IADIS International Journal on Computer Science and Information Systems
Vol. 16, No. 1, pp. 1-15
ISSN: 1646-3692

DYNAMIC CONNECTIVITY: SOME GRAPHS OF INTEREST

George Lagogiannis
Department of Agricultural Economics and Rural Development, Agricultural University of Athens, Iera Odos 75, 11855 Athens, Greece

ABSTRACT

In this paper we deal with the dynamic connectivity problem, targeting deterministic worst-case poly-logarithmic time-complexities. First we show that instead of solving the dynamic connectivity problem on a general graph *G*, it suffices to solve it on a graph we name *aligned double-forest* that has only 2*n*-1 edges where *n* is the number of vertices. Then we present an algorithm that achieves all the operations in logarithmic worst-case time on a graph we name *star-tied forest* that consists of a star and a forest (of trees), both defined on the same set of vertices. The star-tied forest which can be seen as a special case of an aligned double-forest is more complicated than a forest on which deterministic worst-case logarithmic time-complexities have already been obtained by means of the Dynamic Trees algorithm, introduced by Sleator and Tarjan (1983). For implementing the operations we build upon Dynamic Trees.

KEYWORDS

Dynamic connectivity, logarithmic, worst-case, Dynamic Trees

1. INTRODUCTION

The dynamic graph connectivity problem, involves the maintenance of a graph of *n* vertices and initially no edges, under the following operations:

- *insert*(*v*, *w*): insert the edge that connects the vertices *v*, *w*.
- *delete*(*v*, *w*): delete the edge that connects the vertices *v*, *w*.
- *connected*(*v*, *w*): return "yes" if there is a path connecting *v* and *w* and "no" otherwise.

Insert and *delete* operations are called *updates* whereas *connected* operations are called *queries*. In the *decremental* version of the problem we cannot insert new edges thus only *delete(v, w)* and the *connected(v, w)* are supported. We adopt an abstract view on the parameters of the above operations. In particular to insert/ delete an edge we only need a pointer to (the memory space occupied by) the edge and we will assume that such a pointer is given.

For the query operation we need two pointers to (the memory space occupied by) the two vertices and we will assume that these two pointers are given.

Many real world applications exist for the dynamic connectivity problem. For example, the vertices of the graph can be users in a social network and the edges can be relationships of some kind. Connected components in such a graph represent user groups with certain characteristics. The vertices could also be computers (Doulamis et al. 2007) or other mobile devices (Ryu et al., 2013) on a network, digital images (Eppstein, 1997), web pages on the Internet or even transistors on a computer chip.

We can distinguish the solutions presented so far, based on whether or not they introduce randomness or amortization. If neither is introduced, we are led to the deterministic worst-case framework where achieving poly-logarithmic time-complexities for all the operations is a long standing open problem. It has long ago been solved though (Sleator and Tarjan, 1983), in the case that the graph is a forest (i.e. a collection of vertex disjoint trees). In this case, each edge is a *bridge*, and thus the deletion of any edge splits a tree into two trees and the insertion of an edge joins two trees into one tree (because we are not allowed to insert an edge that connects two vertices belonging to the same tree). Then, in order to find if any two vertices belong to the same tree, all we need is to reach the root of the tree containing each of the two vertices and check whether the two reached roots are identical or not. In the following of the paper we will refer to this solution as the *Dynamic Trees*. For a general graph, the first non-trivial solution within the deterministic worst-case framework was presented by Frederichson (1983) and supported constant query time and $O(\sqrt{m})$ update time where m is the number of edges. Eppstein, Galil, Italiano and Nissenzweig (Epstein et al, 1997) improved the update time of Frederichson's solution to $O(\sqrt{n})$ where n is the number of vertices. Recently Kejlberg-Rasmussen, Kopelowitz, Pettie and Thorup (Kejlberg-Rasmussen et al, 2015), managed to improve the update time to $O(\sqrt{n}/\log^{1/4}n)$.

Allowing both amortization and randomization, Henzinger and King (1995) obtained $O(\log^3 n)$ amortized expected time for the update operations and $O(\log n \cdot \log\log n)$ worst-case query time. The update time was improved by Henzinger and Thorup (1997) to $O(\log^2 n)$ and further improved by Thorup (2000) to $O(\log n \cdot (\log\log n)^3)$.

Allowing amortization but not randomization, Holm, De Lichtenberg and Thorup (Holm et al, 1998) achieved deterministic $O(\log^2 n)$ for update operations and $O(\log n/\log\log n)$ for query operations. Wulff-Nilsen (2013) improved the update time to $O(\log^2 n/\log\log n)$.

Allowing randomization but not amortization, worst case poly-logarithmic update time-complexities were finally presented by Kapron, King and Mountjoy (Kapron et al, 2013). In particular they achieved $O(\log^4 n)$ per edge insertion, $O(\log^5 n)$ per edge deletion, and $O(\log n/\log\log n)$ per query. Their query algorithm is correct if the answer is "yes" but correct with high probability if the answer is "no". An improvement on the worst-case update time came by Gibb, Kapron, King and Thorn (Gibb et al, 2015) where they improved the time complexity for deletions to $O(\log^4 n)$.

The lower bound of $\Omega(\log n)$ proved by Patrascu and Demaine (2005) implies that amortization and randomization altogether have in fact allowed for near optimal solutions. In the worst-case framework, poly-logarithmic (but far from optimal) time complexities have been accomplished but not deterministically. Achieving deterministic poly-logarithmic worst-case time complexity for all the operations on a general graph has turned out to be a tantalizing open problem.

We now propose another route for the investigation of this open problem. According to this route, deterministically achieving poly-logarithmic worst-case update time on any graph that is more complex than a forest of trees (which is accomplished by Sleator and Tarjan (1983)) is an open problem. The idea is to obtain poly-logarithmic worst-case update time on graphs that are more complex than a forest, ultimately (and hopefully) reaching a general graph. Let us now define the graphs of interest.

