1 Dynamic graph algorithms

Consider a graph problem: given a graph $G = (V, E)$, one is to compute a function $f(G)$. Some example functions are, what is the size of the maximum matching in G, are two vertices s and t connected in G, what is their distance? Throughout the course of history, many efficient algorithms for a large variety of graphs problems have been developed. However, graphs in practice are dynamic, i.e. they undergo small changes such as edge deletions or insertions. The study of dynamic algorithms takes this into account by developing algorithms that maintain a data structure on G so that any update of G can be performed efficiently, so that queries for $f(G)$ in the current graph can be answered very fast, in near-constant time.

2 Dynamic connectivity

Given an undirected graph $G = (V, E)$, the dynamic connectivity problem is to maintain a graph data structure under edge insertions and deletions in order to efficiently query G to determine whether or not u and v are in the same connected component. More formally, we want to support the following operations:

- `query(u,v)` - Returns whether or not u and v are in the same connected component
- `insert(e)` - Add edge e to E
- `delete(e)` - Delete edge e from E

One simple way to support the above is to recompute the connected components of G after each update. This gives $O(m)$ time updates and $O(1)$ query time. Alternatively, we could use linear time to recompute the connected components on each `query` with constant time `insert` and `delete`. Can one do better?

First let us consider the case in which only insertions and queries are to be supported, but no deletions happen. (Such a dynamic algorithm is called *incremental*.) Then using the union-find data structure, one can support the updates and queries in $O(\alpha(n))$ time! Unfortunately, however, if deletions are to be supported, the union-find approach breaks down. Nevertheless, it turns out that we can support deletions, insertions and queries in $\tilde{O}(1)$ time. This is the subject of this lecture.

Before we state the theorem, let us talk about the runtime of data structure updates. The simplest way to measure running time is *worst case*. $O(t(n))$ worst case time operations means that every single update or query takes $O(t(n))$ time. We often measure the update time in an *amortized* sense.

Definition 2.1. *An dynamic algorithm has amortized $O(t(n))$ update time iff it takes net total $O(mt(n))$ time for m updates starting from an empty data structure.*

We will prove the following theorem.

Theorem 2.1. *There exists a data structure that supports insertions and deletions in amortized $O(\log^2 n)$ time and queries in worst case $O(\log n)$ time.*

For our motivating first attempt, we try using a spanning forest F to represent our dynamic graph G, where say each tree of F is stored in a binary search tree data structure that supposed $O(\log n)$ time merges. This gives us fast $O(\log n)$ `query` and `insert`, but a slow `delete`, as removing an edge from F might not necessarily separate two components in G so an $\Omega(m)$ search may be required to find a potential new "replacement" edge.

To make the search of replacement edges slightly easier we'll assume the existence of a special data structure, ETT.

ETT. An Eulerian Tour Tree (ETT) data structure maintains a forest F with the following $O(\log(n))$ operations:

- `insert(u,v)` - Merge the trees T_u and T_v of F where $u \in T_u$ and $v \in T_v$
- `delete(u,v)` - If $(u, v) \in T$, split T into T_u and T_v where $u \in T_u$ and $v \in T_v$[1]
- `root(u)` - Returns the root of T_u where $u \in T_u$
- `edge(u,v)` - Returns whether or not $(u, v) \in F$?
- `size(u)` - Returns the size of T_u where $u \in T_u$

Additionally, for each $u \in F$, we associate with u a set $L(u)$ such that for each $T \in F$, the ETT can return $\bigcup_{u \in T} L(u)$ in time proportional to $\sum_{u \in T} |L(u)|$. In our application actually, every u will have two sets $L_T(u)$ and $L_{NT}(u)$ so that the ETT F can return $\bigcup_{x \in T_u} L_T(x)$ when one calls $F.list_T(u)$ where T_u is the tree containing u in F, and $F.list_{NT}(u)$ returns $\bigcup_{x \in T_u} L_{NT}(x)$.

