3
Dynamic Algorithms for Graph Connectivity

Work in progress.

Dynamic algorithms is the study of algorithmic problems in the setting where the input changes over time. At each timestep, we get either an ***update*** which tells us how the input has changed, or a ***query*** which demands outputting some aspect of the solution to the underlying problem. In this chapter focus on some basic graph questions, but the area is much broader, and we can consider efficient dynamic algorithms for most algorithmic questions.

For graph problems, the most common model is the *edge-update* model. Hence, the update operations are InsertEdge(e) and DeleteEdge(e), which add and delete some edge e from the graph respectively. (It is useful to also allow InsertNode(v) and DeleteNode(v), which add and delete isolated nodes in the graph; we cannot delete a node that has an edge incident to it.) If we can handle both kinds of edge-update operations, we are said to be in the *fully-dynamic* case, else we may be in the *insert-only* or *delete-only* case. How can we maintain solutions to basic graph problems in these models?

3.1 Dynamic Connectivity

We restrict our attention to the most basic of all graph problems: graph connectivity. As the graph changes via a sequence of Insert and Delete operations, we want to support two types of queries:

- Connected(u, v): are the vertices u, v in the same connected component of the current graph?
- Connected(G): is the current graph G connected.

Here are two naïve approaches for this problem:

1. We keep track of the updates, say using an adjacency matrix, and then run depth-first search each time we get a query. This takes $O(1)$ time for each update, and $O(m+n)$ time for each query.

2. We explicitly maintain the connected components of graph by running depth-first search after each update. Now each query takes $O(1)$ time, but each update may take $O(m)$ time.

If there are no deletions, we can use ***disjoint set union-find*** data structure to implement both updates and queries in $O(\alpha(n))$ amortized time.

These two extreme approaches suggest a trade-off between update time and query time: how can we balance between them to get small query and update times?

3.1.1 *Some Issues to Consider*

The quality of our algorithms will depend on the power we allow them. For example,

- ***worst-case*** versus ***amortized*** update and query times: does the algorithm take at most B time on each step? Or can some of the steps require more time and others less, as long as any sequence of T operations requires at most BT time?

- ***deterministic*** versus ***randomized*** algorithms: does the algorithm flip coins or not?

  Even among randomized algorithms, we may consider questions like ***Las Vegas*** versus ***Monte Carlo*** algorithms: is the algorithm always correct, or is it just correct for each output with high probability? Or that of ***adaptive*** versus ***oblivious*** adversaries: can the request at some step depend on the coin-tosses made by the algorithm in previous steps? We defer these subtler distinctions for now, and just focus on either deterministic algorithms, or on Monte Carlo algorithms against oblivious adversaries, where the input is assumed to be fixed before we flip any coins.

3.1.2 *And Some Results*

The dynamic graph connectivity problem has been studied along all of these axes, which makes it exhausting to give a complete list of the current best results. Let us just list the results we will discuss.

1. For deterministic algorithms with worst-case runtimes, Greg Frederickson showed how to get $O(\sqrt{m})$ update time and $O(1)$ query time. This was improved to $O(\sqrt{n})$ update times by Eppstein et al., but then progress stalled for quite some time. We will discuss Frederickson's approach, and see Eppstein et al.'s extension in a homework.

Only very recently, Chuzhoy et al. broke through the logjam, and obtained $n^{o(1)}$ worst-case update time; the team includes CMU almunus Richard Peng and current student Jason Li.

2. For deterministic algorithms with amortized runtimes, Holm, de Lichtenberg, and Thorup showed $O(\log^2 n)$ update times and $O\left(\frac{\log n}{\log \log n}\right)$ query times.
3. For randomized algorithms with worst-case bounds, Kapron, King, Mountjoy gave an algorithm with poly-logarithmic update and query times.

All three of these algorithms illustrate some clever algorithmic ideas; the running times are not the point here.

The structure of the algorithms will be simple: we maintain a spanning forest of the current graph, which captures the current connectivity information. When a new edge is added, we check if this edge merges two trees in this forest—which is easily done. But when an edge in the spanning forest is deleted, it causes some tree T to break into subtrees L, R—the challenge is to quickly figure out if there is a replacement edge that connect back L and R. The three algorithms handle this challenge in different ways.

3.1.3 *Oh, and a Lower Bound*

A quick word about lower bounds. Sadly, we cannot achieve constant update and query times: a lower bound of Mihai Pătraşcu and Erik Demaine says that if t_u denotes the update time and t_q denotes the query time (both in a deterministic? amortized sense), then

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

This implies, for instance, that $\max\{t_u, t_q\} = \Omega(\log n)$. move later?