Definition 1. Let $G=(V, E)$ be a graph where V is the set of vertices and E is the set of edges. Then G is α *star-tied forest* if there exist two disjoint sets E_1, E_2 of edges such that $E = E_1 \cup E_2$ and $G_1 = (V, E_1)$ is a star whereas $G_2 = (V, E_2)$ is a forest.

Definition 2. Let $G=(V, E)$ be a graph where V is the set of vertices and E is the set of edges. Then G is called *double-forest* if there exist two disjoint sets E_1, E_2 of edges such that $E = E_1 \cup E_2$ and $G_1 = (V, E_1)$ is a forest and $G_2 = (V, E_2)$ is also a forest.

Definition 3. Let $G=(V, E)$ be a graph where V is the set of vertices and E is the set of edges. Then G is called *aligned double-forest* if it is a double-forest and in advance, any two vertices that belong to the same tree in G_2, belong to the same tree in G_1.

In this paper we are going to prove that solving the dynamic connectivity problem on a general graph is equivalent to solving it on an aligned double-forest. In advance we will perform a step to this direction by achieving worst-case logarithmic time-complexity for all the operations of the dynamic connectivity problem on a star-tied forest. It must be noted that the best deterministic worst-case update time in the literature for a star-tied forest is due to Kejlberg-Rasmussen, Kopelowitz, Pettie, and Thorup (Kejlberg-Rasmussen et al, 2015) and it is not poly-logarithmic.

The outline of the paper is the following: In Section 2 we briefly describe the operations of the dynamic connectivity problem on an aligned double-forest. In Section 3 we describe how we can transform a general graph into an aligned double-forest. This transformation includes two steps. First (Subsection 3.1) we transform a general graph into a graph with maximum degree 3 and then (Subsection 3.2) we transform a graph with maximum degree 3 into two aligned double-forests. Finally, (Subsection 3.3) we present the operations of the dynamic connectivity problem on a graph with maximum degree 3, in terms of the corresponding operations on an aligned double-forest. In Section 4 we present a solution for the decremental version on a *star-tied forest* which is a special version of the aligned double-forest. We begin by presenting the data structures used (Subsection 4.1.1) and we proceed (Subsections 4.1.2 and 4.1.3) by showing how to recognize bridges i.e. edges that when deleted, the graph splits. Knowing when an edge is a bridge we proceed to the description of the complete algorithm for the decremental case (Subsection 4.1.4) and finally to its time-complexity and details of implementation (Subsection 4.1.5). The fully dynamic version is addressed in Section 5 and the conclusions follow in Section 6.

2. DYNAMIC CONNECTIVITY ON AN ALIGNED DOUBLE-FOREST

In this subsection we will assume that G_1 and G_2 are maintained as described in the Dynamic Trees algorithm (Sleator and Tarjan, 1983) and we will present the operations of the dynamic connectivity problem on an aligned double-forest, using the corresponding operations in the Dynamic Trees algorithm. In the following, the operations *insert*, *delete* and *connected* as

described in the Dynamic Trees Algorithm are named *DTinsert*, *DTdelete* and *DTconnected* respectively. The Dynamic Trees structure for G_1 and G_2 is named DTG_1 and DTG_2 respectively.

The definition of the aligned double-forest implies that the deletion of an edge from G_1 may create a graph that is not an aligned double-forest. Figure 1 further explains this fact. The black edges belong to G_1 whereas the red edges belong to G_2. It is easy to conclude that Part A shows an aligned double forest since G_1 consists of one tree and G_2 consists of a forest of trees. As a result, any two vertices that belong to the same tree in G_2, belong to the same (and only) tree in G_1. Part B shows the same graph after deleting edge (u, w). Parts C and D show G_1 and G_2 after the deletion. One can observe that after the deletion, vertices u and w belong to different trees in G_1 but to the same tree in G_2 and this does not comply with the definition of the aligned double-forest. For the graph to become an aligned double-forest again, we must transfer (w, v) or (u, z) from G_2 to G_1 (changing the color of the edge from red to black).

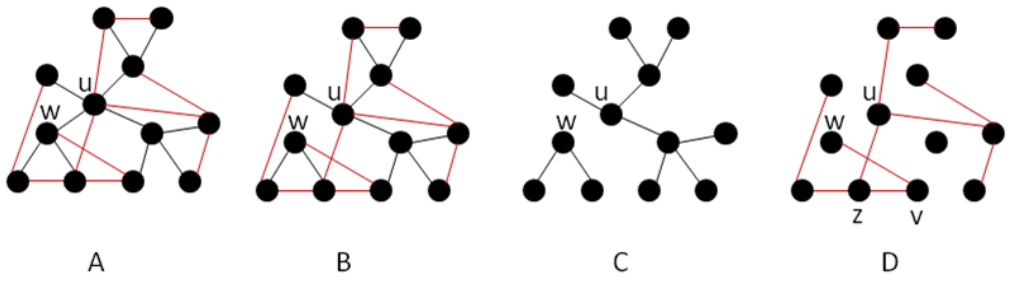

Figure 1. Edge deletions may result to a graph that is not an aligned double-forest

To insert an edge (u, w) into an aligned double-forest G we assume for simplicity that (u, w) does not exist in G prior to the insertion. To delete an edge (u, w) from an aligned double-forest G we assume for simplicity that (u, w) exists in G prior to the deletion. The result of an insertion is and integer which is either implying the forest (G_1 or G_2) in which the edge was inserted, or implying that the edge was not inserted because the insertion would result in a graph that is not an aligned double forest. The result of a deletion is a pair $\{x, p\}$ where x implies the forest from which the edge was deleted and p is a pointer. If x=1 (i.e. the edge was deleted from G_1) then p points to the edge (if such an edge exists) that was deleted from G_2 and inserted in G_1 in order to re-establish the fact that G is an aligned double-forest. The description of the operations of the dynamic connectivity problem on an aligned double-forest follows:

- *AlignedDFconnected*(u, w): Execute *DTconnected*(u, w) in DTG_1 **(**i.e.**,** find the roots of the trees containing u**,** w in G_1 and if the two roots are identical, return "yes" otherwise return "no").
- *AlignedDFinsert*(u, w): Execute *DTconnected*(u, w) in DTG_1. If the result is "no" then execute *DTinsert*(u, w) in DTG_1 and return 1. Otherwise Execute *DTconnected*(u, w) in DTG_2 and if the result is "no" execute *DTinsert*(u, w) in DTG_2 and return 2 otherwise return -1 (returning -1 means that the edge (u, w) cannot be inserted in G because G will not then be an aligned double-forest).
- *AlignedDFdelete*(u, w): If (u, w) belongs to G_2, delete it (by executing *DTdelete*(u, w) in DTG_2) and return $\{2, \text{Null}\}$. Otherwise (i.e. if (u, w) belongs to G_1), delete it from G_1 (by executing *DTdelete*(u, w) in DTG_1) and search in G_2 to find an edge that will reconnect the two newly created trees in G_1. If such an edge is found, let (v, z) be this edge. Execute *AlignedDFdelete*(v, z) in DTG_2 (that is, delete it from G_2) and then execute

AlignedDFinsert(v, z) in DTG_1 (that is, insert it in G_1) and return {1, p} where p is a pointer to (v, z). Otherwise (i.e. if no such edge exists) return {1, Null}.

It is trivial to see that if the graph is an aligned double-forest before an *AlignedDFinsert* or *AlignedDFdelete* operation, it remains an aligned double-forest after the operation. It is also trivial to determine the correctness of the *connected* operation. In all the operations except from *AlignedDFdelete*, the time complexity is ruled by *DTinsert* or *DTconnected* therefore the time complexity is O(log$n$) in the worst-case. In *AlignedDFdelete* the time complexity is ruled by the time needed to find an edge in G_2 that can reconnect the two newly created trees in G_1 because no deterministic worst-case (poly)logarithmic solution exists for this task. As a result, achieving worst-case (poly)logarithmic update operations on an aligned double-forest is still an open problem.

3. FROM A GENERAL GRAPH TO AN ALIGNED DOUBLE-FOREST

3.1 Reducing the Maximum Degree of a General Graph to 3

We are going to use an idea given in (Frederickson, 1983) according to which we can pretend that every vertex in a general graph G has maximum degree 3. The idea is simple but we will cover every detail in this subsection in order for our transformation from a general graph to an aligned double-forest to be clear.

Any vertex u with degree d can be replaced by d vertices connected in a list (which we call adjacenty-list of u) where each vertex (or alternatively, node) of this list is also connected to one of the vertices connected to u in the initial graph. As a result, each node of the list has degree at most 3. This way the number of vertices of G becomes O(m) where m is the number of edges. We are going to use this idea on a general graph. To distinguish the initial graph from the one that results from the above described transformation we use tones that is if our original graph is G, the resulting graph with maximum degree 3 is G'. We distinguish the vertices of G' into *black* vertices which are the vertices of G and *red* vertices which are the vertices added to G for achieving maximum degree 3. This is graphically depicted in Figure 2, the left side of which shows the initial graph whereas the right side shows the resulting graph with maximum degree 3, with the additional red (shaded) nodes. Each black vertex is connected through a pointer (dashed arrows) to the last node of its adjacency-list.

The above described *degree-reduction transformation* alters the problem definition given in the Introduction. In particular, we initially assumed that we start with a graph of n vertices and no edges and during the operations the number of vertices remains unchanged. We now call this version of the dynamic connectivity problem *vertex-static* version. We can still follow the vertex-static version by assuming that initially we have inserted all the possible edges and made them *inactive* thus an edge insertion/deletion is actually an edge activation/inactivation. Obviously this costs $O(n^2)$ space. To keep the space linear to the number of edges, we have to include operations for inserting/deleting red vertices and we call this version, *vertex-dynamic*.

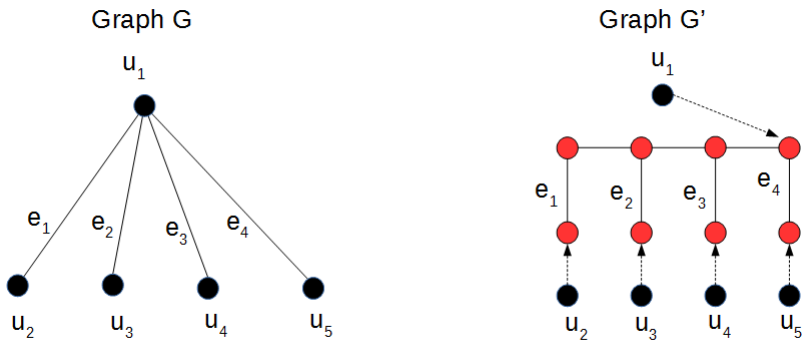

Figure 2. Reducing the maximum degree of any node to 3

Each time a new edge (u, w) is inserted into a general graph G according to the vertex-dynamic version, we insert one red node in the adjacency-lists of u and w (thus we delete at most two edges, and then we insert two new vertices and at most four edges into G'). Then we insert an edge that connects these two new red vertices. Each time an edge (u, w) is deleted from a general graph G according to the vertex-dynamic version, we first find the red node in the adjacency-list of u connected to a node in the adjacency-list of w. Then we delete the edge connecting these two red nodes and finally we delete the red nodes from the adjacency-list they belong to. As a result, the deletion of an edge in G may lead to at most five edge deletions, two edge insertions (to reconnect the two adjacency-lists) and two vertex deletions in G′.

Given a deletion operation in G that has an edge e as a parameter, we need to find the edge e' of G′ that *corresponds* to e. For example, in Figure 2, each named edge of G′ corresponds to the edge of G having the same name. In the vertex-static version, given an edge in G we can find its corresponding edge in G′ in constant time through a table of n rows and n columns (the details are omitted). In the vertex-dynamic version this task is not difficult either: Since the black vertices are known in advance and cannot be deleted, we can give each black node a number (between 0 and n-1) and sort the red nodes in each adjacency-list according to the indirectly pointed black nodes. Then we can reach the red node indirectly pointing to a given black node in logarithmic worst-case time by using a balanced search tree on top of each adjacency-list.