Dynamic Connectivity We begin by representing our graph G by a spanning forest F. Each edge $e \in E$ gets a level $\ell(e) \in \{0, 1, \ldots, L\}$. We show later that $L \leq \log(n)$. For all i, we let $E_i = \{e \in E | \ell(e) \geq i\}$. We let F_i be the induced subforest of F with edges in E_i; here F_i is a spanning forest of E_i. Finally, maintain each F_i in an ETT data structure. For each $u \in F_i$, we maintain two lists, $L_T^i(u) = \{(u, v) \in F_i | \ell((u, v)) = i\}$ (the tree edges incident to u of level i) and $L_{NT}^i(u) = \{(u, v) \notin F_i | \ell((u, v)) = i\}$, the non-tree level i edges incident to u. The dynamic connectivity functions are implemented as in the pseudocode for insert, delete, query and increase-level.

Algorithm 1: insert$((u, v))$

```
insert (u, v) into E;
ℓ((u, v)) ← 0;
if u and v not connected in G then
    F_0.insert(u, v);
    insert (u, v) in L_T^0(u), L_T^0(v) to be stored in F_0;
else
    insert (u, v) in L_NT^0(u), L_NT^0(v) to be stored in F_0;
```

Algorithm 2: query$((u, v))$

return *yes iff* $F_0.root(u) = F_0.root(v)$;

3 Correctness

We will prove two claims. The first states that edge levels never go above $\log n$. The second claim states that any replacement edge for a tree edge e has level at most $\ell(e)$. This latter claim asserts that when e is deleted, one only needs to look for replacements at the levels below e's level, as the current pseudocode does. It also implies that for every i, F_i is a maximum spanning forest of E_i, where the weights of the edges are their levels.

Claim 1. *$F_{\log n}$ contains no edges. That is, no edge has level $\geq \log n$.*

[1] Notice here we do not search for replacement edges.

Algorithm 3: delete((u, v))

```
ℓ ← ℓ((u, v));
if (u, v) ∉ F_0 then
    remove (u, v) from L^ℓ_NT(u), L^ℓ_NT(v);
else
    remove (u, v) from L^ℓ_T(u), L^ℓ_T(v);
    foreach i from ℓ down to 0, as long as no replacement found do
        F_i.delete(u, v);
        T_u ← tree of F_i that u is in;
        T_v ← tree of F_i that v is in;
        if |T_u| > |T_v| then
            swap u and v;
        foreach (x, y) ∈ F_i.list_T(u) do
            //Go through tree edges of level i in T_u;
            increase-level(x, y);
        foreach (x, y) ∈ F_i.list_NT(u) do
            //Go through non-tree edges of level i in T_u.;
            if y ∉ T_v then
                increase-level(x, y);
            else if no replacement found yet then
                found replacement;
                foreach j ≤ i do
                    F_j.delete(u, v);
                    F_j.insert(x, y);
```

Algorithm 4: increase-level((u, v))

```
ℓ ← level(u, v);
level(u, v) ← ℓ + 1;
if (u, v) ∈ F_0 then
    F_{ℓ+1}.insert(u, v);
    remove (u, v) from L^ℓ_T(u), L^ℓ_T(v);
    insert (u, v) into L^{ℓ+1}_T(u), L^{ℓ+1}_T(v);
else
    remove (u, v) from L^ℓ_NT(u), L^ℓ_NT(v);
    insert (u, v) into L^{ℓ+1}_NT(u), L^{ℓ+1}_NT(v);
```

Proof. To prove this claim it suffices to show that the size of any tree in F_i is $\leq n/2^i$. Then $F_{\log n}$ has only single nodes as subtrees and hence contains no edges.

The statement is true for F_0 since the largest possible tree in it has size $n \leq n/2^0$. Consider how the trees in F_i are formed. They are formed when a subtree of some tree in F_{i-1} is moved to level i due to a delete operation.

