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:

```
Example of the algorithm:
```

Kruskal's algorithm: example with a weighted graph

Example of the algorithm:

```
Example of the algorithm:
```

Kruskal's algorithm: example with a weighted graph

Example of the algorithm:

```
Example of the algorithm:
```

Kruskal's algorithm: example with a weighted graph

Example of the algorithm:

```
Example of the algorithm:
```
Example of the algorithm:

```
Example of the algorithm:
```

```
Example of the algorithm:
```

```
Example of the algorithm:
```

```
Example of the algorithm:
```

```
Example of the algorithm:
```

```
Example of the algorithm:
```
Kruskal’s algorithm: example with a weighted graph

Example of the algorithm:

```
  a --- b
  |     |
  8    4
  |     |
  h     g
  1    7

  c --- d
  |     |
  7    2
  |     |
  i     f
  2    10

  e
```

Example of the algorithm:

```
  a --- b
  |     |
  8    4
  |     |
  h     g
  1    7

  c --- d
  |     |
  7    2
  |     |
  i     f
  2    10

  e
```
Kruskal’s algorithm: example with a weighted graph

Example of the algorithm:

```
Example of the algorithm:
```

```
Example of the algorithm:
```

```
Example of the algorithm:
```

```
Example of the algorithm:
```
Example of the algorithm:

Example of the algorithm:

Example of the algorithm:

Example of the algorithm:

Example of the algorithm:

Example of the algorithm:
Kruskal’s algorithm: example with a weighted graph

Example of the algorithm:

```
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...

▶ \text{makeMST(v)} takes \( O(t_m) \) time
▶ \text{findMST(v)} takes \( O(t_f) \) time
▶ \text{union(u, v)} takes \( O(t_u) \) time

Putting it all together:
\[ O(|V| \cdot t_m + |E| \cdot \log(|E|) + |E| \cdot t_f + |V| \cdot t_u) \]

The DisjointSet ADT

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

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.
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.

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))

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 O(n) time – how?)

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

We combine those two trees into one.

**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.

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.

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.

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

Better question: are our trees guaranteed 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:

```
  5
   4
    3
     2
      1
       0
```

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 \( \text{rank}(x) \) be a number representing the upper-bound of the height of \( x \). So, \( \text{rank}(x) \geq \text{height}(x) \).

We then:

1. Keep track of the rank of all trees.
2. When unifying, 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.)

**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 \texttt{union(1, 5)}?

\[
\begin{array}{ccc}
\text{r=2} & \text{r=0} & \text{r=2} \\
6 & 2 & 8 \\
0 & 4 & 3 \\
1 & 5 & 9 \\
\end{array}
\]

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 \texttt{union(5, 11)}?

\[
\begin{array}{ccc}
\text{r=0} & \text{r=3} \\
2 & 6 & 8 \\
0 & 4 & 3 \\
1 & 5 & 9 \\
11 & 10 & 11 \\
\end{array}
\]

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

Net effect? Our trees stay relatively balanced.

So, what are the worst-case runtimes now?

\begin{itemize}
\item \texttt{makeSet(x)}: \[O(1)\] – still the same
\item \texttt{findSet(x)}: \[O(\log(n))\] – since the tree is balanced
\item \texttt{union(x, y)}: \[O(\log(n))\] – since union calls \texttt{findSet}
\end{itemize}

Path compression

Consider the following forest:

\[
\begin{array}{ccc}
1 & 7 \\
2 & 5 & 4 \\
6 & 10 & 9 \\
3 & 11 & 13 \\
\end{array}
\]

Suppose we call \texttt{findSet(3)} a few hundred times.

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

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

\[
\begin{array}{ccc}
1 & 2 & 5 \\
6 & 4 & 9 \\
3 & 10 & 14 \\
11 & 13 & 0 \\
\end{array}
\]

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.

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

- **makeSet(x):**
  \(O(1)\) – still the same

- **findSet(x):**
  In the best case, \(O(1)\), in the worst case \(O(\log(n))\)

- **union(x, y):**
  In the best case, \(O(1)\), in the worst case \(O(\log(n))\)

---

### Back to Kruskal’s

We concluded that the runtime is:

\[
\mathcal{O}\left(\left|V\right| \cdot t_{\text{setup}} + \left|E\right| \cdot \log\left(\left|E\right|\right) + \left|E\right| \cdot t_{r} + \left|V\right| \cdot t_{u}\right)
\]

Well, we just said that in the worst case:

- \(t_{\text{setup}} \in O(1)\)
- \(t_{r} \in O(\log(|V|))\)
- \(t_{u} \in O(\log(|V|))\)

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

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

---

### 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 \(\log^*(n)\).
What is $2^{65536}$?

$2^{65536} = 2003529935548905373932047530222803443018633911047170975471266115557201046762428380558757707...$

A big number

And even then:

$3^{26}$

But wait!

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

This grows even more slowly than

$\log^* n$!