SCHOOL OF ENGINEERING

VANDERBILT UNIVERSITY

# CS 3250
# Algorithms
# Minimum Spanning Trees

## Kruskal's Algorithm and the Union-Find Data Structure

# 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.

VANDERBILT UNIVERSITY

# Announcements

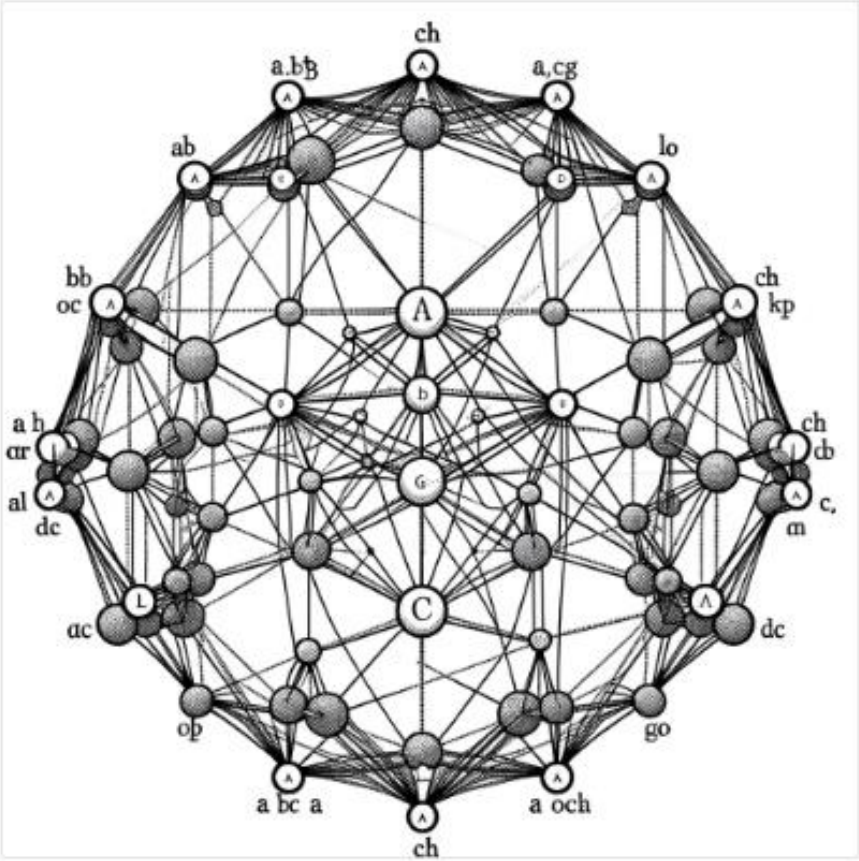
- **On your radar: Exam #1**
  - Wednesday, February 21st in class.
  - Closed book. No notes. No electronics.
  - Stay tuned for more details.

# Studying for the Exam with ChatGPT

**You**

give me an undirected graph so i can practice tarjans articulation point algorithm

**ChatGPT**




Here is an illustration of a complex undirected graph. You can use this graph to practice Tarjan's articulation point algorithm. The graph includes multiple nodes,

# PRIM'S MST Algorithm Analysis

1. Create a set mstSet that keeps track of vertices from G already included in the MST (this is initially empty).
2. Assign an initial key value to all vertices in the input graph. Initialize all key values as INFINITE (some people use null).
3. Assign key value as 0 to the first vertex to be included in the MST so it is picked in the step #4.
4. While mstSet doesn't include all the vertices in G
   a) Pick a vertex **u** not in the mstSet that has minimum key value.
   b) Add u to the mstSet.
   c) Update the key value of all vertices adjacent to u.
      - To update the key values, iterate through all adjacent vertices to u. For each adjacent vertex v, if the weight of edge (u, v) is less than the previous key value of v, update the key value of v to be weight of (u, v).

O(V)

O(1)

O(V)

O(E) total

V times

Where's the bottle neck?