We conclude that by adopting the above described transformation we introduce:

- an extra overhead of a constant number of edge insertions/deletions per update in the vertex-static version.
- an additional logarithmic cost for accessing red edges through black edges and an extra overhead of a constant number of edge/vertex insertions/deletions per update in the vertex-dynamic version.

3.2 From a Graph of Maximum Degree 3 to 2 Aligned Double-Forests

Let G′ be a graph of maximum degree 3 and let us assume that all edges of G' are stored in a list L. We will represent G' through at most 3 forests F_1, F_2 and F_3 by executing the following algorithm called *step*, the i-th execution of which creates forest F_i.

Algorithm Step: for each edge e in L we check if the two end-points of e belong to the same tree in F_i. If they do we do nothing otherwise we delete e from L and insert it in F_i.

It is clear that no more than 3 forests are created, because the maximum degree in G' is equal to 3, and at least one edge per vertex is inserted in each forest per step i.e. after 3 executions of *step*, list L is empty. Also it is trivial to see that algorithm *step* ensures that the following invariant holds:

Invariant 1. If any two vertices belong to the same tree in F_i, they belong to the same tree in all F_j where $j < i$.

Theorem 1. Let $G'=(V, E)$ be a graph of maximum degree 3. Let E_i be the set of edges in forest F_i created by algorithm *step* described above. Then the graph $S_i = (V, E_i \cup E_{i+1})$ $(1 \leq i \leq 3)$ is an aligned double-forest.

Proof. The proof is an immediate consequence of Invariant 1.

3.3 Dynamic Connectivity on a Graph with Maximum Degree 3

In this subsection we present the dynamic connectivity operations on a graph G' of maximum degree 3 (created by the actions of Subsection 3.1 on a general graph G), given that G' is stored in the form of 2 aligned double-forests S_1 and S_2 (as described in the previous subsection). For simplicity we assume that all possible red nodes have been inserted in advance that is, we adopt the vertex-static version. Note that the parameters of the following operations are edges that connect red nodes, i.e. they do not belong to G.

- *MaxD3insert*(u, w). Execute *AlignedDFconnected*(u, w) on S_1. If the result is "no" then execute *AlignedDFinsert*(u, w) on S_1 (observe that in this case, the edge is inserted in forest F_1). Otherwise execute *AlignedDFinsert*(u, w) on S_2. If the result of this operation is 1, then execute *AlignedDFinsert*(u, w) on S_1 (observe that in this case, the edge is inserted in F_2).
- *MaxD3delete*(u, w). Set $\{x, p\}$ = *AlignedDFdelete* (u, w) on S_1. If x=2 (observe that in this case the deleted edge belonged to F_3) return. Otherwise, if x=1 then in case that p=Null return, and if this is not the case, let (z, f) be the edge pointed by p and set $\{y,q\}$=*AlignedDFdelete*(z, f) on S_2. If q = Null return, otherwise let (c, b) be the edge pointed by q (this is the edge that was moved from F_3 to F_2 because of the deletion of (z, f) from F_2). Execute *AlignedDFinsert*(c, b) on S_1 and return.
- *MaxD3connected*(u, w). Execute *AlignedDFconnected*(u, w) on S_1.

It is trivial to see that both S_1 and S_2 continue to be aligned double-forests after a *MaxD3insert* or a *MaxD3delete* operation if they were aligned double-forests before the operation. This fact guarantees the correctness of the *connected* operation.

Theorem 2. Let T_{ADFI}, T_{ADFD} and T_{ADFC} be the deterministic worst-case time-complexities for *AlignedDFinsert*, *AlignedDFdelete* and *AlignedDFconnected* respectively. Then, there exists a deterministic worst-case solution for the static version of the dynamic connectivity problem on a general graph with the following time-complexities:

- Time complexity for the insert operation: $O(T_{ADFI})$
- Time complexity for the delete operation: $O(T_{ADFD})+O(T_{ADFI})$
- Time complexity for the connected operation: T_{ADFC}

Proof. The proof is an immediate consequence of Sections 2 and 3. Each update operation on a general graph results to a constant number of update operations on a graph of maximum degree 3 and each such operation is performed through a constant number of operations on two

aligned double-forests. The connected operation is achieved through a connected operation operation on an aligned double-forest.

Theorem 2 means that in the vertex-static version, if one manages to achieve poly-logarithmic worst-case time complexities for the update operations on an aligned double-forest, one has then achieved poly-logarithmic worst-case time complexities for the update operations on a general graph. As already mentioned, the vertex-static version is based on the premise that we use $O(n^2)$ space.

By adopting the vertex-dynamic version that implies linear (to the number of edges) space, the additional constant number of vertex insertions/deletions per update operation introduces an extra but still logarithmic overhead, since we can insert/delete a red vertex of degree 0 in time proportional to the time needed to access it, and we can access a red vertex in logarithmic time.

4. DYNAMIC CONNECTIVITY ON A STAR-TIED FOREST

4.1 The Decremental Case

4.1.1 The Data Structure

Let us assume that we have a star-tied forest G with n vertices (alternatively, nodes) $u_0, u_1, \ldots, u_{n-1}$. Let u_0 be the *center* of F_1 and let us call all the other nodes of F_1, *leaves* of F_1. No edge exists in both F_1 and F_2 which means that all edges of F_2 connect leaves of F_1 (alternatively, u_0 does not belong to F_2). Figure 3 gives an example of our graph. The black (straight) edges belong to F_1 and the red (curved) ones belong to F_2. We are going to devise deterministic algorithms that achieve logarithmic worst-case time complexities for the following operations:

- delete(v, w) which deletes the edge of the graph connecting vertices v and w
- connected(v, w) which returns "yes" if v and w belong to the same connected component and "no" otherwise.

