Union-Find

In this lecture we describe the *disjoint-sets* problem and the family of *union-find* data structures. This is a problem that captures (among many others) a key task one needs to solve in order to efficiently implement Kruskal's minimum-spanning-tree algorithm. We describe several variants of the union-find data structure and prove that they achieve good amortized costs.

Objectives of this lecture

In this lecture, we will

- Design a very useful data structure called *Union-Find* for the *disjoint sets* problem
- Practice amortized analysis using potential functions

Recommended study resources

- CLRS, *Introduction to Algorithms*, Chapter 21 (3rd edition) or Chapter 19 (4th edition), Data Structures for Disjoint Sets

1 Motivation

To motivate the disjoint-sets/union-find problem, let's recall Kruskal's Algorithm for finding a minimum spanning tree (MST) in an undirected graph. Remember that an MST is a tree that includes all the vertices and has the least total cost of all possible such trees.

Kruskal's Algorithm:

> Sort the edges in the given graph G by weight and examine them from lightest to heaviest. For each edge (u,v) in sorted order, add it into the current forest if u and v are not already connected.

Today, our concern is how to implement this algorithm efficiently. The initial step takes time $O(|E|\log|E|)$ to sort. Then, for each edge, we need to test if it connects two different components. If it does, we will insert the edge, merging the two components into one; if it doesn't (the two endpoints are in the same component), then we will skip this edge and go on to the next edge. So, to do this efficiently we need a data structure that can support the basic operations of (a) determining if two nodes are in the same component, and (b) merging two components together. This is the *disjoint-sets* or *union-find* problem.

2 The Disjoint-Sets / Union-Find Problem

The general setting for the union-find problem is that we are maintaining a collection of disjoint sets $\{S_1, S_2, \ldots, S_k\}$ over some universe. Each set will have a *representative element* (or *canonical element*) that is used to identify it. The representative element is arbitrary, but it is important is that it is consistent so that we can identify whether two elements are in the same set.

Interface: Union-Find

The disjoint-sets/union-find API consists of:

MakeSet(x): Create a new set containing the single element x. Its representative element is x. (x must not be in another set.)

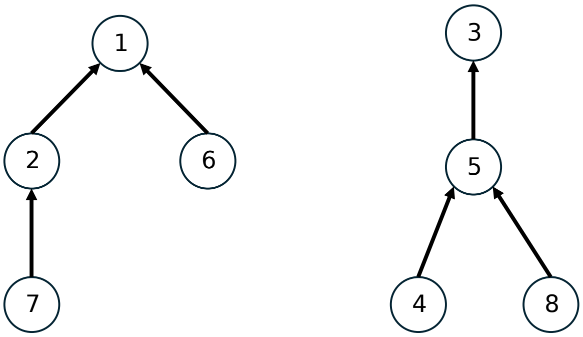
Find(x): Return the representative element of the set containing x.

Union(x, y): x and y are representative elements of two different sets. This operation forms a new set that is the union of these two sets, and removes the two old sets.

Implementing Kruskal's Algorithm Given these operations, we can implement Kruskal's algorithm as follows. The sets S_i will be the sets of vertices in the different trees in our forest. We begin with MakeSet(v) for all vertices v (every vertex is in its own tree). When we consider some edge (v, w) in the algorithm, we first compute v' =Find(v) and w' =Find(w). If they are equal, it means that v and w are already in the same tree so we skip over the edge. If they are not equal, we insert the edge into our forest and perform a Union(v', w') operation. All together we will do $|V|$ MakeSet operations, $|V|-1$ Unions, and $2|E|$ Find operations.

2.1 A Tree-Based Data Structure

We will represent the collection of disjoint sets by a forest of rooted trees. Each node in a tree corresponds to one of the elements of a set, and each set is represented by a separate tree in the forest. The root of the tree is the representative element of the corresponding set. We will refer to the total number of elements in all of the sets (i.e., the number of nodes in the forest) as n.



The trees are defined by parent pointers. So every node x has parent $p(x)$. A node x is a root of its tree if $p(x) = x$. To find the representative element of a set, it suffices to walk up the tree until we reach a root node, which must be the answer. To union two trees, we can simply make one tree a child of the other. To do this, we can first find the roots of the two trees by calling Find, and then setting one as the child of the other.

It will be convenient for our cost analysis to distinguish between the cost incurred by Union when it calls Find, and the actual cost of the step that joins the two trees. To do so, we define a subroutine **Link**, which is not part of the API but just used internally by Union. With all of this set up, here is a basic implementation.