# PRIM'S MST Algorithm Analysis

1. Create a set mstSet that keeps track of vertices from G already included in the MST (this is initially empty).
2. Assign an initial key value to all vertices in the input graph. Initialize all key values as INFINITE (some people use null).
3. Assign key value as 0 to the first vertex to be included in the MST so it is picked in the step #4.
4. While mstSet doesn't include all the vertices in G
   - **a) Pick a vertex u not in the mstSet that has minimum key value.**
   - b) Add u to the mstSet.
   - c) Update the key value of all vertices adjacent to u.
   - To update the key values, iterate through all adjacent vertices to u. For each adjacent vertex v, if the weight of edge (u, v) is less than the previous key value of v, update the key value of v to be weight of (u, v).

O(V)

O(1)

O(V) each time

V times

O(E) total

Here's the bottle neck?

# PRIM'S MST Algorithm Analysis

- In a nutshell:
  1. Look though vertices to find cheapest to add.
  2. Put that one in the MST.
  3. Repeat until all vertices are in the MST.
- Hmmm…Prim's algorithm sounds a lot like selection sort. That's how the analysis looks as well. In the worst case, on the first pass, we consider V-1 vertices, then V-2 vertices, etc.

$$\sum_{i=1}^{V-1} i = \frac{V(V-1)}{2} = O(V^2)$$

# PRIM'S MST Algorithm

- **Spoiler alert:** We will improve this runtime once we learn a few more tricks such as heaps aka priority queues. For now, our version of Prim's runs in $O(V^2)$.
- Let's look at another algorithm, called Kruskal's, which attempts to reduce the bottleneck of repeatedly locating the minimum.


VANDERBILT UNIVERSITY

# Kruskal's MST Algorithm

**Kruskal's algorithm is conceptually simple:**

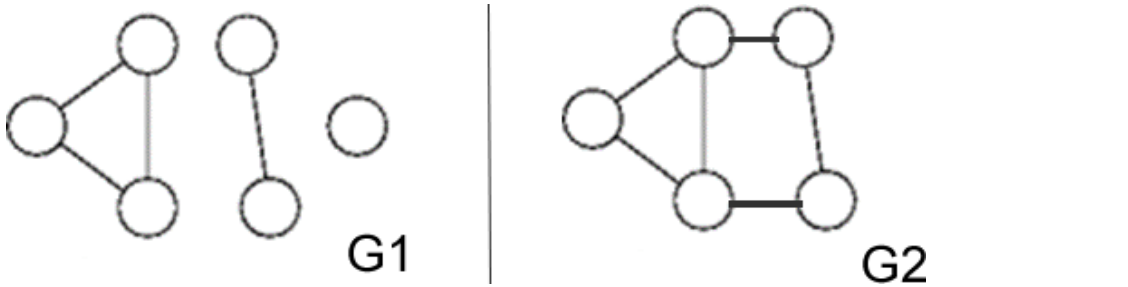
1. Order all the edges by ascending weight into a list L. Set the minimal spanning tree T=Ø (empty)
2. Examine the first edge in the list (the smallest).
   A. IF it forms a cycle, do **not** add it to the MST.
   B. ELSE add the edge/vertex to the MST and remove the edge from list L.
   C. Repeat until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

REAL SIMPLE

VANDERBILT UNIVERSITY

# Kruskal's MST Algorithm

- As we did with Prim's, a couple of assumptions to make our life easier.
  1. **The graph is connected**. If a graph is not connected, one can adapt the algorithm to compute the MSTs of each of its connected components (known as a minimum spanning forest – MSF).




