Design And Analysis of Algorithms
Lecture 4: Dynamic Graph Connectivity.

February 2, 2018

Consider some graph graph $G = (V, E)$. Suppose its edge set is dynamic: our algorithm is given a sequence of updates to the edge set:

- `insert`(u, v)
- `delete`(u, v)

which add or delete edges, respectively. This is referred to as the "edge arrival model". Both of these operations are called *updates*. We want a data structure that answers the questions, or *queries*, about connectivity in G between various sequences of updates, of the form

- `connected` $(u, v) = T$ if u, v are in the same connected component of G
- `connected` $(G) = T$ if G has exactly one connected component

As you might imagine, there will be a trade off in terms of how expensive the updates verses queries to the datastructures we construct. In general, we cann optimize entirely for either, which yield the following two baseline approaches:

1. Track updates and all edges with an edge adjacency list, answer queries with a DFS. This takes $O(1)$ update, and $O(m + n)$ for queries
2. Keep an $n \times n$ matrix tracking pairwise connectivity. This has $O(1)$ query complexity, but the deletion or addition of an edge might change $\Omega(n^2)$ entries in the matrix.

In general, there's a tradeoff between the runtime of updates and queries. A brief history of the study of them is as follows:

Table 1: Dynamic Graph Connectivity

	T_U =Update complexity	T_Q =Query complexity	Notes
Frederickson [1985]	$O(\sqrt{m})$	$O(1)$	Deterministic
Eppstein et al. [1997]	$O(\sqrt{n})$	$O(1)$	Sparsification
Holm et al. [2001]	$O(log^2 n)$	$O\left(\frac{\log n}{\log \log n}\right)$	Amortized
Kapron et al. [2013]	$O(log^5 n0$	$O\left(\frac{\log n}{\log \log n}\right)$	Randomized, Worst-case

As far as lower bounds are concerned, Ptraşcu and Demaine [2004] PD showed for update time T_U, query-time T_Q, both

- $T_U \cdot \log \frac{T_Q}{T_U} \geq \log n$
- $T_Q \cdot \log \frac{T_U}{T_Q} \geq \log n$.

1 Fredrickson's algorithm [Frederickson, 1985]

In order to achieve $o(m)$ update time with constant query complexity, Fredrickson's algorithm uses 2 key ideas. First, assume that max-degree$(G) \leq 3$; otherwise, for a degree d vertex v, we can replace each vertex v by $v_1, \ldots, v_d$ which we connect in a cycle and each has a unique neighbor. This might increase the number of vertices from n to n^2, but it will only increase the number of edges by a constant factor. Second, the algorithm maintains a "clustered" spanning forest F of G.

Definition 1 (z-clustering of T)**.** *Given a tree $T = (V, E)$, we will call $C_1^T, \ldots, C_k^T$ a z-clustering of T if*

- *It is a partition of V*
- *Each C_i^T is connected*
- *For each i, $3z \geq |C_i^T| \geq z$.*

A clustering of this form will guarantee that no cluster is too large (and because they're not too small, there won't be too many of them). The next lemma can be operationalized to form an algorithm for constructing such a clustering of a tree.

Lemma 1. *Given any $T = (V, E)$, with $|V| \geq z$ and parameter z, it is possible to z-cluster T in time $O(z)$.*

Proof. We will prove this by induction on $|V|$. Our inductive hypothesis is that it is possible to z-cluster any tree of size at least z and at most k, where k is the parameter upon which we will do induction.

Our base cases will be all trees of size between z and $3z$; any one of those trees can be placed into its own cluster.

Now, consider $|V| > 3z$, and orient each edge $(a, b) \in T$ to point towards a if $|T_a > |T_b|$ and towards b otherwise. Note that this directed tree must have a sink v_0 (which we assume has degree at most 3). We consider the case where $\deg(v_0) = 3$ as the others are simplifications of this case.

Name the subtrees of v_0 A, B, and C, with sizes s_A, s_B, and s_C respectively. Without loss, assume $s_C \geq \max(s_A, s_B)$. Since $a \to v_0, b \to v_0, c \to v_0$, and the edges are oriented towards the larger of the two subtrees, any two of the subtrees plus v_0 have to have size greater than or equal to the size of the third subtree; e.g. $s_A + s_B + 1 \geq s_C \geq z$, where the second inequality follows from the fact that we assume $|T| > 3z$ and s_C is the largest subtree of v_0. So, removing the edge (v_0, c) would yield two trees of size strictly less than $|T|$ but of size at least z, so we can apply our inductive hypothesis on each one separately to cluster them.

