# 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 then give several data structures for union-find 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, 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 length and examine them from shortest to longest. Put each edge into the current forest if it doesn't form a cycle with the edges chosen so far.

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 *union-find* problem.

## 2 The 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 *canonical element* (or *representative element*) that is used to identify it.

***Interface: Union-Find***

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

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

**Find**($x$): Return the canonical element of the set containing $x$.

**Union**($x, y$): $x$ and $y$ are canonical 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.

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.

In the discussion below, it will be convenient to define

- $n$ as the number of MakeSet operations and
- $m$ as the total number of Union and Find operations

## 3 A Simple 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 the collection of sets, and each set is represent by a separate tree in the forest. The root of the tree is the canonical element of the corresponding set.

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 $p(x) = x$. The nodes also contain a size field, where the size of $x$ is denoted size($x$). For any root node $x$, the invariant will be maintained that the number of nodes in the tree rooted at $x$ is size($x$).

**Cost model:** We will use a simple cost model where each main operation on the data structure costs 1, except for the Find operation which walks up a tree, and costs proportional to the number of nodes it visits.

***Algorithm: Union-Find Forests***

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

**MakeSet**($x$): Set size($x$) to 1 and $p(x)$ to itself. This costs 1.

**Find**($x$): Starting from $x$, follow the parent pointers until you reach the root, $r$. This is the canonical element of the set. It would be sufficient for correctness to return this as the answer.

However, to make the method more efficient we then apply the following additional step called *path compression*. We make an additional pass from $x$ to the root. For every node along this path we assign its parent pointer to $r$.

The cost for Find($x$) is the number of nodes traversed, i.e., the number of nodes (originally) on the path from $x$ to its root.

**Union**($x, y$): We know that the nodes $x$ and $y$ are canonical elements of their respective sets, and are therefore roots of their respective trees. It would be sufficient for correctness to at this point just do the assignments

$$p(y) \leftarrow x \qquad\qquad \text{size}(x) \leftarrow \text{size}(x) + \text{size}(y)$$

But instead we use the size fields to link the smaller tree to the larger tree. (I.e. if size($x$) < size($y$) we reverse the role of $x$ and $y$ in the assignments above.)

This heuristic is called *Union by size*. This operation costs 1.

We will analyze four variants of this data structure, with and without each heuristic.

## 3.1 Analysis with neither union by size nor path compression

It turns out that we need *at least* one of the heurisitcs (path compression or union by size) to achieve good cost bounds. We leave it as an exercise to the reader to prove that the vanilla algorithm with neither heurisitc can perform as badly as $\Omega(mn)$.

## 3.2 Analysis with union by size but not path compression

Adding one of the two heuristics, we start to obtain good costs. Of the four combinations, this one is the most straightforward to analyze and doesn't actually require any amortization.

***Theorem 1: Union by Size without Path Compression***

Consider the forest-based union-find data structure where the union-by-size heuristic is used, but path compression is not. Then MakeSet and Union each have a constant cost of 1 and Find has a worst-case cost of $\log n$.

*Proof.* MakeSet and Union cost 1 by definition in our cost model, and we will not charge any amoritzed cost to them. 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 subtree *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*. □

The total cost of this algorithm in the setting if we perform $n$ MakeSets and $m$ Unions or Finds is therefore at most $O(n + m \log n)$.

## 3.3 Analysis with path compression but not union by size

An interesting special case of this scenario is when all the unions are done before all the finds. In that case, the cost is $O(n + m)$, which the reader can prove as an exercise. In the general scenario, it's still the case that the algorithm costs $O((n + m) \log n)$.

***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 Union is $(1 + \log n)$, the amortized cost of Find is $(2 + \log n)$, and Makeset still costs 1. So in terms of $n$ and $m$ the cost is $O(n + m \log n)$.

