CompSci 201, L26: Disjoint Sets
Logistics, Coming up

• Optional APT 11: Not required, will give APT makeup credit if you do it
  • Makeup credit: Will grade it, add that to apt grade if missing points there.
  • “Due” (for makeup credit) Wednesday 11/30 with grace/late

• Midterm Exam 3 next Monday 12/5

• Project 6: Due next Wednesday 12/7
Midterm Exam 3

• Logistics:
  • 60 minutes, in-person, short answer
  • Can bring 1 reference/notes page

• Major Topics:
  • Trees:
    • binary search trees,
    • binary heaps,
    • recursion
    • Red-black trees: Properties yes, rebalance algorithm, no.
  • Graphs:
    • Recursive & Iterative Stack DFS
    • Iterative Queue BFS
    • Weighted graphs, Dijkstra’s algorithm
Minimum Spanning Tree (MST) Problem

• Given N nodes and M edges, each with a weight/cost...

• Find a set of edges that connect *all* the nodes with minimum total cost. (will be a tree)
Visualizing Kruskal’s Algorithm

In the visualization:

• Edges between all pairs of vertices
• Weights are implicit by distances
• Algorithm greedily grows by cheapest edge that connects disjoint sets/trees.

By Shiyu Ji - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=54420894
Kruskal’s Algorithm in Pseudocode

Input: N node, M edges, M edge weights

• Let MST to an empty set
• Let S be a collection of N disjoint sets, one per node
• While S has more than 1 set:
  • Let (u, v) be the minimum cost remaining edge
  • Find which sets u and v are in. If not equal:
    • Union the sets
    • Add (u, v) to MST
• Return MST
Solving Example MST Problem

[link](leetcode.com/problems/min-cost-to-connect-all-points)

Live Coding
Go to duke.is/wendf

Not graded for correctness, just participation.

Try to answer *without* looking back at slides and notes.

But do talk to your neighbors!
Disjoint Sets and Union-Find
Union Find Data Structure

• Aka Disjoint Set Data Structure
• Start with N distinct (disjoint) sets
  • consider them labeled by integers: 0, 1, ...
• **Union** two sets: create set containing both
  • label with one of the numbers
• **Find** the set containing a number
  • Initially self, but changes after unions
Disjoint-Set Forest Implementation

- Each set will be represented by a parent “tree”: Instead of child pointers, nodes have a parent “pointer”.
- Everything starts as its own tree: a single node

<table>
<thead>
<tr>
<th>parent</th>
<th>itemID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
Disjoint-Set Forest Union

- Union(7,8)
- Just make leaf/root point to parent[7]
Disjoint-Set Forest Union

- Union(3,4)
Disjoint-Set Forest Union

- Union(3, 8)
- Multi-level, make parent[parent[8]] point to parent[3]
Disjoint-Set Forest Find

- **Find(8)**
- Return last ancestor of 8.
- Need to traverse the path up.

```
<table>
<thead>
<tr>
<th>parent</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemID</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
```
Disjoint-Set Forest Array Representation

• The “nodes” and “pointers” are just conceptual – can represent with a simple array, like binary heap.
• Parent array just stores what the itemID node points to.

<table>
<thead>
<tr>
<th>parent</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>3</th>
<th>5</th>
<th>6</th>
<th>3</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemID</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
Disjoint-Set Forest Find

```
18    public int find(int id) {
19          while (id != parent[id]) {
20              id = parent[id];
21          }
22          return id;
23    }
```

"last ancestor" is just when parent[i] = i

Else go to next "node up"

<table>
<thead>
<tr>
<th>parent</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>3</th>
<th>5</th>
<th>6</th>
<th>3</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemID</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

"last ancestor" is just when parent[i] = i

Else go to next "node up"
Disjoint-Set Forest Union Revisited

```java
public void union(int set1, int set2) {
    int root1 = find(set1);
    int root2 = find(set2);
    parent[root2] = root1;
}
```

Make one “point to” other

“last ancestors” from initial set1 and initial set2 “nodes”

<table>
<thead>
<tr>
<th>parent</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>3</th>
<th>5</th>
<th>6</th>
<th>3</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemID</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
Worst-Case Runtime Complexity?

```java
25   public void union(int set1, int set2) {
26       int root1 = find(set1);
27       int root2 = find(set2);
28       parent[root2] = root1;
```

What if we...
union(7,8)
union(6,7)
union(5,6)
...
union(0,1)

Now `find(8)` would have linear runtime complexity!!

<table>
<thead>
<tr>
<th>parent</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemID</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
Optimization 1: Union by Size

Be careful in how you union. Always make the “root” for the set with \textit{fewer} elements point to the “root” for the set with \textit{more} elements.

Sufficient for worst case logarithmic efficiency.
Optimization 1: Union by Size

```java
public void union(int set1, int set2) {
    int root1 = find(set1);
    int root2 = find(set2);
    if (root1 == root2) { return; }
    if (setSizes[root1] < setSizes[root2]) {
        parent[root1] = root2;
        setSizes[root2] += setSizes[root1];
    } else {
        parent[root2] = root1;
        setSizes[root1] += setSizes[root2];
    }
    size--;}
```

- If already in same set, nothing to do.
- Make the smaller set “point to” the bigger set.
Lazy Path Compression

- **Lazy path compression:** When ever you traverse a path in `find`, connect all the pointers to the top.

- Sufficient for **amortized logarithmic** runtime complexity for union/find operations.
Disjoint Set Forest Path Compression

```java
public int find(int id) {
    int idCopy = id;
    while (id != parent[id]) {
        id = parent[id];
    }
    int root = id;
    id = idCopy;
    while (id != parent[id]) {
        parent[idCopy] = root;
        id = parent[id];
        idCopy = id;
    }
    return id;
}
```
Optimized Runtime Complexity

• Optimizations considered separately:
  • Union by size: Worst case logarithmic
  • Path compression: Amortized logarithmic

• Considered together...?
  • Worst case logarithmic, and amortized inverse Ackermann function \( a(n) \).
  • \( a(n) < 5 \) for \( n < 2^{2^{2^{16}}} = 2^{2^{65536}} \)
  • Practically constant for any \( n \) you can write down
Remember Kruskal’s Algorithm Runtime?

Input: N node, M edges, M edge weights

• Let MST to an empty set
• Let S be a collection of N disjoint sets, one per node
• While S has more than 1 set:
  • Let (u, v) be the minimum cost remaining edge
  • Find which sets u and v are in. If not equal:
    • Union the sets
    • Add (u, v) to MST
• Return MST

Looping over (worst case) all M edges

Remove from binary heap, O(log(M))

O(M(log(M)+C)) because C < log(M) for our optimized union find
WOTO

Go to duke.is/zmgqm

Not graded for correctness, just participation.

Try to answer *without* looking back at slides and notes.

But do talk to your neighbors!