required per update. Consequently, few expensive operations can be afforded as long as there are enough operations that pay for those expensive operations.

For the hierarchical decomposition, each edge is assigned a level. The level of an edge is variable and can attain numbers between 0 and $\lfloor \log_2 n \rfloor$. Newly inserted edges start at level 0; at this point it is important to remember that the initial graph consists of $|V| = n$ isolated vertices. The level of an edge can only increase over time, but never decrease. Level increases are used to 'pay' for unsuccessful testing, which is going to be described in the algorithm further down. As the level of an edge can only increase until the maximal level $l_{max} = \lfloor \log_2 n \rfloor$ is reached, an edge can be charged against only $\mathcal{O}(\log n)$ times; then the edge has to be deleted before it

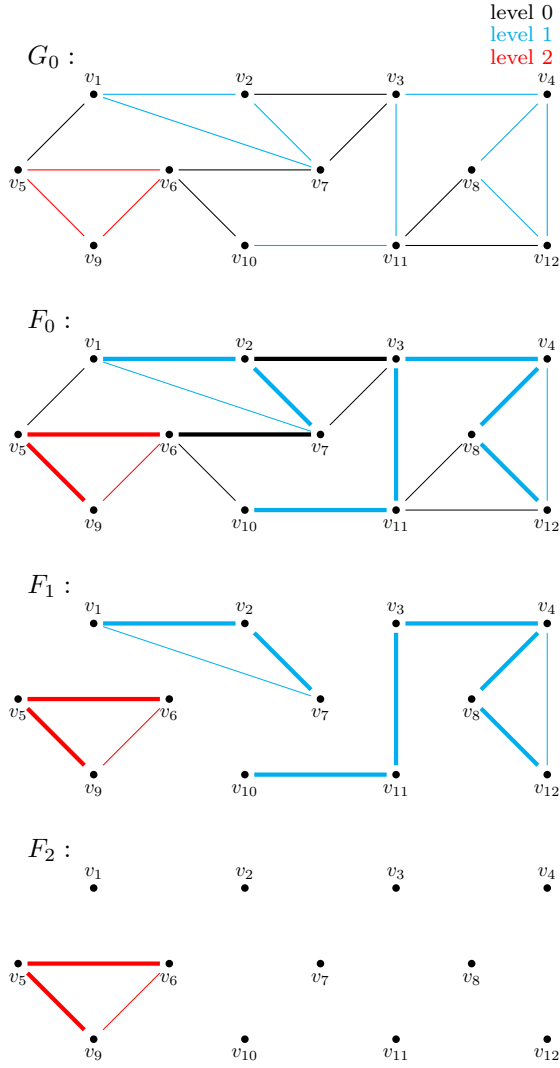


Figure 8: Hierarchical decomposition of G 's spanning forest. Thick edges are tree edges.

becomes relevant again. This property becomes important when the amortised costs of updates will be considered.

Let F be a spanning forest of G . Having introduced the levels of edges, the hierarchical decomposition of the graph G and its spanning forest F is defined as follows: G_i is the subgraph of G that is induced by all edges of G that have a level of at least i . This implies that $G = G_0$. Accordingly, the spanning subforest F_i is defined as the spanning forest of G_i induced by F , or, more formally, $F_i := G_i \cap F$. Consequently, it is $F = F_0$. Figure 8 demonstrates the graph decomposition with an example.

The definition of F_i makes the spanning forests form a nested chain: $F = F_0 \supseteq F_1 \supseteq \dots \supseteq F_{l_{max}}$. In terms of connectivity information, it would suffice to store only F_0 . All other $F_i, i \geq 0$, serve only one purpose and that is to support deletions efficiently.

At this point, the ET trees as a suitable data structure for

trees become important again. Here, every spanning forest F_i is stored in ET trees. Thus, for every connected component of the l_{max} spanning forests an ET tree is used. ET trees have been introduced as either binary or $\Theta(\log n)$ -ary search trees. For F_0 , $\Theta(\log n)$ -ary ET trees are employed. All other spanning forests $F_1, \dots, F_{l_{max}}$ use binary ET trees.

Throughout the entire algorithm two invariants are maintained.

1. F is a maximum spanning forest of G . Here, “maximum” means that the level of an edge is regarded as its weight.
2. The number of vertices in every connected component in F_i is at most $\lfloor n/2^i \rfloor$.