*Proof.* Its time to bring in some amortized analysis using the potential function method. At any point in time there is a forest of trees $F$. Each node $x$ has a size $s(x)$ that is defined as the current number of nodes in the subtree rooted at $x$. Note that this is different from size($x$), which is only kept up-to-date for the root elements. $s(x)$ on the other hand always refers to the *true current size* of the node $x$, which might not be stored in the data structure. The potential function is then defined as:

$$\Phi(F) = \sum_{x \in F} \log s(x).$$

Note that the potential here is summed over *all nodes*, not only the root notes of the forest. 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 Union is done,
4. and it decreases (or stays the same) every time a Find is done.

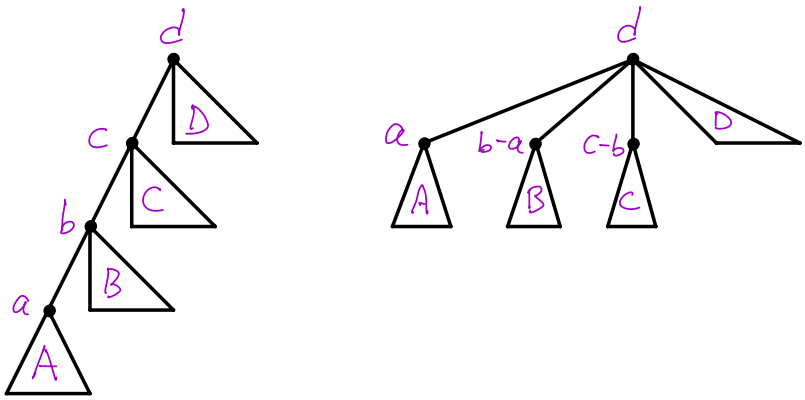
The first two properties mean that we can use the total amortized cost to bound the actual total cost. The last two properties tell us intuitively that the potential should be on the right track for the analysis, since union is the cheap operation (which we therefore want to make more expensive by paying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

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

$$\Delta\Phi_{\text{due to Union}} = \Phi_{\text{final}} - \Phi_{\text{initial}} \leq \log n$$

Since the actual cost of a union is 1, the amortized cost of Union 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.

Given any path from a vertex to its corresponding root node, we cosider all of the vertices on the path except the top two. For every such vertex $u$, whose parent we will denote as $p$, the new size of $p$, which we will denote as $s'(p)$, is given by

$$s'(p) = s(p) - s(u). \tag{1}$$

Notice, importantly, that these changes only cause the potential to decrease!

To analyze the total cost of the Find operation, recall that the actual cost is the number of nodes on the path from $x$ to its root. We can therefore think of each node as contributing one to the cost, and analyze the nodes separately. We will first charge the cost of top two nodes on the find path (e.g., $c$ and $d$) to the Find itself, i.e., we do not amortize them (because the potential of the root does not change, and the root itself has no parent). To analyze the remaining cost, we divide the nodes into two categories. Given a node $u$ and its parent $p$,

1. if $2s(u) > s(p)$, i.e., $u$'s subtree contains a strict majority of $p$'s descendants, call $u$ *big*,
2. else, if $2s(u) \leq s(p)$, i.e., $u$'s subtree contains at most half of $p$'s descendants, call $u$ *small*.

For the small nodes, we are going to analyze them directly without amoritzation. Here is the key observation: since, if a node is small, its parent is at least twice its size, there can be no more than $\log n$ small nodes on any given path to a root! This is because after doubling the size $\log n$ times, we would have to be at a node with size $n$, which would have to contain every node in the entire forest. Since there are at most $\log n$ small nodes, and visiting each of them contributes 1 to the cost of the find operation, we can just accept this cost as is. We don't need to try to amortize it away since we are happy for the final cost to be $\log n$.

