SCHOOL OF ENGINEERING

VANDERBILT UNIVERSITY

CS 3250
Algorithms
Minimum Spanning Trees

Kruskal's Algorithm and the Union-Find Data Structure

VANDERBILT UNIVERSITY

Announcements

- **HW3 will be split into (2) parts.**
  - One part will be due before the first exam.
  - One part will be due after the first exam.
- **HW3 Stop (HW3 – Part 1)**
  - You may work with a partner from this section.
  - You will take a quiz on Graphs in Brightspace. You can see the questions in advance, discuss them with your partner and go back and answer them.
  - You must list your partner's name on the quiz since only one of you "fills in" the quiz answers.
  - **Due date:** HW3 Part 1 (aka "STOP" HW) due by Friday, February 16th at 9AM.

Announcements

- **HW3 GO (HW3 – Part 2)**
  - You will work with you same partner Part 1 STOP (unless you are working by yourself).
  - Create a SlideDoc for one of the problems you answered on the quiz (choose one you got correct).
  - Follow the template for the SlideDoc and look at the examples on Brightspace.
  - **Due date:** HW3 Part 2 (aka "GO" HW) due by Wednesday, February 28th at 9AM.

Announcements

- **On your radar: Exam #1**
  - Wednesday, February 21st in class.
    - 50 minutes. In-class. Closed book.
    - Some multiple choice.
    - Some algorithm walkthrough.
    - Some written response including algorithm design and analysis.
  - Practice review this Friday in class.

Recap: Kruskal’s MST Algorithm

Kruskal’s algorithm is conceptually simple:

1. Order all the edges by ascending weight into a list L. Set the minimal spanning tree T=Ø (empty)
2. Examine the first edge in the list (the smallest).
   A. IF it forms a cycle, do **not** add it to the MST.
   B. ELSE add the edge/vertex to the MST and remove the edge from list L.
   C. Repeat until done.

REAL SIMPLE

Recap: Kruskal's Algorithm via Union/Find

- Many variations of Union/Find. Vandy approach is the one in red.
- **Union**
  - Lazy/Eager (in a bad way) Union
  - Smart Union by Rank
    - Rank by count
    - **Rank by height**
- **Find**
  - Simple Find
  - **Find with path compression**

Recap: Union by Rank via Height

The Basics of How Union by Rank Works:

1. For each set or "tree" maintain a rank that is the upper bound on the height of the tree.
2. When performing a union, choose the root with the smaller rank (shorter tree) and attach to the root with the larger rank (taller tree).
3. If two trees are of equal rank, one is arbitrarily chosen to point to the other. The rank of the tree is then incremented by 1.

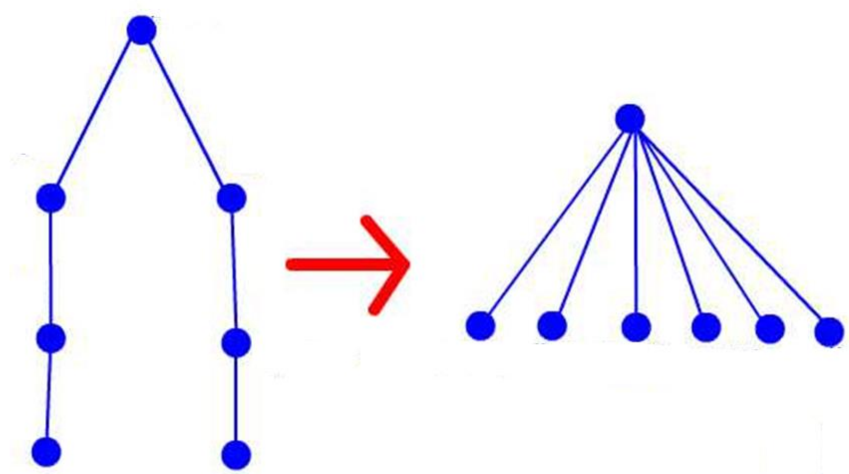
VANDERBILT UNIVERSITY

Recap: FindSet – Who's My Leader?

- In order to union **non-root** nodes, we will need to know their "set leader." This operation is called a FindSet.
- FindSet works by following "pointers" from v up the tree to the root to find the set identifier (aka leader) of v.
- Can we do anything clever here to help us?

Recap: FindSet with Path Compression

- **Idea:** To speed up findSet, we can move **everyone** we encounter on our path to the leader, to point to the leader.
- Under the hood, as recursive calls begin returning the set identifier, we'll change each node we've encountered along the path to "point" to the root.