The definitions of the operations $\text{delete}(v, w)$ and $\text{insert}(v, w)$ have to be designed in such a way that the two invariants are preserved. For a better understanding, the invariants are discussed in greater detail before $\text{delete}(v, w)$ and $\text{insert}(v, w)$ are implemented.

According to **invariant 1**, F is a maximum spanning forest with respect to level. This implies that $F = F_0$ has to prefer higher-level edges. When tree edges are deleted, this invariant defines a search strategy and thereby reduces the search space for a replacement edge. In other words, a level- i edge (v, w) can only be a non-tree edge if there is a path from v to w containing only edges that have at least level i .

For better understanding, consider the following scenarios. On the one hand, a graph that has only level-0 edges may store any spanning forest. On the other hand, a triangle connected component that has one level-0 edge, one level-1, and one level-2 edge has a unique spanning tree. As the spanning tree of the connected component is defined to be maximal with respect to level, it comprises the level-1 and the level-2 edge.

Invariant 2 bounds the size of connected components: Every time the level increases by one, the maximal size of a component goes down by a factor of two. Due to this property, $\lfloor \log_2 n \rfloor$ is the maximum number of levels that an edge can attain. In the algorithm this property allows charging against unsuccessful search for a replacement edge.

For clarification, recall the hierarchical decomposition of F_1 and F_2 in Figure 8. The graph consists of 12 vertices. In F_0 all 12 vertices may be in one connected component. In F_1 , however, the maximal size of a connected component reduces to 6; in F_2 , the maximal size reduces to 3.

3.2 Implementation

Now the query $\text{connected}(v, w)$ and the update operations $\text{insert}(e = \{v, w\})$ and $\text{delete}(e = \{v, w\})$ over the hierarchically decomposed spanning forests are implemented.

- **connected** (v, w) . Check in F_0 , if in the corresponding ET trees the results of $\text{findroot}(v)$ and $\text{findroot}(w)$ coincide. If so, v and w are in the same connected component.

- **insert**($e = \{v, w\}$).

1. Set the level of e to 0.
2. Check if v and w are in the same connected component via the query `connected( $v, w$ )`. If not, add e to F_0 .

Note that an insertion conforms to both invariant 1 and invariant 2.

- **delete**($e = \{v, w\}$).

1. Test if e is a tree edge in F_0 : If neither v 's parent equals w nor w 's parent equals v , then e is a non-tree edge. In this case, delete e and stop. Otherwise:
 2. Let l be the level of e . Remove e from all spanning forests F_i with $0 \leq i \leq l$.
 3. For $i = l, \dots, 0$:
 - (a) Let T_v and T_w be the split subtree of F_i containing v and w , respectively. Assume w.l.o.g. $|T_v| \leq |T_w|$.
 - (b) Increase the level of all edges in T_v to $i + 1$.
 - (c) Check each level- i edge $\{x, y\}$ with $x \in T_v$.
 - i. If $y \in T_w$, a replacement edge is found. $\{x, y\}$ is inserted into all F_k , $0 \leq k \leq i$, and the search stops.
 - ii. Otherwise, $y \in T_v$ and no replacement edge is found. Then the level of $\{x, y\}$ is increased to $i + 1$.

In step 1 it suffices to examine if e is part of F_0 because the spanning forests nest. If e is part of a higher-level spanning forest, it is also contained in F_0 . Provided that e is a tree edge, examining F_0 directly reveals e 's level, which is used for the next steps.

Step 2 removes e from all spanning forests that are affected by the deletion.

Step 3 targets the problem of identifying a replacement edge or concluding that there is no replacement edge. When a replacement edge is searched for, invariant 1 restricts possible candidates to edges that have at most level l , the level of e . As F is supposed to be a *maximum* spanning forest, the search for a replacement edge starts at the highest possible level, which is l .