We assume that our initial graph is *full* that is, F_2 is a tree containing all the vertices of the graph except u_0 (which means that F_2 is full) and for each u_i ($1 \leq i \leq n$-1) the edge (u_0, u_i) exists in the graph (which means that F_1 is full).

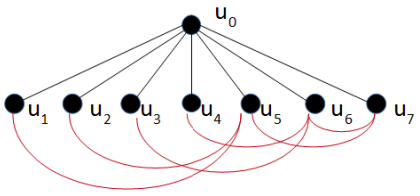

Figure 3. An example of a star-tied forest

Each edge e of F_2 leaves a *mark* on its two end-points that is, (u_i, u_j) leaves a mark in u_i and u_j. We are only interested in the number of edges in F_2 that have marked each leaf of F_1 thus we associate a counter with each leaf of F_1 and when an edge of F_2 marks a leaf of F_1, it increases its counter by 1. For example, in Figure 3 the counter of u_7 is equal to 2 because of (u_6, u_7) and (u_5, u_7). Also, the counter of u_5 is equal to 3 because of (u_5, u_7), (u_5, u_1) and (u_5, u_2). In each

node, we maintain a list that contains the edges (in F_2) attached to this node. We name this list *neighbor-list*.

To detect if a deletion creates a new connected component, we use F_1. As long as F_1 is full, no new connected component is created in our graph. Obviously deleting an edge e_1 of F_1 splits F_1 into two connected components (the leaf is separated from the star center). Let us assume that we are able to find an edge e_2 in F_2 that if transferred to F_1, it will reunite F_1 into one connected component. The problem is that by transferring an edge from F_2 to F_1 we will cancel the fact that F_1 is a star. To solve this problem we *relocate* e_2 which means that e_2 continues to exist "disguised" as e_1. For example, we may relocate (u_3, u_6) because of the deletion of (u_3, u_0). Then (u_3, u_6) takes the place of (u_3, u_0). Thus, the edges of F_1 may be original edges of F_1, which we now call, *Type 1 edges*, or relocated edges of F_2, which we now call, *Type 2 edges*. We use the same notation for the leaves of F_1. Thus, a leaf of F_1 connected to u_0 through a Type 1 edge is called, *Type 1 node*. In the same manner, a leaf of F_1 connected to u_0 through a Type 2 edge is called, *Type 2 node*.

Each connected component of G may be *strong*, if it contains u_0 or *weak* otherwise (obviously there is only one strong component). Each deletion in the strong connected component may create a weak connected component which is separated from the strong one. Every deletion in a weak connected component definitely splits the weak connected component into two weak connected components because a weak connected component contains only edges of F_2 that is, it is a tree.

4.1.2 An algorithm for Performing "Safe" Deletions

We call a deletion *safe* if we can guarantee that no new connected component is created after the deletion. Otherwise, we call the deletion *non-safe*. Let us now devise an algorithm that allows us to delete edges and ends at the point where a non-safe deletion occurs. A trivial solution is to do nothing (i.e., to have no algorithm), assuming that all deletions are non-safe. Our algorithm is a step forward compared to the trivial solution, and incorporates our basic idea for solving the problem. Before we start deleting edges, all the edges of F_1 are of Type 1 and our graph is full. Let us assume that we delete an edge e of G. We distinguish the following cases:

1. Edge e is a Type 1 edge of F_1. Let us assume that e is (u_i, u_0). If the counter of u_i is equal to 0, then the algorithm ends. Otherwise let (u_i, u_j) be the first edge in the neighbor-list of u_i. Edge (u_i, u_j) becomes a Type 2 edge (alternatively, node u_i becomes a Type 2 node). We decrease by one the counters of u_i and u_j. We delete (u_i, u_j) from the neighbor-list of u_i and from the neighbor-list of u_j. Figure 4 depicts this case and it seems as if we deleted (u_i, u_j) instead of (u_i, u_0). An intuitive view of what happened is that (u_i, u_0) was deleted and (u_i, u_j) has taken its place. Said otherwise, (u_i, u_j) now exists "disguised" as (u_i, u_0).
2. Edge e belongs to F_2 and it has not been relocated. We delete e from the two neighbor-lists that contain it, and we also delete its marks (we reduce two counters by one).
3. Edge e is a Type 2 edge of F_1. We execute case 1 assuming that e belongs to F_1. That is, if e has been relocated as (u_i, u_0) we execute case 1 and delete (u_i, u_0).

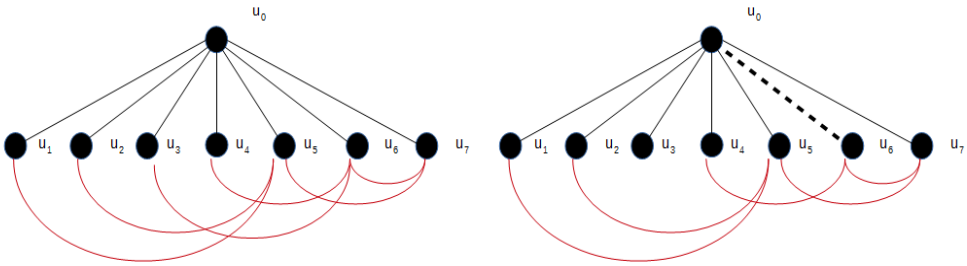

Figure 4. The right part visualizes the graph of the left part after deleting (u_0, u_6). Edge (u_0, u_6) was replaced by (u_3, u_6) which becomes a Type 2 edge of F_1. Edge (u_3, u_6) is now "disguised" as (u_0, u_6)

Theorem 3. When the above algorithm terminates, our graph remains connected i.e. only one connected component exists.

