Kruskal’s Algorithm; Disjoint Sets
CSE 332 Spring 2021

Instructor: Hannah C. Tang

Teaching Assistants:
Aayush Modi  Khushi Chaudhari  Patrick Murphy
Aashna Sheth  Kris Wong  Richard Jiang
Frederick Huyan Logan Milandin  Winston Jodjana
Hamsa Shankar  Nachiket Karmarkar
How many times do you need to take the log of 20,035,299,304,068,464,649,790,723,515,602,557,504,478,254,755,697,514,192,650,169,737 to get to a value <= 1?
- Hint: that value is equal to $2^{65536}$, and $65536 = 2^{16}$

**Bonus:** find an MST using Prim’s Algorithm on this graph:
Lecture Outline

❖ Disjoint Sets ADT (aka Union/Find ADT)

❖ Kruskal’s Algorithm, for realz
  ▪ Review and Example
  ▪ Correctness Proof

❖ Up-Trees Data Structure
  ▪ Representation
  ▪ Optimization: Weighted Union
  ▪ Optimization: Path Compression
Disjoint Sets ADT (1 of 2)

❖ The Disjoint Sets ADT has two operations:
  ▪ find(e): gets the id of the element’s set
  ▪ union(e1, e2): combines the set containing e1 with the set containing e2

❖ Example: ability to travel to drive to a country
  ▪ union(france, germany)
  ▪ union(span, france)
  ▪ find(span) == find(germany)?
  ▪ union(england, france)
Disjoint Sets ADT (2 of 2)

❖ Applications include percolation theory (computational chemistry) and .... Kruskal’s algorithm

❖ Simplifying assumptions
  ▪ We can map elements to indices quickly
  ▪ We know all the items in advance; they’re all disconnected initially

❖ Later this lecture, we’ll see:
  ▪ We can do union() in constant time
  ▪ We can get find() to be *amortized* constant time
    • Worst case $O(\log n)$ for an individual find operation
Lecture Outline

❖ Disjoint Sets ADT (aka Union/Find ADT)

❖ Kruskal’s Algorithm, for realz
  ▪ Review and Example
  ▪ Correctness Proof

❖ Up-Trees Data Structure
  ▪ Representation
  ▪ Optimization: Weighted Union
  ▪ Optimization: Path Compression
Kruskal’s Algorithm

❖ Kruskal’s thinks edge by edge
  ▪ Eg, start from lightest edge and consider by increasing weight
  ▪ Compare against Dijkstra’s and Prim’s, which think vertex by vertex

❖ Outline:
  ▪ Start with a forest of \(|V|\) MSTs
  ▪ Successively connect them (ie, eliminate a tree) by adding edges
  ▪ Do not add an edge if it creates a cycle
Kruskal’s Algorithm: Pseudocode

```
kruskals(Graph g) {
    mst = {}
    forests = buildDisjointSets(g.vertices)
    numforests = g.vertices
    edges = buildHeap(g.edges)

    while (numForests > 1):
        e = edges.deleteMin()
        u_id = forests.find(e.u)
        v_id = forests.find(e.v)
        if (u_id != v_id):
            mst.addEdge(e)
            forests.union(e.u, e.v)
            numforests--
}
```

Runtime: $|E|(\log|E| + 2\log|V| + 1) + |V|(1 + 1 + 1) \in O(|E|\log|V| + |V|\log|V|)$

However, since we know $E \in O(|V|^2)$, runtime \in O(|E|\log|V|)
Kruskal’s Algorithm: Example

Weight | Edges
--- | ---
1 | (A,D), (C,D), (B,E), (D,E)
2 | (A,B), (C,F), (A,C)
3 | (E,G)
5 | (D,G), (B,D)
6 | (D,F)
10 | (F,G)

MST:
Num Trees: 7
Kruskal’s Algorithm: Example

MST:
(A, D)

Num Trees: 6
Kruskal’s Algorithm: Example

MST: (A, D), (C, D)
Num Trees: 5
Kruskal’s Algorithm: Example

MST:
(A, D), (C, D), (B, E)
Num Trees: 4

<table>
<thead>
<tr>
<th>Weight</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(A,D), (C,D), (B,E), (D,E)</td>
</tr>
<tr>
<td>2</td>
<td>(A,B), (C,F), (A,C)</td>
</tr>
<tr>
<td>3</td>
<td>(E,G)</td>
</tr>
<tr>
<td>5</td>
<td>(D,G), (B,D)</td>
</tr>
<tr>
<td>6</td>
<td>(D,F)</td>
</tr>
<tr>
<td>10</td>
<td>(F,G)</td>
</tr>
</tbody>
</table>
Kruskal’s Algorithm: Example

