15-850: Advanced Algorithms **CMU, Fall 2018**
Lecture #3: Dynamic Graph Connectivity algorithms 08/31/18
Lecturer: Anupam Gupta Scribe: Anish Sevekari and Justin Wang

Dynamic algorithms is the study of standard algorithmic problems in the setting where the input changes over time. In this lecture, we will focus on dynamic graph algorithm. Consider the setting where we have a sequence of *Updates* and *Queries*. Updates tell us how the graph is changing, and Query operations ask to output some aspect of the solutions to the underlying graph problems.

For instance, we may assume that the vertex set of this graph is fixed and we have two kinds of update operations: Insert(u, v), which adds edge (u, v), and Delete(u, v), which deletes edge (u, v). We will stick with this model for rest of the lecture, unless otherwise specified. (Sometimes we can get better results by restricting ourselves to only inserts or only deletes.) Algorithms that support both inserts and deletes are called *fully dynamic algorithms*.

1 Dynamic Connectivity

Dynamic Connectivity is the dynamic version of the classical graph connectivity problem. The graph changes by a sequence of Insert(u, v) and Delete(u, v) operations as described above. Moreover, we want to support two types of queries:

- Connected(u, v): asks if two vertices u, v are in the same connected component of the current graph, and
- Connected(G), which asks if the entire graph G is connected.

There are two naive approaches for this problem:

1. We can keep track of the graph, say as an adjacency matrix, and then run dfs each time we get a query. This take $O(1)$ time for each update, and $O(m + n)$ time for each query.
2. We can keep track of all connected components of graph. E.g., after each update we run dfs, and find the connected components. Now each query takes $O(1)$ time, but each update takes $O(m)$ time. Observe that if there are no deletions, we can use union-find data structure to implement updates in $O(\alpha(m, n))$ amortized time, where m is total number of updates.

These two approaches illustrate a trade-off between update time and query time. But if we want to balance between them to get small query and update times, what should we do?

The dynamic graph connectivity problem has been widely studied, but still it has not been completely solved yet. We list some known results about dynamic connectivity below:

authors	update time	query time	comments
Frederickson [Fre85]	$O\left(m^{\frac{1}{2}}\right)$	$O(1)$	deterministic worst case
Eppstein et al [EGIN97]	$O\left(n^{\frac{1}{2}}\right)$	$O(1)$	deterministic worse case
Holm, de Lichtenberg, Throp [HdLT98]	$O(\log^2 n)$	$O\left(\frac{\log n}{\log\log n}\right)$	amortized
Kapron, King, Mountjoy [KKM13]	$O(\log^5 n)$	$O\left(\frac{\log n}{\log\log n}\right)$	randomized worst case

Lower bound: A lower bound of Pătraşcu and Demaine [PD04]) says the following: if t_u denotes the update time and t_q denotes the query time (both in amortized sense), then following inequalities hold:

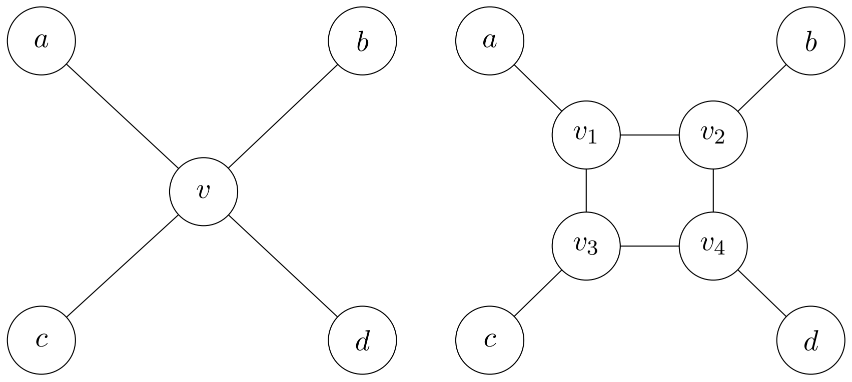
$$t_q \log(t_u/t_q) = \Omega(\log n)$$
$$t_u \log(t_q/t_u) = \Omega(\log n)$$

This can be used to infer that $\max\{t_u, t_q\} = \Omega(\log n)$.

