CSE 373: Disjoint sets

Michael Lee
Wednesday, Feb 28, 2018

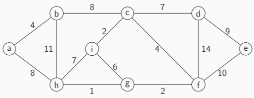
Review

Last time...

- **Prim's algorithm:**
  Nearly identical to Dijkstra's, except we use the distance to any already-visited node as the cost.
- **Kruskal's algorithm:**
  Loop over edges, from smallest to largest. Use the edge only if it doesn't introduce a cycle.

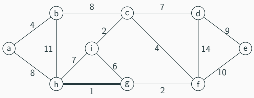
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



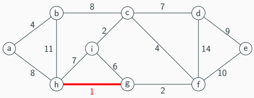
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



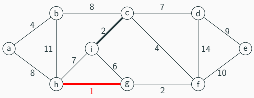
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



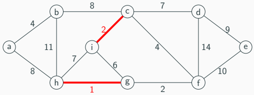
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



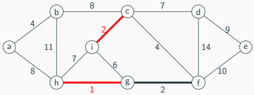
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



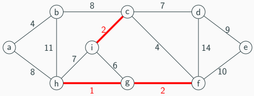
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



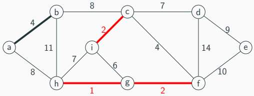
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



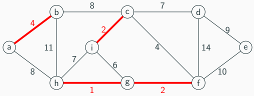
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



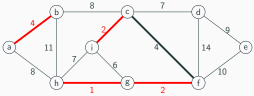
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



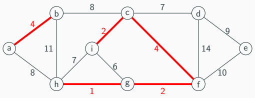
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



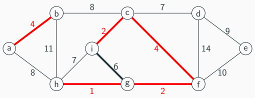
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



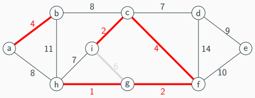
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



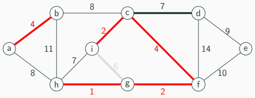
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



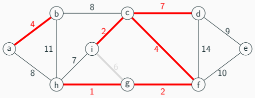
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



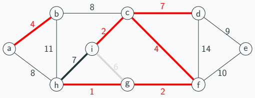
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



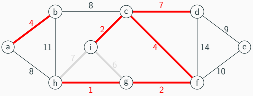
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



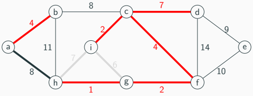
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



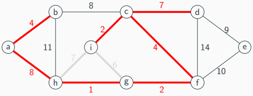
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



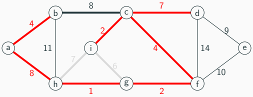
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



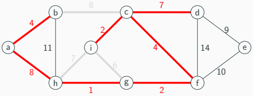
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



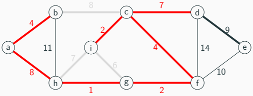
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



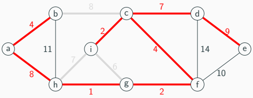
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



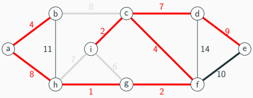
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



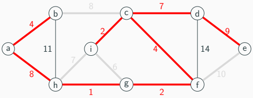
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



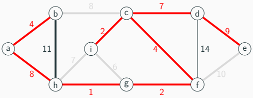
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



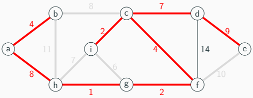
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



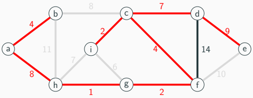
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



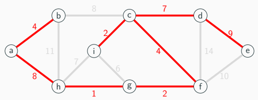
Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



Kruskal's algorithm: example with a weighted graph

Example of the algorithm:



Kruskal's algorithm: analysis

Runtime analysis:

```
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
```

Note: assume that...

- `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

Kruskal's algorithm: analysis

- Making the $|V|$ MSTs takes $\mathcal{O}(|V| \cdot t_m)$ time
- Sorting the edges takes $\mathcal{O}(|E| \cdot \log(|E|))$ time, assuming we use a general-purpose comparison sort
- The final loop takes $\mathcal{O}(|E| \cdot t_f + |V| \cdot t_u)$ time

Putting it all together:

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

The DisjointSet ADT

But wait, what exactly is t_m, t_f, and t_u? How exactly do we implement `makeMST(v)`, `findMST(v)`, and `union(u, v)`?

We can do so using a new ADT called the `DisjointSet` ADT!

Interlude: What is a set?

Review: what is a set?

- A set is a "bag" of elements arranged in no particular order.
- A set may not contain duplicates.

