CSE 373: Disjoint sets continued

Michael Lee

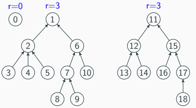
Friday, Mar 2, 2018

Warmup

Consider the following disjoint set.

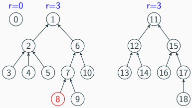
What happens if we run `findSet(8)` then `union(4, 17)`?

Note: the `union(...)` method internally calls `findSet(...)`.



Warmup

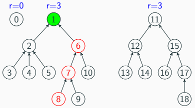
What happens when we run `findSet(8)`?



Step 1: We find the node corresponding to 8 in $\mathcal{O}(1)$ time

Warmup

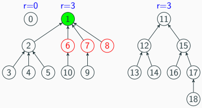
What happens when we run `findSet(8)`?



Step 2: We travel up the tree until we find the root

Warmup

What happens when we run `findSet(8)`?

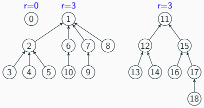


Step 3: We move each node we passed by (every red node) to point directly at the root.

Note: we do not update the rank (too expensive)

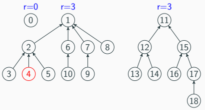
Warmup

What happens if we run `union(4, 17)`?



Warmup

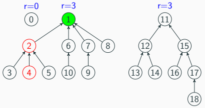
What happens if we run union(4, 17)?



Step 1: We first run findSet(4)

Warmup

What happens if we run union(4, 17)?

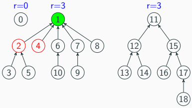


Step 1: We first run findSet(4).

So we need to crawl up and find the parent...

Warmup

What happens if we run union(4, 17)?



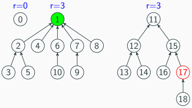
Step 1: We first run findSet(4).

So we need to crawl up and find the parent...

...and make node "4" point directly at the root.

Warmup

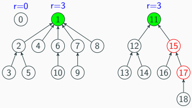
What happens if we run union(4, 17)?



Step 2: We next run findSet(17) and repeat the process.

Warmup

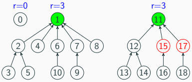
What happens if we run union(4, 17)?



Step 2: We next run findSet(17) and repeat the process.

Warmup

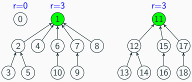
What happens if we run union(4, 17)?



Step 2: We next run findSet(17) and repeat the process.

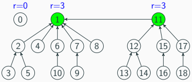
Warmup

We've finished `findSet(4)` and `findSet(17)`, so now we need to finish the rest of `union(4, 17)` by linking the two trees together.



Warmup

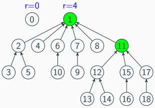
We've finished `findSet(4)` and `findSet(17)`, so now we need to finish the rest of `union(4, 17)` by linking the two trees together.



The ranks are the same, so we arbitrarily make set 1 the root and make set 11 the child.

Warmup

We've finished `findSet(4)` and `findSet(17)`, so now we need to finish the rest of `union(4, 17)` by linking the two trees together.



We then update the rank of set 1 and "forget" the rank of set 11.

Path compression: runtime

Now, what are the worst-case and best-case runtime of the following?

- **`makeSet(x)`**:
  $\mathcal{O}(1)$ – still the same
- **`findSet(x)`**:
  In the best case, $\mathcal{O}(1)$, in the worst case $\mathcal{O}(\log(n))$
- **`union(x, y)`**:
  In the best case, $\mathcal{O}(1)$, in the worst case $\mathcal{O}(\log(n))$

Back to Kruskal's

Why are we doing this? To help us implement Kruskal's algorithm:

```
def kruskal():
    for (v : vertices):
        makeMST(v)

    sort edges in ascending order by their weight

    mst = new SomeSet<Edge>()
    for (edge : edges):
        if findMST(edge.src) != findMST(edge.dst):
            union(edge.src, edge.dst)
            mst.add(edge)

    return mst
```

- `makeMST(v)` takes $\mathcal{O}(t_m)$ time
- `findMST(v)`: takes $\mathcal{O}(t_f)$ time
- `union(u, v)`: takes $\mathcal{O}(t_u)$ time

Back to Kruskal's

We concluded that the runtime is:

$$\mathcal{O}\left(\underbrace{|V|\cdot t_m}_{\text{setup}} + \underbrace{|E|\cdot\log(|E|)}_{\text{sorting edges}} + \underbrace{|E|\cdot t_f + |V|\cdot t_u}_{\text{core loop}}\right)$$

Well, we just said that in the worst case:

- $t_m \in \mathcal{O}(1)$
- $t_f \in \mathcal{O}(\log(|V|))$
- $t_u \in \mathcal{O}(\log(|V|))$

So the worst-case overall runtime of Kruskal's is:

$$\mathcal{O}\left(|V| + |E|\cdot\log(|E|) + (|E| + |V|)\cdot\log(|V|)\right)$$

Back to Kruskal's

Our worst-case runtime:

$$\mathcal{O}(|V| + |E| \cdot \log(|E|) + (|E| + |V|) \cdot \log(|V|))$$

One minor improvement: since our edge weights are numbers, we can likely use a *linear sort* and improve the runtime to:

$$\mathcal{O}(|V| + |E| + (|E| + |V|) \cdot \log(|V|))$$

We can drop the $|V| + |E|$ (they're dominated by the last term):

$$\mathcal{O}((|E| + |V|) \cdot \log(|V|))$$

...and we're left with something that's basically the same as Prim.