In step 3(a) the smaller of the two split subtrees is determined. The subsequent steps aim at finding a replacement edge to reconnect the split subtrees whilst preserving the invariants. Therefore the impact of the deletion of the tree edge on subtree sizes has to be considered. Immediately before the deletion of e , the tree $T = T_v \cup \{\{v, w\}\} \cup T_w$ complied with invariant 2, i.e. T was a spanning tree on level i with at most $\lfloor n/2^i \rfloor$ vertices. Regarding the sizes of the two subtrees, $|T| = |T_v| + |T_w| \leq \lfloor n/2^i \rfloor$ entails $2 \cdot |T_v| \leq \lfloor n/2^i \rfloor$ and thereby also $|T_v| \leq \lfloor n/2^{i+1} \rfloor$. Thus, the size of the smaller subtree T_v shrinks by a factor of at least two.

At this point it is not known whether T_v and T_w really belong to disconnected components in F_i as there might be a replacement edge that has not been discovered yet. What is known, however, is that T_v and T_w

are in disconnected components on level $i + 1$ because F is a *maximum* spanning forest. One can therefore afford to increase the level of all edges in T_v to $i + 1$, which is done in step 3(b), and still comply with invariant 2. At this point, increasing the levels is not necessary for finding a replacement edge, but in terms of an amortised analysis, pushing the level up pays in advance for future operations.

In step 3(c), level- i edges emanating from any vertex in the smaller tree T_v are considered one after another. Let f be an edge that is being considered during the process. If f reconnects T_v and T_w , f is inserted as a replacement edge in $F_i, \dots, F_0$ and the search stops. Otherwise, f does not reconnect T_v and T_w . Then its level is increased to $i + 1$ in order to pay for considering the edge and the search continues. Once all level- i edges have been considered, the search continues on level $i - 1$. If there are no more edges that can be considered on level $i = 0$, there is no replacement edge and T_v and T_w are indeed split.

The procedure for deletions contains two nontrivial steps. First, in step 3(a) the sizes of the subtrees T_v and T_w have to be determined. Augmenting the ET trees with an argument that stores its size resolves this problem. Second, step 3(c) requires the data structure to allow iterating over incident level- i edges efficiently. Therefore, each vertex stores separate incidence lists for each level for tree respectively non-tree edges. Thus, one single vertex maintains $2 \times \lfloor \log_2 n \rfloor$ incidence lists, namely

- level-0 non-tree edges,
- level-0 tree edges,
- level-1 non-tree edges,
- level-1 tree edges,
- ...
- level- $\lfloor \log_2 n \rfloor$ non-tree edges and
- level- $\lfloor \log_2 n \rfloor$ tree edges.

Summing up the space requirement for all vertices, in total $\mathcal{O}(m + n \log n)$ space is required. This bound incorporates the $\mathcal{O}(m)$ space requirement for non-tree edges at any level. Furthermore, the tree edges for each spanning forest have to be stored. Each spanning forest contains at most $n - 1$ tree edges. Due to the maximal edge level increase of $\mathcal{O}(\log n)$, there are $\mathcal{O}(\log n)$ spanning forests. Hence, the data structure uses in total $\mathcal{O}(m) + \mathcal{O}(\log n \cdot (n - 1)) = \mathcal{O}(m + n \log n)$ space.

3.3 Runtime Analysis

According to Holm, de Lichtenberg and Thorup [9] connectivity queries take $\mathcal{O}((\log n)/(\log \log n))$ time and updates require $\mathcal{O}(\log^2 n)$ amortised time. This section analyses the implementation of the operations `connected( $v, w$ )`, `insert( $e = \{v, w\}$ )` and `delete( $e = \{v, w\}$ )`, as presented in the previous section, step-by-step.

A **connectivity query** operates on F_0 . As F_0 's connected components are stored in $\Theta(\log n)$ -ary ET trees, the cost

of a connectivity query adds up the cost of two findroot ET tree operations, i.e. $\mathcal{O}((\log n)/(\log \log n))$. Note at this point that the trick of extending the branching factor to $\Theta(\log n)$ is only a feasible improvement because the search is restricted to finding the root of the original tree. Never is an arbitrary vertex searched for [9].

Considering edge **insertions**, step 1 can be performed in $\mathcal{O}(1)$. The retrieval of the connectivity status in step 2 costs $\mathcal{O}((\log n)/(\log \log n))$ time. If the new edge becomes a tree edge, the insertion corresponds to linking two $\Theta(\log n)$ -ary ET trees. This link operation costs $\mathcal{O}((\log^2 n)/(\log \log n))$. In total, an insertion costs $\mathcal{O}((\log^2 n)/(\log \log n))$ worst-case time. This runtime is acceptable against a backdrop of deletions requiring $\mathcal{O}(\log^2 n)$ amortised time, which is discussed in the next paragraph. Aiming for *updates* in $\mathcal{O}(\log^2 n)$ amortised time, it is sufficient to support edge insertions in $\mathcal{O}((\log^2 n)/(\log \log n)) = \mathcal{O}(\log^2 n)$ time.

