CSE 100
Disjoint Set, Union Find
Today’s Lecture

- Eager Simple Union and Find
- Eager Smart Union and Find
- Lazy Simple Union and Find (using up trees)
CLICKERS OUT
Q1: Which of the following is **not** an equivalence relation?

A. \( \leq \)

B. Are two cities in the same country?

C. Are two integers equal?
Q2: Which of the following are we manipulating when building a minimum spanning tree using Kruskal’s algorithm?

A. Equivalence relations  
B. Equivalence classes  
C. Both A and B
Q3: What is the overall Big O run time of Kruskal’s algorithm if BFS is used to check whether adding an edge creates a cycle?

A. O(1)
B. O(|V|)
C. O(|V| |E|)
Q4: Path compression is used to speed up the run time of which operation in the union-find data structure?

A. Find  
B. Union  
C. Path compression is unrelated to union-find
The Union-Find Data Structure

Efficient way of maintaining partitions

- find (v0) return set that v0 belongs to
- union (v1, v5): merge v5 to v1
- union (v2, v5): merge v5 to v2
- union (v1, v2): merge v2 to v1

Find is the condition for partition E(xy)

E(xy) = \{ x \in A | \exists y \in A, x \neq y \}

0 1 2 3 4 5
0 1 3 4 5
Equivalence Relations

• An equivalence relation $E(x,y)$ over a domain $S$ is a boolean function that satisfies these properties for every $x,y,z$ in $S$

  • $E(x,x)$ is true \hspace{1cm} (reflexivity)
  • If $E(x,y)$ is true, then $E(y,x)$ is true \hspace{1cm} (symmetry)
  • If $E(x,y)$ and $E(y,z)$ are true, then $E(x,z)$ is true \hspace{1cm} (transitivity)

• For example:
  • $E(x,y)$: Are cities $x$ and $y$ in the same country?

• Another example
  • $E(x,y)$: Are the integers $x$ and $y$ equal?
  • Then $E()$ is an equivalence relation over integers
Equivalence Classes

• An equivalence relation $E()$ over a set $S$ defines a system of *equivalence classes* within $S$
  
• The equivalence class of some element $x \in S$ is that set of all $y \in S$ such that $E(x,y)$ is true

• Note that every equivalence class defined this way is a subset of $S$

• The equivalence classes are disjoint subsets: no element of $S$ is in two different equivalence classes

• The equivalence classes are exhaustive: every element of $S$ is in some equivalence class
Equivalence Classes

Q: What are the equivalence classes defined over all cities in Asia by the equivalence relation - \( E(x,y) \): Are cities \( x \) and \( y \) in the same country?

A. The countries in Asia that share a border
B. The country of either city \( x \) or city \( y \)
C. All the cities in Asia
D. All the countries in Asia
Equivalence Classes for Kruskal’s

For Kruskal’s algo we will partition all vertices of the graph into disjoint sets, based on the equivalence relation: Are vertices x and y connected?

Q: The above equivalence relation partitions the graph into which of the following equivalence classes?

A. Connected subgraphs

B. Fully-connected (Complete) subgraphs

C. Disconnected subgraphs
Application of Union-Find to Kruskal’s MST

- Vertices that form a connected component will be in the same group
- Connected components that are isolated from each other will be in different groups

Q1: How can we check if adding an edge \((v, w)\) to the graph creates a cycle?

\[ \text{find} \left( v_1 \right) = \text{find} \left( v_6 \right) \]

then add \((v_1, v_6)\) will create cycle

Q2: What do we need to do (in Kruskal’s) if adding the edge does not create a cycle?

union \((c_0, c_2)\)
Implementation details: Eager Union-find

1. In each group, select one of the vertices to represent the leader
2. Augment each vertex to have a pointer to the leader vertex
3. How do we implement find(x)?
4. What is the run time of find(x)?