VANDERBILT UNIVERSITY

Recap: FindSet with Path Compression

- Improved FindSet – We follow pointers from v "up the tree" to the root to find the set identifier of v.
  - ✓As recursive calls begin returning the set identifier, change each node we've encountered along the path to point to the root.
  - ✓This is known as **path compression** (also called a collapsing find).
  - ✓This will keep our **find** nearly constant time!

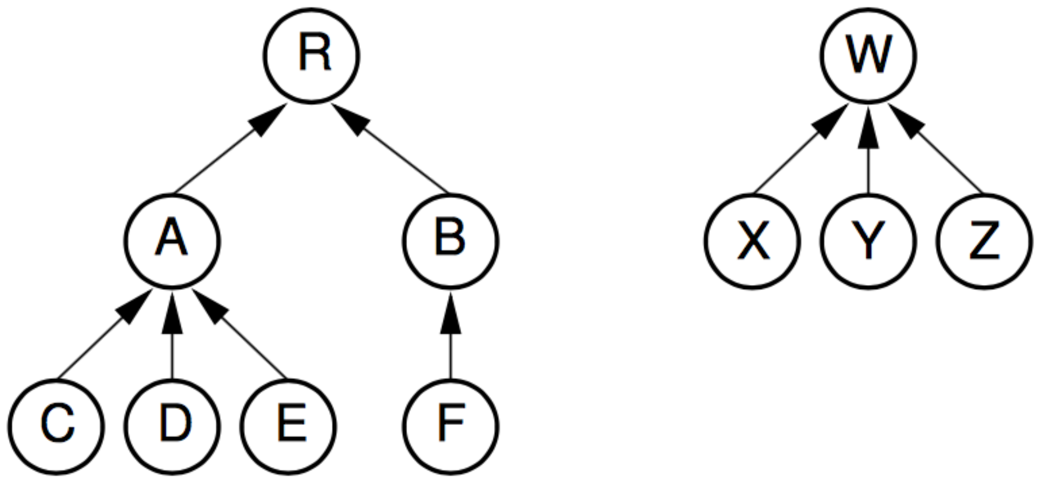
VANDERBILT UNIVERSITY

Recap: FindSet – Who's My Leader?

- A couple of additional notes about our FindSet:
  1. We represent union/find visually as a "tree."
  2. **Plot Twist!** Under the hood, we use an array of parent indices instead of pointers. This allows us to go instantly to any node and follow its parent "up" to determine the set leader.
  3. We do not update any height ranks during a FindSet operation, only during a union operation.

Under the Hood: Union/Find

- We use an array to help us locate the set leader easily.

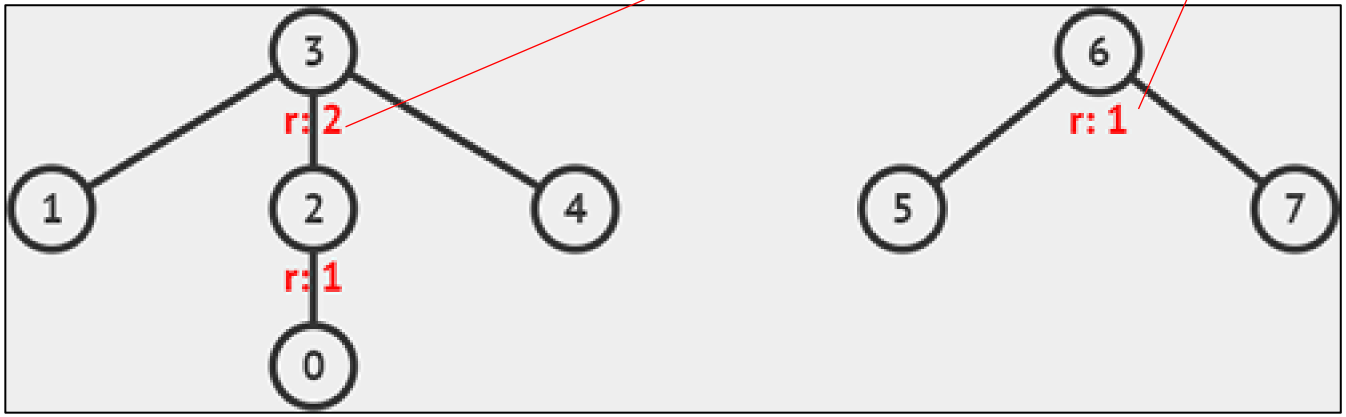


Parent's Index		0	0	1	1	1	2		7	7	7
Label	R	A	B	C	D	E	F	W	X	Y	Z
Node Index	0	1	2	3	4	5	6	7	8	9	10

Example: How Union/Find Works

- Let’s walkthrough a Union(2, 5).
  - We first need a FindSet(2).
  - We go directly to 2 (via array access).
  - Discover its parent is 3 which is the set leader.