MST: 
(A, D), (C, D), (B, E), (D, E) 
Num Trees: 3
Kruskal’s Algorithm: Example

<table>
<thead>
<tr>
<th>Weight</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(A, D), (C, D), (B, E), (D, E)</td>
</tr>
<tr>
<td>2</td>
<td>(A, B), (C, F), (A, C)</td>
</tr>
<tr>
<td>3</td>
<td>(E, G)</td>
</tr>
<tr>
<td>5</td>
<td>(D, G), (B, D)</td>
</tr>
<tr>
<td>6</td>
<td>(D, F)</td>
</tr>
<tr>
<td>10</td>
<td>(F, G)</td>
</tr>
</tbody>
</table>

MST: (A, D), (C, D), (B, E), (D, E)
Num Trees: 3
Kruskal’s Algorithm: Example

MST:
(A, D), (C, D), (B, E), (D, E), (C, F)
Num Trees: 2

Weight | Edges
--- | ---
1 | (A,D), (C,D), (B,E), (D,E)
2 | (A,B), (C,F), (A,C)
3 | (E,G)
5 | (D,G), (B,D)
6 | (D,F)
10 | (F,G)
Kruskal’s Algorithm: Example

MST:
(A, D), (C, D), (B, E), (D, E), (C, F)
Num Trees: 2

<table>
<thead>
<tr>
<th>Weight</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(A, D), (C, D), (B, E), (D, E)</td>
</tr>
<tr>
<td>2</td>
<td>(A, B), (C, F), (A, C)</td>
</tr>
<tr>
<td>3</td>
<td>(E, G)</td>
</tr>
<tr>
<td>5</td>
<td>(D, G), (B, D)</td>
</tr>
<tr>
<td>6</td>
<td>(D, F)</td>
</tr>
<tr>
<td>10</td>
<td>(F, G)</td>
</tr>
</tbody>
</table>
Kruskal’s Algorithm: Example

MST:
(A, D), (C, D), (B, E), (D, E), (C, F), (E, G)
Num Trees: 1

Hark, an MST!!!
Total Cost: 9

Weight | Edges
--- | ---
1 | (A,D), (C,D), (B,E), (D,E)
2 | (A,B), (C,F), (A,C)
3 | (E,G)
5 | (D,G), (B,D)
6 | (D,F)
10 | (F,G)
Find an MST in this graph using Kruskal’s Algorithm:

MST:

<table>
<thead>
<tr>
<th>Weight</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(B, C), (G, H)</td>
</tr>
<tr>
<td>2</td>
<td>(A,B), (B, F), (C,D), (F,G)</td>
</tr>
<tr>
<td>4</td>
<td>(E,G)</td>
</tr>
</tbody>
</table>
Kruskal’s Algorithm: Demos and Visualizations

❖ Prim’s Visualization
  ▪ [https://www.youtube.com/watch?v=6uq0cQZOyoY](https://www.youtube.com/watch?v=6uq0cQZOyoY)
  ▪ Prim’s jumps around the fringe, adding edges by edge weight

❖ Kruskal’s Visualization:
  ▪ [https://www.youtube.com/watch?v=ggLyKfBTABo](https://www.youtube.com/watch?v=ggLyKfBTABo)
  ▪ Kruskal’s jumps around the graph – not just the fringe – because it chooses edges by edge weight independent of the “tree under construction”

❖ Conceptual demo:
  ▪ [https://docs.google.com/presentation/d/1RhRSYs9Jbc335P24p7vR-6PLXZU1-1EmeDtqieL9ad8/present?ueb=true&slide=id.g375bbf9ace_0_645](https://docs.google.com/presentation/d/1RhRSYs9Jbc335P24p7vR-6PLXZU1-1EmeDtqieL9ad8/present?ueb=true&slide=id.g375bbf9ace_0_645)
Lecture Outline

❖ Disjoint Sets ADT (aka Union/Find ADT)

❖ Kruskal’s Algorithm, for realz
  ▪ Review and Example
  ▪ Correctness Proof

❖ Up-Trees Data Structure
  ▪ Representation
  ▪ Optimization: Weighted Union
  ▪ Optimization: Path Compression
Kruskal’s Algorithm: Correctness

- Kruskal’s algorithm is clever, simple, and efficient
  - But does it generate a minimum spanning tree?

