Lecture 19: Disjoint Sets
Warmup

Run Kruskal’s algorithm on the following graph to find the MST (minimum spanning tree) of the graph below.
Below is the provided pseudocode for Kruksal’s algorithm to choose all the edges.

```
KruskalMST(Graph G)
    initialize each vertex to be an independent component
    sort the edges by weight
    foreach(edge (u, v) in sorted order){
        if(u and v are in different components){
            add (u,v) to the MST
            update u and v to be in the same component
        }
    }
```
Announcements
Disjoint Sets
Selecting an ADT

Kruskal’s needs to **find** what MST a vertex belongs to, and **union** those MSTs together

- Our existing ADTs don’t lend themselves well to “unioning” two sets...
- Let’s define a new one!

```plaintext
kruskalMST(G graph)
    Set(?), msts; Set finalMST;
    initialize msts with each vertex as single-element MST
    sort all edges by weight (smallest to largest)
    for each edge (u,v) in ascending order:
        uMST = msts.find(u)
        vMST = msts.find(v)
        if (uMST != vMST):
            finalMST.add(edge (u, v))
            msts.union(uMST, vMST)
```
Disjoint Sets ADT (aka “Union-Find”)

Kruskal’s will use a Disjoint Sets ADT under the hood

- Conceptually, a single instance of this ADT contains a “family” of sets that are disjoint (no element belongs to multiple sets)