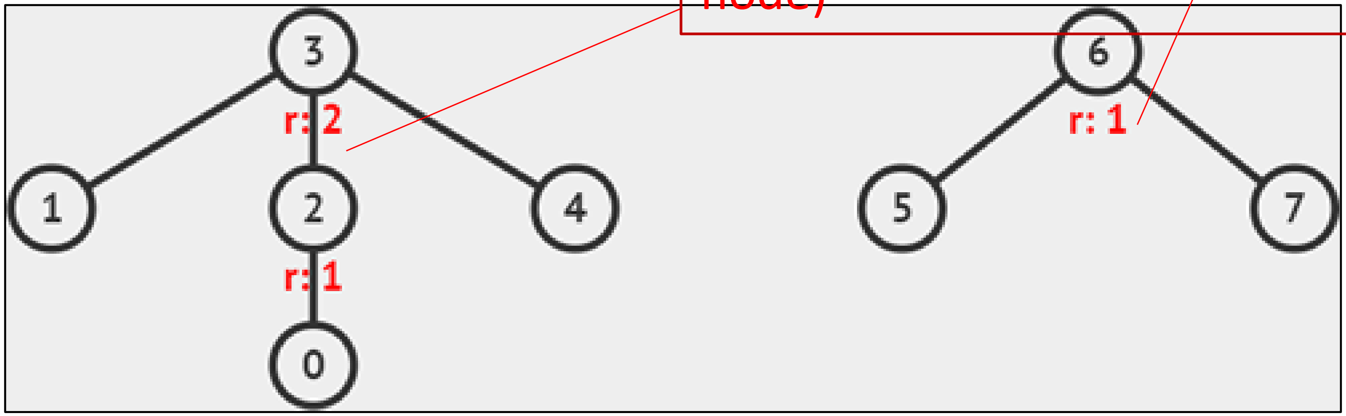
r is the rank (indicates the height of the node)



Example: How Union/Find Works

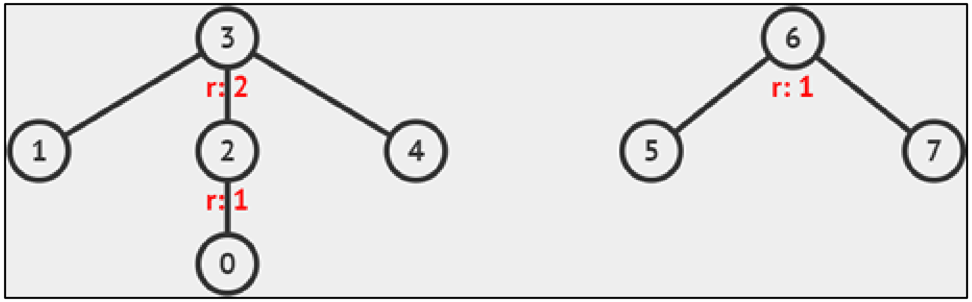
- Let’s walkthrough a Union(2, 5)
  - Now we need to do FindSet(5).
  - We go directly to 5 (via array access).
  - Discover its parent is 6 (the set leader).
  - This is the shorter tree so it will get updated during the actual union operation.

r is the rank (indicate height of the node)

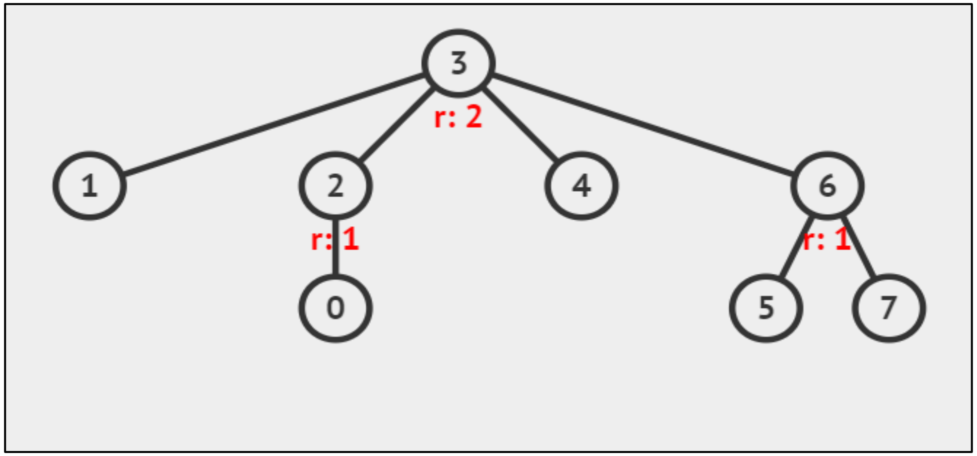


Example: How Union/Find Works

- Lastly, we do the Union operation for Union(2, 5)
  - The smaller ranked tree joins the other set.
  - We update ranks if necessary.