- First: it generates a spanning tree
  - To show treeness, need to show lack of cycles
  - To show that it’s a single tree, need to show it’s connected
  - To show spanningness, need to show that all vertices are included

- Second: there is no spanning tree with lower total cost ...
Kruskal’s Output is a Spanning Tree (1 of 2)

- To show **treeness**, need to show lack of cycles
  - **By definition**: Kruskal’s doesn’t add an edge if it creates a cycle

- To show that it’s a **single** tree, need to show it’s connected
  - **By contradiction**: suppose Kruskal’s generates >1 tree. Since the original graph \( G \) was connected, there exists an edge in \( G \) that connects Kruskal’s trees. Adding this edge would not create a cycle, so Kruskal’s would have included it. **CONTRACTION**
Kruskal’s Output is a Spanning Tree (2 of 2)

- To show *spanningness*, need to show that all vertices are included
  - *By contradiction*: suppose Kruskal’s tree $T$ does not include *any* edges adjacent to some vertex $v$. Since the original graph $G$ was connected, there exists at least one edge in $G$ that is adjacent to $v$. The minimum of these edges would not have created a cycle with $T$, so Kruskal’s would have included it. **CONTRACTION**
Kruskal’s Optimality: Inductive Proof Setup

❖ Let $F$ (stands for “forest”) be the set of edges Kruskal has added at some point during its execution.

❖ **Claim**: $F$ is a subset of one or more MSTs for the graph
  - *(Therefore, once $|F|=|V|-1$, we have a single MST)*

❖ **Proof**: By induction on $|F|$
  - **Base case**: $|F|=0$. The empty set is a subset of all MSTs
  - **Inductive case**: $|F|=k+1$. By induction, before adding the $(k+1)^{th}$ edge (call it $e$), there was some MST $T$ such that $F-\{e\} \subseteq T$ ...
Staying a Subset of Some MST

- **Claim:** $F$ is a subset of one or more MSTs for the graph
- **Things we know so far:**
  - $F - \{e\} \subseteq T$

- **Proof:** Two disjoint cases:
  A. If $\{e\} \subseteq T$, then $F \subseteq T$ and proof is done
  B. Else, $e$ forms a cycle with some simple path (call it $p$) in $T$
    - Must be a cycle since $T$ is a spanning tree
Staying a Subset of Some MST

- **Claim**: $F$ is a subset of one or more MSTs for the graph
- **Things we know so far**:
  - $F - \{e\} \subseteq T$
  - $e$ forms a cycle with $T$

- **New claim**: There is an edge $e_2$ on $p$ such that $e_2$ is not in $F$
  - Otherwise, Kruskal’s would not have added $e$

T is “the real” MST
$F$ is Kruskal’s output at the $k+1$th step
$e$ is the “wrong” edge Kruskal’s will add
$e_2$ is an edge in $T$ (but not $F$) along a cycle
Staying a Subset of Some MST

- **Claim:** $F$ is a subset of one or more MSTs for the graph

- **Things we know so far:**
  - $F - \{e\} \subseteq T$
  - $e$ forms a cycle with $T$
  - $e_2$ (on $p$) is not in $F$

- **New claim:** $e_2$.weight == $e$.weight
  - If $e_2$.weight > $e$.weight, then $T$ is not an MST
    - $T - \{e_2\} + \{e\}$ is a spanning tree with lower cost. **Contradiction!!**
  - If $e_2$.weight < $e$.weight, then Kruskal’s would have already considered $e_2$
    - Would have added it since $F - \{e\}$ has no cycles ($T$ has no cycles and $F - \{e\} \subseteq T$)
    - But $e_2$ is not in $F$. **Contradiction!!**
Staying a Subset of Some MST

**Claim:** $F$ is a subset of one or more MSTs for the graph

**Things we know so far:**
- $F - \{e\} \subseteq T$
- $e$ forms a cycle with $T$
- $e_2$ (on $p$) is not in $F$
- $e_2$.weight == $e$.weight

