Algorithm
Design and Analysis

Union-Find (More Amortized Analysis!)

Roadmap for today

- Design the ***Union-Find*** data structure for the ***disjoint sets problem***
- Practice ***potential functions*** by analyzing Union-Find

Motivation: Kruskal's Algorithm

Review (Minimum Spanning Tree): A spanning tree of an undirected graph with the least total (edge) cost of all possible spanning trees

Review (Kruskal's Algorithm): For each edge (u, v) in sorted order by cost, add the edge to the spanning tree if u and v are not connected.

How do we do that part??

The disjoint-sets problem

Problem (Disjoint Sets): We want to support the following API:

- **MakeSet**$(\boldsymbol{x})$: Create a set consisting of the single element $\{x\}$
- **Find**$(\boldsymbol{x})$: Return the *representative element* of the set containing x
- **Union**$(\boldsymbol{x}, \boldsymbol{y})$: Merge the two sets $S_x \ni x$ and $\mathrm{S_y} \ni y$ into a single set.

How to use it for Kruskal's?

The disjoint-set forest data structure

- ***Key idea***: Represent the sets as **trees**. Use the roots of the trees as the representative element.
- Representation: Store a parent pointer for each node. Roots have no parent (by convention, $p(x) = x$ for roots).

Implementation (basic version)

- **MakeSet**(x):
- **Find**(x):
- **Union**(x, y):
- **Link**(x, y)

Performance

Theorem: Let n be the current number of elements in the sets (i.e., the number of MakeSet operations performed so far). There exists inputs for which every find costs $\Theta(n)$.

Making Union better?

- The bad performance was caused by long chains of nodes…
- Can we just… not do that?

Idea (Union-by-size): When performing a Union, make the smaller tree a child of the larger tree. If they're the same size, then pick arbitrarily.

- We should store an extra field $s(x)$ that knows the size of the trees
- $s(x)$ is the size of the tree rooted at x (we don't care about non-roots)

Union-by-size implementation

- **Link(x, y):**

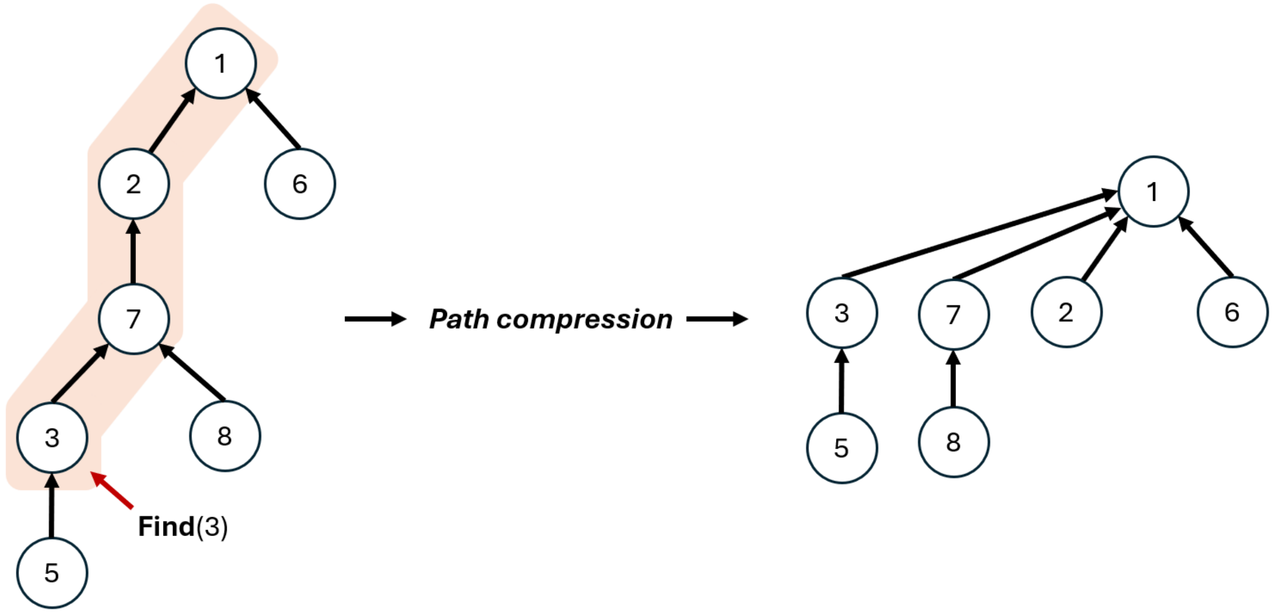
Performance of union-by-size

Theorem: Let n be the current number of elements in the sets. Using *union-by-size*, every Link operation costs $O(1)$ and every Find operation costs $O(\log n)$ **worst-case**.

