Disjoint Data Sets
Outline

- Disjoint set data structure
- Applications
- Implementation
Data Structures for Disjoint Sets

- A disjoint-set data structure is a collection of sets \( S = \{S_1, \ldots, S_k\} \), such that \( S_i \cap S_j = \emptyset \) for \( i \neq j \).

- The methods are:
  - \( \text{find}(x) \): returns a reference to \( S_i \in S \) such that \( x \in S_i \)
  - \( \text{merge}(x, y) \): results in \( S \leftarrow S - \{S_i, S_j\} \cup \{S_i \cup S_j\} \)
    where \( x \in S_i \) and \( y \in S_j \)
    - \( \text{merge}([a], [d]) \) is executed by a union \( ([a], [d]) \) and update of the collection
      \( S = \{[a, d], [b], [c], [e]\} \)
Application of disjoint-set data structure

• Problem: Find the connected components of a graph.

1. Make a set of each vertex
2. For each edge do:
   if the two end points are not in the same set, merge the two sets

In the end, each set contains the vertices of a connected component.

• We can now answer the question: Are vertices \( x \) and \( y \) in the same component?
Example: Find Connected Vertices

$G = \begin{array}{ccc}
1 & 2 & 3 \\
\text{---} & \text{---} & \text{---} \\
5 & 4 & \\
\end{array}
$

$E = \{ (1,2), (1,5), (2,5), (3,4) \}$

1. Make a set of each vertex

Set of sets of vertices
$V = \{ \{1\}, \{2\}, \{3\}, \{4\}, \{5\} \}$

2. For each edge in $E$ do:

merge(1,2)
$V = \{ \{1, 2\}, \{3\}, \{4\}, \{5\} \}$

merge (1,5)
$V = \{ \{1, 2, 5\}, \{3\}, \{4\} \}$

merge (2,5)
$V = \{ \{1, 2, 5\}, \{3\}, \{4\} \}$

merge(3,4)
$V = \{ \{1, 2, 5\}, \{3, 4\} \}$
Disjoint Set Implementation in an array

- We can use an array, or a linked list to implement the collection. In this lecture we examine only an array implementation.
  - The size of the array is $N$ for a total of $N$ elements.
  - One element is the *representative* of the set.
  - In the array $Set$, each element $i$ for $i = 1, \ldots, N$ has the value $rep$ of the representative of its set. ($Set[i] = rep$)
  - We use the smallest “value” of the elements in a set as the representative.
Using an Array to implement DS

\[ Set = \{ \{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\} \} \]

\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8
\end{array}

\textit{merge ("4", "7")}

\[ Set = \{ \{1\}, \{2\}, \{3\}, \{4,7\}, \{5\}, \{6\}, \{8\} \} \]

\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 4 & 8 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8
\end{array}
DS implemented as an array

\[
\text{find}_1(x) \\
\text{return Set}[x]; \quad \text{// } \Theta(1).
\]

\[
\text{union}_1(\text{repx}, \text{repy}). \\
\quad \text{smaller } \leftarrow \min (\text{repx}, \text{repy}); \\
\quad \text{larger } \leftarrow \max (\text{repx}, \text{repy}); \\
\quad \text{for } k \leftarrow 1 \text{ to } N \text{ do} \\
\quad \quad \text{if } \text{set}[k] = \text{larger} \text{ then } \text{set}[k] \leftarrow \text{smaller};
\]

\(\Theta(N)\) in every case. After \(N-1\) union operations
the computation time is \(\Theta(N^2)\) which is too slow.
**DS is implemented as an array**

- For the following sequence of *merges* we show the resulting array

<table>
<thead>
<tr>
<th>Initial array</th>
<th>After merge ( {5}, {6})</th>
<th>After merge ( {4}, {5, 6})</th>
<th>After merge ( {3}, {4, 5, 6})</th>
<th>merge ( {2}, {3, 4, 5, 6})</th>
<th>merge ( {1},{2, 3, 4, 5, 6})</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6</td>
<td>1 2 3 4 5 5</td>
<td>1 2 3 4 4 4</td>
<td>1 2 3 3 3 3</td>
<td>1 2 2 2 2 2</td>
<td>1 1 1 1 1 1</td>
</tr>
</tbody>
</table>
Backward forests

- Sets are represented by “backward” rooted trees, with the element in the root representing the set.
- Each node points to its parent in the tree.
- The root points to itself.
- Backward forests can be stored in an array.
Backward forests stored in an array

\[ \text{find}_2(x) \]
\[ \text{rep} \leftarrow x; \]
\[ \text{while } (\text{rep} \neq \text{Set}[\text{rep}]) \]
\[ \quad \text{rep} \leftarrow \text{Set}[\text{rep}]; \]
\[ \text{return rep} \]

