Notes on Fully Dynamic Connectivity

Sarah Clapoff

December 12, 2022

1 Introduction

1.1 Previously: Incremental and Decremental Connectivity

Recall the dynamic graph problem, in which we are given a graph *G(V,E)* $G(V, E)$, and are interested in the connectivity of any two nodes *i, j* in G. We are making these connectivity queries on-line, meaning that G may be updated between queries. Previously, we have considered Incrementally and Decrementally Connectivity problems in partially dynamic graphs, in which edges are either added or deleted.

Tarjan [1975] provided a solution to the **Incremental Connectivity** (or `Union-Find`) problem in $O(m \cdot \alpha(m, n))$ time, where α is the inverse Ackermann's function. This was achieved through using union by rank and path compression heuristics on a disjoint-set data structure.[1] Even and Shiloach [1981] provided a solution to the **Decremental Connectivity Problem** using a data structure in which each node is assigned a component label, and the components are updated as edges are deleted. This algorithm answers queries constant time, and the data structure is maintained in $O(m \cdot n)$ giving an overall time of $O(q + m \cdot n)$ for answering q queries. [2]

1.2 Now: Fully Dynamic Connectivity

Now, we will combine the two problems to consider the **Fully Dynamic Connectivity Problem.** Let us formally define the problem:

Problem (Fully Dynamic Connectivity)**.** *Given a fully dynamic graph* $G = (V, E)$*, with a fixed set of nodes* V*,* $|V| = n$*, and* $|E| = m$*,* $m = 0$ *initially. The operations are, for any two nodes* $i,\ j$*:*

Update: Insert Insert a new edge between $i,\ j$*.*
Update: Delete Delete the edge e between $i,\ j$*.*
Query: Are i and j connected in G?

J. Holm et al. [2001] present a solution to this problem that maintains a spanning forest F for G in amortized time $O(\log^2 n)$ per update and $O(\log n / \log\log n)$ per query [3]. We will focus on the high-level description of this solution, and later will focus on the implementation using ET-trees developed by Henzinger and King [1999] [4].

2 The Solution

2.1 Solution Overview

Henzinger and King [1999] presented a randomized algorithm that maintained a spanning forest in $O(\log^3 n)$ amortized time per update, and their 1997 paper presented a technique to adapt a deletions-only data structure to a fully dynamic structure [5]. They approach the problem from the point of view of edge deletions first, then adding edge insertions. The intuition is that, similar to in the Decremental Connectivity problem, we look at every edge incident to the smaller component when looking for a replacement edge. In addition, as we traverse components, we create new levels for those we have visited, so that we don't have to traverse them again in future queries. An augmented spanning forest data structure is created to keep track of this information.

Holm et al. [2001] use a similar intuition to improve on Henzinger and King's solution. In their algorithm, G is represented by a spanning forest, F, and the levels are represented by subforests $F_i,\ F = F_0$. We will call edges in these forests *tree edges.* Insertions and queries are easily implemented using ET-trees (or a similar dynamic forest data structure). If we are wondering about the connectivity of two nodes, we simply need to check if they are in the same tree in F. To insert an edge $(v,\ w)$ we will check if v and w are connected in F; if they are we will add $(v,\ w)$ to the list of non-tree edges, if not we add it to F. Deletions of non-tree edges are also handled easily, the edge is just removed from the list. However, if a tree edge is deleted then it will split a tree in F. Since each tree in F represents a corresponding component in G, if the component is not split then we will have to find a *replacement edge* to repair the split tree. This is where our subforest data structure comes in.

When looking for a replacement edge to connect v and w, we will start searching at the highest possible level in the smaller of the two trees created from the split, T_v and T_w. When visiting an edge, if it does not connect T_v and T_w, then we will increase its level. This increase is what pays for the visitation of that edge. In addition, we have gained more information about the connectivity of G. So, as we *learn* about the connectivity of nodes, we promote their edges so that we don't have to visit them again later. Since we are searching the smaller component first, we know that $|F_{i-1}|$ is never larger than $|F_i|$. In addition, since edge levels are never decreased, we have at most $\lfloor \log_2 n \rfloor$ increases per edge.

So, we will create a data structure to represent G, composed of spanning forests. Edges in these trees are called *tree edges.* The general idea is to maintain levels $i = 0, 1, 2, ..., \ell_{max}$, $\ell_{max} = \lfloor \log_2 n \rfloor$, and for each level maintain a spanning forest F_i. Initially all edges have level 0; we increase the level as we visit edges during a deletion, since its endpoints are close enough to fit in a tree on a higher level. That is, as we *learn* about the connectivity of nodes, we promote their edges so that we don't have to visit them again later. Since we are searching the smaller component first, we know that $|F_{i-1}|$ is never larger than $|F_i|$. In addition, since edge levels are never decreased, we have at most ℓ_{max} increases per edge.

2.2 The Spanning Forest Structure

Let us look at the spanning forest closer. For each edge e in G we assign a level $\ell(e) \leq \ell_{max} = \lfloor \log_2 n \rfloor$. For each level i we have a subforest F_i with edges of level $\geq i$, as well as a subgroup G_i, and a set of nontree edges in G_i, E_i. We have the following invariants:

(i) F_i is contained in F_{i-1}. That is, $F = F_0 \supseteq F_1 \supseteq \cdots \supseteq F_{\ell_{max}}$

(ii) F_0 is a maximum spanning forest for G. So if (v, w) is a nontree edge, then v and w are connected in $F_{\ell(v,w)}$.