Not connected

A Connected Graph

# Kruskal's MST Algorithm

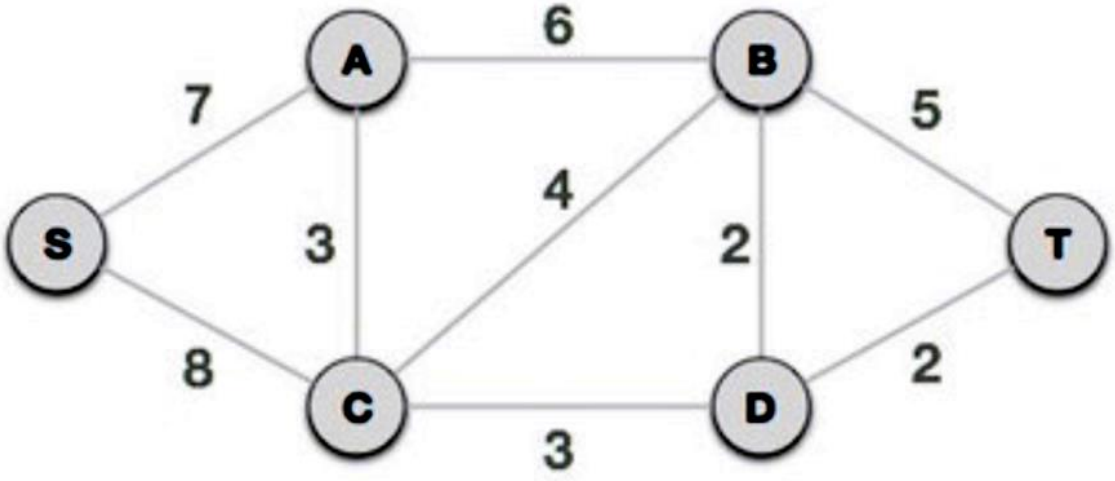
- As we did with Prim's, a couple of assumptions can make our life easier.
  2. **The edge weights are unique.** If the edge weights are unique, this guarantees a unique MST. Kruskal's works even if the edges are not unique. Assuming uniqueness gives a single correct answer and makes any proof of correctness a little easier.


VANDERBILT UNIVERSITY

# Kruskal's MST Algorithm

- **Example:** Walk through Kruskal's algorithm for the pictured graph below.
- **Step 1:** Order all edges in non-decreasing order by weight. This gives us:

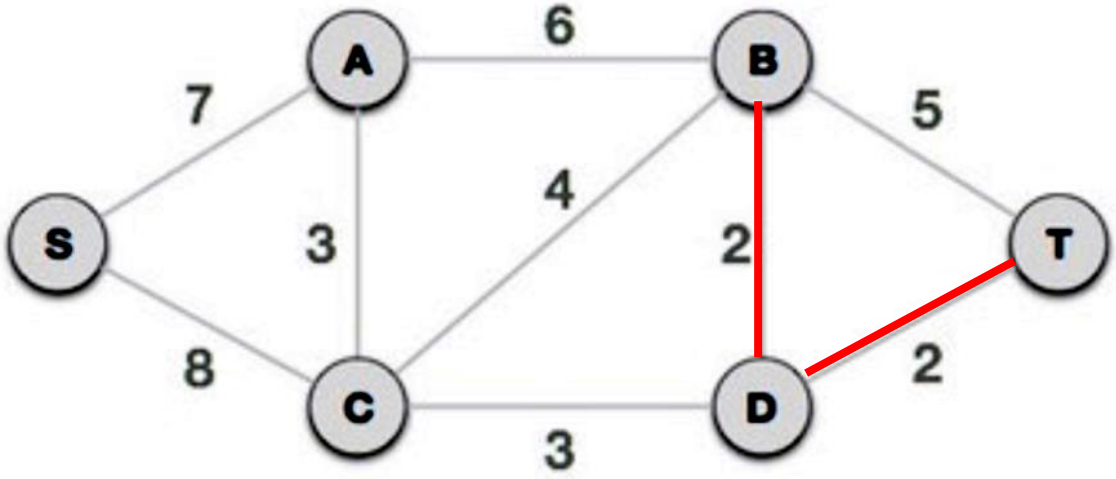
  L = {(B,D,2), (D,T,2), (A,C,3), (C,D,3), (C,B,4), (B,T,5), (A,B,6), (S,A,7), (S,C,8)}