It remains to show that we can amortize the cost of the big nodes. We will do so by charging the cost of a big node to the decrease in potential of its parent. Recall that a big node $u$ and its parent $p$ satisfy $2s(u) > s(p)$, so using (1), we have

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

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

$$\begin{aligned} \Delta\Phi &= \log(s'(p)) - \log(s(p)), \\ &< \log\left(\frac{1}{2}s(p)\right) - \log(s(p)), \\ &= -1. \end{aligned}$$

Therefore, the potential of any node whose child is big *decreases by at least one*, which counterbalances the cost of touching the big nodes, so the amortized cost of a find operation is

$$\begin{aligned} \text{Amortized cost of Find} &= \underbrace{2 + \#\text{ big nodes} + \#\text{ small nodes}}_{\text{actual cost}} - \underbrace{\#\text{ big nodes}}_{\Delta\Phi}, \\ &\leq 2 + \log n. \end{aligned}$$

□

Lastly, we can note down the total cost of $n$ MakeSets and $m$ Unions and Finds. Since we have $\Phi_{\text{initial}} = 0$, and $\Phi_{\text{final}} \geq 0$, we know that the total cost of any operation set is at most the total amortized cost, and hence the total cost is $O(n + m\log n)$.

This upper bound turns out to be tight. There's an example with $n$ Makesets and $n$ Unions and Finds which incurrs a cost of $\Omega(n\log n)$. The construction first builds a fibonacci tree of $n/2 = 2^k$ nodes, then proceeds to do a Find (of $\log(n/2)$ steps) and then an unbalanced union which almost exactly reproduces the same tree. So you repeat this $n/2$ times giving the result.

## 3.4 Analysis with path compression and union by size

This problem has been extensively analyzed. In the 1970s, a series of upper bounds of the form $O(n+mf(n))$ were proven, where $f(n)$ shrank from $\log\log n$ 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. 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. We will prove the $O(n+m\log\log n)$ upper bound here[1]. Our proof is based on recent research of Richard Peng, et.al[2].

The proof is very similar in structure to the proof of Theorem 2, except that we use a different potential function. As before we define $s(x)$ to be the size of a node $x$. We start our analysis using the following potential function:

$$\Phi(F)=\sum_{x\in F}\sqrt{s(x)}$$

We've replaced the $\log(s(x))$ by $\sqrt{s(x)}$. As before, Unions cause the potential to increase, and Finds cause it to decrease (or stay the same). Since all nodes have size at least 1, it is the case that the minimum value of this potential is $n$. We first prove some lemmas about the behavior of the Union-Find algorithm in terms of this potential. We'll need to get an upper bound how much the potential can increase. The first lemma does this.

***Lemma 1***

Starting from the initial $n$ disjoint sets, the potential can increase by at most $\frac{n}{\sqrt{2}}$.

*Proof.* We've already established that that sizes of nodes can only increase as a result of a Union. And which trees are allowed to be joined by the Union is not affected by the intervening Finds. Therefore for our purposes here we can just ignore the Finds and think about how big the potential could get by doing Unions starting from $n$ separate singleton trees.

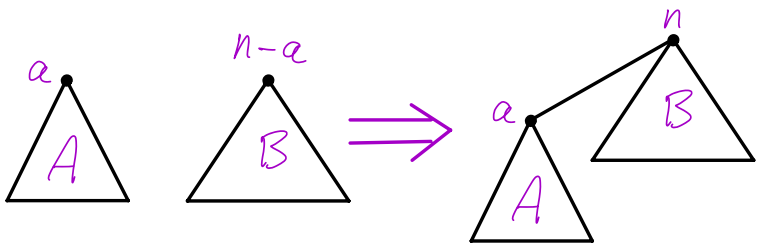
To this end, let's make this definition:

$f(n)=$ The greatest potential increase that can occur when going from $n$ separate singleton nodes to a tree of $n$ nodes.

To write a recurrence for $f()$ consider the following picture. Suppose that trees $A$ and $B$ of sizes $a$ and $n-a$ have been built in a way that has the greatest total potential (for trees of those sizes). Now we link them together as shown. The change in potential caused by this link is $\sqrt{n}-\sqrt{n-a}$.

[1] For context, if $N$ is the number of atoms in the universe, then $\log\log(N)\simeq 8$.