Regarding **deletions**, step 1 requires $\mathcal{O}(1)$ time. Step 2 is reached if a tree edge is deleted. In the worst case, the deleted edge e has level l_{max} . Removing e from F_0 , i.e. using an ET tree cut, costs $\mathcal{O}((\log^2 n)/(\log \log n))$. Removing e from any of the binary ET trees $F_1, \dots, F_{l_{max}}$ costs $\mathcal{O}(\log n)$ each. Hence, step 2 takes $\mathcal{O}((\log^2 n)/(\log \log n)) + \mathcal{O}(\log n \cdot \log n) = \mathcal{O}(\log^2 n)$ time. For step 3, 3(c) is the expensive operation in each loop. If a replacement edge is found, it has to be inserted into all relevant spanning forests. The insertion, of course, corresponds to an ET tree link and therefore adds up to

$$\begin{aligned} & \underbrace{\mathcal{O}\left(\frac{\log^2 n}{\log \log n}\right)}_{\text{insertion in } F_0} + \underbrace{(l-1) \cdot \mathcal{O}(\log n)}_{\substack{\text{forests} \\ F_1, \dots, F_l}} + \underbrace{\mathcal{O}(\log n)}_{\substack{\text{link of} \\ \text{binary} \\ \text{ET tree}}} \\ &= \mathcal{O}\left(\frac{\log^2 n}{\log \log n}\right) + \mathcal{O}(\log n) \cdot \mathcal{O}(\log n) \\ &= \mathcal{O}(\log^2 n) \end{aligned}$$

time. Otherwise, the cost amortised over level increases is $\mathcal{O}(\log n \cdot |\text{level increases}|) = \mathcal{O}(\log^2 n)$.

At this point it should become clear, why two types of ET trees, namely binary and $\Theta(\log n)$ -ary ET trees, are employed. In the section about ET trees, the claim was that the branching factor of the ET trees is the crucial factor for the trade-off between update bounds and query bounds. To speed up queries, F_0 's connected components are stored in $\Theta(\log n)$ -ary rather than binary ET trees. One can afford to slow down updates for this particular forest because one amortises over level increases. If all spanning forest used $\Theta(\log n)$ -ary ET trees, step 2 and 3c) in a tree edge deletion would require $\mathcal{O}((\log^3 n)/(\log \log n))$ amortised time. Therefore, only F_0 utilises $\Theta(\log n)$ -ary ET trees, whereas $F_1, \dots, F_{l_{max}}$ use binary ET trees yielding the desired update time.

3.4 Example of a Tree Edge Deletion

As deletions are the most complicated operations, a detailed example is presented for the purpose of a better understanding. Therefore, reconsider Figure 8, which will be the start-