- \text{find}_2 \text{ is } O(\text{height}) \text{ of the tree in the worst case}

Example: \text{find}_2(4)
Backward forests stored in an array

```
union2(repx, repy).
  smaller ← min (repx, repy);
  larger ← max (repx, repy);
  set [larger ] ← smaller;
```

• union2 is O(1)
Disjoint-set implemented as forests

- Example: $\text{merge2}(2,5)$
- $\text{find2}(2)$ traverses up one link and returns 1. $\text{find2}(5)$ traverse up 2 links and returns 3.
- $\text{union2}$, adds a back link from the root of tree with rep=3 to the root of the tree with rep=1.
Disjoint-set implemented as backward forests

What is the worst case height?

- The following example shows that \( N - 1 \) merges may create a tree of height \( N - 1 \)
- Now \( N - 1 \) unions take a total of \( O(N) \) time.
- \( n \) find operations take \( O(nN) \) in the worst case.
- Initially:

```
<table>
<thead>
<tr>
<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>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
```

```
1 -> 2    3 -> 4    5 -> 6
```
Disjoint-set implemented as forests

- The order of execution of the "merge2" affects the height of the trees. Consider the following sequence of merge:

  - \( \text{merge2} (\{5\}, \{6\}) \)
  - \( \text{merge2} (\{4\}, \{5, 6\}) \)
  - \( \text{merge2} (\{3\}, \{4, 5, 6\}) \)
  - \( \text{merge2} (\{2\}, \{3, 4, 5, 6\}) \)
  - \( \text{merge2} (\{1\}, \{2, 3, 4, 5, 6\}) \)

Tree of height \( N - 1 \)
Disjoint-set forests with improved height

- A heuristic to improve time by decreasing the height of the trees
- Requires another array that contains heights. Initialized to 0
- We modify union2 to decrease the height of the trees to $O(\lg N)$ in the worst case
- $union3$ links the root of the tree with the smaller height to the root of the tree with the larger height
- $\text{Now find2} = O(\lg N)$ and $union3 = O(1)$
Disjoint-set forests with improved height

union3(rep\(x\), rep\(y\))

if (height[repx] == height [repy])
    height[repx]++;
    Set[repy] \leftarrow repx; // y’s tree points to x’s tree

else
    if height[repx] > height [repy]
        Set[repy] \leftarrow repx; // y’s tree points to x’s tree
    else
        Set[repx] \leftarrow repy; // x’s tree points y’s tree
**Merge with reduced height**

- *Example: merge3*(2,5)
- *find2*(2) traverses up one link and returns 1. *find2*(5) traverses up 2 links and returns 3.
- *union3*, adds a back link from the root of tree of height =1 with rep=1, to the root of the tree of height = 2 with rep=3.
Disjoint-set forests also with path compression

• Another heuristic to improve time:
  – Path compression (done during \textit{find3}). The nodes along a path from \( x \) to the \textit{root} will now point directly to the root.

• Useful when the number of finds \( n \) is very large, since most of the time \textit{find3} will be \( O(1) \)
**Find and compress**

`find3(x)`  
//find root of tree with x  
`root ← x;`  
`while (root != Set[root])`  
`root ← Set[root];`  
//compress path from x to root  
`node ← x;`  
`while (node != root)`  
`parent ← Set[node]`  
`Set[node] ← root; node points to root`  
`node ← parent`  
return `root`
Summary

• The worst case time to perform $n$ finds and $m$ unions for Backward forest with improved height and path compression
  – Approximately linear in $n$ finds + $m$ unions in most practical cases
    • To be precise, it’s $O((n + m) \alpha(n + m, n))$ where $\alpha(n + m, n)$ is the inverse of the Ackermann function
    • For all practical $n + m$ and $n$, $\alpha(n + m, n) \leq 3$, and time for $n$ finds and $m$ unions is linear in $n + m$
    • Proof is beyond the scope of this class
Supplementary info: Ackermann function

• Careful analysis shows that when a sequence of \( n \) finds and \( m < N \) unions are performed:
  - Computation time using path compression becomes \( O((n + m) \alpha(n + m, n)) \) where \( \alpha(n + m, n) \) is the inverse of the Ackermann function.