(iii) E_i is the set of nontree edges in G_i. That is, E_i contains all nontree edges that have been looked at $i - 1$ times. E_i is disjoint for all i.

(vi) The maximum number of nodes in a tree F_i is $\lfloor n/2^i \rfloor$. The maximum level is thus $\ell_{max} = \lfloor \log_2 n \rfloor$.

Initially, all edges have level 0. That is, G_i =emptyset, for $i > 0$. Note also that since edge levels are never decreased, each edge may have at most ℓ_{max} increases.

2.3 High-Level Solution Description

We will now consider the algorithm for fully dynamic insertions and deletions, as put forward by Holm et al. [2001]. Keep invariants (ii) and (iv) in mind.

`Insert(`e`):`

1: Insert e into G_0.
2: If the endpoints were not connected in F_0, add e to F_0.

`Delete(`e`):`

1: $i \leftarrow \ell(e)$
2: **if** $e \in E_0$ **then**
3: Remove e from E_i.
4: **else**
5: `Replace(`e, i`)` *Delete e and find a replacement edge which reconnects F at the highest possible level.*
6: **end if**

`Replace(`$(v, w), i$`):`
Assuming no replacement edge on level $> i$ (since F is a max spanning forest), finds a replacement edge of the highest level $\leq i$, if it exists.

1: $T_v \leftarrow$ the tree in F_i containing v.
2: $T_w \leftarrow$ the tree in F_i containing w. *Let $|T_v| \leq |T_w|$.
3: **for all** edges of T_v with level i **do**
4: $\ell = i + 1$ *Now T_v is a tree in F_{i+1}.
5: **end for**
Visit all edges level i incident to T_v, until a replacement edge is found or all edges have been considered:
6: **for** edge f in T_v, $\ell(f) = i$, **do**

7: Found = False
8: **if** f does not connect T_v and T_w **then**
9: $\ell(f) \leftarrow i+1$ *This increment pays for the consideration of f.*
10: **else if** $T_{(f,v)} == T_{(f,w)}$ **then**
11: insert f as replacement edge *f connects T_v and T_w, so we have found a replacement edge.*
12: **return**
13: **end if**
14: **end for**
If we have reached this point, then all edges on level i have been considered. So we look on the level above $(i-1)$ for a replacement edge.
15: **if** $i == 0$ **then**
16: There is no replacement edge for e.
17: **return**
18: **else**
19: `Replace`$((v,w), i-1$
20: **end if**

`Query`(v,w):
Answering the question: Are v and w connected?
1: **if** $T_{(v,F_0)} == T_{(w,F_0)}$ **then**
2: **return** True
3: **else**
4: **return** False
5: **end if**

3 Analysis of The Algorithm

Our algorithm is implemented by representing each forest F_i by an ET-tree. For each node, every incident edge on level i will maintain a key representing it's level. Since the trees on lower levels contain those on higher levels (by invariant (i)), an edge with level i will also appear on all F_h, $h \leq i$, that is, on $O(\log n)$ levels. We will also separate the tree- and non-tree edges.

Theorem 1. *Given a graph G with m edges and n nodes, there exists a fully dynamic algorithm that answers connectivity queries in* O(logn/log log n) *time worst case, and handles inserts or deletes in* $O(\log^2 n)$. *[3]*

Proof. The keys to this proof are the invariants of the spanning forest structure, and the efficiencies of ET-Trees. [4]

Answering connectivity queries comes directly as a property of ET-Trees. For the general ET-Trees covered in this course, queries can be answered in $O(\log n)$ time, as we are simply

comparing roots. However, if we apply a $\Theta(\log n)-$ary restriction (see [4]), we can reduce this time to $O(\log n/\log\log n)$.

An edge is initially inserted on level 0, which requires a cost of $O(\log n)$. The maximum number of increases is $\log n$. At each increase, adjusting the ET-Trees and finding the edge costs $O(\log n)$ time. In total, the amortized cost to insert and maintain an edge is $O(\log^2 n)$.

Similarly, deleting an edge takes $O(\log^2 n)$ total amortized time. Deleting the edge itself takes $O(\log n)$ time, and all forests on levels lower than $\ell(e)$ must be cut (costing $O(\log^2 n)$ time). Then, `Replace` (which takes $O(\log n)$ time to execute) is called $O(\log n)$ times. Finally, if a replacement edge is found, we must link $O(\log n)$ forests. Overall, this is a total cost of $O(\log^2 n)$ time.

Thus, we have shown that connectivity queries can be answered in $O(\log n/\log\log n)$ time, and insertions and deletions can be handled in $O(\log^2 n)$ time. [3] □

4 References

References

[1] Tarjan, R.E. Efficiency of a good but not linear set union algorithms. *J. ACM*, 22(2):215–225, April 1975.

[2] Even S. and Shiloach Y. An on-line edge-deletion problem. *J. ACM*, 28(1):1–4, Jan. 1981.

[3] Holm, J. and de Lichtenberg, K. and Thorup, M. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, Jul. 2001.

[4] Henzinger, M.R. and King, V,. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, Jul. 1999.

[5] Henzinger, M.R. and King, V.,. Maintaining minimum spanning trees in dynamic graphs. *ICALP*, 1256:594–604, 1997.