3.2 *Frederickson's algorithm*

We describe a weaker version of Frederickson's algorithm, which has an $O(m^{2/3})$ update time, and $O(1)$ query time. As we mentioned, the algorithm maintains a spanning forest of the graph. To find a replacement edge fast, it clusters the forest into "clusters" that are roughly the same size. Now it keeps track of which clusters have edges going between them. So, when searching for a replacement edge, it can just look at pairs of clusters, instead of at all nodes.

3.2.1 *Reduction to the Sub-Cubic Case*

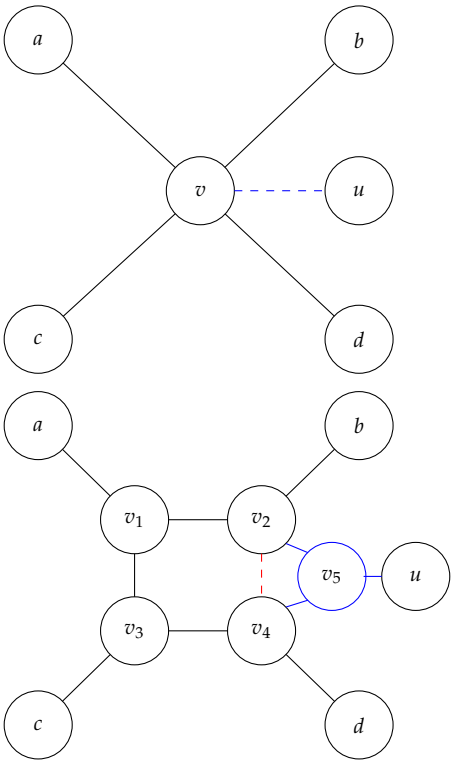
We can reduce to the case where the maximum degree of any node is at most 3. Indeed, given updates to a graph $G = (V, E)$, we instead maintain a graph $H = (V', E')$ with $V \subseteq V'$, such that the answer to

the queries on G and H have the same. The graph H has $O(m)$ nodes and edges.

The mapping is simple: pick any vertex v in G with degree $d \geq 2$, create a cycle $v = v_1, v_2, \ldots, v_d$ in H, and connect each vertex on this cycle to a unique neighbor of v. A vertex of degree one is unchanged. Moreover, this mapping can be maintained dynamically: if an edge $e = uv$ is inserted into G, we may have to increase the size of the cycles for both endpoints (or create a cycle, in case the degree has gone from 1 to 2), and then add an edge. This requires a constant number of InsertNode and InsertEdge operations in H (and maybe one DeleteEdge as well), so the number of updates is maintained up to a constant. Deleting an edge in G is the exact inverse of this process.

While we did not discuss the data structures needed to maintain this mapping, it is easy to do this using arrays or hash-tables; we omit the details here. Give more?

We don't really need the cycle for nodes that have degree at most 3 in G, but it is easier to maintain a uniform rule.



This is tight for a star with three leaves.

3.2.2 *Tree Separators*

The advantage of a sub-cubic graph is that any spanning forest F also has maximum degree 3. This allows us to use the following elementary clustering algorithm:

Lemma 3.1. *Given a positive integer z, and any tree T with at least z nodes and maximum degree 3, we can partition its vertex set into clusters such that each cluster (a) induces a connected subtree, and (b) contains between z and $3z$ nodes.*

Proof. We start with a single edge-separator result:

Claim 3.2. Given any tree T on n vertices with maximum degree at most 3, we can find an edge e whose deletion forms two connected components, each having sizes in $[\frac{n-1}{3}, \frac{2n+1}{3}]$.

Proof. For each node v in T, the remaining $n-1$ nodes of the tree lie in one of the (at most) three subtrees hanging off it. Draw a directed arc to its neighbor u which has the largest number of nodes in its subtree (breaking ties arbitrarily). Since there are n nodes and $n-1$ edges in T, there is at least one edge $e = uv$ in it whose endpoints draw arcs to each other. We claim that e is the desired edge.

Indeed, say the other two trees at u have sizes a, b, and those at v have sizes c, d, as in the figure. Then the nodes in u's component upon deleting e will have $a + b + 1$ nodes. And

$$\begin{aligned} & a+b+1 \geq c \qquad \text{and} \qquad a+b+1 \geq d \\ \implies & a+b+1 \geq \frac{(a+b+1)+c+d}{3} = \frac{n-1}{3}. \end{aligned}$$

A similar argument shows that $c + d + 1 \geq \frac{n-1}{3}$, which means $a + b + 1 = n - (c + d + 1) \leq \frac{2n+1}{3}$. □

Now to prove the clustering theorem: consider a collection of trees each of size at least z (initially containing only T). If any of these trees has more than $3z$ nodes, use the edge-separator to delete an edge and form two smaller subtrees. Both these subtrees will have size at least $\frac{(3z+1)-1}{3} \geq z$. Repeat until all trees in the collection have sizes in $[z, 3z]$—these trees give the clustering. □

Important Exercise: a naïve application of this theorem may take $O(n^2)$ time. Given an implementation that clusters T in $O(n)$ time.