• The Ackermann function grows very fast. But the inverse of the Ackermann function grows more slowly than \( \lg^* n \) that grows very slowly.
Kruskal’s Algorithm
solution = {} 
while ( more edges in $E$) do 
{ 
  // Selection 
  select minimum weight edge 
  remove edge from $E$ 

  // Feasibility 
  if (edge creates a cycle with solution so far) 
    then reject edge 
  else add edge to solution 

  // Solution check 
  if $|solution| = |V| - 1$ return solution 
} 
return null // when does this happen?
Kruskal's Algorithm:

1. Sort the edges $E$ in non-decreasing weight
2. $T \leftarrow \emptyset$
3. For each $v \in V$ create a set.
4. repeat
5. Select next shortest edge $\{u,v\} \in E$
6. $u_{\text{comp}} \leftarrow \text{find}(u)$
7. $v_{\text{comp}} \leftarrow \text{find}(v)$
8. if $u_{\text{comp}} \neq v_{\text{comp}}$ then
   8. add edge $(u,v)$ to $T$
9. union($u_{\text{comp}}, v_{\text{comp}}$)
10. until $T$ contains $|V| - 1$ edges
11. return tree $T$

$C = \{ \{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}, \{h\} \}$

$C$ is a forest of trees.
Kruskal - Disjoint set
After Initialization

1. Sort the edges $E$ in non-decreasing weight
2. $T \leftarrow \emptyset$
3. For each $v \in V$ create a set.

Sorted edges

- A B 2
- B C 5
- A C 6
- B D 7

Disjoint data set for G
Kruskal - add minimum weight edge if feasible

5. for each \{u,v\} ∈ in ordered \(E\)
6. \(ucomp ← \text{find}(u)\)
7. \(vcomp ← \text{find}(v)\)
8. if \(ucomp \neq vcomp\) then
9. add edge \((v,u)\) to \(T\)
10. union( \(ucomp,vcomp\) )

Sorted edges

\[
\begin{align*}
\text{Disjoint data set for } G \\
\text{Find(A)} & \quad \text{Find(B)} \\
(A, B) & \quad (A, B) \\
(A, B) & \quad (A, B) \\
\end{align*}
\]
Kruskal - add minimum weight edge if feasible

5. for each \{u,v\} in ordered E
6. \ ucomp \leftarrow \text{find} \ (u)
7. \ vcomp \leftarrow \text{find} \ (v)
8. if \ ucomp \neq \ vcomp then
9. add edge \ (v,u) \ to \ T
10. union \ (\ ucomp,\ vcomp \ )
Kruskal - add minimum weight edge if feasible

5. for each \{u,v\} ∈ in ordered E
6. ucomp ← \text{find}(u)
7. vcomp ← \text{find}(v)
8. if ucomp ≠ vcomp then
9. add edge (v,u) to T
10. union ( ucomp,vcomp )

Sorted edges

\begin{tabular}{c|c}
A & B 2 \\
B & C 5 \\
C & A 6 \\
D & B 7 \\
\end{tabular}

A and C in same set → Reject edge (A,C)
Kruskal - add minimum weight edge if feasible

5. for each \{u,v\} ∈ in ordered \(E\)
6. \(u\)comp ← find \((u)\)
7. \(v\)comp ← find \((v)\)
8. if \(u\)comp ≠ \(v\)comp then
9. add edge \((v,u)\) to \(T\)
10. union \((u\)comp,\(v\)comp\)
Kruskal's Algorithm: Time Analysis

Kruskal ($G$)

1. Sort the edges $E$ in non-decreasing weight
2. $T \leftarrow \emptyset$
3. For each $v \in V$ create a set.
4. repeat
5. \{$u,v\} \in E$, in order
6. $u_{\text{comp}} \leftarrow \text{find}(u)$
7. $v_{\text{comp}} \leftarrow \text{find}(v)$
8. if $u_{\text{comp}} \neq v_{\text{comp}}$ then
9. add edge $(v,u)$ to $T$
10. union ($u_{\text{comp}}, v_{\text{comp}}$)
11. until $T$ contains $|V| - 1$ edges
12. return tree $T$

$\textbf{Count}_1 = \Theta(E \lg E)$

$\textbf{Count}_2 = \Theta(1)$

$\textbf{Count}_3 = \Theta(V)$

$\textbf{Count}_4 = O(E)$

Using Disjoint set-height and path compression

Sorting dominates the runtime.

We get $T(E,V) = \Theta(E \lg E) = \Theta(E \lg V)$

For a sparse graph we get $\Theta(V \lg V)$

For a dense graph we get $\Theta(V^2 \lg V)$