**Disjoint Sets ADT**

<table>
<thead>
<tr>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Family of Sets</td>
</tr>
<tr>
<td>• disjoint: no shared elements</td>
</tr>
<tr>
<td>• each set has a representative (either a member or a unique ID)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>makeSet(value)</strong> - new set with value as only member (and representative)</td>
</tr>
<tr>
<td><strong>find(value)</strong> - return representative of the set containing value</td>
</tr>
<tr>
<td><strong>union(x, y)</strong> - combine sets containing x and y into one set with all elements, choose single new representative</td>
</tr>
</tbody>
</table>

```python
def kruskalMST(G graph):
    DisjointSets<V> msts; Set finalMST;
    initialize msts with each vertex as single-element MST
    sort all edges by weight (smallest to largest)

    for each edge (u,v) in ascending order:
        uMST = msts.find(u)
        vMST = msts.find(v)
        if (uMST != vMST):
            finalMST.add(edge (u, v))
            msts.union(uMST, vMST);  ```
Disjoint Sets in mathematics

“In mathematics, two sets are said to be disjoint sets if they have no element in common.” – Wikipedia

disjoint = not overlapping

These two sets are disjoint sets

These two sets are not disjoint sets
In computer science, disjointsets can refer to this ADT/data structure that keeps track of the multiple “mini” sets that are disjoint (confusing naming, I know). This overall grey blob thing is the actual disjoint sets, and it’s keeping track of any number of mini-sets, which are all disjoint (the mini sets have no overlapping values).

Note: this might feel really different than ADTs we’ve run into before. The ADTs we’ve seen before (dictionaries, lists, sets, etc.) just store values directly. But the Disjoint Set ADT is particularly interested in letting you group your values into sets and keep track of which particular set your values are in.
DisjointSets ADT methods

Just 3 methods (and makeSet is pretty simple!)

- findSet(value)
- union(valueA, valueB)
- makeSet(value)
**findSet(value)**

**findSet(value)** returns some ID for which particular set the value is in. For Disjoint Sets, we often call this the **representative** (as it’s a value that represents the whole set).

Examples:

- `findSet(Brian)` 3
- `findSet(Sherdil)` 2
- `findSet(Velocity)` 2
- `findSet(Kevin) == findSet(Aileen)` true
union(valueA, valueB)

**union(valueA, valueB)** merges the set that A is in with the set that B is in. (basically add the two sets together into one)

Example: `union(Kevin, Nishu)`
**makeSet(value)**

**makeSet(value)** makes a new mini set that just has the value parameter in it.

Examples:
- makeSet(Elena)
- makeSet(Anish)
Disjoint Sets ADT Summary

### Disjoint-Sets ADT

**state**
- Set of Sets
  - **Mini sets are disjoint**: Elements must be unique across mini sets
  - No required order
  - Each set has id/representative

**behavior**
- `makeSet(value)` – creates a new set within the disjoint set where the only member is the value. Picks id/representative for set
- `findSet(value)` – looks up the set containing the value, returns id/representative of that set
- `union(x, y)` – looks up set containing x and set containing y, combines two sets into one. All of the values of one set are added to the other, and the now empty set goes away.
New ADT

Set ADT

**state**
- Set of elements
  - Elements must be unique!
  - No required order
- Count of Elements

**behavior**
- `create(x)` - creates a new set with a single member, x
- `add(x)` - adds x into set if it is unique, otherwise add is ignored
- `remove(x)` - removes x from set
- `size()` - returns current number of elements in set

Disjoint-Set ADT

**state**
- Set of Sets
  - **Disjoint**: Elements must be unique across sets
  - No required order
  - Each set has representative
- Count of Sets

**behavior**
- `makeSet(x)` - creates a new set within the disjoint set where the only member is x. Picks representative for set
- `findSet(x)` - looks up the set containing element x, returns representative of that set
- `union(x, y)` - looks up set containing x and set containing y, combines two sets into one. Picks new representative for resulting set
Example

new()
makeSet(a)
makeSet(b)
makeSet(c)
makeSet(d)
makeSet(e)
findSet(a)
findSet(d)
union(a, c)
Example

new()
makeSet(a)
makeSet(b)
makeSet(c)
makeSet(d)
makeSet(e)
findSet(a)
findSet(d)
union(a, c)
union(b, d)
Example

new()
makeSet(a)
makeSet(b)
makeSet(c)
makeSet(d)
makeSet(e)
findSet(a)
findSet(d)
union(a, c)
union(b, d)

findSet(a) == findSet(c) true
findSet(a) == findSet(d) false
Questions?
That’s all!
Disjoint-Set ADT

**state**
- Set of Sets
- **Disjoint**: Elements must be unique across sets
- No required order
- Each set has representative
- Count of Sets

**behavior**
- `makeSet(x)` – creates a new set within the disjoint set where the only member is x. Picks representative for set
- `findSet(x)` – looks up the set containing element x, returns representative of that set
- `union(x, y)` – looks up set containing x and set containing y, combines two sets into one. Picks new representative for resulting set

TreeDisjointSet<E>

**state**
- Collection<TreeSet> forest
- Dictionary<NodeValues, NodeLocations> nodeInventory

**behavior**
- `makeSet(x)` – create a new tree of size 1 and add to our forest
- `findSet(x)` – locates node with x and moves up tree to find root
- `union(x, y)` – append tree with y as a child of tree with x

TreeSet<E>

**state**
- SetNode overallRoot

**behavior**
- TreeSet(x)
- add(x)
- remove(x, y)
- getRep() – returns data of overallRoot

SetNode<E>

**state**
- E data
- Collection<SetNode> children

**behavior**
- SetNode(x)
- addChild(x)
- removeChild(x, y)
Implement `makeSet(x)`

```
makeSet(0)
makeSet(1)
makeSet(2)
makeSet(3)
makeSet(4)
makeSet(5)
```

Worst case runtime?

$O(1)$
QuickUnion Data Structure

Fundamental idea:
- QuickFind tracks each element’s ID
- QuickUnion tracks each element’s *parent*. Only the root has an ID!
  - Each set becomes tree-like, but something slightly different called an *up-tree*: store pointers from children to parents!

Joyce, Sam, Ken, Alex
Aileen, Santino
Paul

Abstract Idea of “Disjoint Sets”

Implementation using QuickUnion
QuickUnion: Find

**find**(Ken):
- jump to Ken node
- travel upward until root
- return ID

Key idea: can travel upward from any node to find its representative ID

How do we jump to a node quickly?
- Also store a map from value to its node (Omitted in future slides)

\[
\begin{align*}
\text{find}(\text{Santino}) & \rightarrow 1 \\
\text{find}(\text{Ken}) & \rightarrow 2 \\
\text{find}(\text{Santino}) & \neq \text{find}(\text{Ken}) \\
\text{find}(\text{Santino}) & =\text{find}(\text{Aileen})
\end{align*}
\]
QuickUnion: Union

Key idea: easy to simply rearrange pointers to union entire trees together!

Which of these implementations would you prefer?

union(Ken, Santino):
    rootS = find(Santino)
    set Ken to point to rootS

RESULT:

union(Ken, Santino):
    rootK = find(Ken)
    rootS = find(Santino)
    set rootK to point to rootS

RESULT:
QuickUnion: Union

union(Ken, Santino):
rootS = find(Santino)
set Ken to point to rootS

RESULT:

union(Ken, Santino):
rootK = find(Ken)
rootS = find(Santino)
set rootK to point to rootS

We prefer the right implementation because by changing just the root, we effectively pull the entire tree into the new set!

○ If we change the first node instead, we have to do more work for the rest of the old tree
○ A rare example of constant time work manipulating a factor of \( n \) elements
QuickUnion: Why bother with the second root?

**Key idea:** will help minimize runtime for future `find()` calls if we keep the height of the tree short!

- Pointing directly to the second element would make the tree taller