Let T_u be a subtree of some tree T in F_{i-1}, so that due to a deletion of some edge (u, v) of level $\ell \geq i-1$, all level $i-1$ tree edges of T_u are moved to level i. Consider the tree T' in F_i that these tree edges become part of. Because F_i is a subforest of F_{i-1}, T' is actually a subtree of T_u (and is hence actually T_u). We can assume inductively that before the delete operation, F_i contained trees of size $\leq n/2^i$ and that F_{i-1} contained trees of size $\leq n/2^{i-1}$. After the delete operation, the maximum size of a tree of F_{i-1} stays at most $\leq n/2^{i-1}$ and the maximum size of a tree in F_i goes up to $\leq \max\{n/2^i, |T_u|\}$. The size of T_u is however at most half of the size of T (the tree in F_{i-1} that was split), and since $|T| \leq n/2^{i-1}$, we have $|T_u| \leq n/2^i$. This completes the proof. □

Claim 2. *Suppose that (x, y) is a replacement edge for some edge (u, v) of level ℓ. Then $\ell(x, y) \leq \ell$.*

Proof. Let's look at when (x, y) first became a replacement edge of (u, v). Being a replacement edge means that there is a tree path P in F_0 between x and y that includes (u, v) and that (x, y) is a non-tree edge. (x, y) becoming a replacement edge is due to some update:

- Inserting (x, y) when the $x - y$ path P including (u, v) is already there. In this case, the level of (x, y) is set to 0 and the level of (u, v) is at least 0.
- The insertion of an edge on P into the graph cannot complete P as a tree path since then (x, y) would be there and (x, y) (or some other path of G between x and y) would be the tree path.
- The deletion of some edge e' causes the insertion of an edge $e \neq (u, v)$ of P that was a replacement for e' and also completed P. However in this case, before the deletion of e', there was a tree path between x and y including (u, v), namely the path $P \setminus \{e\}$ together with the path between the endpoints of e including e'. Hence (x, y) was already a replacement edge for (u, v) so such an update could not be the first to make (x, y) a replacement edge.
- The deletion of some edge e' causes the insertion of (u, v) to complete P, and (u, v) was a replacement edge for e'. In this case, before the update there was a tree-path including e' between x and y so that (x, y) is also a replacement edge for e'. We can assume via an induction on the number of updates done by the algorithm (base case: no updates), that $\ell(x, y) \leq \ell(e')$. When $delete(e')$ was searching for a replacement, it chose (u, v) instead of (x, y) (and both are viable since their levels were $\leq \ell(e')$), and thus $\ell(x, y) \leq \ell(u, v)$.

Now suppose that (x, y) is a replacement for (u, v) where $i = \ell(x, y) \leq \ell(u, v)$ and at some point $\ell(x, y)$ increased to $i + 1$ due to the deletion of some edge e. Now, in F_i, e split some tree into T_a and T_b, where T_a is the smaller one. Both x and y are in T_a since only replacement edges with both endpoints in the smaller tree have their levels increased by delete. This means that there is a tree path within T_a between x and y and since (u, v) is a tree edge on such a path, (u, v) must also be in T_a. Thus, either $\ell(u, v) \leq i + 1$ and so after the increase of $\ell(x, y)$ we still have $\ell(u, v) \geq \ell(x, y)$, or $\ell(u, v) = i$ and hence delete also increased $\ell(u, v)$ to $i + 1$ and again $\ell(u, v) \geq \ell(x, y)$. This finishes the proof. □

4 Runtime

Here are the proofs of runtimes.

- `query(e)` - Checking the roots of ETT costs $O(\log(n))$ for each ETT structure.
- `insert(e)` - We pay $O(\log n)$ for an insertion into the ETT and give $\Theta(\log^2(n))$ credits to e.

- `delete(e)` - Deleting e from each $F_i, i \leq \ell(e)$ and inserting its replacement edges into F_i costs $O(\log(n))$ for each of the $O(\log n)$ ETT data structures, and we give $\Theta(\log^2(n))$ credits to e to do this. Moving each edge e' up a level within delete is payed for with credit from `insert`(e').