I am thinking of taking advantage of our course web page in the following way:

1. announce problem set on tuesday as usual, but only on the web
2. expect some intrepid souls to read the web version and ask questions
3. hand out a revised version on paper in class thursday.

Thoughts?
Scribing sign-up sheet.

1 Dynamic Connectivity

Discuss history, henzinger-king.

1.1 Trees

Let's start with easy case: trees.
Insertions only easy (union find)
Deletions only

- start with all vertices labelled
- when delete edge, search smaller half, relabel
- claim: vertex relabeled $O(\log n)$ times
- proof: vertex's component halves on each relabel.
- total cost over full process (down to empty): $O(n \log n)$
- amortized $O(\log n)$ per operation, *if* finish with empty struct.

1.2 Non Trees

Deletions only non-tree.

- as before, label vertex with component
- with each vertex, store incident non-tree edges
- delete non-tree trivial
- if delete tree, must find **replacement edge**
- traverse smaller (relabeled) half
- find edge with original label on other endpoint
- note must connect to other half of broken tree
- if so, use to connect back up

Analysis.

- on failed search, tree edges get promoted.
- but note: can also promote failed non-tree edges (both endpoints in same piece)
- so, tree or non-tree, at most $\log n$ unsuccessful searches.

Problem:

- successful searches not paid for.
- must charge cost m
- but note: there was a "smaller half."
- some sampling approaches, but won't discuss
- Can we remember it somehow? Yes.

But first, a digression.

1.3 Euler Tours

Fully dynamic on trees (deletions+insertions).
Direct approach:

- just add/remove edges as inserted deleted
- great for those opps
- problem with connectivity queries: must search whole tree
- idea from union/find:
  - root tree,
  - do "find" to identify component for vertex
  - unfortunately, cost equals depth of tree
  - unlike union-find, cannot keep shallow
- solution: "encode" tree so it is shallow
  - one idea (Sleator-Tarjan): compress paths in tree.
  - simpler (Tarjan-Vishkin): represent tree as a *list*, use balanced search tree

ET structure:

- introduce Euler tour sequence
- each edge stores its two endpoint occurrences

- necessary operations: split, join, find-root on a *sequence*
- store in $2n-1$-node balanced search tree (eg splay, 2-3 tree)
- store one *active* copy of each vertex, point at from actual vertex
- supports "find" by walk up active vertex
- supports split, join by operations on tree
- time for ops: $O(\log n)$
- called *ET-tree*
- **note:** sequence is not initially ordered. Tree *imposes* order. So can't search, but who cares?
- **note:** unlike normal tree, path information is lost. Only connectivity information maintained.

1.4 Thorup's new method

Amplifies "repeated halving" concept.

- recall idea: when search a tree. look only in smaller half
- so tree edges get searched $O(\log n)$ times
- "failed searches are free" because all nontree edges move to smaller tree/different level
- thorup makes successful searches free too
- remembers smaller half, even on successful search
- $O(\log^2 n)$ time per operation

Idea:

- spanning forest F
- $L = \log n$ levels
- level i has trees of size $n/2^i$
- F_i is F intersect edges at level i and higher (to L)
- all edges (including tree edges) start at level 0, move up a level each time accessed
- so total promotions of any edge is $O(\log n)$

Data structure:

- ET-tree structures for F_i
- edges stored at (active copy of) vertex in ET-tree at their level

Invariants:

- $F_0 \supseteq F_1 \cdots F_L$ (note made up of edges from many levels)
- F_i spans all edges at level i or higher
- any tree in F_i has size at most $n/2^i$

Operations:

- query: check in F_0
- insert: add to F_0
- delete nontree: remove from current level
- delete tree:
  - remove from all L forests F_j where present
  - find replacement edge at some level,
  - add to all F_j below its level (ET-tree ops)
  - $O(\log n)$ forests, so $O(\log^2 n)$ time (modulo searching work)

Finding replacement edge:

- as before, issue to find replacement edge for e
- deleted from level i (and below)
- replacement cannot be at higher level (would violate spanning invariant for level i)
- so start search at i.
- delete e, splits ET tree in 2
- check smaller half (by size of tree) until find replacement edge
- time is size of tree plus number of failed tests
- how pay?
  - tree was $n/2^i$. took smaller half T so $n/2^{i+1}$
  - move all its tree edges up a level
  - subtlety: some of its edges might already be at higher level
  - doesn't matter: final tree still has size $n/2^{i+1}$
    - tree above was subtree of broken tree

    * so only edge leaving T's above-edges was deleted
    * so even if push T up, doesn't connect to anything else.
  - failed tests: both endpoints in T
  - so move up to next level (maintains spanning invariant)
  - Note: we don't inspect tree edges, so promotions unneccessary **except** to maintain spanning invariant.