3.2.3 *The Algorithm*

We maintain a spanning forest F of the dynamically changing subcubic graph H. For each tree in F, we also maintain a clustering of the kind promised in Lemma 3.1 for some value of z we will choose later. (If some tree in F has size less than z, it forms a *trivial* cluster by itself.) Note that there are $O(m/z)$ non-trivial clusters. Further, since all vertices have degree at most 3 in H, each cluster contains $O(z)$ edges from H.

We keep track of the following pieces of information:

- Each vertex tracks its cluster, and its neighbors in both F and H.
- Each cluster tracks which tree in F it belongs to.
- Each pair of clusters C_i, C_j maintains a list of all edges not in F that go between C_i and C_j. This list is non-empty only for clusters in the same tree.

We need data structures, that given an edge, can locate its two vertices, etc. Details? Now we will describe how we process updates and queries:

1. INSERT(uv): Look up u's cluster C_u, and its tree T_u. Do the same for v. If u, v belong to the same tree, do nothing. Else, u, v are in different trees in F, so merge them. Combine their clusters C_u, C_v in these trees into a single cluster C. If it has size more than $3z$, break it into two clusters in $O(z)$ time (using the Exercise above).

2. DELETE(uv): If edge uv is not in the spanning forest F, then F does not change. Just remove uv from the list for the pair of clusters C_u, C_v.

   Else $uv \in T$ for some tree T in the spanning forest F, and we must check if deleting uv disconnects H or not. Deleting it breaks T into T_u, T_v containing u, v respectively. There are two cases:

   (a) Suppose uv was not within a cluster. Then for all clusters $C_i \in T_u$ and $C_j \in T_v$, check if there is an edge xy from C_i to C_j. Upon

finding the first such edge, add this edge to the tree T, and remove it from the list of non-forest edges between C_i and C_j. (You can think of this as first deleting this edge xy and then immediately adding it back in.)

If no edge is found between any clusters of T_u, T_v, then these two must replace T in the spanning forest. Iterate over the $O(m/z)$ clusters to reassign them clusters to their new trees. This runtime is dominated by the $O(m^2/z^2)$ time taken to check all pairs of clusters (C_i, C_j) for a connecting edge.

(b) Now suppose uv lies within some cluster C. Removing it divides the subtree $T[C]$ induced by this cluster into two parts, say L, R. First, check if there is an edge within C from L to R: since there are $O(z)$ nodes in C each of degree at most 3, this takes $O(z)$ time. (This is where we use the bounded cluster size!) If such an edge e is found, add e to T, and thereby reconnect cluster C as well; there is no need to change any of the other metadata we are maintaining. If there is no such edge, then expand the "search area": look at the two parts T_u, T_v of tree T, and try to find an edge from T_u to T_v as in part (a), which still takes $O(m^2/z^2)$ time.

Finally, L and R may still be separate clusters, but their sizes might be less than z. If so, merge each of them with any one of their adjacent clusters, and use Lemma 3.1 to get their sizes back in the range $[z, 3z]$. Since each cluster only has $O(z)$ edges, using a linear-time implementation would recluster in only $O(z)$ time. Hence, the total runtime in this case is $O(z + m^2/z^2)$.

To minimize this worst-case runtime bound, we can set $z = m^{2/3}$ to balance the various update times above, and get $O(m^{2/3})$ time for both InsertEdge and DeleteEdge. The InsertNode and DeleteNode operations can be implemented in $O(1)$ time.

3. Query: for queries of type Query(u,v) we check whether the clusters for u, v belong to the same tree. To check connectivity of H we check whether the spanning forest contains one tree or not. Both these can takex $O(1)$ time.

As mentioned above, the $O(m^{2/3})$ worst-case update time was improved by Frederickson by maintaining an *Euler-tree* data structures to reduce the $O(m^2/z^2)$ time for searching over all pairs of clusters down to $O(m/z)$. Now balancing terms gets an $O(m^{1/2})$ update time. The homeworks may ask you to derive a cool and fairly generic way to sparsify graphs for dynamic algorithms, which improves the algorithm's dependence on the number of edges m down to a dependence on the number of vertices n.