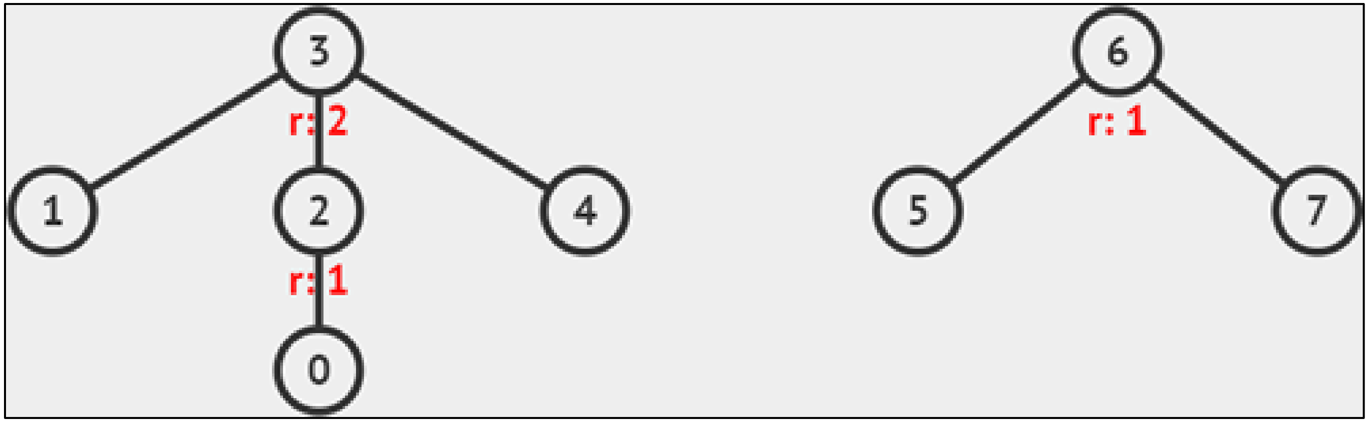
before union(2, 5)



after union(2, 5)

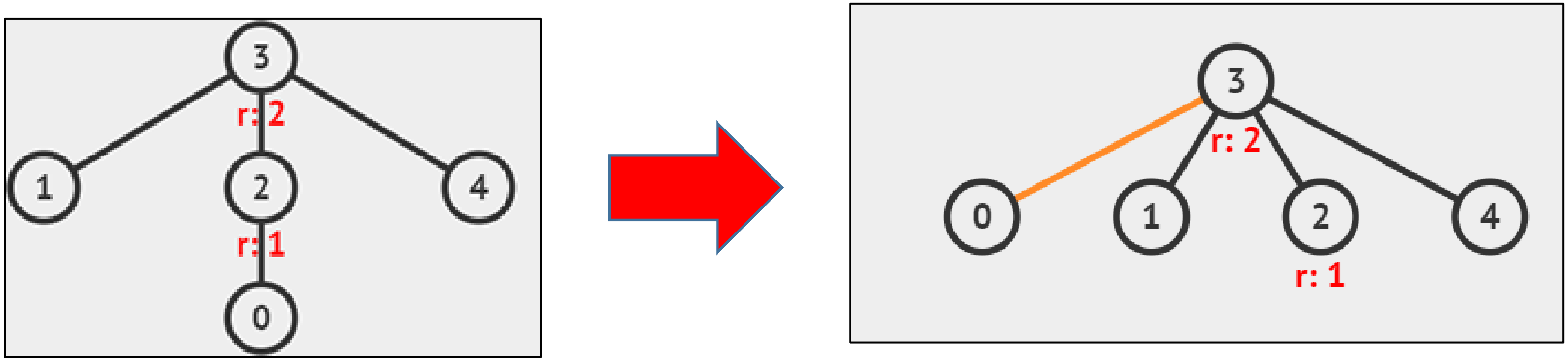
Example: How Union/Find Works

- Let's walkthrough a Union(0, 7)
  - First, we need to FindSet(0).
  - We go directly to 0 (array access).
  - Discover its parent is 2. We discover that 2's parent is 3 (the set leader).