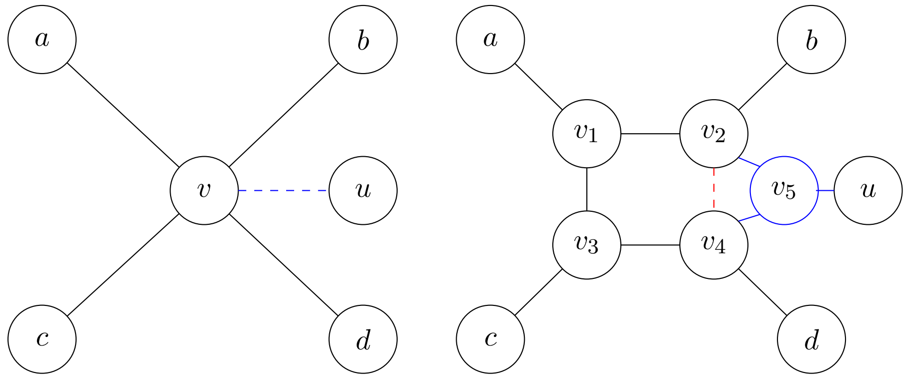
2 Frederickson's algorithm

We describe a weaker version of Frederickson's algorithm [Fre85] below, which gives $O(m^{2/3})$ update time and $O(1)$ query time. Throughout this algorithm, we maintain a spanning forest. We cluster this forest into parts which are roughly of the same size and maintain these clusters. There are two important ideas to achieve this:

1. We may assume that max degree of a graph is 3. We will show this by reducing any connectivity problem to a graph with max degree 3 without blowing up size of the graph. Pick any vertex with degree $d > 3$. We split this vertex into d vertices which form a cycle, and connect exactly one of these vertices to a neighbor of v. Note that all the new vertices have degree 3. Case for $d = 4$ is illustrated in the figure below:



After doing this for all vertices, we get a graph of degree at most 3. To perform an INSERT, we might have to increase one of the cycles, which takes 3 total INSERT operations and one DELETE. Therefore, we preserve number of updates up to a constant. Delete is an exact inverse of insert. Note that we need to add vertices for INSERT, but this does not change algorithm. Picture below shows INSERT, with blue edges being added and red being removed:



Note that we add at most $O(m)$ new vertices, therefore, number of total vertices might go up to $O(m)$, but number of edges is still $O(m)$.

2. Tree Separators! We claim that we can cluster any spanning forest into clusters whose sizes are in $[z, 3z]$ for any z.

Lemma 3.1. *Given any tree T on n vertices with max degree at most 3, we can find and edge e such that removing e divides T into 2 connected components, both having sizes in $[n/3, 2n/3]$*

Proof. Root the tree at some node r. We'll find the desired separating edge by walking out from the root. If one of the subtrees has size more than $\frac{2n}{3}$, we walk to that subtree. Note that only one subtree could possibly have this property, because otherwise the total node count $\frac{2n}{3} + \frac{2n}{3}$ would excced n.

Now consider the first node u at which there is no longer a subtree of size more than $\frac{2n}{3}$. If u has one child, then the edge going to this child is our separating edge. Thus there are two nontrivial cases to consider: when u has 3 children (in which case u is the root) or when u has 2 children.

In the first case, suppose the children are c_1, c_2, c_3 with corresponding subtrees S_1, S_2, S_3. By the Pigeonhole Principle, at least one of the children has subtree of size bigger than $n/3$, say it's S_1. But by assumption, all subtrees have size at most $\frac{2n}{3}$. Thus we pick S_1 as one component and rest of the tree as other. Note that $k \in [\frac{n}{3}, \frac{2n}{3}]$ implies $n - k \in [\frac{n}{3}, \frac{2n}{3}]$, so it's enough to see that one component is in the desired size range.

Otherwise, suppose the children are c_1, c_2 with subtrees S_1, S_2. Because u is the first node whose subtrees' size do not exceed $\frac{2n}{3}$, the size of its subtree is greater than $\frac{2n}{3}$. Thus, we have $|S_1| + |S_2| \geq \frac{2n}{3}$. Again, by Pigeonhole, at least one of S_1 or S_2 has size more than $\frac{n}{3}$, say it's S_1. Then, $|S_1| \in [\frac{n}{3}, \frac{2n}{3}]$, so we choose S_1 as one of the components and rest of the tree as the other component. Note that in this process, we never travel any edge inside one of the components that we get. □

Lemma 3.2. *Given any tree and a parameter z, we can divide tree into clusters such that each cluster is connected and they all have sizes in $[z, 3z)$.*

Proof. This immediately follows from the lemma above. Suppose any cluster has size $\geq 3z$, then we can split it into two connected clusters using lemma 3.1. Components thus produced will never have size less than z. We can keep doing this until all clusters have size $\leq 3z$. ☐

Note that given a tree, we can find these clusters in $O(|E|)$, since while doing this process above, we scan each edge at most once.