Disjoint-sets, amortized analysis

...or are we?

Observation: each call to `findSet(x)` improves all future calls. How much of a difference does that make?

Interesting result:

It turns out `union` and `find` are *amortized* $\log^*(n)$.

Disjoint-sets, amortized analysis

Iterated log

The expression $\log_b^*(n)$ is equivalent to the number of times we repeatedly compute $\log_b(x)$ to bring x down to at most 1.

What does this mean?

Interlude: repeated exponentiation

Observation:

- Multiplication is a shorthand for repeated addition*

$$2 \times 5 = 2 + 2 + 2 + 2$$

- Exponentiation is a shorthand for repeated multiplication*

$$2^5 = 2 \times 2 \times 2 \times 2 \times 2$$

- Is there a way of expressing repeated exponentiation?

$$2 \,??\, 5 = 2^{2^{2^{2^2}}}$$

- Why stop there – is there a way of expressing repeated whatever-it-is-we-did up above?

$$2 \,??!!???\, 5 = 2 \,??\, 2 \,??\, 2 \,??\, 2 \,??\, 2$$

*assuming we use only integers

Interlude: Knuth's up-arrow notation

Yes – it's called **Knuth's up-arrow notation**

- Repeated addition (multiplication) is still the same:

$$2 \times 5 = 2 + 2 + 2 + 2$$

- A single arrow means *repeated multiplication* – exponentiation

$$2 \uparrow 5 = 2 \times 2 \times 2 \times 2 \times 2 = 2^5 = 16$$

- Two arrows means *repeated exponentiation* – tetration

$$2 \uparrow\uparrow 5 = 2 \uparrow 2 \uparrow 2 \uparrow 2 \uparrow 2 = 2^{2^{2^{2^2}}}$$

- Three arrows means *repeated tetration*

$$2 \uparrow\uparrow\uparrow 5 = 2 \uparrow\uparrow 2 \uparrow\uparrow 2 \uparrow\uparrow 2 \uparrow\uparrow 2$$

- etc...

Interlude: Knuth's up-arrow notation

These functions all also have *inverses*

- Division is the inverse of multiplication:

$$\frac{(2 \times 5)}{2} = 5$$

- $\log(...)$ is the inverse of $\uparrow$ (exponentiation)

$$\log_2(2 \uparrow 5) = \log_2(2^5) = 5$$

- $\log^*(...)$ is the inverse of $\uparrow\uparrow$ (tetration)

$$\log_2^*(2 \uparrow\uparrow 5) = \log_2^*(2^{2^{2^{2^2}}}) = 5$$

Up-arrows and iterated log

A slightly modified definition:

Iterated log

The expression $\log^*_b(n)$ is equivalent to the number of times we repeatedly compute $\log_b(x)$ to bring x down to at most 1.

This is equivalent to the inverse of $b \uparrow\uparrow x$.

What does this look like?

- $\log^*(2 \uparrow\uparrow 1) = \log *(2) = \log(2) = 1$
- $\log^*(2 \uparrow\uparrow 2) = \log^*(2^2) = \log(\log(4)) = 2$
- $\log^*(2 \uparrow\uparrow 3) = \log^*(2^{2^2}) = \log(\log(\log(8))) = 3$
- $\log^*(2 \uparrow\uparrow 4) = \log^*(2^{2^{2^2}}) = \log(\log(\log(\log(65536)))) = 4$
- $\log^*(2 \uparrow\uparrow 5) = \log^*(2^{2^{2^{2^2}}}) =$ $\log(\log(\log(\log(\log(2^{65536}))))) = 5$

A big number

And what exactly is 2^{65536}?

= 2003529930406846464979072351560255750447825475569751419265016973710894059556311453089506130880933348101038234342907263181822949382118812668869506364761547029165041871916351587966347219442930927982084309104855990570159318959639524863372367203002916969592156108764948889254090805911457037675208500206671563702366126359747144807111774815880914135742720967190151836282560618091458852699826141425030123391108273603843767876449043205960379124490905707560314035076162562476031863793126484703743782954975613770981604614413308692118102485959152380195331030292162800160568670105651646750568038741529463842244845292537361442533614373729088303794601274724958414864915930647252015155693922628180691650796381064132275307267143998158508811292628901134237782705567421080070065283963322155077831214288551

A big number

Note: in the interests of saving space, the handouts only contain the first 800 or so digits of the number.