Runtime:

- an up-level move costs $O(\log n)$
- All examinations paid for by promotions of edges
- edge promoted at most $\log n$ times
- cost per edge: $O(\log^2 n)$

Can't afford to traverse half tree, because many of its edges were already promoted.

- Problem: can't tell smaller half
- Solution: augment ET-tree to maintain size of all subtrees
- maintain on rotations/rebalances

Problem: even if know smaller, can't traverse to find level-i edges

- Instead, traverse ET tree to visit only level i edges (tree and non-tree).
- augment ET tree: in each node, store if any level-i edge below
- deduce: time $O(\log n)$ to reach per edge (skips empty subtrees)
- already paid for

Minor tweak to $\log n$-way trees gives $\log \log n$ speedup.

2 Maximum Flow

2.1 Definitions

Tarjan: *Data Structures and Network Algorithms*
Ford and Fulkerson, *Flows in Networks*, 1962 (paper 1956)
Ahuja, Magnanti, Orlin *Network Flows.* Problem: do min-cost.
Problem: in a graph, find a *flow* that is *feasible* and has maximum *value.*
Directed graph, edge *capacities* $u(e)$ or $u(v, w)$. Why not c? reserved for costs, later.
source s, *sink* t
Goal: assign a *flow* value to each edge:

- *skew symmetry:* $f(v,w) = -f(w,v)$
- *conservation:* $\sum_w f(v,w) = 0$ unless $v = s,t$
- *capacity:* $f(e) \le u(e)$ (flow is *feasible/legal*)

Alternative formulation: no skew symmetry

- *conservation:* $\sum_w f(v,w) = 0$ unless $v = s,t$
- *capacity:* $0 \le f(e) \le u(e)$ (flow is *feasible/legal*)

Equivalence: second formulation has "gross flow" g, first has "net flow" $f(v,w) = g(v,w) - g(w,v)$. To go other way, sign of f defines "direction" of flow in g.
We'll focus on net flow model for now.
Flow *value* $|f| = \sum_w f(s,w)$ (in net model).
Water hose intuition. Also routing commodities, messages under bandwidth constraints, etc. Often "per unit time" flows/capacities.
Maximum flow problem: find flow of maximum value.
Path decomposition (another picture):

- claim: any s-t flow can be decomposed into paths with quantities
- proof: induction on number of edges with nonzero flow
- if s has out flow, find an s-t path (why can we? conservation) and kill
- if some vertex has outflow, find a cycle and kill
- corollary: flow into t equals flow out of s (global conservation)

Cuts:

- partition of vertices into 2 groups
- s-t-cut if one has s, other t
- represent as $(S, \overline{S})$ or just S
- $f(S)$ = net flow leaving S
- lemma: for any s-t cut, $f(S) = |f|$ (all cuts carry same flow)

$$\begin{aligned}
|f| &= \sum_{v \in S} \sum_w f(v,w) && \text{(flow conservation)} \\
&= \sum_{e \in S \times S} f(e) + \sum_{e \in S \times \overline{S}} f(e) && \text{(skew)} \\
&= \sum_{e \in S \times \overline{S}} f(e)
\end{aligned}$$

Flows versus cuts:

- Deduce: $|f| \leq u(S) = \sum_{e \in S \times \overline{S}} c(e)$.
- in other words, max-flow $\leq$ minimum s=t cut value.
- soon, we'll see *equal*
- first, need more machinery.

Residual network.

- Given: flow f in graph G
- define G_f to have capacities $u'_e = u_e - f_e$
- if f feasible, all capacities positive
- Since f_e can be negative, some residual capacities **grow**
- Suppose f' is a feasible flow in G_f
- then $f + f'$ is feasible flow in G of value $f + f'$
  - flow
  - feasible
- Suppose f' is feasible flow in G
- then $f' - f$ is feasible flow in G_f (value —f'—-—f—)
- **corollary**: max-flows in G correspond to max-flows in G_f
- Many algorithms for max-flow:
  - find some flow f
  - recurse on G_f

How can we know a flow is maximum?

- check if residual network has 0 max-flow
- augmenting path: s-t path of positive capacity in G_f
- if one exists, not max-flow

Max-flow Min-cut

- Equivalent statements:
  - f is max-flow
  - no augmenting path in G_f
  - $|f| = u(S)$ for some S

Proof:

- if augmenting path, can increase f
- let S be vertices reachable from S in G_f. All outgoing edges have $f(e) = u(e)$
- since $|f| \leq u(S)$, equality implies maximum