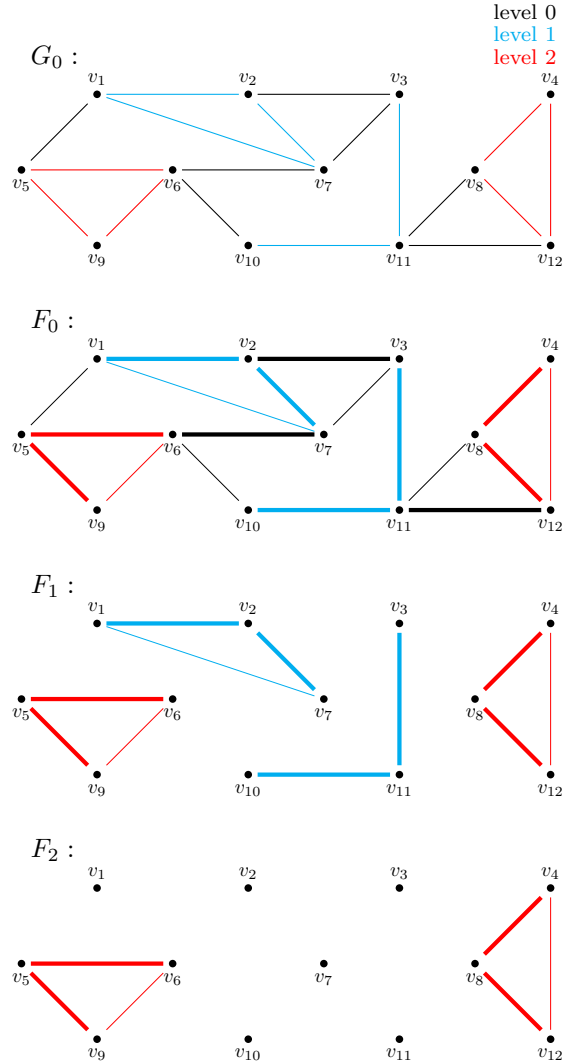


Figure 9: Updated spanning forest after the deletion of the level-1 tree edge $\{v_3, v_4\}$.

ing point for this extended example. This example demonstrates in a step-by-step fashion how the deletion of a tree edge is processed.

The level-1 tree edge (v_3, v_4) is deleted and removed from G_0 , F_0 and F_1 . F_2 is not considered because a level-1 edge cannot appear in any spanning forest F_i where i is greater than the level of the deleted edge. The deletion splits the spanning tree into T_{v_3} 's component and T_{v_4} 's component, respectively. Thus, a replacement edge has to be searched for.

The search for a replacement edge starts in F_1 since (v_3, v_4) was a level-1 edge. In F_1 , both T_{v_3} and T_{v_4} have three vertices. At this point, any of the two subtrees can be elected for further processing; in this example, T_{v_4} is chosen. T_{v_4} 's tree edges are increased in level to pay for the search of a replacement edge; T_{v_4} contains two level-1 edges, which become level-2 edges. Now level-1 edges emanating from T_{v_4} are tested for reconnectivity. There is only one level-1 edge and that is $\{v_4, v_{12}\}$. As this edge does not reconnect the

split components, its level is increased to pay for considering the edge. As there are no more level-1 edges, the search for a replacement edge recurses on level 0.

The order in which the level-0 edges are considered is random. Here, the level-0 edge $\{v_{11}, v_{12}\}$ is tested first. This edge does indeed reconnect the split components and is therefore added to the spanning forest. The search for a replacement edge stops immediately. Figure 9 displays how the graph looks like afterwards.

4. FURTHER IMPROVEMENTS

This section outlines how the results presented in the previous section can be exploited to speed up updates even further. The resulting algorithm by Thorup [12] applies randomisation in the form of sampling, so that time bounds in this section are *expected* amortised.

This section only describes the high-level idea, but omits details about the implementation. The critical operation is once again the search for a replacement edge, when a tree edge has been deleted. The key idea to deal with deletions remains the same, meaning that the increase of the level of an edge charges against the search for a replacement edge.

The goal of Thorup’s algorithm is to support deletions in $\mathcal{O}(\log n \cdot (\log \log n)^3)$ expected amortised time. As an edge can be charged against up to $l_{max} = \lfloor \log_2 n \rfloor$ times, testing a non-reconnecting edge may cost at most $\mathcal{O}((\log \log n)^3)$. How the latter bound is achieved, will not be discussed here. Thorup uses a special data structure, namely a structural forest, for maintaining the hierarchically decomposed graph rather than separate ET trees for each connected component of every spanning forest F_i . Here, “structural forest” does not refer to the possibly ambiguous spanning forests F_i , but to the hierarchy of connected components induced by G_i . This allows him to handle modifications of the underlying graph more flexibly. Combined with a more even distribution of level charges due to randomisation and the maintenance of additional structural information such as tree sizes, this gives an $\mathcal{O}(\log n \cdot (\log \log n)^3)$ bound.

In contrast to the algorithm from the previous section, Thorup’s method aims at considering *all* non-tree edges independent of whether a replacement edge has been found or not. The problem with this idea is that only non-reconnecting edges can pay for being considered. Here, “paying” again means increasing the level of an edge. Reconnecting edges themselves cannot pay for being considered because the hierarchical decomposition must comply with invariant 2, i.e. the size of connected components on each level is bound. Consequently, there is a threshold for the number of reconnecting edges whose consideration can be afforded. This threshold is $\mathcal{O}(\log n)$ with respect to the update bound of $\mathcal{O}(\log n \cdot (\log \log n)^3)$ being aimed for.