We've omitted the remaining digits, which take up an additional 20 slides.

A big number

If we count, $2 \uparrow\uparrow 5$ has **19729 digits**!

And yet, $\log^*(2 \uparrow\uparrow 5)$ equals just 5!

Punchline? $\log^*(n) \leq 5$, for basically any reasonable value of n.

Runtime of Kruskal?

$$\mathcal{O}\left((|E| + |V|)\log^*(|V|)\right) \leq \mathcal{O}\left((|E| + |V|)5\right) \approx \mathcal{O}\left(|E| + |V|\right)$$

Inverse of the Ackermann function

But wait!

Somebody then came along and proved an even tighter bound. It turns out `findSet(...)` and `union(...)` are amortized $\mathcal{O}\left(\alpha(n)\right)$ – the inverse of the Ackermann function.

The Ackermann function

The Ackermann function is a recursive function designed to grow extremely quickly:

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

This function grows even more quickly then $m \uparrow\uparrow n$ – this means the inverse Ackermann function $\alpha(...)$ grows even more slowly then $\log^*(...)$!

So, the runtime of Kruskal's is even better! It's

$$\mathcal{O}\left((|E| + |V|)\alpha(|V|)\right) \leq \mathcal{O}\left((|E| + |V|)4\right)$$

...for any practical size of $|V|$.

Are we done yet?

But wait, there's more!

Recap

To recap, we found that the runtimes of `findSet(...)` and `union(...)` were...

- Originally $\mathcal{O}(n)$
- After applying **union-by-rank**, $\mathcal{O}(\log(n))$
- After applying **path compression**, $\mathcal{O}(\alpha(n)) \approx \mathcal{O}(1)$
- One final optimization: **array representation**.
  It doesn't lead to an asymptotic improvement, but it does lead to a constant factor speedup (and simplifies implementation).

Array representation

So far, we've been thinking about disjoint sets in terms of nodes and pointers.

For example:

```java
private static class Node {
    private int vertexNumber;
    private Node parent;
}
```

Observation: It seems wasteful to have allocate an entire object just to store two fields

Array representation

Java is technically allowed to store and represent its objects however it wants, but in a modern 64-bit JDK, this object will probably be 32 bytes:

- The int field takes up 4 bytes
- The pointer to the parent takes up 8 bytes (assuming 64-bit)
- The object itself also uses up an additional 16 bytes
- This adds up to 28, but in a 64 bit computer, we always "pad" or round up to the nearest multiple of 8. So, this object will use up 32 bytes of memory.

Array representation

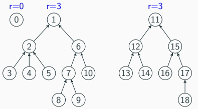
Idea: Just use an array of ints instead!

Core idea:

- Make the index of the array be the vertex number
- Make the element in the array be the index of the parent

Array representation

Example:



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
-	-	1	2	2	2	1	6	7	7	7	-	11	12	12	11	15	15	17

Array representation

So, rather then using 32 bytes per element, we use just 4!

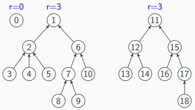
Question: Where do we store the ranks?

Observation: Hey, each root has some unused space...

Idea 1: Rather then leaving the root cells empty, just stick the ranks there.

Array representation

Example:



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	3	1	2	2	2	1	6	7	7	7	3	11	12	12	11	15	15	17

What's wrong with this idea?

Array representation

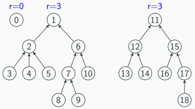
Problem: How do we tell whether a number is supposed to be a rank or an index to the parent?

A trick: Rather then storing just the rank, let's store the negative of the rank!

So, if a number is positive, it's an index. If the number is negative, it's a rank (and that node is a root).

Array representation

Example:



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
-0	-3	1	2	2	2	1	6	7	7	7	-3	11	12	12	11	15	15	17

What's wrong with this idea?

Array representation

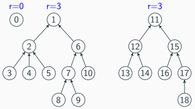
Problem: What's the difference between 0 and -0?

Solution: Instead of just storing $-\text{rank}$, store $-\text{rank} - 1$.

(Alternatively, redefine the rank to be the upper bound of the number of *levels* in the tree, rather then the *height*.)

Array representation

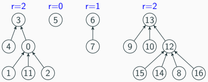
Example:



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
-1	-4	1	2	2	2	1	6	7	7	7	-4	11	12	12	11	15	15	17

Array representation

Now you try – what is the array representation of this disjoint set?



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
3	0	0	-3	3	-1	-2	6	12	13	13	0	13	-3	12	12	12

Recap

And that's it for graphs. Topics covered:

- Graph definitions, graph representations
- Graph traversal: BFS and DFS
- Finding the shortest path: Dijkstra's algorithm
- Topological sort
- Minimum spanning trees: Prim's and Kruskal's
- Disjoint sets

Recap

Next time: What does it mean for a problem to be “hard”?