3.2.4 Data Structure Details

3.3 An Amortized, Deterministic Algorithm

If we are happy with amortized update times instead of worst-case ones, we can avoid the clustering approach. Indeed, the reason we wanted the clustering was to quickly find a replacement edge (if possible) whenever deleting a tree edge uv caused some tree T to break into components L and R. The clusters meant that we could focus the search on pairs of clusters, instead on doing the naïve search.

Without the clustering, we use a charging scheme to keep track of the work we do. So we associate a ***level*** with each edge, which is zero when the edge is just inserted. Now when T breaks into L, R because some edge is deleted, we want to scan the edges incident to one of the sides (say the side L with fewer vertices) to check if they remain within L, or if they go to R. (Because T was a tree in a spanning forest, there can be no other options: do you see why?) When we scan an edge e and it fails to connect L, R, we increase its level by 1—this "pays" for the failed work we did in scanning e.

Of course, this charging is completely useless if the levels can rise unboundedly. So we maintain a careful structure that ensures that the level of any edge is at most $\log_2 n$.

3.3.1 A Collection of Spanning Forests

Here is how we do it. We imagine there being many graphs

$$G = G_0 \supseteq G_1 \supseteq \cdots \supseteq G_i \cdots,$$

where graph G_i consists of the edges with level i or higher. We maintain a spanning forest F_0 of G_0, with the additional property that $F_i := F_0 \cap G_i$ is also a spanning forest of G_i. It can be useful to think as follows:

($\star$) F_0 is a max-weight spanning forest of G_0, w.r.t. the edge-levels.

Adding an edge is easy: set the level to zero, and add it to F_0 if it does not create a cycle in F_0. Deleting an edge not in F_0 is also easy: just delete it. But if an edge $e \in F_0$ (say with level ℓ) is deleted, we need to search for a replacement. And this replacement must be at level ℓ or lower—this follows from property ($\star$). Moreover, to maintain that property, we should add a replacement edge of the highest level possible. So the off-tree edges should be considered from level ℓ downwards.

Remember, when we scan an edge, we want to raise its level. That could mess with probability ($\star$), because this may cause off-tree

edges to have higher level/weight than tree edges. To fix that problem, we first raise the levels of some of the tree edges. Specifically, we do the following steps:

1. Say deleting e breaks tree $T \in F_\ell$ into L_ℓ, R_ℓ. Let $|L_\ell| \leq |R_\ell|$; we'll scan the edges incident to L_ℓ. (We'll see how this choice of the smaller side can bound the number of levels by $\log_2 n$.) We raise the all level-ℓ edges in L_ℓ to have level $\ell + 1$.

2. Scan level-ℓ edges incident to L_ℓ. If some edge e connects to R_ℓ, this is the replacement edge. Add e to F_0 (and hence to $F_1, \ldots, F_\ell$). Do not raise its level, though. Stop: you are done.

   Else, if the edge stays within L_ℓ, raise its level, and try another.

If we fail to find a replacement edge at level ℓ, we lower ℓ by one, and try again. And if we don't find a replacement edge even at level 0, we stop—there is no replacement edge, and the deletion has increased the number of components in the graph. (Convince yourself that $(\star)$ remains satisfied when we do all this.)

3.3.2 *Bounding the Maximum Level*

It just remains to show that no edge can have level more than $\log_2 n$—this is what all the steps above have been leading up to. This is simple: we prove a second invariant.

$(\star\star)$ Each tree in forest F_ℓ has at most $\lfloor n/2^\ell \rfloor$ nodes.

This is clearly true for F_0. Now a tree in F_ℓ is formed by raising the levels of some "left" piece $L_{\ell-1}$. It was the smaller half of some tree in level $\ell - 1$, which by induction has size at most $\lfloor n/2^{\ell-1} \rfloor$. Half of that is at most $\lfloor n/2^\ell \rfloor$; this means invariant $(\star\star)$ is maintained by the algorithm.

Fianlly, each tree must have size at least 2, so invariant $(\star\star)$ means the level ℓ of any edge is at most $\log_2 n$. Observe: we're not saying that only a few edges have level $\approx \log_2 n$. Just that each component at a high level is small. I find this particular charging very clever, even though variants of this idea (of "charging to the smaller side") arise all the time.