Proof: Let us assume that Theorem 3 does not hold. When the algorithm terminates, all the initial leaves of F_1 continue to be connected to u_0 through Type 1 or Type 2 edges. Let C_1 and C_2 be the strong and the weak connected component respectively. Let S be the set of vertices that belong to C_2. Let $K = |S|$. Clearly S contains Type 2 nodes only, because otherwise at least one vertex of C_2 would belong to C_1. This means that C_2 contains K vertices and at least K edges (the Type 2 edges connecting the vertices of G_2 to u_0). Since F_2 was initially a tree, these K vertices and K edges should then form a forest in C_2 which is not possible because this forest has too many edges i.e. there is a cycle in C_2 and this is not true by assumption.

We have determined that according to our algorithm, all the deletions are safe up to the point where we delete an edge of F_1 such that the involved leaf of F_1 has a zero counter.

4.1.3 Deleting an Edge having a Zero Counter: Observations

Let us now assume that we have deleted an edge (u_0, u_i) (either of Type 1 or of Type 2), and the counter of u_i is equal to 0. Clearly, if no edge is attached on u_i after the deletion, we conclude that u_i now forms a singleton connected component. Otherwise the possibilities grow. Figure 5 shows the possible cases.

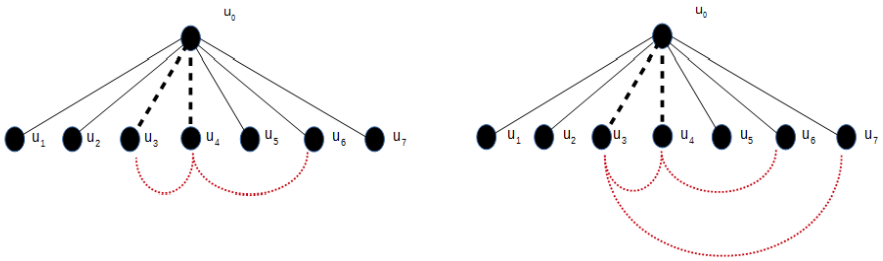

Figure 5. Deleting (u_0, u_6) in the left part creates a new connected component that contains u_3, u_4 and u_6. In the right part, the deletion of u_0, u_6 does not create a new connected component

In the left part of Figure 5, (u_3, u_4) and (u_4, u_6) have been relocated as (u_0, u_3) and (u_0, u_4) respectively. The counter of (u_0, u_6) is equal to 0. If (u_0, u_6) is deleted, it becomes obvious that u_3, u_4 and u_6 form a connected component. However, the right part of Figure 5 shows a slightly different scenario. In particular, (u_0, u_6) still has a zero counter, however after deleting (u_0, u_6), no new connected component is created. This is because of the existence of (u_3, u_7), which now connects the subgraph consisted of u_3, u_4, u_6 and the edges between them, to the rest of the graph (through (u_7, u_0)). Our next task is to enhance our algorithm so that we can without any doubt decide what actually happens after deleting a Type 1 or Type 2 edge of F_1, where the counter of the involved leaf of F_1 is equal to 0.

4.1.4 The Final Algorithm for the Decremental Case

In the final algorithm for the decremental case we are going replace the "relocated" status, by the "Type 2" status (because the "relocated" status was introduced in a more intuitive sense). In particular, let u_i be a vertex of the graph, other than the star center (which is u_0). If u_i is not a root in F_2, let u_j be the father of u_i. If (u_i, u_0) is deleted, then edge (u_i, u_j) and node u_i are of *Type 2*. Otherwise (i.e. if (u_i, u_0) is not deleted), edge (u_i, u_j) and node u_i are of *Type 1*. In comparison to the previous subsections this means that if (u_i, u_0) is deleted, the only F_2 edge we are now allowed to relocate is (u_i, u_j), whereas in the previous subsections we could relocate any F_2 edge adjacent to u_i.

In each node u we maintain a counter which is equal to the number of Type 1 nodes in its subtree. If u_i is a Type 2 root in F_2 and the counter of u_i is positive then u_i becomes a *Type 3 node*, i.e. it becomes a special Type 1 node, connected to u_0 through a virtual edge. Thus the root of a tree in F_2 may be of Type 1 (which is directly connected to u_0), Type 2 (which is not directly connected to u_0 and has a zero counter) or Type 3 (which is not directly connected to u_0 and has a positive counter). Intuitively when a node u_i becomes a root in F_2, its counter determines whether the tree rooted at u_i has been separated from the star center or not (if its counter is equal to 0 then the tree has been separated from the star center).

Let us now assume that we want to delete an edge e of the graph. We distinguish the following cases:

- Edge e is of Type 1 and let (u_0, u_i) be e. Node u_i becomes of Type 2. We distinguish the following subcases:
  - Node u_i is a root in F_2. We decrease the counter of u_i by one. If the counter of u_i is now zero, u_i becomes of Type 2.
  - Node u_i is not a root in F_2. Let u_j be the father of u_i. The edge (u_i, u_j) becomes a Type 2 edge (observe that (u_i, u_j) was of Type 1, because it can become of Type 2 as a result of deleting (u_i, u_0) only). This means that u_i also becomes a Type 2 node. We decrease by 1 the counter of all the ancestors of u_i in F_2 (because their subtree now has one less Type 1 node). Let u_k be the root of the tree that contains u_i. If the counter of u_k is equal to 0 and u_k is of Type 3, u_k becomes of Type 2.
- Edge e is an edge of F_2, and let (u_i, u_j) be e. Let us assume w.l.o.g. that u_j is the father of u_i in F_2. We go to the root of the tree (in F_2) containing u_i. Let u_k be that node. We decrease the counter of all the ancestors of u_i, up to u_k (u_k included) by the counter of u_i (because all the Type 1 nodes included in the subtree of u_i are not now included in the subtree of all the ancestors of u_i). We delete (u_i, u_j). Node u_i is now a root in F_2. If u_k is of Type 3, and its counter is now 0 then u_k becomes of Type 2. If u_i is of Type 2 and its counter is greater than 0, then u_i becomes of Type 3.