Algorithm: Union-Find Forests

Union-Find forests (a.k.a. disjoint-set forests) implement the API as follows:

MakeSet(x): Create node x and set $p(x) \leftarrow x$.

Find(x): Starting from x, follow the parent pointers until you reach the root, r, then return r.

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

Link(x, y): Set $p(y) \leftarrow x$.

We are going to analyze several variants of this data structure.

Analysis with no optimizations The vanilla version of the data structure unfortunately has terrible worst-case (even in an amortized analysis) performance. Too see this, suppose we create n sets and then Union them into a long chain of length n. Now every find operation on the bottom-most element of the chain costs $\Theta(n)$.

We will now explore two optimizations that substantially improve the performance of the data structure: *union-by-size* and *path compression*.

3 The union-by-size optimization

Our first optimization, and the simpler of two to analyze is *union-by-size*. The pathological cost of the no-optimization example was caused by the fact that we were able to build a long chain of nodes with the Union operation, allowing all subsequent Finds to be very expensive.

Key Idea: Union by size optimization

When performing the Union (Link) operation, always make the smaller (by number of nodes) tree the child of the larger tree.

To implement this, we augment each element x with an additional field $s(x)$. If x is a root of a tree, then $s(x)$ contains the size of the subtree rooted at x. If x is not the root of a tree, then we do not care about the value of $s(x)$ since it would be too expensive to update $s(x)$ on every single node, and its value is never needed on nodes other than roots.

Algorithm: Link with union-by-size optimization

Augment each element x with a field $s(x)$. MakeSet(x) sets $s(x) \leftarrow 1$.

```
Link(x, y):  - if $s(x) < s(y)$ then swap$(x, y)$,
             - Set $p(y) \leftarrow x$,
             - Set $s(x) \leftarrow s(x) + s(y)$.
```

With this minor change to the algorithm, the claim is that we now get great performance! Even better, we don't need amortized analysis for this one, we get a worst-case bound.

Theorem 1: Union by size performance

Consider the union-find forest data structure where the union-by-size optimization is used. Then MakeSet and Link each cost $O(1)$ and Find has a worst-case cost of $O(\log n)$.

Proof. MakeSet and Link each perform a constant number of operations regardless of the input, so they they cost $O(1)$.

To analyze the cost of Find, we observe that every edge of the tree is created by the union of two sets, and using union by size, the smaller tree must have always been made into a child of the larger one. Therefore, whenever the Find algorithm moves from a node to its parent, the total size of the rooted at the current node *at least doubles*. Since no tree has size more than n, the number of edges on any path to the root is at most $\log n$, so any Find operation costs at most $O(\log n)$ in *the worst case*. □

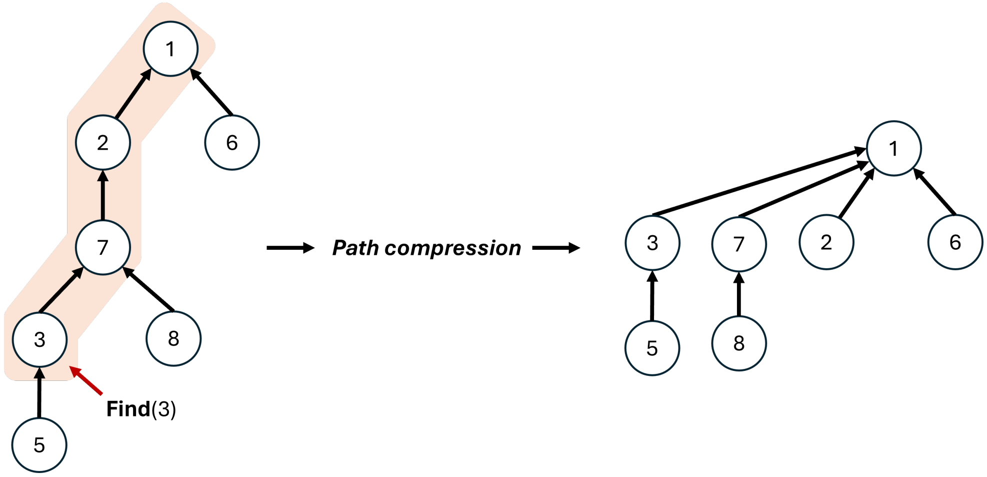
4 The path compression optimization

In the previous section, we improved the worst-case performance of Find by improving the Union operation, but leaving the Find operation unchanged. What if we instead want to improve the Find operation itself? Assuming worst-case Unions, is there a way to make the Find operation $O(\log n)$ cost on its own? With the power of amortized analysis, the answer is yes, and the technique is elegant and intuitive (though the analysis is hard!)