[2] The proof appears in a recent paper by Zhiyi Huang, Chris Lambert, Zipei Nie, and Richard Peng `https://arxiv.org/pdf/2308.09021`




This gives us the following recurrence:

$$f(n) = \begin{cases} 0 & \text{if } n = 1 \\ \max\limits_{1 \le a \le n/2} f(a) + f(n-a) + \sqrt{n} - \sqrt{n-a} & \text{otherwise} \end{cases}$$

We're going to do a proof by induction. The induction hypothesis is that for all $i < n$ we have that $f(i) \le c * (i - \sqrt{i})$ for some contstant $c$. The base case of $i = 1$ holds for any constant $c$. So what we need to complete the induction for $n$ is that the following holds for all $1 \le a \le \frac{n}{2}$:

$$c(a - \sqrt{a}) + c(n - a - \sqrt{n-a}) + \sqrt{n} - \sqrt{n-a} \;\le\; c(n - \sqrt{n}).$$

Rearranging and cancelling terms, this is equivalent to:

$$\sqrt{n} - \sqrt{n-a} \;\le\; c(\sqrt{a} + \sqrt{n-a} - \sqrt{n}).$$

The term multiplied by $c$ is non-negative because $\sqrt{a} + \sqrt{n-a} \ge \sqrt{n} \Leftrightarrow (\sqrt{a} + \sqrt{n-a})^2 \ge n$ $\Leftrightarrow a + 2\sqrt{a}\sqrt{n-a} + n - a \ge n$, which is clearly true. So we can use this inequality to bound $c$ as follows:

$$\frac{\sqrt{n} - \sqrt{n-a}}{\sqrt{a} + \sqrt{n-a} - \sqrt{n}} \le c.$$

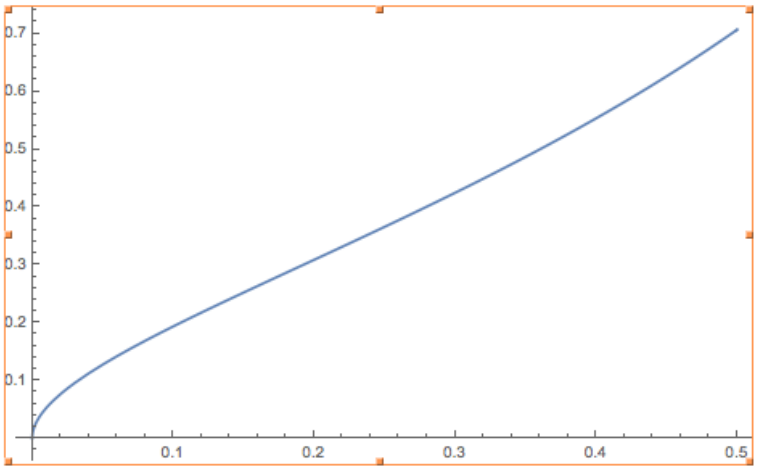
To determine the maximum value of the expression on the left, we can divide through the numerator and denominator by $\sqrt{n}$, then replace $a/n$ by $x$, where $0 < x \le 1/2$. This gives:

$$\frac{1 - \sqrt{1-x}}{\sqrt{x} + \sqrt{1-x} - 1} \le c.$$

I graphed this function in Mathematica:

```
In[20]:= Plot[(1 - Sqrt[1 - x]) / (Sqrt[x] + Sqrt[1 - x] - 1), {x, 0, .5}]
```

Out[20]=




The maximum of the function occurs when $x = 1/2$, and its value is $\sqrt{1/2}$. So with $c = \sqrt{1/2}$ the induction proof goes through. It follows that

$$f(n) \leq \frac{1}{\sqrt{2}}(n - \sqrt{n}) < \frac{n}{\sqrt{2}}.$$

□

For the analysis of Find, we again classify the find steps into three categories: (1) top two nodes on the path, (2) small nodes, which are defined as those where $a < \sqrt{b}$, and (3) big nodes, which are defined as those where $a \geq \sqrt{b}$. (As in the proof of Theorem 2, $a$ here denotes the size of the node and $b$ denotes the size of its parent.)