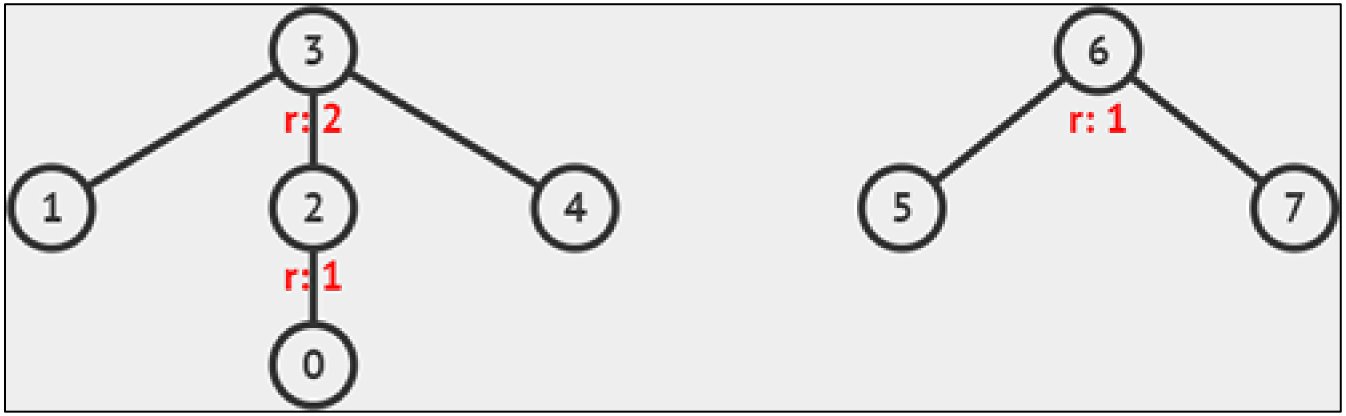
Example: How Union/Find Works

- Let's walkthrough a Union(0, 7) [continued]
  - First, we need to FindSet(0).
  - We go directly to 0 (array access).
  - Discover its parent is 2. We discover that 2's parent is 3 (the set leader).
  - We update 0 to "point" to the set leader.