The worst case for a Find operation is encountering a long root-to-leaf path. When that happens, we can't avoid having to traverse it to the root, **but**, we can at least try to make the situation better for future Find operations that happen to come along one of the same nodes.

Key Idea: Path compression optimization

Each time the data structure performs a Find operation, it changes the parent of every node encountered on the path to point directly to the root so that future Finds do not have to travel the same long path.



Unlike union by size, this does not require augmenting the data structure with any additional fields, and the implementation is nice and elegant.

Algorithm: Path compression implementation

Find with path compression can be implemented elegantly with recursion.

```
Find(x):  - if $p(x) \neq x$ then set $p(x) \leftarrow$ Find($p(x)$),
          - return $p(x)$
```

Note that we are not combining union by size with path compression (yet). We are just using path compression with worst-case unions. The claim is that this algorithm is also efficient, but we are going to need amortized analysis to prove it. The idea is that a single Find operation can be inefficient (the worst-case cost is still the same), but after performing an inefficient Find, the forest must become shallower as a result, making future Finds more efficient. To make the analysis clean, we will define a simple cost model that we will use.

Cost model We will analyze path compression under the cost scheme:

- **MakeSet**(x) costs 1,
- **Link**(x, y) costs 1,
- **Find**(x) costs the number of nodes on the path from x to the root.

Note that these costs are accurate up to constant factors in the word RAM, so our results will be asymptotically valid in the word RAM, but without having to deal with the arbitrary constants.

Theorem 2: Path Compression without Union by Size

Consider the forest-based union-find data structure where path compression is applied but union by size is not. Then the amortized cost of Link is $(1+\log n)$, the amortized cost of Find is $(2+\log n)$, and Makeset still costs 1.

The proof is not as simple as the proof for union by size, and uses a very non-trivial potential function. Before diving right into it, lets try to get a handle on how we might measure the potential. The main observation that we need is that *balance* defines the difference between an efficient Find and an inefficient Find (which must therefore be paid for by the stored potential). A very balanced tree can have low potential since operations will be cheap anyway, but an imbalanced tree will need a lot more potential to pay for the Finds.

Heavy and light nodes While balance appears to be the key thing that we care about, a particular Find operation doesn't care how balanced the tree is as a whole, it only cares about the specific nodes along the $x \to r$ path that it encounters. So what we need is a way of measuring how balanced a tree is on a per-node level. In other words, what would it mean for a single node to be good or bad for balance?

In a balanced tree, all of the children of a particular node x would have an even share of x's descendants as their descendants. Nodes that deviate significantly from this (such as a long chain of nodes) will break this. This motivates us to define the concept of **heavy** and **light** nodes. Let $\text{size}(x)$ be the number of nodes that are descendants of x (including x).

Definition: Heavy and light nodes

In a tree, a node u (other than the root) with parent $p(u)$ is called:

1. **heavy** if $\text{size}(u) > \frac{1}{2}\text{size}(p(u))$, i.e., u's subtree has a majority of $p(u)$'s descendants,
2. **light**, if $\text{size}(u) \le \frac{1}{2}\text{size}(p(u))$, i.e., u's subtree has at most half of $p(u)$'s descendants.

A perfectly balanced tree would have no heavy nodes, only light nodes (except for the root). A maximally imbalanced tree (a chain) would consist entirely of heavy nodes (except for the root). We have the following very useful observation about heavy and light nodes.

Lemma 1: Heavy-light lemma

On any root-to-leaf path in a tree on n vertices, there are at most $\log n$ light nodes.

Proof. Consider a node x. If x is light, then by definition, $\text{size}(p(x)) \ge 2\text{size}(x)$. Since there are no more than n nodes in any tree, the number of times one can double the size of the current node is at most $\log n$. Therefore there are at most $\log n$ light nodes on any root-to-leaf path. □

This gives us an idea. We want the amortized cost of Find to be $O(\log n)$, and it costs us 1 for every node we touch on the root-to-leaf path. So, we can afford to traverse a path of light nodes, since there are at most $\log n$ of them. We only get sad when we have to touch heavy nodes, because there could be a lot of them, so lets try to use a *potential function* to save up and pay for the heavy nodes!