***Lemma 2***

The total number of small nodes on a find path is at most $1 + \lceil \log \log n \rceil$.

*Proof.* Consider the operator $x \to x^2$. Let $f(n)$ be the number of times you have to apply that operator (starting from 2) to get a number that is at least $n$. It is the case that $f(n) = \lceil \log \log n \rceil$. (You can prove it by induction.).

Consider the find path that starts at node $a$ whose parent is $b$. If we start from the second to bottom node $b$ on the find path, its size is at least 2. So the number of small nodes on the find path starting from $b$ is at most $\lceil \log \log n \rceil$. The bottom node may also be small, so the bound becomes $1 + \lceil \log \log n \rceil$. □

***Lemma 3***

Each big node on the find path causes the potential decreases by at least $\frac{1}{2}$.

*Proof.* Looking back at the figure from Theorem 2, if $a \geq \sqrt{b}$. The $\Delta\Phi$ of this step is $\sqrt{b-a}-\sqrt{b}$. We will show that

$$\sqrt{b-a}-\sqrt{b} < -\frac{1}{2}.$$

This is equivalent to

$$\sqrt{b}-\sqrt{b-a} > \frac{1}{2}.$$

This would follow if (because $\sqrt{b} \leq a$).

$$\sqrt{b}-\sqrt{b-\sqrt{b}} = \sqrt{b}-\sqrt{\sqrt{b}\left(\sqrt{b}-1\right)} > \frac{1}{2}.$$

But we know that $\sqrt{b}(\sqrt{b}-1) < (\sqrt{b}-\frac{1}{2})^2$ (Because $x(x-1) < (x-\frac{1}{2})^2$ for any $x > 1$.) So our result will follow if

$$\sqrt{b}-\sqrt{\left(\sqrt{b}-\frac{1}{2}\right)^2} \geq \frac{1}{2}.$$

The last statement is true, because the left hand size is $\frac{1}{2}$. □

Finally, we can use all this to prove the following theorem.

***Theorem 3: Path Compression with Union by Size***

Consider the forest-based union-find data structure where path compression and union by size are applied. Then $n$ Makesets and $m$ Unions and Finds have total cost bounded by $O(n + m \log\log n)$.

*Proof.* We're going to use a potential that is twice the one we were working with above. Here's the definition:

$$\Phi'(F) = 2\Phi(F) = 2\sum_{x \in F} \sqrt{s(x)}.$$

Lemma 1 tell us that the Unions can increase $\Phi'$ by at most $\sqrt{2}n$. Thus the amortized cost of the unions is thus at most $n-1+\sqrt{2}n$. Lemma 2 tells us that the cost of the all the small nodes on find path is at most $m(1+\lceil\log\log n\rceil)$. Lemma 3 tells us that the amortized cost of the big nodes on find path have zero amortized cost. The top nodes contribute $2m$ to the cost. Lastly, the Makesets cost $n$.

So putting all of these together, along with the fact that the final potential is at least the initial potential, we have that $n$ makesets and $m$ Unions and Finds costs at most

$$n+n-1+\sqrt{2}n+m(1+\lceil\log\log n\rceil)+2m = -1+(2+\sqrt{2})n+(3+\lceil\log\log n\rceil)m = O(n+m\log\log n).$$

□

# Exercises: Union Find

**Problem 1.** Consider the most basic version of the forest-based union find data structure which performs neither union by size nor path compression. Show a sequence of $n$ MakeSet and $m$ Union and Find operations such that the cost incurred is $\Omega(mn)$.

**Problem 2.** Consider a version of the forest-based union find data structure with the path compression heuristic but no union by size, and suppose we process a set of $m$ Union and Find operations such that all of the Unions come before all of the Finds. Prove that the cost in this case is $O(n+m)$. (Hint: define a potential function that is the degree of the root of the tree.)