Ref: Tim Roughgarden (stanford)
Run time of find operation

Finding a path between two vertices
  => finding if they are both in the same group
  => checking if they both point to the same leader: O(1)
Implementation of Eager-simple union operation

- union(group1, group2): update leader pointers of vertices in one of the groups to point to the leader of the other group

\[
\text{union}(v_0, v_5) \rightarrow \text{Make all elements of group } v_5 \text{ point to } v_0
\]
Run time of Eager simple unions

Q: If $|V|$ is the total number of elements, in the worst case, how many leader pointer updates are needed when fusing two groups in the union method:

A. $O(1)$  
B. $O(\log |V|)$  
C. $O(|V|)$  
D. $O(|V|^2)$

Ref: Tim Roughgarden (stanford)
Eager smart union operation

Idea 1: When two groups are merged, elect the leader of the larger group to be the leader of the new group and update the leader pointers of the smaller group.

Q: In the worst case, how many leader pointer updates are needed when fusing two groups in the union method?

A. \(O(1)\)
B. \(O(\log|V|)\)
C. \(O(|V|)\)
D. \(O(|V|^2)\)
Implementation of smart union operation

Idea 1: When two groups are merged, elect the leader of the larger group to be the leader of the new group and update the leader pointers of the smaller group.

Q: (Revised question) In the worst case, how many leader pointer updates does a single vertex experience overall (until the MST is constructed by Kruskal’s)?
A. \( O(1) \)
B. \( O(\log|V|) \)
C. \( O(|V|) \)
D. \( O(|V|^2) \)
Q: In the worst case, how many leader pointer updates does a single vertex experience overall (until the MST is constructed by Kruskal’s)?

A. $O(1)$
B. $O(\log|V|)$
C. $O(|V|)$
D. $O(|V|^2)$

Since the leader pointer of a node is updated only when it joins another group with a larger size i.e. the size of its group doubles or more, there can be at most $\log |V|$ such merge operations, resulting in a max of $O(|V|\log|V|)$ updates overall.
Running Time of Kruskal’s algorithm with union find data structure:

1. Sort edges in increasing order of cost \( \mathcal{O}(|E| \log |V|) \)

2. Set of edges in MST, \( T = \{ \} \)

3. For \( i = 1 \) to \( |E| \)
   
   \( |E| \) iterations

   If \( T \cup \{ e_i = u,v \} \) has no cycles i.e. \( \text{find}(u) \neq \text{find}(v) \)\{ 
   
   Add \( e_i \) to \( T \)

   \( \mathcal{O}(|E|) \)

   \( \text{union} (\text{find}(u), \text{find}(v)) \)

\}

Ref: Tim Roughgarden (stanford)
Implementation of Union-Find

- The implementation of the Eager (smart) union-find is in the text. You can use that for the PA
Eager Unions vs. Lazy Unions

Perform these operations:
Union(1, 0) =
Find(4) =
Find(3) =

Eager Union

Lazy Union

Find(4) = O(1)
Array representation of Up-trees

- A compact and elegant implementation
- Each entry is the up index
- -1 for the roots
- Write the forest of trees, showing parent pointers and node labels, represented by this array

Find(4)
Performing a union operation

Union(6, 7) (Assume first element becomes the parent of the second)

Fill in 6, 7 and 8 in the array

<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>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>-1</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
Disjoint subsets using up-trees (Lazy simple Union and Find)

Start with 4 items: 0, 1, 2, 3

Suppose $\text{Union}(i,j)$ makes the node $\text{Find}(i)$ the parent of the node $\text{Find}(j)$

Perform these operations:

$\text{Union}(2,3)$

$\text{Union}(1,2)$

$\text{Find}(0) =$

$\text{Find}(3) =$

$\text{Union}(0,1)$

$\text{Find}(1) =$

$\text{Find}(0) =$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
Run time of Lazy simple union and find (using up-trees)

If we have no rule in place for performing the union of two up trees, what is the worst case run time of find and union operations in terms of the number of elements N?

A. find: O(N), union: O(1)
B. find: O(1), union: O(N)
C. find: O(\log_2 N), union: O(1)
D. find: O(1), union: O(\log_2 N)