We maintain a spanning forest F divided into clusters of size z, where the optimal z will be picked after analyzing the runtime. If a tree has size less than z, we maintain it as one cluster. Note that there are $O(m/z)$ clusters. Further, since all vertices have degree at most 3, each cluster contains $O(z)$ edges.

For each vertex, we will have to keep track of which cluster it lies in. For each cluster, we track which tree it lies in. For each pair of clusters C_i, C_j, we maintain a list of all edges not in F that connect C_i and C_j. We also need to keep track of the adjacency matrix of graph. See Anupam's blog post for further details.

Now we will describe how we process updates and queries:

1. Insert(u, v): If u, v are in different trees, then we merge the trees. Combine clusters containing u and v, and then reform the clusters using lemma 3.2. Note that since each cluster has $O(z)$ edges, this takes $O(z)$ time.

2. Delete(u, v): If u, v is not in the spanning forest, then the spanning forest doesn't change. If it is, then we need to check if deleting (u, v) disconnects the graph or not. Deleting (u, v) divides some tree T into two parts, say T_L, T_R. There are two cases:

   (a) If (u, v) is not in any cluster, then for all clusters $C_i \in T_L$ and $C_j \in T_R$, we check if there is an edge from C_i to C_j. If yes, we add this edge into the tree and update the information about non-forest edges between C_i and C_j. We do not change the actual membership of any clusters or connected components. If no edge is found, T_L and T_R become disjoint trees, and it's an $O(m/z)$ operation to reassign the clusters to their new trees. The runtime is dominated by the time it takes to check all pairs of clusters (C_i, C_j) for a connecting edge, which takes time $O(m^2/z^2)$.

   (b) If (u, v) is in some cluster C, then removing edge (u, v) divides $T \cap C$ into two parts, say L and R. First, we check if there is an edge in C from L to R, which takes $O(z)$ time. *Note, this is where we take advantage of the fact that we can bound our cluster size!*

   If we find such an edge e, then we add e to T. In this case, we don't need to change any of the other metadata we are maintaining. If there is no such edge, then we try to find an edge from T_L to T_R by doing the same process in part (a), which still takes $O(m^2/z^2)$ time.

   Finally, note that after this, we might have L and R as separate clusters, whose sizes might be less than z. If so, we merge them with any one of the adjecent clusters and use lemma 3.2 to find a clustering that obeys the size constraint. Since each cluster only has $O(z)$ edges, reclustering takes only $O(z)$ time.

   The total runtime in this case is $O(z + m^2/z^2)$.

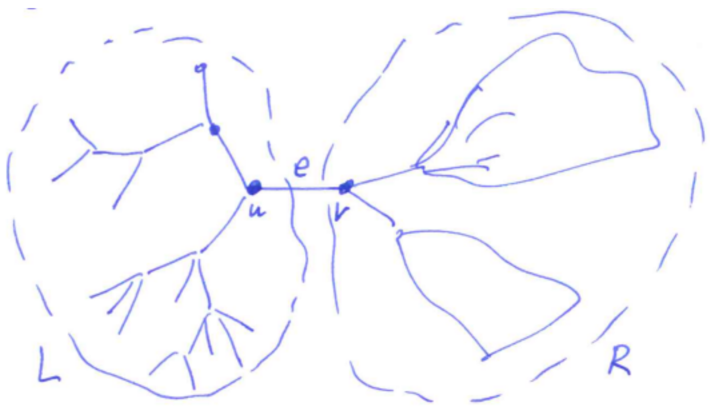
We see that, in the worst case, the runtime is $O(z + m^2/z^2)$. We pick $z = m^{\frac{2}{3}}$, so that in the limit, neither the z nor m^2/z^2 terms dominates the other. We thus conclude that Delete takes $O(m^{\frac{2}{3}})$ time.