We implemented a set in project 2: `ChainedHashSet`

Interesting note: sets come up all the time in math.

The DisjointSet ADT

Properties of a disjoint-set data structure:

- A disjoint-set data structure maintains a collection of many different sets.
- An item **may not** be contained within multiple sets. Each set must be *disjoint*.
- Each set is associated with some *representative*. What is a representative? Any sort of unique "identifier". Examples:
  - We could pick some arbitrary element in the set to be the "representative"
  - We could assign each set some unique integer id.

The DisjointSet ADT

A disjoint-set has the following core operations:

- **`makeSet(x)`** – Creates a new set where the only member is *x*. We assign that set a representative.
- **`findSet(x)`** – Looks up the set containing *x*. Then, returns the representative of that set.
- **`union(x, y)`** – Looks up the set containing *x* and the set containing *y*. We combine these two sets together into one. We (arbitrarily) pick one of the two representatives to be the representative of this new set.

The DisjointSet ADT

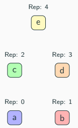
Example:

```
makeSet(a)
makeSet(b)
makeSet(c)
makeSet(d)
makeSet(e)
```


The DisjointSet ADT

Example:

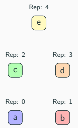
```
makeSet(a)
makeSet(b)
makeSet(c)
makeSet(d)
makeSet(e)
```



The DisjointSet ADT

Example:

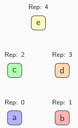
```
makeSet(a)
makeSet(b)
makeSet(c)
makeSet(d)
makeSet(e)

print(findSet(a))
print(findSet(d))
```



The DisjointSet ADT

Example:

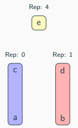
```
makeSet(a)
makeSet(b)
makeSet(c)
makeSet(d)
makeSet(e)

print(findSet(a))
print(findSet(d))

union(a, c)
union(b, d)
print(findSet(a) == findSet(c))
print(findSet(a) == findSet(d))
```



The DisjointSet ADT

Example:

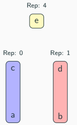
```
makeSet(a)
makeSet(b)
makeSet(c)
makeSet(d)
makeSet(e)

print(findSet(a))
print(findSet(d))

union(a, c)
union(b, d)
print(findSet(a) == findSet(c))
print(findSet(a) == findSet(d))
```



The DisjointSet ADT

Example:

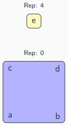
```
makeSet(a)
makeSet(b)
makeSet(c)
makeSet(d)
makeSet(e)
print(findSet(a))
print(findSet(d))
union(a, c)
union(b, d)
print(findSet(a) == findSet(c))
print(findSet(a) == findSet(d))
union(c, b)
print(findSet(a) == findSet(d))
```



The DisjointSet ADT

Example:

```
makeSet(a)
makeSet(b)
makeSet(c)
makeSet(d)
makeSet(e)
print(findSet(a))
print(findSet(d))
union(a, c)
union(b, d)
print(findSet(a) == findSet(c))
print(findSet(a) == findSet(d))
union(c, b)
print(findSet(a) == findSet(d))
```



The DisjointSet ADT

What operations does a disjoint-set **NOT** support?

Answer: The ability to actually get the entire set.

We can *make* a set, *check* if an item is in a set, and *combine* two sets, but we don't have a built-in way of *getting* the entire set itself.

Insight: The few operations we need to support, the more creative our implementation can be.

(If the client really wants the sets, they can get it themselves in $\mathcal{O}(n)$ time – how?)

DisjointSet: implementation

So, how do we implement these?

Core idea:

- We represent each set as a tree
- The disjoint-set keeps track of a "forest" of trees

Intuitions:

- We want union-ing to be cheap.
  Combining two trees is cheap; we just manipulate pointers.
- We want a single "representative" per set.
  A tree has a single root!

DisjointSet: implementation

High-level overview:

- **`makeSet(x)`**: Adds a new tree (of size 1) to our "forest"
- **`findSet(x)`**: Looks up the node, then finds root of tree
- **`union(x, y)`**: Combines two trees into one

DisjointSet: implementation

Suppose we call `makeSet(...)` on 0 through 5.

Each `makeSet(...)` adds a new tree to our "forest".

Note that right now, each tree has only one element.

DisjointSet: implementation

Suppose we call `union(3, 5)`.

We combine those two trees into one.

Assumption: we have an $\mathcal{O}(1)$ way of getting each node.
(E.g. maintain a hashmap of numbers to node objects.)

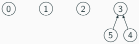
Question: how do we implement `findSet(...)`?

Once we find a node, move upwards until we're looking at root.