# Kruskal's MST Algorithm

L = {(B,D,2), (D,T,2), (A,C,3), (C,D,3), (C,B,4), (B,T,5), (A,B,6), (S,A,7), (S,C,8)}

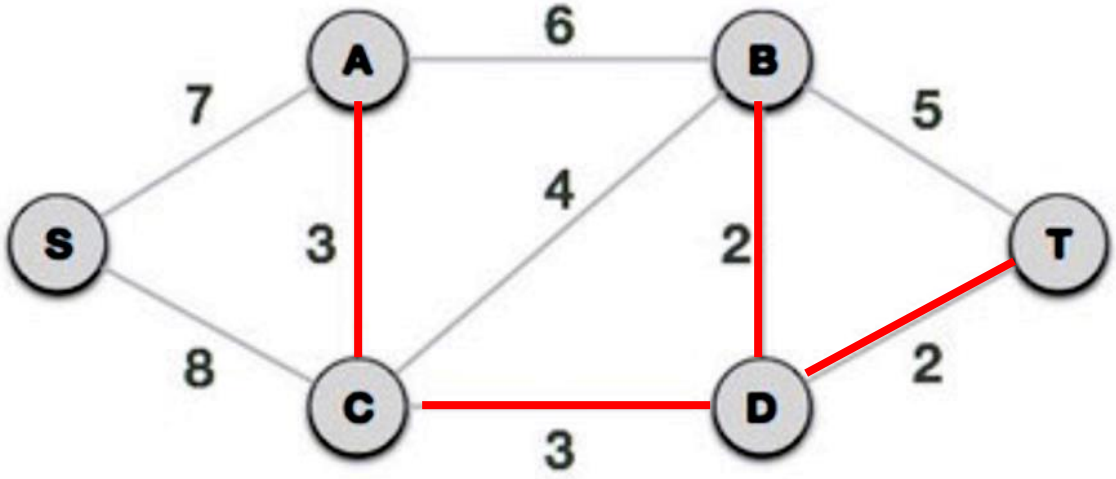
- **Step 2:** Start adding edges from the edge list L provided adding the edge does not create a cycle.
- Add (B,D,2) and (D,T,2) to our MST.



# Kruskal's MST Algorithm

L = {~~(B,D,2)~~, ~~(D,T,2)~~, (A,C,3), (C,D,3), (C,B,4), (B,T,5), (A,B,6), (S,A,7), (S,C,8)}

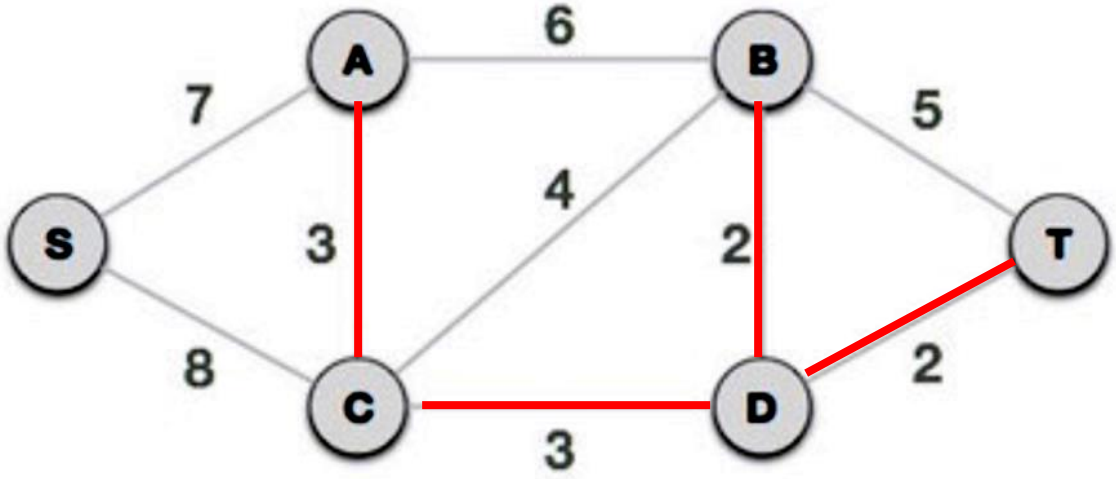
- **Step 3:** Continue adding edges from our set as long as they do not create a cycle. Next, we will add (A,C,3) and (C,D,3) to our MST.