Assume that T_v is the smaller subtree of the split spanning tree. Furthermore, assume that the search is proceeding on level i . Due to the threshold, Thorup distinguishes between two cases.

1. T_v has at most $\mathcal{O}(\log n)$ incident level- i non-tree edges. In this case, considering *all* edges is possible, even if

all non-tree edges are feasible replacement edges. If no replacement edge is found, the level of every edge considered is increased to $i + 1$. In case of finding at least one replacement edge, this edge pays to the procedure. In either scenario the update bound is satisfied.

2. T_v has at least $\Omega(\log n)$ incident level- i non-tree edges. In other words, there may be more reconnecting edges than the maximally affordable number. A sampling procedure is applied to check if the number of reconnecting edges is *probably* below the threshold. Therefore, an expected constant number of level- i non-tree edges emanating from T_v is picked at random. If a replacement edge is found amongst the samples, this replacement edge pays for the procedure. Otherwise, having considered a sufficiently high number of samples without detecting a replacement edge, it is likely that the number of replacement edges is below the threshold. Therefore, the procedure of case 1 can be applied.

In sum, Thorup’s algorithm based on sampling improves the update time to $\mathcal{O}(\log n \cdot (\log \log n)^3)$. Queries slow down slightly and can be answered in $\mathcal{O}((\log n)/(\log \log \log n))$. A side benefit of using the structural forest, a custom-made data structure, is that the space requirement reduces to $\mathcal{O}(m)$ compared to $\mathcal{O}(m + n \cdot \log n)$ in the algorithm from the previous section.

5. RELATED RESULTS

This section presents some results on the fully-dynamic connectivity problem on general graphs. In this context, a pertinent question is how far improvements can possibly go, i.e. what is a lower bound. In addition, this section targets open questions.

For a better overview, Table 1 shows how algorithms that are based on an amortisation argument have evolved over time (cf. [2, 12]). The two best results known have been presented in this paper. On the one hand, there is the algorithm [9] that supports updates in $\mathcal{O}(\log^2 n)$ and queries in $\mathcal{O}((\log n)/(\log \log n))$ time, which has been discussed in greater detail in this paper. On the other hand, it is possible to speed up updates at the expense of queries [12], which has been outlined in Section 3. Updates then require $\mathcal{O}(\log n \cdot (\log \log n)^3)$ expected amortised time, whereas connectivity queries slow down to $\mathcal{O}((\log n)/(\log \log \log n))$.

Table 1 suggests that there is a relation between the performance achievable for updates and queries. Seemingly it is not possible to speed up updates without slowing queries at some point. Indeed, updates and queries mutually depend on each other with regard to lower bounds.

In 2004 Pătraşcu and Demaine [11] proved an $\Omega(\log n)$ amortised lower bound on any data structure meaning that either updates or queries have to take $\Omega(\log n)$ time. In other words, it is not possible to achieve both sublogarithmic updates and sublogarithmic queries. Reconsidering the complexities mentioned in the previous paragraph, updates in either case satisfy the lower bound $\Omega(\log n)$ allowing queries to be sublogarithmic. In addition, Pătraşcu and Demaine identified how updates and queries mutually relate each other:

update	query	year	source
$\mathcal{O}(\sqrt{m})$	$\mathcal{O}(1)$	1983	[5]
$\mathcal{O}(\sqrt{n})$	$\mathcal{O}(1)$	1992	[4]
$\mathcal{O}(\log^3 n)$	$\mathcal{O}\left(\frac{\log n}{\log \log n}\right)$	1995	[7] ^{*,‡}
$\mathcal{O}(\log^2 n)$	$\mathcal{O}\left(\frac{\log n}{\log \log n}\right)$	1996	[8] ^{*,‡}
$\mathcal{O}(\sqrt[3]{n} \cdot \log n)$	$\mathcal{O}(1)$	1997	[6] [‡]
$\mathcal{O}(\log^2 n)$	$\mathcal{O}\left(\frac{\log n}{\log \log n}\right)$	1998	[9] [‡]
$\mathcal{O}(\log n \cdot (\log \log n)^3)$	$\mathcal{O}\left(\frac{\log n}{\log \log \log n}\right)$	2000	[12] ^{*,‡}
$\mathcal{O}(\log^5 n)$	$\mathcal{O}\left(\frac{\log n}{\log \log n}\right)$	2012	[10]

Table 1: Advances on the fully-dynamic connectivity problem on general graphs. The algorithms tagged with ‡ are based on an amortisation argument. The star (*) marks algorithms that apply randomisation.