3. Query: for queries of type Query(u,v) we just need to check if u, v are in the same tree or not. To check connectivity of whole graph we need to check if there is exactly one tree or not. This takes $O(1)$ time.

Therefore, this algorithm handles updates in $O(m^{2/3})$ worse case time and queries in $O(1)$ time.

3 A Randomized Algorithm for Dynamic Graph Connectivity

We've seen that one fundamental way of approaching dynamic graph connectivity is to maintain a spanning forest F for our graph. If an edge e in tree $T \in F$ is deleted, we must make sure F is still a valid spanning forest. To do this, we observe that e will separate T into components L and R, and we either must find a replacement edge between L and R, or determine that no such edge exists (in which case L and R become distinct trees in F).



In the previous section, we saw Frederickson's approach for quickly doing this: by subdividing trees into clusters of bounded size, and keeping track of the additional edges between clusters. Now, we consider a randomized approach to the process of either finding a replacement edge or determining that none exists, which is due to Kapron, King, and Mountjoy.

We'll first consider a simplified version of the problem, where only *one* edge deletion ever happens. This will be enough to highlight two interesting techniques of their algorithm.

3.1 Giving Nodes a Bit Representation, and the Power of XOR

We begin by assigning to each node v a $(\log n)$-bit label, which we denote by $\ell(v)$. Let there be some total ordering on the nodes, such that each edge has a canonical ordering (u, v) where $u < v$. We can then give each edge $e = (u, v)$ a label by concatenating the names of its two vertices: $\ell(e) := [\ell(u), \ell(v)]$.

Definition 3.3. The fingerprint $\mathcal{F}(v)$ of any node v is defined as:

$$\bigoplus_{e \in \partial v} \ell(e) .$$

Note, we are taking the fingerprint over all edges in the graph that are incident to v, not just the edges in tree incident to v.

Suppose we have computed $\mathcal{F}(v)$ for each vertex v. When we delete an edge (u, u'), we can maintain correctness of the values of $\mathcal{F}$ in $O(1)$ time, by just XOR'ing $\ell(v)$ with the old values of $\mathcal{F}(u)$ and $\mathcal{F}(u')$.

To see how this can be useful, let's further simplify our problem: *Suppose that after deleting e, there exists a unique replacement edge e' from L to R.*

Then, we have the following useful fact:

Fact 3.4.

$$\bigoplus_{v \in L} \mathcal{F}(v) = \ell(e')$$

It's easy to see why: any edge with both endpoints in L will be XOR'd twice, and, hence, cancel off. The only thing remaining will be $\ell(e')$.

For details on data structures foe how XOR can be quickly computed, see Anupam's blog post.

3.2 Multiple Edges? Subsample!

If there are multiple edges from L to R, then the result of the calculation in 3.4 will not result in any meaningful edge. Rather, it will just be the XOR of all edges from L to R, which is not useful. However, we can get an idea for how to generalize this trick by thinking about what makes it work.

Observation 3.5. The process in 3.4 works for any subset of edges E' of the graph, so long as exactly one edge $e' \in E'$ goes from L to R.

The idea will be to use random subsampling to create different subsets of E, with success being when we generate a subset having exactly one edge crossing from L to R. If we can show that the probability of success is $\theta(1)$, and we repeat the process $\Omega(\log n)$ times, then the probability we generate a successful subset will be $1 - \frac{1}{\text{poly}(n)}$, i.e. we succeed with high probability.

For simplicity, assume there are 2^i edges from L to R. Generate subsets $E_0, E_1, \ldots, E_{\log m}$ by picking each edge $e \in E$ to be in E_j with probability $\frac{1}{2^j}$ (each E_j is generated independently).

Claim 3.6. **Pr**(*exists a unqiue L-to-R edge in E_i*) $= \theta(1)$.