Then, return the root's data field.

DisjointSet: implementation

Suppose we call `union(5, 4)`.

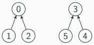


Algorithm: Find the roots of both trees and add one tree as a subchild of the other.

Which tree becomes the new root? For now, pick randomly.

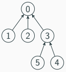
DisjointSet: implementation

Suppose we call `union(0, 1)`, then `union(2, 0)`.



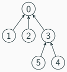
DisjointSet: implementation

Now, suppose we call `union(2, 4)`. What happens?



DisjointSet: implementation

Now, suppose we call `union(2, 4)`. What happens?



We look up 2 and 3, find their roots, and nest one tree inside the other

DisjointSet: Analysis

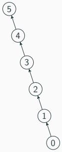
What's the worst-case runtime of our methods?

Better question: are our trees guaranted to be balanced?

Hint: When union-ing, we pick which tree is nested randomly.
Does that guarantee we'll get a balanced tree?

DisjointSet: Analysis

The worst-case scenario:



Possible outcome of calling union(0, 5)

DisjointSet: implementation

So, what are the worst-case runtimes?

- **makeSet(x)**:
  $\mathcal{O}(1)$ – creating the tree takes constant time
- **findSet(x)**:
  $\mathcal{O}(n)$ – if it's a linked list, we need to traverse *n* elements!
- **union(x, y)**:
  $\mathcal{O}(n)$ – union calls findSet(...) on both elements

...where *n* is the total number of items added to the disjoint-set.

Improving DisjointSet

How can we improve disjoint sets?

1. **Union-by-rank:**
   Strategy to make sure trees are balanced
2. **Path compression:**
   Hijack findSet(x) and make it do a little extra work to improve overall performance.
3. **Array representation:**
   Takes advantage of cache locality, simplifies implementation, etc.

Union-by-rank

Problem: Our trees could be unbalanced

Solution:

Let rank(x) be a number representing the upper-bound of the height of x. So, rank(x) $\geq$ height(x).

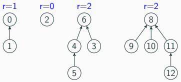
We then...

1. Keep track of the rank of all trees.
2. When unioning, make the tree with the larger rank the root!
3. If it's a tie, pick one randomly and increase the rank by one.

(Why not keep track of the height? When we look at path compression, keeping track of the height becomes more challenging.)

Union-by-rank

Example: Suppose we call union(1, 5)?

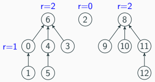


The tree with the root of "6" has the larger rank, so we make it the root.

Note: we're not really "removing" the rank from node 0 – it's just

Union-by-rank

Example: Suppose we call union(1, 5)?

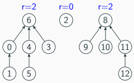


The tree with the root of "6" has the larger rank, so we make it the root.

Note: we're not really "removing" the rank from node 0 – it's just

Union-by-rank

Example: Suppose we call `union(1, 5)`?

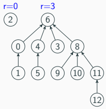


The tree with the root of "6" has the larger rank, so we make it the root.

Note: we're not really "removing" the rank from node 0 – it's just irrelevant, so we're ignoring it and omitting it from the diagram to save space. We only care about the ranks at the roots.

Union-by-rank

Example: Suppose we call `union(5, 11)`?



Here, there's a tie. We break the tie arbitrarily, and increment the rank of the new tree by one.

Union-by-rank

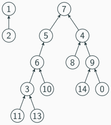
Net effect? Our trees stay relatively balanced.

So, what are the worst-case runtimes now?

- **`makeSet(x)`**:
  $\mathcal{O}(1)$ – still the same
- **`findSet(x)`**:
  $\mathcal{O}(\log(n))$ – since the tree is balanced
- **`union(x, y)`**:
  $\mathcal{O}(\log(n))$ – since union calls `findSet`

Path compression

Consider the following forest:

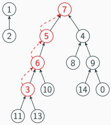


Suppose we call `findSet(3)` a few hundred times.

Why do we have to keep finding the root again and again?

Path compression

Observation: To find root, we must also traverse these nodes:

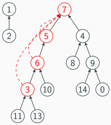


What if, next time, we could just jump straight to the root?

Same for the other nodes we visited

Path compression

Observation: To find root, we must also traverse these nodes:

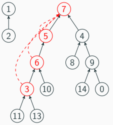


What if, next time, we could just jump straight to the root?

Same for the other nodes we visited

Path compression

Observation: To find root, we must also traverse these nodes:

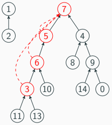


What if, next time, we could just jump straight to the root?

Same for the other nodes we visited

Path compression

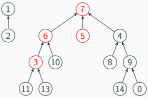
So, let's do it!



