Lecture 20: Disjoint Sets
Practice

Given the following disjoint-set what would be the result of the following calls on union if we always add the smaller tree (fewer nodes) into the larger tree (more nodes). Draw the forest at each stage with corresponding ranks for each tree.

union(2, 13)
union(4, 12)
union(2, 8)
Practice

Given the following disjoint-set what would be the result of the following calls on union if we add the “union-by-weight” optimization. Draw the forest at each stage with corresponding ranks for each tree.

union(2, 13)
Given the following disjoint-set what would be the result of the following calls on union if we add the “union-by-weight” optimization. Draw the forest at each stage with corresponding ranks for each tree.

union(2, 13)
union(4, 12)
Practice

Given the following disjoint-set what would be the result of the following calls on union if we add the “union-by-weight” optimization. Draw the forest at each stage with corresponding ranks for each tree.

union(2, 13)
union(4, 12)
union(2, 8)
Practice

Given the following disjoint-set what would be the result of the following calls on union if we add the “union-by-weight” optimization. Draw the forest at each stage with corresponding ranks for each tree.

union(2, 13)
union(4, 12)
union(2, 8)

Does this improve the worst case runtimes?

findSet is more likely to be $O(\log(n))$ than $O(n)$
Announcements

● P4 releases today
  ○ Due Wednesday 6/7 (finals week)
● EX3 regrade requests due Sunday
● EX5 due Monday
● EX6 releases Monday
● Sorry about lecture audio issues...
  ○ We are posting the lectures from last year
  ○ I added a video walk through of Bellman Ford (missing from last year)
  ○ I recorded a shorter video just going over the Dijkstra's implementation slides to help you on P4
Disjoint Set Implementation

Weighted Union
Path Compression
Array Implementation
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
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
- Set<TreeSet> forest
- Map<NodeValues, NodeLocations> nodeInventory

behavior
- makeSet(x) – creates 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
- SetNode<E> parent

behavior
- SetNode(x)
- updateParent(x)
Disjoint Sets are built as a collection of three objects:
- The TreeDisjointSet<E> is the top level object with two fields:
  - Set<TreeSet> forest
  - Map with node references
- Each TreeSet<E> is a collection of SetNodes<E>
- Each SetNode<E> can have an unlimited number of children, but only one parent. Nodes store parents instead of children.

Set<TreeSet> forest = TreeSet<E>

Map<E, SetNode<E>> nodeInventory = SetNode<E>
Implement makeSet(x)

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

Worst case runtime?

**O(1)**
Implement find(X)

**find(Ken):**
- Jump to Ken node
- Travel upward until root Joyce
- Return root “Joyce”

**Key Idea:** Jump to the node given. Travel upward to parent until parent field is null, nodes with null parents are roots and their data will act as the representative for the set

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

**Runtime**
- Jump to node $O(1)$
- Travel up to root
  - Based on height of TreeSet
  - Worst case: $O(n)$ if TreeSet is degenerate Tree

```
find(Santino) -> Aileen
find(Ken) -> Joyce
find(Santino) != find(Ken)
find(Santino) == find(Aileen)
```

Map<E, SetNode<E>> nodeInventory =

![Diagram showing the relationship between nodes and trees](image-url)
Implement \text{union}(x, y)

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

- it doesn’t matter what the order of the trees are, only that all the nodes from one tree are connected to the other tree

\textbf{union}(Ken, Santino):
- \texttt{rootK} = \texttt{find}(Ken) //Joyce
- \texttt{rootS} = \texttt{find}(Santino) //Aileen
- set \texttt{rootK} to point to \texttt{rootS}

RESULT:
Union: Why bother with the second root?

Key idea: keeping the height of each tree short will help minimize runtime for future $\text{find}(x)$
- Pointing directly to the individual element instead of the root can grow the tree height

Why not just use:

$\text{union}(\text{Ken, Santino})$:
- root$K = \text{find}(\text{Ken})$
- set root$K$ to point to Santino

$\text{root}_{K} = \text{find}(\text{Ken})$
$\text{set root}_{K}$ to point to Santino

Height = 3

Height = 4
Union runtime

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) n runtime :

union(A, B):
    rootA = find(A)
    rootB = find(B)
    set rootA to point to rootB
Analyzing the union 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.**
  - Instead of always putting rootA under rootB what if we could ensure the smaller tree went under the larger tree?

```plaintext
union(C, D)
```