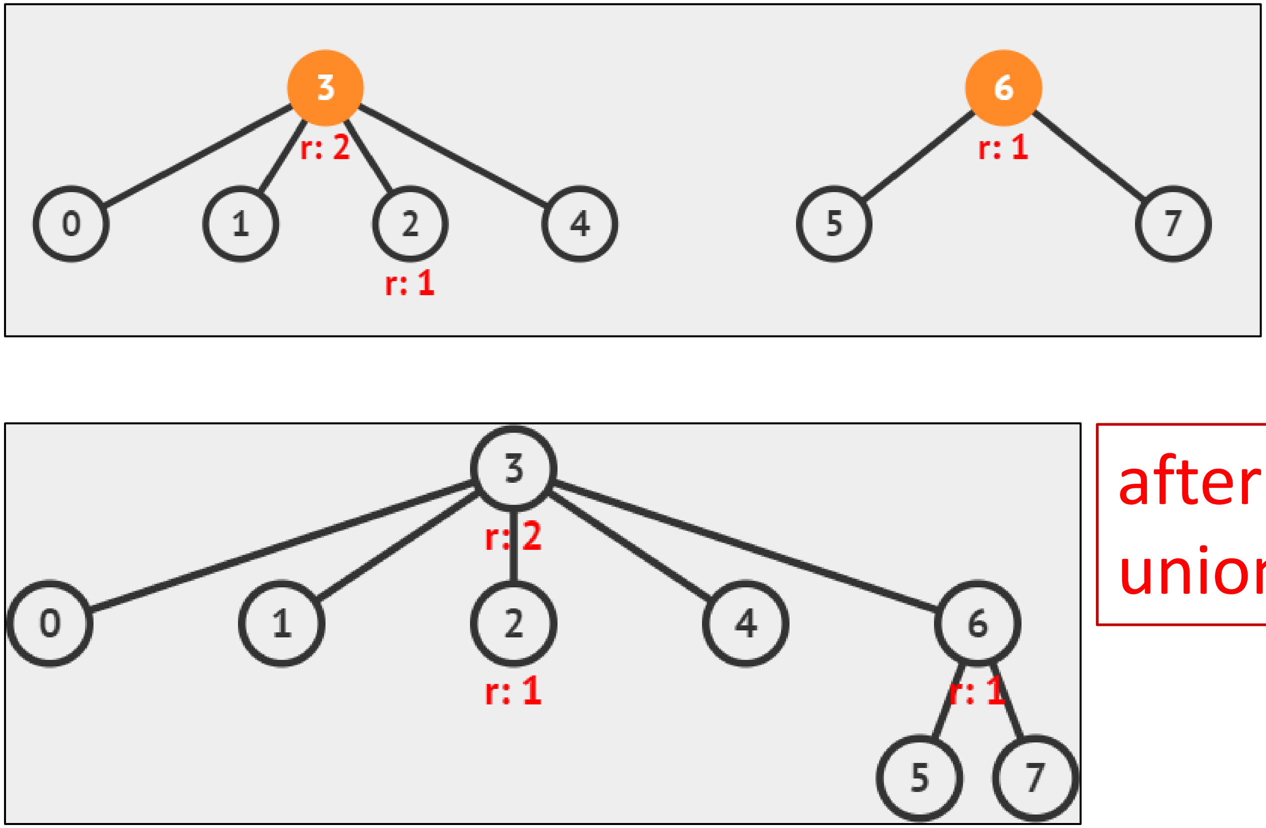
Example: How Union/Find Works

- Let’s walkthrough a Union(0, 7) [continued]
  - Now we need to FindSet(7)
  - We go directly to 7 (array access)
  - Discover its parent is 6 (the set leader).
  - This is the “shorter” tree so it will get updated during the actual union operation.



Example: How Union/Find Works

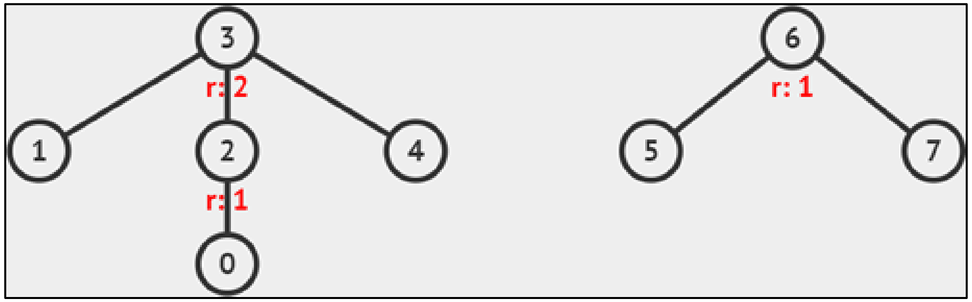
- Now we do the Union operation for Union(0, 7)
  - The smaller ranked tree gets joins the other set.
  - We update ranks if necessary.



after union(0, 7))

VisualGo.Net – A Visual Tool for Graphs

- Let’s watch the Union and FindSet in action on VisualGo.
- https://visualgo.net/en/ufds



Kruskal’s via Union/Find

- **Question:** If we implement union in the manner just described with a smart union, what do you think is the maximum depth our trees might achieve?
  - ElogE
  - VlogV
  - logV
  - V^2

Kruskal's via Union/Find

- **Question:** If we implement union in the manner just described, what do you think is the maximum depth our trees might achieve?
- **Answer:**
    - log(V)