So, how many heavy nodes can there be? Well, a lot, unfortunately, but what about how many heavy *children* can a particular node have? By definition, a node can only have one heavy child at a time since a heavy child must contain a majority of the descendants! When a Find operation path compresses a heavy child, another child *might* become a heavy child in its place. However, the node must therefore *lose over half of its descendants*, sine the heavy child was a majority of the subtree. This can not happen very many times. In particular, a node u with $\text{size}(u)$ descendants can only halve its size $\log(\text{size}(u))$ times before it has no children anymore!

Therefore, the number of heavy children a single node could ever have is bounded by $\log(\text{size}(u))$, so the total number of heavy children we could *ever have in our tree* is

$$\Phi(F) = \sum_{u \in F} \log(\text{size}(u)),$$

which will be our potential function! As usual, log is base-2. It is important to note that this sum is over *every node in the forest*, not only the roots. This potential function is quite powerful and is also used in the analysis of other data structures, such as Splay Trees. It is, at some intuitive level, a very good potential function for measuring *how imbalanced a tree is*. It can be shown as an exercise that a perfectly balanced tree on n vertices has $\Theta(n)$ potential, while a long chain (the most imbalanced tree) has $\Theta(n \log n)$ potential.

This potential has a few nice important properties:

1. The potential is initially zero (all trees have size 1),
2. at any point in time the potential is non-negative,
3. the potential increases every time a Link is done,
4. and it decreases (or stays the same) every time a Find is done.

The first two properties mean that we can use the total amortized cost to bound the actual total cost. The last two properties tell us intuitively that the potential should be on the right track for the analysis, since union is the cheap operation (which we therefore want to make more expensive by paying into the potential) and find is the expensive operation, which we want to make cheaper by withdrawing from the potential.

Analysis of MakeSet Calling **MakeSet**(x) creates a new singleton set $\{x\}$ represented by a node with no parent or children and does not modify any of the existing sets. Therefore the potential contributed by all of the existing nodes does not change, and we just get one new term in the potential from the new node.

$$\Delta\Phi_{\text{MakeSet}} = \log(\text{size}(x)) = \log(1) = 0.$$

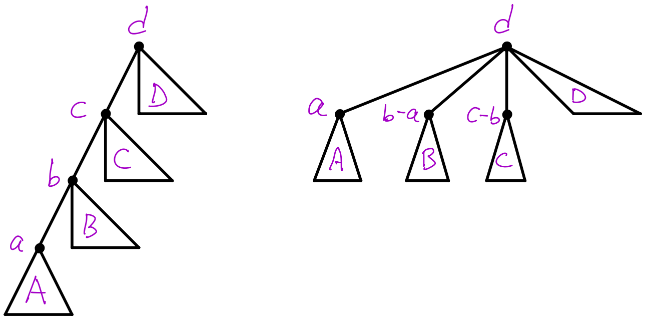
Since we just created it, the size of x's subtree is just one, and $\log 1 = 0$, so the potential does not change at all. Therefore the actual and amortized cost of **MakeSet** is 1.

Analysis of Link Consider a Link operation. Suppose the operation links a node y to a node x. The only node whose size changes is x. Let size(x) be the size of x before the Link and size$'(x)$ be the size after the Link. We know that size$(x) \geq 1$, because any tree has at least one node in it. So $\log(\text{size}(x)) \geq 0$. Similarly size$'(x) \leq n$ and $\log(\text{size}'(x)) \leq \log n$. So we have:

$$\begin{aligned}\Delta\Phi_{\text{Link}} &= \Phi_{\text{final}} - \Phi_{\text{initial}},\\ &= \log(\text{size}'(x)) - \log(\text{size}(x)),\\ &\leq \log n\end{aligned}$$

Since the actual cost of a Link is 1, the amortized cost of Link is at most $1 + \log n$.

Analysis of Find Let's consider the Find operation. The following figure shows a typical Find with path compression. On the left is the tree before and on the right is after. The triangles labeled with capital letters represent arbitrary trees, which are unchanged. A Find operation is applied to the node labeled a on the left. The cost of this operation is 4 (it touches 4 nodes).



The find path passes through vertices a, b, c, and d. These labels are the sizes of the nodes. Note that the node labeled b has size b before the Find, and has size $b - a$ after the Find. Similarly the node labeled c has size c before the Find and has size $c - b$ after the Find. Those are the only two nodes whose size changes as a consequence of the Find. In general, all nodes except for the first and last have their size decreased, while the first and last remain the same. Since sizes only decrease, the potential from every node can only decrease, never increase!