**New claim:** $T - \{e_2\} + \{e\}$ is (also) an MST
- It’s a spanning tree because $p - \{e_2\} + \{e\}$ connects the same nodes as $p$
- It’s minimal because its cost equals cost of $T$, an MST

Since $F \subseteq T - \{e_2\} + \{e\}$, $F$ is a subset of one or more MSTs  

Done!
Lecture Outline

❖ Disjoint Sets ADT (aka Union/Find ADT)

❖ Kruskal’s Algorithm, for realz
  ▪ Review and Example
  ▪ Correctness Proof

❖ Up-Trees Data Structure
  ▪ \textbf{Representation}
  ▪ Optimization: Weighted Union
  ▪ Optimization: Path Compression
Implementing the Disjoint Sets ADT (1 of 2)

❖ If we have \( n \) elements, what is the total cost of \( m \) find()s + \( \leq n-1 \) union()s?
  ▪ Can we have \( >n \) union()s?

❖ Goal: \( O(m+n) \) total for these operations
  ▪ i.e. \( O(1) \) amortized for all operations!

❖ Is our goal possible?
  ▪ Can get \( O(1) \) worst-case union()
  ▪ Would be nice if we could also get \( O(1) \) worst-case find(), but...
  ▪ *Known result*: both find() and union() can’t have worst-case \( O(1) \)
Implementing the Disjoint Sets ADT (2 of 2)

- **Observation:**
  - Trees let us find many elements given a single root

- **Idea:**
  - If we reverse the pointers (i.e., point up from child to parent), we can find a single root from many elements

- **Decision:**
  - One up-tree for each set
  - The ID of the set is (hash of) the tree root
  - *(as before, we will use integer elements for in-lecture examples)*
Up-Trees Data Structure for Disjoint Sets ADT

❖ Initial State:

1 2 3 4 5 6 7

❖ After several union()s:

1 3 7

2 5 6

Roots are the IDs for each set: 1, 3, 7
Up-Trees Find

- **find(x)**: follow x to the root and return the root ID
  - Eg: find(6) = 7
Up-Trees Union

- `union(x, y)`: assuming `x` and `y` are roots, point `y` to `x`
  - If `x` or `y` are not roots, can require caller to call `find()` first or do a `find()` internally
  - Eg: `union(1, 7)` vs `union(2, 5)`
Up-Trees Representation (1 of 2)

- Up-trees can be represented as an array of indices, where the element is the index of the parent
  - `up[x] = 0` means `x` is a root
  - *Note: in these slides, array is 1-indexed; 0-indexed is also fine*
Up-Trees Representation (2 of 2)

- Up-trees can be represented as an array of indices, where the element is the index of the parent
  - Can contain non-integer values if we use a hash table to map values to indices