Let us now focus on the query algorithm. Assume that we are given two vertices u_i, u_j of G and we want to find out if they belong to the same connected component. Let u_k be the root of the tree in F_2 containing u_i. Let u_v be root of the tree in F_2 containing u_j. If both u_k and u_v are not of Type 2 (which means that they are both belong to the strong connected component) or u_v is identical to u_k (which means that u_i and u_j both belong to the same tree of F_2) we conclude that u_i and u_j belong to the same component, otherwise u_i and u_j belong to different connected components.

The correctness of the query operation is obvious. The time-complexity for all the above operations is discussed in the following subsection.

4.1.5 Implementation and Time-Complexity of the Decremental Algorithm

The decremental algorithm described in the previous subsection includes some tasks that involve O(n) nodes and the time-complexity for performing these tasks rules the overall time-complexity of the operations. In particular, given a node u we need to be able to

- access the root of the tree that contains u (i.e., root(u)) in logarithmic time
- modify in logarithmic time the counters of all ancestors of u in F_2 by subtracting from them the same number.

From a first glance it seems that the time needed to perform the above tasks is proportional to the height of the tree, that is, O(n) in the worst case. Fortunately, according to Dynamic Trees (see Sleator and Tarjan (1983)) both tasks mentioned above can be performed in O(log$n$) time.

According to the Dynamic Trees algorithm, a tree is represented through a number of vertex disjoint paths and each edge is associated with a *cost*. Given a node u of the tree we are able to access root(u) in logarithmic time. We can also subtract the same value from the cost of all edges on a tree path from u to root(u) in logarithmic time. To use this result without having to deal with its technical details, we define the cost of an edge (u_i, u_j) where u_j is the parent of u_i to be the number of Type 1 nodes in the subtree rooted at u_i. In the Dynamic Trees algorithm, the cost of an edge is an inherent property of the edge whereas now it is depended on the subtree defined by the child node of the edge. This poses no problem as long as we never change the root of a tree. Changing the root of a tree may reverse the parent-child relationship between the two end-points of an edge and obviously the cost of the edge (as we defined it) changes. We need to change the root of a tree only when we insert edges into the graph. Since we are now considering the decremental case, we never insert any edges. It is now easy to see that all the operations of our decremental algorithm can be achieved in O(log$n$) worst-case time by using the Dynamic Trees algorithm for representing F_2.

5. INTRODUCING INSERTIONS: THE FULLY DYNAMIC CASE

The fully dynamic algorithm is described at a lower level, using the operations and notation of Dynamic Trees and we assume that the operations and details of Dynamic Trees are familiar to the reader of this subsection.

In Dynamic Trees, each tree edge may be solid or dashed and each tree of F_2 is maintained as a collection of vertex disjoint solid paths connected to each other through dashed edges. In particular, each leaf-to-root path of the tree is composed of O(log$n$) such solid paths and each solid path is kept in the form of a binary tree of logarithmic height. The leaves of such a binary

tree correspond to the vertices of the solid path and the internal nodes of the binary tree correspond to edges of the solid path (see Figure 6). The operation *expose*(*v*) creates a solid path from *v* to root(*v*) (see Figure 6). In the following, by "tree" we mean a tree in F_2 and by "binary tree" we mean the tree that represents a solid path of a tree in F_2.

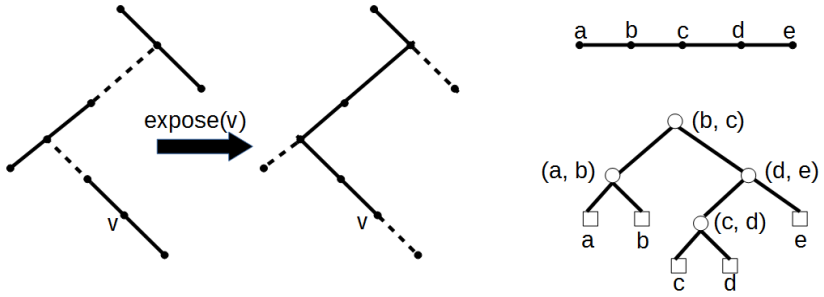

Figure 6. Each edge may be either solid, or dashed. Observe that we have solid paths but no solid subtrees. The operation expose(*v*) creates a solid path that extends from *v* up to root(*v*). Each solid path is represented as a binary tree, as shown in the right part of the figure. Moving to the right, we move towards the root

In Dynamic Trees, the *size* of a vertex *u* is defined to be the number of vertices in the subtree rooted at *u* and the *weight* of *u* is equal to size(*u*) if no solid edge enters *u* or equal to size(*u*) - size(*w*) if the solid edge (*w*, *u*) enters *u*. The weight of each internal node in a binary tree is defined to be the sum of the weights of its children. It is easy to verify that the weight of the root of a binary tree is equal to the size of the subtree rooted at the top-most node of the solid path.

To adjust the Dynamic Trees structure into our case, we define the *volume* of a vertex *u* to be the number of Type 1 nodes in the subtree rooted at *u*. Then we define the *income* of a vertex *u* to be equal to volume(*u*) if no solid edge enters *u* or equal to volume(*u*) - volume(*w*) if the solid edge (*w*, *u*) enters *u*. The income field of each internal binary tree node is defined to be the sum of the income fields of its children. We manipulate income in a way identical to the manipulation of weight i.e. we just insert under each instruction (in the operations of Dynamic Trees) that manipulates weights, a copy of itself and replace weight by income. It then follows that the weight and income fields have identical values under the operations defined in Dynamic Trees.

In order to set the values of the income fields according to our definition of income, we introduce the following two operations into the Dynamic Trees algorithm, one for deleting an edge e from F_1 and one for inserting an edge e into F_1.

- Delete_F_1(*e*). Let (u_i, u_0) be *e*. We execute expose(u_i) in F_2. Node u_i is now the leftmost leaf of a binary tree that represents the solid path from u_i to root(u_i). Starting from u_i we traverse the path to the root of the binary tree decreasing by one the income field.
- Insert_F1(*e*). Let (u_i, u_0) be *e* and let us assume that *e* does not exist. We execute expose(u_i) in F_2. Node u_i is now the leftmost leaf of a binary tree that represents the solid path from u_i to root(u_i). Starting from u_i we traverse the path to the root of the binary tree increasing by one the income field.