Proof. If e is some edge from L to R, the probability that e is the unique L to R edges in E_i is given by

$$\frac{1}{2^i}\left(1 - \frac{1}{2^i}\right)^{2^i - 1}.$$

By assumption, there are 2^i edges from L to R, and any of these edges can be the unique one from L to R. Since the events are all disjoint, the total probability is

$$2^i \frac{1}{2^i}\left(1 - \frac{1}{2^i}\right)^{2^i - 1} \approx \left(1 - \frac{1}{k}\right)^k \approx \frac{1}{e}.$$

□

Zooming back out to the overall algorithm, the idea is for each subset of edges E_i and each vertex v, to calculate the signature of v "induced" by that particular subset:

$$\mathcal{F}(v) := \bigoplus_{e' \in \partial_G(v) \cap E_i} \ell(e').$$

After taking the XOR of these signatures, the result is some $2 \log n$ length sequence of bits. If this sequence of bits does not correspond to any edge in the graph, then we know that E_i must have contained multiple L-to-R edges, and we try another subset.

Even if the sequence of bits corresponds to a valid edge, it was pointed out in lecture that we still need to check that this edge is actually an L-to-R edge. There can be a false positive, because the result of XOR'ing multiple L-to-R edges could happen to be some totally different edge in the graph (for example, it could be an edge purely within L; or, if our graph consists of multiple spanning trees, the edge could be from a different tree).

One way to handle false positives is to maintain a forest with a $\text{Findroot}(v)$ operation. When we delete (u, v), either u or v becomes the root of a new subtree in the forest. If (u', v') is the candidate replacement edge, we should check that the sets $\{\text{Findroot}(u'), \text{Findroot}(v')\}$ and $\{\text{Findroot}(u), \text{Findroot}(v)\}$ are identical. For more details, see Anupam's blog post.

4 An Amortized, Deterministic Algorithm

Finally, we briefly sketched (i.e. in under 10 minutes) an algorithm due to Holm, et al., which runs in $O(\text{polylog} n)$ amortized time, and is deterministic. Again, we model the graph as a forest of maximal spanning trees, and focus on efficiently finding a replacement edge to connect the components L and R that are split by an edge deletion.

The naive thing to do is to just start scanning edges from whichever component is smaller, and check each edge to see if it will re-connect the tree.

Idea: Keep each edge in some level in $[0, \log_2 n]$. Each time we scan an edge that does not help connect the tree, we "charge" it by raising its level.

Let E_i denote the edges that are level i or higher. For G_i, the graph induced by E_i, let F_i be a spanning forest.

We will maintain the following invariants:

1. F_i is always a spanning forest, i.e. if $x, y \in E_i$ are connected, then they are connected in F_i.
2. $F_0 \supset F_1 \supset \cdots \supset F_{\log n}$
3. Each component in F_i has at most $\frac{n}{2^i}$ nodes.

When we insert an edge, we insert it at level 0. So before we have deleted any edges, the invariants obviously hold.

When we delete an edge, say it splits the tree into components L and R with $|L| \leq |R|$. If e was at level ℓ, we raise all edges of L that were at level ℓ to level $\ell + 1$

Note: It's crucial that we "charge" the smaller half (in this case, L), to maintain our third invariant.

Now we start scanning nontree edges to find a replacement, starting with all potential edges at level ℓ and then dropping down level by level. At each level, if we scan an edge that goes within L, we raise it up one level.

Otherwise, we've found an edge that goes to R (otherwise, $L \cup R$ would not have been a maximal spanning tree before the edge deletion). Stop when we find such an edge, and add it to $F_\ell, F_{\ell-1}, \ldots, F_0$.

You can think of the levels as a way of organizing edges, so that we do not scan any particular edge too often. In the worst case, each edge is charged $O(\log n)$ times. If we use link/cut trees for the F_i at each level, the cost of maintaining the trees is $O(\log n)$, so in total updates cost $O(\log^2 n)$.

To query, we just check if u and v are in the same component (i.e. have the same root), which takes $O(\log n)$ time.

Acknowledgments

These lecture notes were scribed by Anish Sevekari and Justin Wang, based on previous scribe notes of Hui Han Chin and Jacob Imola.

References

[EGIN97] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification;a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, September 1997. 1

[Fre85] Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 14(4):781–798, 1985. 1, 2

[HdLT98] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 79–89, New York, NY, USA, 1998. ACM. 1

[KKM13] 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*, SODA '13, pages 1131–1142, Philadelphia, PA, USA, 2013. Society for Industrial and Applied Mathematics. 1

[PD04] Mihai Pǎtraşcu and Erik D. Demaine. Lower bounds for dynamic connectivity. In *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing*, STOC '04, pages 546–553, New York, NY, USA, 2004. ACM. 1