Aqua | Cerulean | Glaucous
---|---|---
Bleu de France | Everton | Denim | Fluorescent

```
int[] up = [0, 1, 0, 7, 7, 5, 0, 7]
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aqua</td>
<td>1</td>
</tr>
<tr>
<td>Bleu de France</td>
<td>2</td>
</tr>
<tr>
<td>Cerulean</td>
<td>3</td>
</tr>
<tr>
<td>Denim</td>
<td>4</td>
</tr>
<tr>
<td>Everton</td>
<td>5</td>
</tr>
<tr>
<td>Fluorescent</td>
<td>6</td>
</tr>
<tr>
<td>Glaucous</td>
<td>7</td>
</tr>
</tbody>
</table>
Up-Trees Implementation

void union(int x, int y) {
    up[y] = x;
}

int find(int x) {
    while (up[x] != 0) {
        x = up[x];
    }
    return x;
}

❖ Worst-case runtime for union():

❖ Worst-case runtime for find():

❖ Total runtime for n-1 union()s and m find()s:

Remember: we can’t have ≥n calls to union()
What is the runtime for ...

- union(), worst-case
- find(), worst-case
- \( n-1 \) union()s + \( m \) find()s

\[ \Theta(1) / O(1) / O(n + m) \]

\[ \Theta(1) / O(h) / O(n + mh) \]

- \( h \) is the height of the up-tree

\[ \Theta(1) / O(n) / O(n^2) \]

\[ \Theta(1) / O(n) / O(n + mn) \]

\[ \Theta(1) / O(n) / O(n + m^2) \]
Worst-case Union

union(A, B)
union(B, C)
union(C, D)
union(D, E)
union(E, F)

🤔 If only I could keep these trees (semi-?)balanced
Lecture Outline

❖ Disjoint Sets ADT (aka Union/Find ADT)

❖ Kruskal’s Algorithm, for realz
  ▪ Review and Example
  ▪ Correctness Proof

❖ Up-Trees Data Structure
  ▪ Representation
  ▪ Optimization: Weighted Union
  ▪ Optimization: Path Compression
Weighted Union (1 of 3)

- Our naïve union() always picked the same argument (the second one) to become the child in the unioned result
Weighted Union (2 of 3)

❖ Our naïve union() always picked the same argument (the second one) to become the child in the unioned result

❖ Let’s make it smarter:
  ▪ Pick the smaller tree (i.e., tree with fewer nodes) to be the new child
    • i.e., “weight” = “num nodes”
  ▪ Add the new child to the heavier-tree’s root
Weighted Union (3 of 3)

❖ Our naïve union() always picked the same argument (the second one) to become the child in the unioned result

❖ Weighted union:
  - Pick the smaller tree (i.e., tree with fewer nodes) to be the new child
    - i.e., “weight” = “num nodes”
  - Add the new child to the heavier-tree’s root
Weighted Union: Representation

- Need to store number of nodes (or “weight”) of each tree

- Instead of ‘0’, we can store the root’s weight instead!
  - Use negative values to indicate they’re not indices
  - See Weiss, 8.4
Weighted Union: Implementation

```c
void union(int x, int y) {
    up[y] = x;
}
```

```c
weightedUnion(int x, int y) {
    wx = weight[x];
    wy = weight[y];
    if (wx < wy) {
        up[x] = y;
        weight[y] = wx + wy;
    } else {
        up[y] = x;
        weight[x] = wx + wy;
    }
}
```

union()’s runtime is still O(1)!

Does this (slightly) added complexity help us balance the up-trees and improve find()?
Weighted Union: Performance

- Consider the worst case: tree height grows as fast as possible
  - i.e., up-tree and up-subtrees are “spindly”

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

- Consider the worst case: tree height grows as fast as possible
  - ie, up-tree and up-subtrees are “spindly”

<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>
Weighted Union: Performance

- Consider the worst case: tree height grows as fast as possible
  - i.e., up-tree and up-subtrees are “spindly”

<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>
Weighted Union: Performance

- Consider the worst case: 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>
Weighted Union: Performance

- Consider the worst case: tree height grows as fast as possible
  - i.e., up-tree and up-subtrees are “spindly”

<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>
Weighted Union: Performance

- Consider the worst case: tree height grows as fast as possible
  - ie, up-tree and up-subtrees are “spindly”

- Worst-case height and worst-case find() is $\Theta(\log N)$

<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>$2^n$</td>
<td>n</td>
</tr>
</tbody>
</table>
Weighted Union Performance: Proof

- An up-tree with height $h$ using weighted union has weight at least $2^h$

- Proof by induction
  - Base-case: $h = 0$. The up-tree has one node and $2^0 = 1$
  - Inductive step: Assume true for all $h' < h$

We know:

- $W(T_1) \geq 2^{h-1}$
- $W(T_2) \geq 2^{h-1}$
- $W(T_1) \geq W(T_2)$

Since $W(T) = W(T_1) + W(T_2)$, we know that

$$W(T) \geq W(T_1) + W(T_2) = 2^{h-1} + 2^{h-1} = 2^h$$

Therefore $W(T) \geq 2^h$
What is the runtime for ...

- weighted union(), worst-case
- find(), worst-case
- n-1 union()s + m find()s

A. \( \Theta(1) / \Theta(1) / O(n + m) \)
B. \( \Theta(1) / \Theta(n) / O(n + m^2) \)
C. \( \Theta(1) / \Theta(\log n) / O(n + m \log n) \)
D. \( \Theta(1) / \Theta(\log n) / O(n + m^2) \)