In both the above operations it is easy to see (following the logic and details of Dynamic Trees), that the income field of the root of the binary tree having u_i as its leftmost leaf contains the volume of the tree of F_2 that contains u_i (in the same way that by summing the weights we produce the size). Let us now present the operations of the fully dynamic connectivity problem on a star-tied forest:

- Insert(e). We execute Insert_F_1(e) if one of the two end-points of e is u_0. Otherwise we execute the link operation of the Dynamic Trees structure. Observe that this operation needs the cost of the new edge as a parameter, however in our case we do not need edge costs. To solve this without getting into the details of Dynamic Trees, we can simply assume that each edge of F_2 has cost a cost of 1.
- Delete(e). We execute Delete_F_1(e) if one of the two end-points of e is u_0. Otherwise we execute the cut operation of Dynamic Trees.
- Connected (u, v). Let z, s be the roots of the trees that contain u, w respectively. If z is identical to s, we conclude that u and w belong to the same connected component. Otherwise, we execute expose(u). Let k_1 be the income field of the root of the binary tree that contains u, which is equal to the volume of z (i.e. it is equal to the number of Type 1 nodes in the tree rooted at z). Then we execute expose(w). Let k_2 be be the income field of the root of the binary tree that contains w, which is equal to the volume of s (i.e. it is equal to the number of Type 1 nodes in the tree rooted at s). If k_1 is positive and k_2 is positive, we conclude that u and w belong to the same connected component (since they belong to the same connected component with u_0). If, on the other hand, at least one of k_1, k_2 is equal to 0, we conclude that u and w do not belong into the same connected component.

The time complexity of all the above operations is logarithmic in the worst-case since since:

- We use the operations of the Dynamic Trees algorithm.
- The two additional operations we introduce (Insert_F1(e) and Delete_F1(e)) contain tasks achieved in logarithmic worst-case time.
- The additional instructions we introduce to the operations of the Dynamic Trees algorithm (i.e. the ones for manipulating the income field) are identical to the already existing ones that manipulate the weight field.

6. CONCLUSION

We presented an algorithm for achieving all the operations of the dynamic connectivity problem on a graph that consists of a forest and a star defined on the same set of vertices, in worst-case logarithmic time. We based our solution on the Dynamic trees algorithm which was introduced for solving the same problem on a forest of trees. It turns out that although our graph has n-1 additional edges compared to a forest with n vertices, the tools provided by the Dynamic Trees algorithm suffice for achieving all the additional tasks in worst-case logarithmic time. An interesting question that now arises is whether the tools provided by the Dynamic trees algorithm suffice for achieving the same result on an even more complicated graph. Finally we have shown that this “more complicated graph” on which one needs to advance in is not a general graph but an aligned double-forest.

REFERENCES

Doulamis N. D. et al, 2007. Cluster-based proactive replication of multimedia files in peer-to-peer networks, Proceedings of the second International Conference on Digital Information Management, Lyon, France, pp. 368–375.

Eppstein D., 1997. Dynamic connectivity in digital images. Information Processing Letters, Volume 62, Issue 3, 121–126.

Eppstein, D. et al., 1997. Sparsification—a technique for speeding up dynamic graph algorithms. Journal of the ACM (JACM), Volume 44, no.5, pp. 669–696.

Frederickson G. N., 1983. Data structures for on-line updating of minimum spanning trees. Proceedings of the fifteenth annual ACM symposium on Theory of computing (STOC '83). Boston, USA, pp 252-257.

Gibb, D. et al., 2015. Dynamic graph connectivity with improved worst case update time and sublinear space. arXiv preprint arXiv:1509.06464.

Henzinger M. R. and King V. 1995. Randomized dynamic graph algorithms with polylogarithmic time per operation, Proceedings of the 27th Symposium on Theory of Computing, Las Vegas, USA, pp. 519–527.

Henzinger, M. R. and Thorup, M. 1997. Sampling to provide or to bound: With applications to fully dynamic graph algorithms, Random Structures and Algorithms, Volume 11, no 4, pp. 369–379

Holm, J. et al, 1998. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. Proceedings of the thirtieth annual ACM symposium on Theory of computing, Dallas, USA pp. 79-89

Kapron, B. et al., 2013. Dynamic graph connectivity in polylogarithmic worst case time. In Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms. New Orleans, USA, pp. 1131-1142

Kejlberg-Rasmussen et al., 2015. Deterministic worst case dynamic connectivity: Simpler and faster. CoRR, abs/1507.05944.

Patrascu, M. and Demaine, E., 2005. Logarithmic Lower Bounds in the Cell-Probe Model. SIAM Journal on Computing, Vol 35. No 4, pages 932–963.

Ryu S. et al, 2013. Development of device-to-device (D2D) communication based new mobile proximity multimedia service business models. Proceedings of the IEEE International Conference on Multimedia and Expo Workshops, San Jose, USA, pp. 1–6.

Sleator D. D. and Tarjan R. E., 1983. A data structure for dynamic trees. Journal of Computer and System Sciences, Volume 26, Issue 3, pp 362-391

Thorup, M., 2000. Near-optimal fully-dynamic graph connectivity. Proceedings of the thirty-second annual ACM symposium on Theory of computing, Portland, USA, pp. 343-350.

Thorup M., 2000. Near-optimal fully-dynamic graph connectivity. Proceedings of the 32nd annual ACM symposium on Theory of computing (STOC), Portland, USA, pp. 343–350.

Wulff-Nilsen, C., 2013. Faster deterministic fully-dynamic graph connectivity. Proceedings of the twenty-fourth Annual ACM-SIAM Symposium on Discrete Algorithms New Orleans, USA, pp. 1757-1769