CS 573: Topics in Analysis of Algorithms	Spring 2006

Lecture 9 — February 14

Lecturer: Jeff Erickson *Scribe: David Klempner*

9.1 Introduction

Consider the problem of dynamic graph connectivity. Given a graph G with no edges, we want to be able to add or remove edges to it and then answer whether two vertices in the graph are connected.

In particular, we want to be able to maintain an arbitrary graph G quickly under the following operations:

- INSERT(v, w) – insert the edge vw into G
- DELETE(v, w) – remove the edge vw from G
- CONNECTED(v, w) – return whether v and w are connected in G

In [2], Jacob Holm*, Kristian de Lichtenberg*, and Mikkel Thorup describe a data structure which can do INSERT(u, v) and DELETE(u, v) in $O(\log^2 n)$ amortized time and CONNECTED(u, v) in $O(\log n/\log\log n)$ amortized time.

The idea is to maintain a spanning forest in conjunction with an adjacency list representation of the graph. We maintain this spanning forest F of G in an Euler-tour tree or a rake-compress tree. It is easy to maintain the forest under insertion operations. The difficulty is in deletions; if we delete an edge from the forest, do we need to add another edge to keep the trees connected?[1] We use an auxiliary data structure similar to the Bentley-Saxe* construction in [1]. Every edge has an integer level, level(uv), between 0 and $\lg n$. The level of each edge in G starts at $\lg n$, and occasionally decreases. We then maintain a spanning tree for each level, consisting only of edges at that level or lower.

9.2 Definitions

- level(e) is the integer level of edge e.
- G_i is the subgraph containing edges in G whose level is at most i; $G_{\lg n}$ is the entire graph.
- F_i is the edges in F with level $\leq i$, or equivalently the spanning forest for G_i; F_i is a spanning forest for the entire graph.

[1] Note that, if we did not need to support deletion, we could just use UNION-FIND. It is therefore not surprising that deletion is the difficult operation.

9.3 Invariants

We maintain two invariant properties of this family of trees:

- F_i is a minimum spanning forest, where the weight of an edge is its level; $F_i = MSF(G_i)$.
- Any component of G_i or F_i has at most 2^i vertices.

9.4 Operations

Connectivity and insertion are both easy:

```
CONNECTED(v, w)
    return FINDROOT(v, F) = FINDROOT(w, F)
```

The two FINDROOT(v, F) operations both take amortized $O(\log n)$ time, thus connectivity queries take $O(\log n)$ time. We will make a minor tweak later to improve this cost to $O(\log n / \log\log n)$.

```
INSERT(v, w)
    level(vw) ← lg n
    add v and w to each other's adjacency lists
    if v and w are in different components of F
        then add vw to F
```

INSERT(v, w) involves doing a FINDROOT for each of v and w, then a JOIN if they are not already in the same component. Thus, INSERT immediately has an amortized cost of $O(\log n)$. However, we will later be charging the cost of level changes to INSERT, which will increase the amortized cost to $O(\log^2 n)$.

```
DELETE(v, w)
    remove v from w's adjacency list
    remove w from v's adjacency list.
    if vw ∈ F
    delete vw from every F_i with i ≥ level(vw)
    for i ← level(vw) to lg n
        do T_v ← tree in F_i containing v
           T_w ← tree in F_i containing w
           wlog |T_v| ≤ |T_w|
           for each edge e ∈ T_v such that level(e) = i
               do level(e) ← level(e) − 1
                  Add e to v's tree in level i − 1
           for each edge xy such that x ∈ T_v, xy ∉ T_v, level(xy) = i
               do if y ∈ T_w
                     then add xy to F_i, F_{i+1}, ... , F_{lg n} = F
                     else  level(xy) ← i − 1
```

Deleting an edge while maintaining that F is a spanning tree is the interesting part of this data structure. Doing so is no problem at all if the edge $vw \notin F$. However, if $vw \in F$, we need to delete it from every spanning tree and then find out if it needs to be replaced; v and w may well still be connected in G.