Since each edge is removed at most once out of all the recursive calls, there is at most $O(m)$ work done by this recursion. □

So, our algorithm for generating a new datastructure for a graph G, deleting or adding an edge to G, or answering a connectivity query will be as follows.

```
Maintain a spanning forest F of G;
For each T ∈ F, if |T| ≥ z then
  | maintain C_1^T, ..., C_k^T;
end
else
  | Place T in a single cluster;
end
Each C_i^T will track edges to and in C_i^T in A(T);
```

We now describe an analysis of the runtime of the algorithm. `Connected`

Algorithm 1: `delete`(u,v)

```
if u ∈ T, v ∈ T' for T ≠ T' then
    do nothing, return
end
else
    if u, v ∈ T, with u ∈ C_u^T, v ∈ C_v^T then
        if C_u^T = C_v^T then
            Update C_u^T edge set;
            if DFS of C_u^T shows disconnected then
                break C_u^T into two clusters;
            end
        end
        else
            Remove edge from A(T) over clusters;
        end
        if DFS on cluster graph shows T is disconnected then
            Recluster the two components of T into T_1, T_2
        end
        if Some cluster was broken into two then
            recluster relevant trees;
        end
    end
end
```

Algorithm 2: `insert`(u,v)

If $u, v \in T$ then add (u, v) to $A(T)$;
If $u \in T, v \in T'$ then merge T, T' and recluster the union of u and v's cluster;

Algorithm 3: `Connected`(u,v)

If $u, v \in T$ for some $T \in F$, return true;
Otherwise return false ;

`Connected` is constant time by tracking with pointers the name of the tree each vertex belongs to in F; `insert` takes $O(z)$ time (one needs to update $A(T), A(T')$ in a constant number of locations and possibly recluster the union of the two clusters, which might take $O(z)$ time). For `delete`(u, v), one must do a DFS to determine whether the cluster(s) or tree they belong to. A given cluster has $O(z)$ vertices and $O(z)$ edges (since each vertex has degree at most 3); therefore this DFS will take $O(z)$ time. If there are k clusters in T, there are $O(k^2)$ edges between clusters in $A(T)$, so DFS on this will take $O(k^2)$ time. If there is more than a single cluster then each cluster has $\Omega(z)$ vertices, there are $O(n/z) = O(m/z)$ clusters, thus the DFSes will take at most $O(z + \left(\frac{m}{z}\right)^2)$ time. Setting $z = m^{2/3}$ gives us $O(m^{2/3})$ update complexity.

2 Randomization for Dynamic Connectivity in $O(\textbf{poly}\log n)$ time [Kapron et al., 2013]

We will now develop a randomized algorithm for dynamic connectivity with $O(\text{poly}\log n)$ update and query time. This will come at some cost: namely, that the answer the algorithm will give to a query will sometimes

be wrong. In particular, the type of mistakes the algorithm might make are called "one-sided", in that when the algorithm says "Connected", that is always correct, but when the algorithm says "disconnected", that is correct with probability $1 - \frac{1}{n^c}$ for some constant c. The algorithm works similarly to the previous one, in that it keeps track of a spanning forest and adds a fancy idea for replacing an edge on delete if one exists.

We simplify to the case where there is a sequence of inserts followed by a single deleted, with interleaved queries. This is not as general as allowing multiple deletes, but is a simplification that makes the algorithm substantially simpler. The idea is to keep track of *signatures* for each vertex, which then extend to signatures on edges and sets of vertices that have a nice operational meaning.

- For each vertex v, let $\ell(v)$ be the label of v, a unique $O(\log n)$-length binary string for v.
- Similarly define for each edge $e = (u, v) = \langle \ell(u), \ell(v) \rangle$ the lexicographically ordered concatenation of the labels of its endpoints.
- Then, define the signature of a vertex $\text{sign}(v) = \oplus_{e=(u,v)\in E} \ell(e)$ to be the XOR of the edges adjacent to v.
- For a set of vertices $V' \subseteq V$, define $\text{sign}(V') = \oplus_{v\in V'} \text{sign}(v)$ and notice that $\text{sign}(V') = \oplus_{e=(u,v):u\in V', v\notin V'} \ell(e)$.