```plaintext
union(Ken, Santino):
  rootK = find(Ken)
  rootS = find(Santino)
  set rootK to point to rootS
```

Why not just use:

```plaintext
union(Ken, Santino):
  rootK = find(Ken)
  set rootK to point to Santino
```

![Diagram showing the effect of not using the second root](image)
QuickUnion: Checking in on those runtimes

<table>
<thead>
<tr>
<th></th>
<th>Maps to Sets</th>
<th>QuickFind</th>
<th>QuickUnion</th>
</tr>
</thead>
<tbody>
<tr>
<td>makeSet(value)</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>findSet(value)</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>union(x, y)</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>

Only if we discount the runtime from union’s calls to find!
Otherwise, $\Theta(n)$.

- However, for Kruskal’s not a bad assumption: we only ever call union with roots anyway

union(A, B):
- rootA = find(A)
- rootB = find(B)
- set smallerRoot to point to largerRoot

kruskalMST(G graph)

```java
DisjointSets<V> msts; Set finalMST;
initialize msts with each vertex as single-element MST
sort all edges by weight (smallest to largest)

for each edge (u,v) in ascending order:
    uMST = msts.find(u)
    vMST = msts.find(v)
    if (uMST != vMST):
        finalMST.add(edge (u, v))
        msts.union(uMST, vMST);
```
QuickUnion: Let’s Build a Worst Case

Even with the “use-the-roots” implementation of union, try to come up with a series of calls to union that would create a worst-case runtime for find on these Disjoint Sets:

```
find(A):
  jump to A node
  travel upward until root
  return ID

union(A, B):
  rootA = find(A)
  rootB = find(B)
  set rootA to point to rootB
```
QuickUnion: Let’s Build a Worst Case

Even with the “use-the-roots” implementation of union, try to come up with a series of calls to union that would create a worst-case runtime for find on these Disjoint Sets:

union(A, B)
union(B, C)
union(C, D)
find(A)

union(A, B):
  rootA = find(A)
  rootB = find(B)
  set rootA to point to rootB

find(A):
  jump to A node
  travel upward until root
  return ID
Analyzing the QuickUnion Worst Case

- How did we get a degenerate tree?
  - Even though pointing a root to a root usually helps with this, we can still get a degenerate tree if we put the root of a large tree under the root of a small tree.
  - In QuickUnion, rootA always goes under rootB
    - But what if we could ensure the smaller tree went under the larger tree?

union(C, D)

What currently happens

What would help avoid degenerate tree
WeightedQuickUnion

Goal: Always pick the smaller tree to go under the larger tree

Implementation: Store the number of nodes (or “weight”) of each tree in the root

- Constant-time lookup instead of having to traverse the entire tree to count

union(A, B):
rootA = find(A)
rootB = find(B)
put lighter root under heavier root

union(A, B)
union(B, C)
union(C, D)
find(A)

Now what happens?

Perfect! Best runtime we can get.
WeightedQuickUnion: Performance

union()’s runtime is still dependent on find()’s runtime, which is a function of the tree’s height

What’s the worst-case height for WeightedQuickUnion?

union(A, B):
    rootA = find(A)
    rootB = find(B)
    put lighter root under heavier root
WeightedQuickUnion: Performance

Consider the worst case where the tree height grows as fast as possible.

<table>
<thead>
<tr>
<th>N</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
WeightedQuickUnion: Performance

Consider the worst case where the tree height grows as fast as possible

<table>
<thead>
<tr>
<th>N</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
WeightedQuickUnion: Performance

Consider the worst case where the tree height grows as fast as possible

<table>
<thead>
<tr>
<th>N</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>?</td>
</tr>
</tbody>
</table>
WeightedQuickUnion: Performance

Consider the worst case where the tree height grows as fast as possible

\[
\begin{array}{c|c|c}
N & H \\
1 & 0 \\
2 & 1 \\
4 & 2 \\
\end{array}
\]
WeightedQuickUnion: Performance

Consider the worst case where the tree height grows as fast as possible.
WeightedQuickUnion: Performance

Consider the worst case where the tree height grows as fast as possible

<table>
<thead>
<tr>
<th>N</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>
WeightedQuickUnion: Performance

Consider the worst case where the tree height grows as fast as possible.
Worst case tree height is $\Theta(\log N)$
Why Weights Instead of Heights?

We used the number of items in a tree to decide upon the root

Why not use the height of the tree?
- HeightedQuickUnion’s runtime is asymptotically the same: $\Theta(\log(n))$
- It’s easier to track weights than heights, even though WeightedQuickUnion can lead to some suboptimal structures like this one:
WeightedQuickUnion Runtime

<table>
<thead>
<tr>
<th></th>
<th>Maps to Sets</th>
<th>QuickFind</th>
<th>QuickUnion</th>
<th>WeightedQuickUnion</th>
</tr>
</thead>
<tbody>
<tr>
<td>makeSet(value)</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>find(value)</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>union(x, y) assuming root args</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>union(x, y)</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
</tbody>
</table>

This is pretty good! But there’s one final optimization we can make: path compression
Modifying Data Structures for Future Gains

- Thus far, the modifications we’ve studied are designed to preserve invariants
  - E.g. Performing rotations to preserve the AVL invariant
  - We rely on those invariants always being true so every call is fast

- Path compression is entirely different: we are modifying the tree structure to improve future performance
  - Not adhering to a specific invariant
  - The first call may be slow, but will optimize so future calls can be fast
Path Compression: Idea

This is the worst-case topology if we use WeightedQuickUnion

Idea: When we do find(15), move all visited nodes under the root

- Additional cost is insignificant (we already have to visit those nodes, just constant time work to point to root too)
Path Compression: Idea

This is the worst-case topology if we use WeightedQuickUnion

- Idea: When we do find(15), move all *visited nodes* under the root
  - Additional cost is insignificant (we already have to visit those nodes, just constant time work to point to root too)

Perform Path Compression on every find(), so future calls to find() are faster!
Path Compression: Details and Runtime

Run path compression on every find()!
- Including the find()s that are invoked as part of a union()

Understanding the performance of M>1 operations requires amortized analysis
- Effectively averaging out rare events over many common ones
- Typically used for “In-Practice” case
  - E.g. when we assume an array doesn’t resize “in practice”, we can do that because the rare resizing calls are amortized over many faster calls
- In 373 we don’t go in-depth on amortized analysis
Path Compression: Runtime

M find()s on WeightedQuickUnion requires takes $\Theta(M \log N)$

... but $M$ find()s on WeightedQuickUnionWithPathCompression takes $O(M \log^* N)$!

- $\log^* n$ is the “iterated log”: the number of times you need to apply log to $n$ before it’s $\leq 1$
- Note: $\log^*$ is a loose bound
Path Compression: Runtime

Path compression results in find()s and union()s that are very very close to (amortized) constant time

- \(\log^*\) is less than 5 for any realistic input
- If \(M\) find()s/union()s on \(N\) nodes is \(O(M \log^* N)\) and \(\log^* N \approx 5\), then find()/union()s amortizes to \(O(1)!\)

<table>
<thead>
<tr>
<th>(N)</th>
<th>(\log^* N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>(2^{16})</td>
<td>3</td>
</tr>
<tr>
<td>65536</td>
<td>4</td>
</tr>
<tr>
<td>(2^{65536})</td>
<td>5</td>
</tr>
</tbody>
</table>

Number of atoms in the known universe is \(2^{256}ish\)