Now what happens if we try calling `findSet(3)`?

Path compression

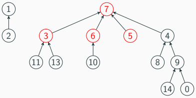
So, let's do it!



Now what happens if we try calling `findSet(3)`?

Path compression

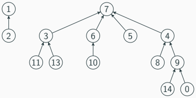
So, let's do it!



Now what happens if we try calling `findSet(3)`?

Path compression

So, let's do it!



Now what happens if we try calling `findSet(3)`?

Path compression

One additional note: path compression changes the heights of our trees.

This means it could be the case that rank $\neq$ height.

Is this a problem?

Answer: No; proof is beyond the scope of this class

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:

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

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 $|V| + |E|$, since 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's algorithm.

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 $\log^*(n)$ is equivalent to the number of times you need to compute $\log(x)$ to bring the value down to at most 1

Example:

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

A big number

What is 2^{65536}?

2^{65536} =

2003529930406846464979072351560255750447825475569751419
26501697371089405955631145308950613088093334810103823434290726
31818229493821188126688695063647615470291650418719163515879663
47219442930927982084309104855990570159318959639524863372367203
00291696959215610876494888925409080591145703767520850020667156
37023661263597471448071117748158809141357427209671901518362825
60618091458852699826141425030123391108273603843767876449043205
96037912449090570756031403507616256247603186379312648470374378
29549756137709816046144133086921181024859591523801953310302921
62800160568670105651646750568038741529463842244845292537361442
53361437372908830379460127472495841486491593064725201515569392
26281806916507963810641322753072671439981585088112926289011342
37782705567421080070065283963322155077831214288551

675554073345107213112427399562982719769150054883905223804357045848197956393157853510018992000024141963706813559840464039472194016069517690156119726982337890017641517190051133466306898140219383481435426387306539552969691388024158161859561100640362119796101859534802787167200122604642492385111393400464351623867567078745259464670903886547743483217897012764455529409092021959585751622973333576159552394885297579954028471943529913

A big number

4376370598692891375715374000198639433246489005254310662966916
52434191746913896324765602894151997754777031380647813423095961
90960654591300890188887588084733625956065444888501447335706058
81709016210849971452956834406197969056546981363116205357936979
14032363284962330464210661362002201757878518574091620504897117
81820400187282939943446186224328009837323764931814789848119452
71300744022076568091037620399920349202390662626449190916798546
15157788390603977207592793788522412943010174580868622633692847
25851403039615558564330385450688652213114813638408384778263790
45960718687672850976347127198889068047824323039471865052566097
81507298611414303058169279249714091610594171853522758875044775
92218301158780701975535722241400019548102005661773589781499532
32520858975346354700778669040642901676380816174055040511767009
36732028045493390279924918673065399316407204922384748152806191
66900933805732120816350707634351669869625020969023162859350071
87419057916124153689751480826190484794657173660100589247665544
58408383347905441448176842553272073155863493476051374197795251
90365032198020108764738368682531025183377533908861426184800374
00808223810407646887847164755294532694766170042446106331123802
113458869453220011656407632702307429242605

A big number

634069650308442258559670392718694611585137933864756997485686700798239606043934788508616492603049450617434123658283521448067266768418070837548622114082365798029612000274413244384324023312574035450193524287764308802328508558860899627744581646808578751158070147437638679769550499916439982843572904153781434388473034842619033888414940313661398542576355771053355802066221855770600825512888933322264362819848386132395706761914096385338323743437588308592337222846442879962456054769324289984326526773783731732880632107532112386806046747084280511664887090847702912081611049125555983223662448685566514026846412096949825905655192161881043412268389962830716548685255369148502995396755039549383718534059000961874894739928804324963731657538036735867101757839948184717984982469480605320819960661834340124760966395197780214411997525467040806084993441782562850927265237098986515394621930046073645079262129759176982938923670151709920

915315678144397912484757062378046000099182933213068805700465914583872080880168874458355579262584651247630871485663135289341661174906175266714926721761283308452739364692445828925713888778390563004824837998396920292222154861459023734782226825216399574408017271441461795592261750838890200741699262383002822862

A big number

...I got tired of copying and pasting, but we're not even a fourth of the way through.

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

Runtime of Kruskal? $\mathcal{O}\left((|E| + |V|) \log^*(|V|)\right) \approx \mathcal{O}\left(|E| + |V|\right)$

Inverse of the Ackerman function

But wait!

Somebody then came along and proved that find and union are amortized $\mathcal{O}\left(\alpha(n)\right)$ – the inverse of the Ackermann function.

This grows even more slowly then $\log^*(n)$!