9.5 Correctness of Delete

We remove wv from every level, then we search upwards by level looking for a replacement edge. To maintain the minimum spanning forest invariant, this edge needs to be of minimum level. Any replacement can not be of lower level than level(vw), because of that same invariant. Thus, the idea is to amortize the cost of this search by decreasing the level of an edge whenever we look at it. Since an edge can change level only $\log n$ times and each level change costs at most $\log n$, the cost of level changes is at most $O(\log^2 n)$ per edge inserted. We can thus charge the cost of this search to Insert.

The only correctness issue remaining is to show that we do not break the first invariant when we decrease the level of an edge. There are only two times we actually do so. We decrease all of the edges of T_v on a level when we first start looking at that level. Since $|T_v| \leq |T_w|$ and $|T_v + T_w| \leq 2^i$, $|T_v| \leq 2^{i-1}$, so moving all of T_v to $i-1$ does not break that invariant. Then, when we decrease the edge xy during the search, we do not need to add it to level $i-1$ because there is no way it can connect two components.[2] Since we are not adding it to any trees, it can not break the first invariant.

9.6 Analysis of Delete (and amortization over Insert)

1. We do a FindRoot on vw, which costs $O(\log n)$
2. Deleting vw from every F_i takes $O(\log n)$ Cuts.
3. FindRooting v and w through $O(\log n)$ levels costs $O(\log^2 n)$.
4. Ensuring $|T_v| < |T_w|$ requires only maintaining the size of each tree, which can be done in $O(1)$.
5. Decrementing levels of edges sometimes requires a Join, costing $O(\log n)$ per level decrease.
6. Finding edges in the search costs $O(\log n)$ per edge. All edges but the last have their level decreased, so $O(\log n)$ per level decrease, plus $O(\log n)$ for the replacing edge.
7. Adding xy to $F_i, F_{i+1}, \ldots, F_{\lg n}$ is $O(\log n)$ Joins, costing $O(\log^2 n)$.

Thus, the cost of a Delete is $O(\log^2 n)$ plus $O(\log n)$ per level change. Since the number of level changes is bounded by $\lg n$ per edge, we charge the level changes to Insert, leaving both Insert and Delete at $O(\log^2 n)$.

[2] The technique from lecture does not work. We would need to not only add the edge to $i-1$, we would also need to add it to every higher level, which would be too expensive.

9.7 Improving CONNECTED to $O(\log n/\log\log n)$

The trick to improving CONNECTED from $O(\log n)$ to $O(\log n/\log\log n)$ is to use a b-ary tree rather than a binary tree to store the Euler tour of $F = F_{\lg n}$. CUT and JOIN become $O(b\log_b n)$ and FINDROOT becomes $O(\log_b n)$. Set $b = \Theta(\log n)$. FINDROOT becomes $O(\log n/\log b) = O(\log n/\log\log n)$, CUT and JOIN are $O(b\log n/\log b) = O(\log^2 n/\log\log n)$. Since this only affects the top level forest and INSERT and REMOVE do only $O(1)$ changes to that forest, this cost is dominated by the existing $O(\log^2 n)$ amortized cost. On the other hand, since CONNECTED only touches the top forest, optimizing FINDROOT on that level is worthwhile.

9.8 Concluding remarks

This data structure supports connectivity queries in a fully dynamic graph, supporting both insertions and deletions of edges, with an amortized cost of $O(\log^2 n)$ to change the graph and $O(\log n/\log\log n)$ to test connectivity. It turns out that this performance is optimal in the sense that any algorithm that has better performance on one operation must necessarily have worse performance on the other.

Bibliography

[1] J. L. Bentley and J. B. Saxe. Decomposable searching problems i: Static-to-dynamic transformation. *J. Algorithms*, 1(4):301–358, 1980.

[2] 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*, 48(4):723–760, July 2001.