There are (1+# heavy nodes+# light nodes) on the Find path (the root is neither heavy nor light). We recall from Lemma 1 that there are at most $\log n$ light nodes on the path, so we are happy to pay for those in the final cost. Our goal is therefore to cancel out the cost of the heavy nodes using the potential function. Given any path from a vertex to its corresponding root node, we consider all of the *heavy nodes* on the path except the child of the root (because it does not move as a result of path compression). For every such vertex u, whose parent we will denote as p, the new size of p, which we will denote as size$'(p)$, is given by size$'(p) = \text{size}(p) - \text{size}(u)$.

By the definition of heavy, u and p satisfy size$(u) > \frac{1}{2}$size(p), so from this equation, we have

$$\begin{aligned}\text{size}'(p) &= \text{size}(p) - \text{size}(u),\\ &< \text{size}(p) - \frac{1}{2}\text{size}(p),\\ &< \frac{1}{2}\text{size}(p),\end{aligned}$$

i.e., the size of p at least halves! This means that the change in potential at node p is

$$\begin{aligned}\Delta\Phi_{\text{at node } p} &= \log(\text{size}'(p)) - \log(\text{size}(p)),\\ &< \log\left(\frac{1}{2}\text{size}(p)\right) - \log(\text{size}(p)),\\ &= -1.\end{aligned}$$

Therefore, the potential of every node (other than the root) whose child on the Find path is heavy *decreases by at least one*. The overall decrease in the potential is at least (# heavy nodes−1) and hence the amortized cost of a find operation is

$$\begin{aligned}\text{Amortized cost of Find} &\le \underbrace{1 + \#\text{ heavy nodes} + \#\text{ light nodes}}_{\textbf{actual cost}} - \underbrace{(\#\text{ heavy nodes} - 1)}_{\Delta\Phi},\\ &\le 2 + \#\text{ light nodes},\\ &\le 2 + \log n.\end{aligned}$$

5 Both optimizations at once

This problem has been extensively analyzed. In the 1970s, a series of upper bounds were proven for the amortized cost of Union and Find, going from $\log n$ to $\log\log n$ to $\log^* n$, and finally to $\alpha(n)$. Bob Tarjan proved the final result and matched it with a corresponding lower bound, thus "closing" the problem and showing that $O(1)$ time is impossible.

Here $\log^* n$ denotes the function that counts the number of times you have to apply log to n until it doesn't exceed 1. The function $\alpha(n)$ is an inverse of Ackermann's function, and grows insanely slowly[1], even slower than $\log^* n$. We won't be studying the proofs of these results since that would take another entire lecture, but they are useful results to know so that you can analyze the runtime of algorithms that use Union-Find as an ingredient.

Theorem 3: Union-Find with union-by-size and path compression

Consider the forest-based union-find data structure with both path compression and the union-by-size optimizations. Then the amortized cost of MakeSet is $O(1)$, and the amortized costs of Link and Find are $O(\alpha(n))$.

The proof uses a very complicated potential function. If you want to read it, you can find it in CLRS (Chapter 19 in the fourth edition, or Chapter 21 in the third edition).

[1] For n up to the number of particles in the universe, $\alpha(n) \le 4$, so while it may not be a constant for theoretical purposes, in practice it is indistinguishable from a small constant

Exercises: Union Find

Problem 1. Consider the union-find forest data structure with path compression. Suppose we perform a sequence of n MakeSet operations, followed by m Unions, then f Finds *in that order*, i.e., the operations are not interleaved. Show that the total cost of this sequence of operations is $O(n+m+f)$, i.e., each operation takes constant amortized time. (Hint: define a potential function that is the degree of the root of the tree.)

Problem 2. Show that the potential function from Section 4 is $\Theta(n)$ for a perfectly balanced tree on n vertices and $\Theta(n \log n)$ for a chain of n vertices.

Problem 3. Suppose we implement the union-find forest data structure with both union-by-size and path compression. Show, using the same potential function as Section 4 that the cost of Union is $O(1)$.

Problem 4. Consider the union-find forest data structure with union by size. Show that the worst-case bound of $O(\log n)$ cost per Find operation is tight, i.e., describe an input for which the Find operation costs $\Omega(\log n)$.

Problem 5. Consider the union-find forest data structure with path compression. Show that the amortized bound of $O(\log n)$ cost per Union and Find operation is tight, i.e., describe an input with m Union operations and f Find operations for which the total cost is at least

$$\Omega(m \log n + f \log n).$$