Another improvement

- We just made Union better. Can we instead/also make Find better?

Idea (Path compression): When performing a Find, point every node along the path at its current root/representative element.



Path compression implementation

- **Find**(x):

Cost model for amortization

- To avoid arbitrary constants in the analysis, we will once again work in a simplified cost model. All our analyses will be asymptotically valid in the word RAM up to constant factors.
  - **MakeSet** costs 1
  - **Link** costs 1
  - **Find** costs number of nodes touched

- **Goal**: Amortized costs of $O(\log n)$ for each operation

Performance of path compression

Theorem: Let n be the current number of elements in the sets. Using *path compression* (but not union-by-size), the amortized cost of MakeSet is 1, Link is $(1 + \log n)$, and Find is $(2 + \log n)$.

- **Observation**: ***Balance*** is what matters
- Balanced trees are always fast, imbalanced trees are slow
- How do we measure how balanced a tree is at a ***per-node basis***?

Balanced or imbalanced?

Definition (heavy/light): Given a node u and its parent p, call a node:

1. ***Heavy*** if $\mathrm{size}(u) > \frac{1}{2}\,\mathrm{size}(p)$, i.e., u contains a majority of p's descendants
2. ***Light*** if $\mathrm{size}(u) \leq \frac{1}{2}\mathrm{size}(p)$, i.e., u contains at most half of p's descendants

- Root is neither heavy nor light (it has no parent)
- In a perfectly balanced tree, every node is light (except root)
- In a chain (the worst-balanced tree), every node is heavy (except root)

Balanced or imbalanced?

Lemma (heavy/light): On any root-to-leaf path in any tree of n nodes, there are at most $\log n$ light nodes.

Reaching your potential

- Find costs $(1 + \#\text{heavy} + \#\text{light})$.
- We know that $\#\text{light} \leq \log n$
- So, we don't need to amortize the light nodes. We are happy to just pay directly for the light nodes.
- We want to define a ***potential function*** that will save up and **pay for the cost of touching the heavy nodes**

Reaching your potential

- ***Observation***: A node can have ***at most one heavy child***
- What happens when you remove (compress) a node's heavy child?
- A different child might become the heavy child!
- But how many times can this happen?

Reaching your potential

- Define our potential function to be:

$$\Phi(F) =$$

- **Nice properties**:
  - Initially zero (all trees start at size 1)
  - Always non-negative
  - Increases when we perform a Link
  - Decreases when we perform a Find

no debt

Links save up $$$ to pay for Finds

Analysis of MakeSet

Lemma (cost of MakeSet): MakeSet does not change $\Phi(F)$

Corollary: The MakeSet operation has an amortized cost of 1

Analysis of Link

Lemma (cost of Link): A link operation at most increases $\Phi(F)$ by $\log n$

Corollary: A link operation has amortized cost at most $1 + \log n$

Analysis of Find

Lemma: A Find operation decreases $\Phi(F)$ by at least $\#\text{heavy nodes} - 1$

- Consider ***heavy nodes*** u with parent p (other than r) on the **Find** path

Analysis of Find

Corollary (cost of Find): Find has amortized cost at most $(2 + \log n)$

Summary of Union-Find complexity

- ***Union-Find with union by size***:
  - ***Link***: $O(1)$
  - ***Find***: $O(\log n)$ **worst-case**
- ***Union-Find with path compression***:
  - ***Link***: $O(\log n)$ **amortized**
  - ***Find***: $O(\log n)$ **amortized**
- ***Union-Find with both! (Not proven in this class)***:
  - ***Link***: $O\big(\alpha(n)\big)$ **amortized**
  - ***Find***: $O\big(\alpha(n)\big)$ **amortized**
  - $\Omega\big(\alpha(n)\big)$ is also a lower bound so this is optimal!