1) A superlogarithmic update time $\mathcal{O}(x \cdot \log n)$, $x > 1$, causes an $\Omega((\log n)/(\log x))$ query time. Note how the two algorithms discussed above match this definition; hence, the algorithms are optimal with respect to the trade-off between updates and queries. 2) As Table 1 reveals, all update times known so far take at least $\Omega(\log n)$. It is an open problem if it is possible to design a dynamic data structure in such a way that updates are supported in sublogarithmic time. If so, an $\mathcal{O}(x \cdot \log n)$, $x > 1$, query time goes along with an $\Omega((\log n)/(\log x))$ update time.

So far all results have been based on an amortised analysis, but also worst-case bounds have been examined. For more than a decade the best results performed updates in $\mathcal{O}(\sqrt{n})$ and queries in $\mathcal{O}(1)$ time. For a long time it was an open question if polylogarithmic worst-case updates and queries are possible. This question could be answered recently, as Kapron, King and Mountjoy [10] have found an algorithm that solves the dynamic graph connectivity problem in polylogarithmic worst-case time; an insertion costs $\mathcal{O}(\log^4 n)$, a deletion $\mathcal{O}(\log^5 n)$ and a query $\mathcal{O}((\log n)/(\log \log n))$ time.

6. CONCLUSION

In this paper, solutions to the fully-dynamic graph connectivity problem have been presented in a step-by-step approach. The initial idea of maintaining a spanning forest has been refined, once the deletion of tree edges has been identified to be the critical operation. The deletion of a tree edge is potentially expensive because a replacement edge has to be found. The focus therefore shifted onto narrowing down the search space for a replacement edge. The solution to this problem is a hierarchical decomposition of the spanning forest. Exploiting this, a deterministic algorithm supporting connectivity queries in $\mathcal{O}((\log n)/(\log \log n))$ and updates in $\mathcal{O}(\log^2 n)$ amortised time has been presented. Its implementation heavily relies on ET trees as the underlying data structure. Applying sampling, it is possible to speed up

updates even further, but at the expense of slower queries. Finally, the results have been discussed in the context of related work.

7. REFERENCES

- [1] P. Bakalov et al. Maintaining connectivity in dynamic multimodal network models. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 1267–1276. IEEE, 2008.
- [2] E. D. Demaine. Advanced Data Structures. Lecture notes, <http://courses.csail.mit.edu/6.851/spring10/scribe/lec17.pdf>, April 2010. Online source, accessed: October 2, 2013.
- [3] D. Eppstein. Dynamic connectivity in digital images. *Information Processing Letters*, 62(3):121–126, 1997.
- [4] D. Eppstein et al. Sparsification - a technique for speeding up dynamic graph algorithms. *Journal of the ACM (JACM)*, 44(5):669–696, 1997.
- [5] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 14(4):781–798, 1985.
- [6] M. R. Henzinger and V. King. Maintaining minimum spanning trees in dynamic graphs. In *Automata, Languages and Programming*, pages 594–604. Springer, 1997.
- [7] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM (JACM)*, 46(4):502–516, 1999.
- [8] M. R. Henzinger and M. Thorup. Sampling to provide or to bound: with applications to fully dynamic graph algorithms. *Random Structures and Algorithms*, 11(4):369–379, 1997.
- [9] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM (JACM)*, 48(4):723–760, 2001.
- [10] B. M. Kapron, V. King, and B. Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *SODA*, pages 1131–1142, 2013.
- [11] M. Pătraşcu and E. D. Demaine. Lower bounds for dynamic connectivity. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 546–553. ACM, 2004.
- [12] M. Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 343–350. ACM, 2000.
- [13] L. Zhao, E. L. Lloyd, and S. Ravi. Topology control in constant rate mobile ad hoc networks. *Wireless Networks*, 16(2):467–480, 2010.