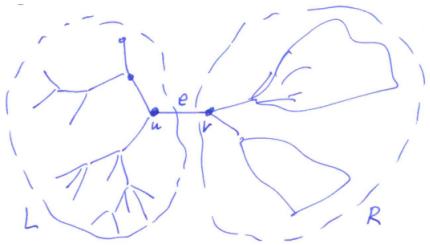
To summarize: we raise levels of edges to pay for the work we do in finding a replacement edge. Each edge-level is at most $O(\log n)$, so the amortized number of scans is $O(\log n)$. And we can do each of these operations of raising levels and maintaining spanning forests in amortized $O(\log n)$ time, giving $O(\log^2 n)$ amortized update time overall.

Of course, we have not talked about how to do this efficient implementation. The trick is to use the ***link-cut trees*** data structure of

Danny Sleator and Bob Tarjan. The details are not difficult: once you see what operations the data structure supports, you can work them out yourself. Or you can check out the paper of Holm, de Lichtenberg, and Thorup.

3.4 *A Randomized Algorithm for Dynamic Graph Connectivity*

Finally, let's sketch a randomized approach to finding replacement edges, which leads to a Monte-Carlo randomized (albeit worst-case) update time algorithm due to Bruce Kapron, Valerie King, and Alan Mountjoy. To simplify matters, consider the case where a *single* edge-deletion occurs: this will be enough to highlight two interesting techniques of their algorithm.



3.4.1 *Giving Nodes a Bit Representation, and the Power of XOR*

Suppose we delete edge $e = uv$, causing the tree T to split into L and R. As a thought experiment, suppose there is a *single* replacement edge f in G that goes between L and R: all other edges incident to L and R do not cross. How would we find f?

We begin by assigning to each node v a $(\log_2 n)$-bit label $\ell(v)$. Assume some total ordering on the nodes, so the edge uv (where $u \prec v$ in this ordering) has a $2\log_2 n$-bit label that concatenates the names of its two vertices in that order:

$$\ell(e) := [\ell(u), \ell(v)].$$

Definition 3.3 (Node Fingerprint)**.** The *fingerprint* $\mathcal{F}(v)$ of node v is defined to be the exclusive-or of labels of all edges incident to v in G

$$cF(v) := \bigoplus_{e \in \partial_G v} \ell(e).$$

Note: we consider all neighbors in the graph G, not just in the spanning forest.

It is easy and fast to maintain the fingerprint $\mathcal{F}(v)$ for each vertex v as edges change. Now when we delete an edge uv (and if there is a unique replacement edge f), consider the following:

Fact 3.4.

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

Proof. Indeed, each edge with both endpoints in L will be present twice in the exclusive-or, and hence will be zeroed out. The only contribution remaining will be $\ell(f)$. □

3.4.2 *Multiple Edges? Subsample!*

What if there are multiple possible replacement edges going from L to R? The result of the calculation in Fact 3.4 will be the XOR of the labels of all these edges, which is not useful. However, all is not lost—our secret weapon will be randomness, which we have not used yet.

Let us make an weaker assumption this time: *suppose there are k replacement edges, but we know this value k.* We can do the following: we keep a subsampled graph G' obtained by picking each edge in G with probability $1/k$. Then for any $k \geq 1$,

$$\Pr(\text{exists unique } L\text{-to-}R \text{ edge in } G') = k \cdot \frac{1}{k}\left(1 - \frac{1}{k}\right)^{k-1} \geq \frac{1}{e}. \quad (3.1)$$

So with constant probability, there is a unique edge, and we run the previous fingerprinting algorithm. Observe that if we don't know k precisely, but instead use some value in $[k/2, 2k]$, the probability **??** is no longer $1/e$, but still remains a constant. Moreover, by repeating this process $O(\log n)$ times independently, we can ensure that one of these repetitions will give a unique crossing edge with probability $1 - 1/\operatorname{poly}(n)$.

Finally, what if we don't even have a crude estimate of k? We can try subsampling at rates $1/2, 1/4, \ldots, 1/2^i, \ldots, 1/n$, i.e., at *all powers of* 2 between $1/2$ and $1/n$. There are $\log_2 n$ such values, and one of them will be the correct one, up to a factor of 2. This means that one of the $O(\log n)$ trials for one of these $O(\log n)$ sampling rates will succeed with probability $1 - \frac{O(\log n)}{\operatorname{poly}(n)}$, by a trivial union bound.

3.4.3 *Wrapping Up*

There are several details that remain to work out: the answers in the random experiments where the sample contains multiple L-to-R edges may not make sense. Or they could give us valid names of edges that do not cross between L and R. In this case, we need a mechanism to check that the answer is indeed an L-to-R edge. This, again, can be done by the link-cut tree data structure mentioned above. Details of this can be found in the Kapron, King, and Mountjoy paper.

A closely related algorithm is due to Ahn, Guha, and MacGregor. Future developments.