What currently happens

What would help avoid degenerate tree
Disjoint Set Implementation

Weighted Union

Path Compression

Array Implementation
WeightedUnion

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)  O(1) runtime :)
WeightedUnion: 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>
WeightedUnion: 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>
WeightedUnion: 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>
WeightedUnion: 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>
</tbody>
</table>
WeightedUnion: 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>?</td>
</tr>
</tbody>
</table>
WeightedUnion: 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>
WeightedUnion: Performance

Consider the worst case where the tree height grows as fast as possible
Worst case tree height is $O(\log N)$

<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>
<tr>
<td>16</td>
<td>4</td>
</tr>
</tbody>
</table>
Runtime so far…

<table>
<thead>
<tr>
<th>Operation</th>
<th>Worst Case Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>makeSet(value)</td>
<td>O(1)</td>
</tr>
<tr>
<td>find(value)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>union(x, y)</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>

This is pretty good! But there’s one final optimization we can make:

path compression

```
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);
```

O(ElogV)
Disjoint Set Implementation
Weighted Union
Path Compression
Array Implementation
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 WeightedUnion

Key Idea: When we do $\text{find}(15)$, 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 WeightedUnion

Key Idea: When we do \( \text{find}(15) \), 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 \( \text{find()}, \) so future calls to \( \text{find()}, \) 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

\( M \text{ find()} \)s on WeightedUnion requires takes \( O(M \log N) \)

... but \( M \text{ find()} \)s using the WeightedUnion and PathCompression optimizations 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>16</td>
<td>3</td>
</tr>
<tr>
<td>( 2^{16} )</td>
<td>( 2^{65536} )</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
Kruskal’s Runtime

\[ \Theta(|E| \log |E|) \quad \text{or} \quad \Theta(|V| \log |V|) \]

- \( \Theta(|E| \log |V|) \) iterations
- \( \Theta(|E|) \) worst-case, but \( \Theta(|V| \log |V|) \) total
- \( \Theta(|V|) \) operations

Find and union are \( \log |V| \) in worst case, but amortized constant “in practice” 😞

- For an MST to exist, \( E \) can’t be smaller than \( V \), so assume it dominates
- Note: some people write \( |E| \log |V| \), which is the same (within a constant factor)

```
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 Set Implementation
Weighted Union
Path Compression
Array Implementation
Using Arrays for Up-Trees

Since every node can have at most one parent, what if we use an array to store the parent relationships?

Proposal: each node corresponds to an index, where we store the index of the parent (or -1 for roots). Use the root index as the representative ID!

Just like with heaps, tree picture still conceptually correct, but exists in our minds!
Using Arrays: Find

Initial jump to element still done with extra Map

But traversing up the tree can be done purely within the array!

Can still do path compression by setting all indices along the way to the root index!

```
find(A):
    index = jump to A node’s index
    while array[index] > 0:
        index = array[index]
    path compression
    return index
```

```
find(Alex) = 0
```

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
```
Using Arrays: Union

For **WeightedUnion**, we need to store the number of nodes in each tree (the weight).

Instead of just storing -1 to indicate a root, we can store -1 * weight!

```
union(Ken, Santino)
```

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>-4</td>
<td>0</td>
<td>-2</td>
<td>6</td>
<td>-1</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Joyce  Sam  Aileen  Alex  Paul  Santino  Ken

**union(A, B):**

- rootA = find(A)
- rootB = find(B)
- use -1 * array[rootA] and -1 * array[rootB] to determine weights
- put lighter root under heavier root
Using Arrays: Union

For WeightedUnion, we need to store the number of nodes in each tree (the weight).

Instead of just storing -1 to indicate a root, we can store -1 * weight!

union(Ken, Santino)

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-6</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>-1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Joyce</td>
<td>Sam</td>
<td>Aileen</td>
<td>Alex</td>
<td>Paul</td>
<td>Santino</td>
<td>Ken</td>
<td></td>
</tr>
</tbody>
</table>
```
Array Implementation Practice

Fill in the array representing this Disjoint set. Remember to Store (weight * -1) - 1 as the “parent” of the root nodes. Each “node” now only takes 4 bytes of memory instead of 32.
Array Implementation Practice

Update the Array with the correct values after a call of `union(14, 11)` using `WeightedUnion` and `PathCompression`
Array Implementation Practice

Update the Array with the correct values after a call of \texttt{union(14, 11)} using \texttt{WeightedUnion} and \texttt{PathCompression}.
Questions?
That’s all!
Appendix