Kruskal's Algorithm via Union/Find

1. Order the edges by weight into a list L. Set the minimal spanning tree T=Ø.
2. Examine the first edge in the list (the smallest).
   A. If it forms a cycle, do not add it to the tree
   B. Otherwise, add it to the tree

```
MST-KRUSKAL(G, w)
A = Ø
for each vertex v ∈ G.V
    MAKE-SET(v)
sort the edges of G.E into nondecreasing order by weight w
for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```

Kruskal’s via Union/Find: Analysis

- Sorting edges O(E log V)
- Each edge will be examined O(E)
- Total for all find and union operations is O(E) using the nearly constant time using smart **union by rank** with a **find incorporating path compression**.
- In other words, this implementation of Kruskal’s runs in O(E log V). The bottleneck is now the sorting of edges!
- Nice improvement!

VANDERBILT UNIVERSITY

Kruskal’s Algorithm via Union/Find

- Let’s see how another one of these approaches might do. Let’s try Lazy Union with Simple Find.
  - Lazy/Eager Union
  - Smart Union by Rank
    - Rank by count
    - Rank by height
  - Simple Find
  - Find with path compression

Who Cares Which Union/Find We Use?

- What about doing a Lazy Union/Simple Find?
  - Lazy Union with simple Find. Lazy means we attach one tree onto the other tree randomly.
  - No other enhancements, nor path compression during the Find.
  - Where is the bottleneck?

Who Cares Which Union/Find We Use?

- Question: Where is the problem in a lazy union/find?

  A. The tree height may grow to O(logV)

  B. The tree height may grow to O(V)

  C. The tree height may grow to O(V^2)

  D. We are unable to perform path compression

VANDERBILT UNIVERSITY

Who Cares Which Union/Find We Use?

- Let’s try a different idea…
  - How about having every vertex/node store their set leader.
  - Then we can instantly determine the set leader.
  - Sounds like a good idea, right?

Who Cares Which Union/Find We Use?

- When we "union" two components using this approach, one group of nodes always gets assigned a new "leader". How much work will it be to update the leader fields over the entire algorithm assuming n nodes?

A. Constant time

B. O(logn)

C. O(n)

D. O(n^2)

Who Cares Which Union/Find We Use?

- When we “union” two components using this approach, one group of nodes always gets assigned a new “leader”. How much work will it be to update the leader fields over the entire algorithm assuming n nodes?

  A. Constant time

  B. O(logn)

  C. O(n)

  D. O(n^2)

- Let’s talk about why

Who Cares Which Union/Find We Use?

- How much work is it to update the leader fields?
  - We can determine whether we should union quickly since each node knows who its leader is…O(1). That's good.