```c
def int find(int x) {
    while (up[x] > 0) {
        x = up[x];
    }
    return x;
}
```
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?

- Heighted Union’s runtime is asymptotically the same: \( \Theta(\log N) \)
  - Proof is left as an exercise to the reader ;)
- Easier to track weights than heights, and heighted union doesn’t combine very well with the next optimization technique for find()
Lecture Outline

❖ Disjoint Sets ADT (aka Union/Find ADT)

❖ Kruskal’s Algorithm, for realz
   ▪ Review and Example
   ▪ Correctness Proof

❖ Up-Trees Data Structure
   ▪ Representation
   ▪ Optimization: Weighted Union
   ▪ **Optimization: Path Compression**
Modifying Data Structures To Preserve Invariants

❖ Thus far, the modifications we’ve studied are designed to *preserve invariants* (aka “repair the data structure”)
  ▪ **Tree rotations**: preserve AVL tree balance
  ▪ **Promoting keys / splitting leaves**: preserve B-tree node sizes (eg, L+1 keys stored in a leaf node)

❖ Notably, the modifications don’t improve runtime *between identical method calls*
  ▪ If `avl.find(x)` takes 2 µs, we expect future calls to take ~2 µs
  ▪ If we call `avl.find(x)` *m* times, the total runtime should be ~2*m* µs
Modifying Data Structures for Future Gains

- Path compression is entirely different: we are modifying the up-tree to improve future performance
  - If `uptree.find(x)` takes 2 µs, we expect future calls to take <2 µs
  - If we call `uptree.find(x) m` times, the total runtime should be <2m µs
    - ... and possibly even << 2m µs
Path Compression: Idea

❖ Recall the worst-case structure if we use weighted union:

❖ **Idea**: When we find(8), move all visited nodes under the root
   - Additional cost is insignificant (same order of growth), so run path compression on every find()
Path Compression: Example

- Recall the worst-case structure if we use weighted union

![Path Compression Diagram]

- Idea: When we find(8), move all visited nodes under the root
  - Additional cost is insignificant (same order of growth), so run path compression on every find()
  - Doesn’t meaningfully change runtime for this invocation of find(8), but subsequent find(8)s (and subsequent find(7)s and find(5)s and ...) will be faster!
Path Compression: Details and Runtime

- With “enough” find()s, we end up with a very shallow tree:

![Tree Diagram]

- How much is “enough”? Probably m>n

(hopefully we're finding unique values... )
Path Compression: Implementation

```c
int find(int x) {
    while (up[x] != 0) {
        x = up[x];
    }
    return x;
}
```

```c
int pathCompressionFind(int x) {
    while (up[x] > 0) {
        x = up[x];
    }
    int root = x;

    // Change the parent for all
    // nodes along this path
    while (up[x] > 0) {
        x = up[x];
        up[x] = root;
    }
    return root;
}
```

find()’s **worst-case** runtime is still $O(\log n)$!

Does this (slightly) added complexity help us make the 
up-trees shallower and improve sequences of find()?
Path Compression: Runtime

- A sequence of $m$ find()s on $n$ elements has total $O(m \log^* n)$ time
  - Assumes weighted union and path compression
  - See Weiss for proof

- $\log^* n$ is really cheap!
  - $\log^* n$ is the “iterated log”: the number of times you need to apply log to $n$ before the result is $\leq 1$
  - For all practical purposes, $\log^* n < 5\,$
  - So $O(m \cdot 5)$ for $m$ operations!

- So find() is amortized $O(1)$
  - And union() is still worst case $O(1)$

<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>$65536$</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
Interlude: A Really Slow Function

- Ackermann’s function is a really big function $A(x, y)$ with inverse $\alpha(x, y)$ which is really small

- $\alpha$ shows up in:
  - Computation Geometry (surface complexity)
  - Combinatorics of sequences

- How fast does $\alpha(x, y)$ grow?
  - Even slower than iterated log!
  - For all practical purposes, $\alpha(x, y) < 4$
Path Compression: **Tighter Runtime**

- A sequence of $m$ `union()`s + `find()`s on a set of $n$ elements has **worst-case** total $O(m \cdot \alpha(m, n))$ time
  - Assumes weighted union and path compression
  - Proved by Robert Tarjan in 1984
    - (Tarjan is also known for Fibonacci heaps and splay trees)
  - Complex analysis, but inverse-Ackermann’s is a tighter bound than iterated-log

- So `find()` is still **amortized** $O(1)$
  - Since $O(m \cdot 4)$ for $m$ operations!
  - And `union()` is still **worst case** $O(1)$