Note that it is possible to compute $\text{sign}(u)$ in $O(\log n)$ time using link-cut trees or other heaplike dynamic structures. Now, given a `delete`$(e = (u, v))$ command, with T_u, T_v the respective subtrees on either side of e. Consider the values of $\text{sign}_{G\setminus e}(T_u), \text{sign}_{G\setminus e}(T_v)$. Note that:

1. If exactly 0 edges in $G \setminus e$ go between T_u and T_v, then $\text{sign}(T_u) = \text{sign}(T_v) = 0$.
2. If exactly 1 edge f goes between the two, $\text{sign}(T_u) = \text{sign}(T_v) = f$.
3. If multiple edges cross that cut, there could be many values of these computations.

So, how should we use this to find a replacement edge f when they may not be unique for a graph $G \setminus e$? The answer is by sampling edges such that we're likely to have only one edge crossing any cut (or zero if there weren't any to begin with). We'll construct $O(\log n)$ subgraphs of G, where G_j includes each edge $e \in E$ independently with probability $\frac{1}{2^j}$ for each $j \in [0, \log n]$. Then, we can compute $\text{sign}_{G_j}(V')$ only over those edges in G_j. If there are roughly 2^j many edges across a cut in $G \setminus e$, then there is constant probability that exactly one edge survives this sampling into G_j. So, repeating this process $O(\log n)$ times shows this will succeed with probability $1 - \frac{1}{n^c}$ in at least one of them. In total then there will be $O(\log^2 n)$ many subsampled graphs, and we will do $O(\log n)$ computation on each of them to search for a replacement edge for e, so this will take $\text{poly}\log n$ work in total.

This doesn't work for multiple deletes due to interdependency of the sampling, but there is a way to make it work. Check out the original paper for more details.

3 Amortizeed deterministic poly $\log n$ algorithm using link-cut trees [Holm et al., 2001]

Again this algorithm relies upon a fancy idea for finding (or failing to find) replacement edges upon a delete of an edge e. In this instance, the authors keep track of G as $O(\log n)$ subgraphs, and for each edge e some *level* $\ell(e) \in [0, \log n]$, where graph G_i contains all edges of level i and higher. So, $G_0 = G$ and $G_i \subseteq G_{i+1}$ for all i. Then, let F_i be a spanning forest for G_i; so if two vertices are connected in G_i they are also connected in F_i (and all G_{i-j}). The algorithm will maintain the forest F_i with link-cut trees and the two invariants

- The size of the connected components of F_i are less than $\frac{n}{2^i}$
- As previously mentioned, $F_0 \subseteq \ldots \subseteq F_{\log n}$.

`insert`(u, v) adds an edge to G_0 and update F_0. `delete`(u, v) for an edge at level ℓ. remove the edge, and if T_L, T_R are the trees in F_ℓ remaining with $|L| \leq |R|$, move T_L's edges to level $\ell + 1$, which maintains the first invariant. For each non-tree edge e between vertices in T_L at level ℓ, raise the edges to $\ell + 1$. For edge e between T_L and T_R at level ℓ, e is a replacement edge for (u, v); add it to all levels $\ell, \ell - 1, 0$. If a replacement doesn't exist at level ℓ, look at level $\ell - 1$, then $\ell - 2$, and so on.

Edges can't increase in level more than $O(\log n)$ times, and each increase takes $O(\log n)$ time using link-cut trees; thus deletes in total take $O(\log^2 n)$ amortized time. Checking whether x, y are in the same connected component takes $O(\log n)$ time as well.

References

David Eppstein, Zvi Galil, Giuseppe F Italiano, and Amnon Nissenzweig. Sparsificationa technique for speeding up dynamic graph algorithms. *Journal of the ACM (JACM)*, 44(5):669–696, 1997.

Greg N Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 14(4):781–798, 1985.

Jacob Holm, Kristian De Lichtenberg, and Mikkel 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.

Bruce M Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*, pages 1131–1142. Society for Industrial and Applied Mathematics, 2013.

Mihai Ptraşcu and Erik 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.

Mikkel 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.