  - Hmmm…but when we "union" two components, one group of nodes gets assigned a new leader.
  - We will need to update the leader field in those nodes. That's bad.
  - We fixed one bottleneck and created another bottleneck. Let's check out this bottleneck.

VANDERBILT UNIVERSITY

Who Cares Which Union/Find We Use?

- How much work is it to update the leader fields?
  - Imagine the last union where we have n/2 nodes in each component. That means n/2 nodes need their leader updated.
  - Now, think backwards…we've had to do this every time while building up the components into a single MST component.
  - That's 1 + 2 + 3 + 4 + …. + n/2 = O(n^2).
  - This doesn't improve Kruskal's.
  - Let's stick with Union by Rank.

Under the Hood: Union/Find

- An example union-find Data Structure (courtesy of CMU). Notice they use rank by count rather than rank |

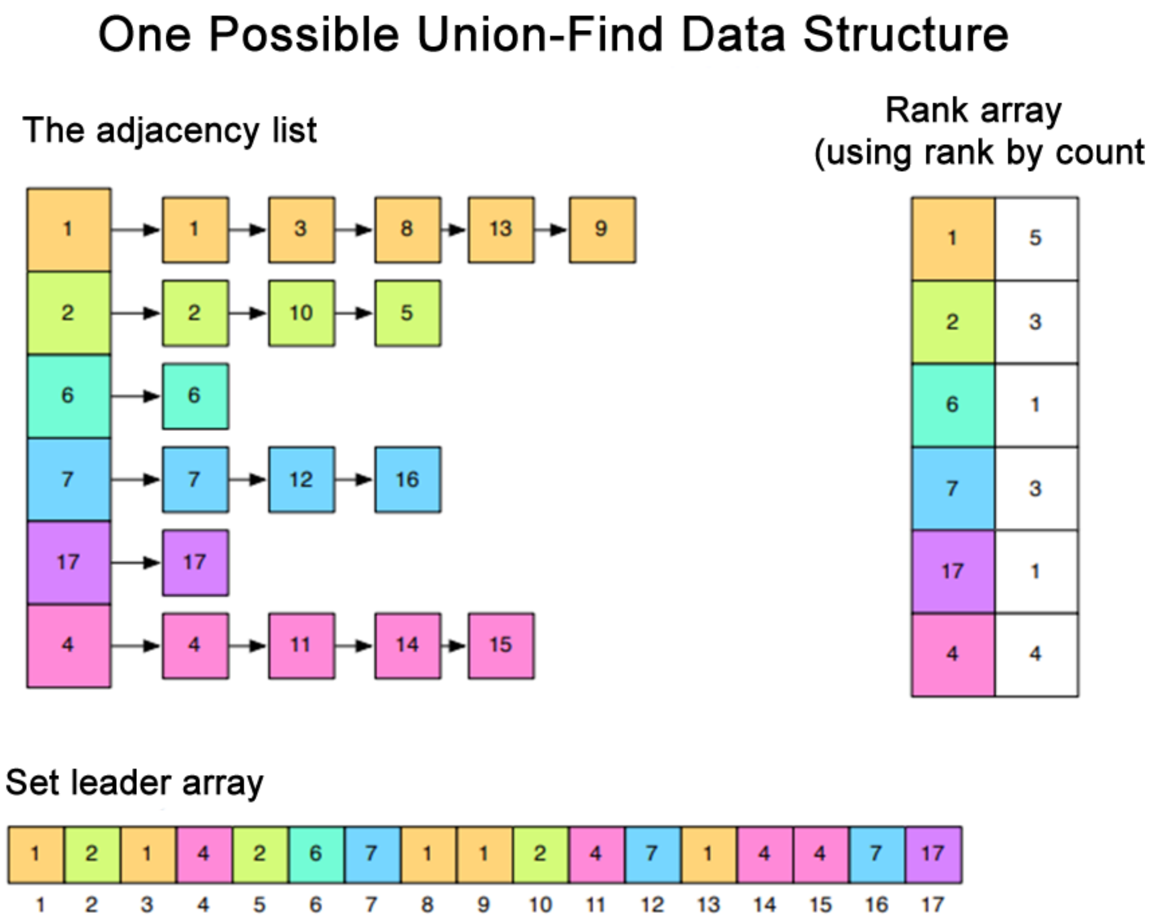


 Image courtesy of CMU. See https://www.cs.cmu.edu/~ckingsf/class/02713-s13/lectures/lec03-othermst.pdf

VANDERBILT UNIVERSITY

Graphs: Single Source Shortest Path

- **Question:** Suppose we have an undirected weighted graph where w is on the **shortest path** from u to v. Is the path from u to w (on the way to v) guaranteed to be the shortest path from u to w? Explain.

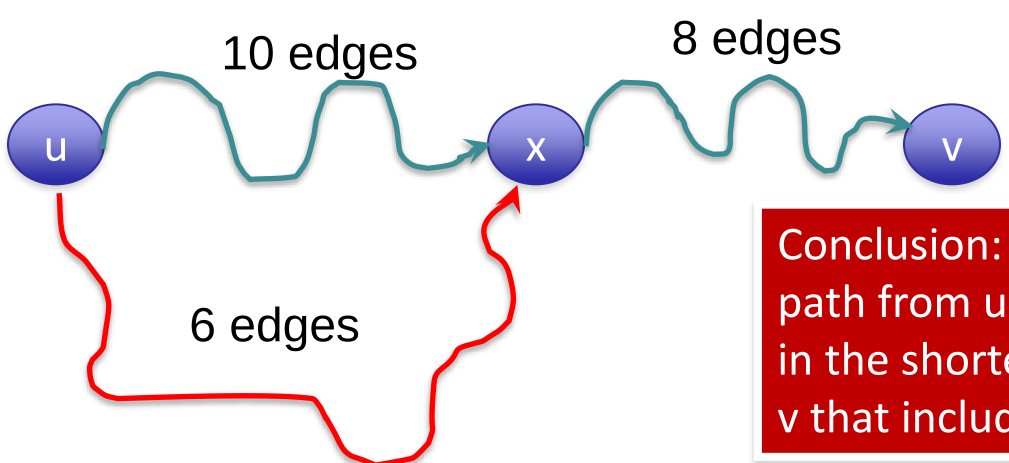
Graphs: Single Source Shortest Path

- **Question:** Suppose we have an undirected weighted graph where w is on the **shortest path** from u to v. Is the path from u to w (on the way to v) the shortest path from u to w? Explain.
- **Answer:**
- **Yes.** If there were a shorter path from u to w on the way to v, we would use that path to get from u to w on the way to v instead of the longer path.

VANDERBILT UNIVERSITY

Greedy Algorithms – Optimal Substructure

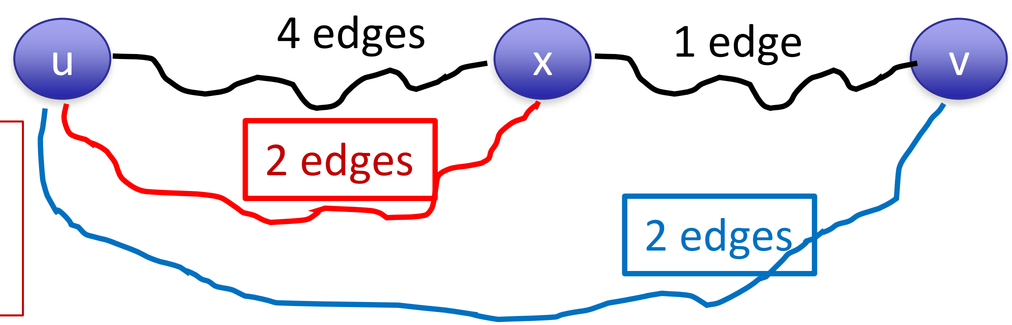
- **Shortest path** – Consider the general scenario depicted below. Is it possible that the shortest path from u to v that includes x does not house the shortest path from u to x?
- No. If there was a shorter path from u to x, we would use it to shorten the path from u to v.



Conclusion: The shortest path from u to x is contained in the shortest path from u to v that includes x

Greedy Algorithms – Optimal Substructure

- **Let's take a closer look…**
- Consider the scenarios below on an unweighted, undirected graph.



This path can't exist. If it existed, we would have used the red path instead.

This path can't exist or the shortest path from u to v wouldn't include x

That’s All For Now…

- Coming to a Slideshow Near You Soon…
  1. Shortest Path (Dijkstra’s Algorithm)

That’s All For Now