# Kruskal's MST Algorithm

L = {~~(B,D,2)~~, ~~(D,T,2)~~, ~~(A,C,3)~~, ~~(C,D,3)~~, (C,B,4), (B,T,5), (A,B,6), (S,A,7), (S,C,8)}

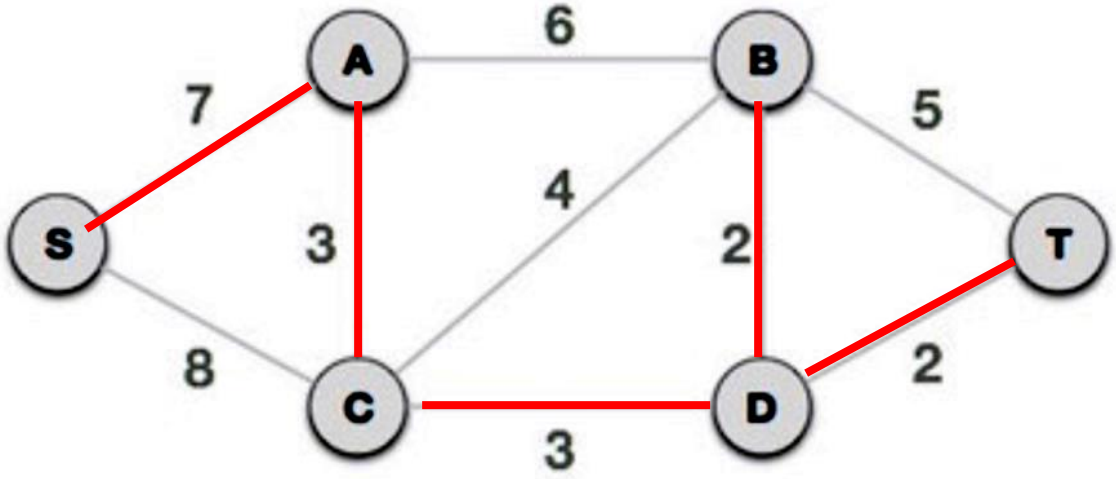
- **Step 4:** Continue adding edges from our set as long as they do not create a cycle. Skip (C,B,4), (B,T,5) and (A,B,6) since they would create a cycle.



# Kruskal's MST Algorithm

L = ~~{(B,D,2)~~, ~~(D,T,2)~~, ~~(A,C,3)~~, ~~(C,D,3)~~, ~~(C,B,4)~~, ~~(B,T,5)~~, ~~(A,B,6)~~, (S,A,7), (S,C,8)}

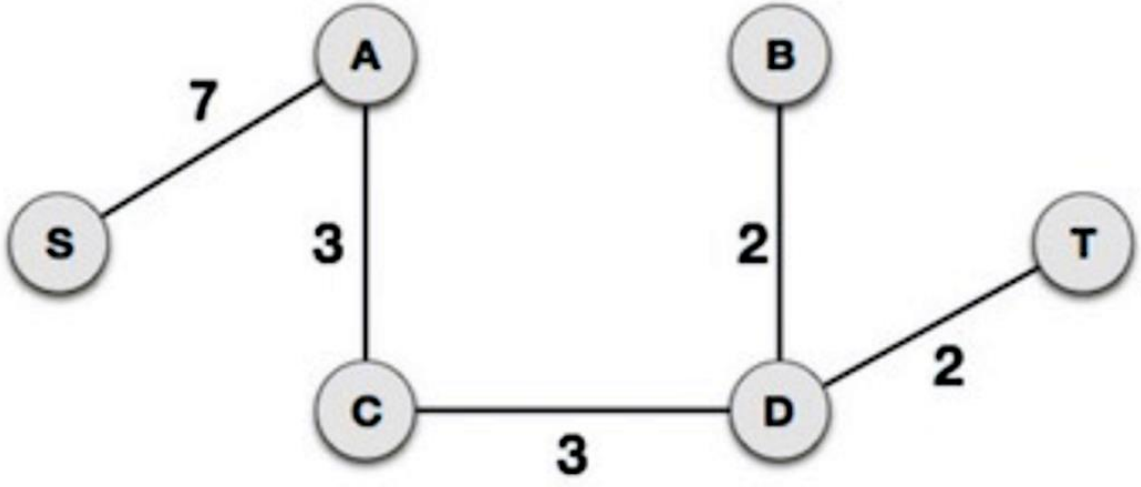
- **Step 5:** The next edge is (S,A,7) which we can add to the MST.



# Kruskal's MST Algorithm

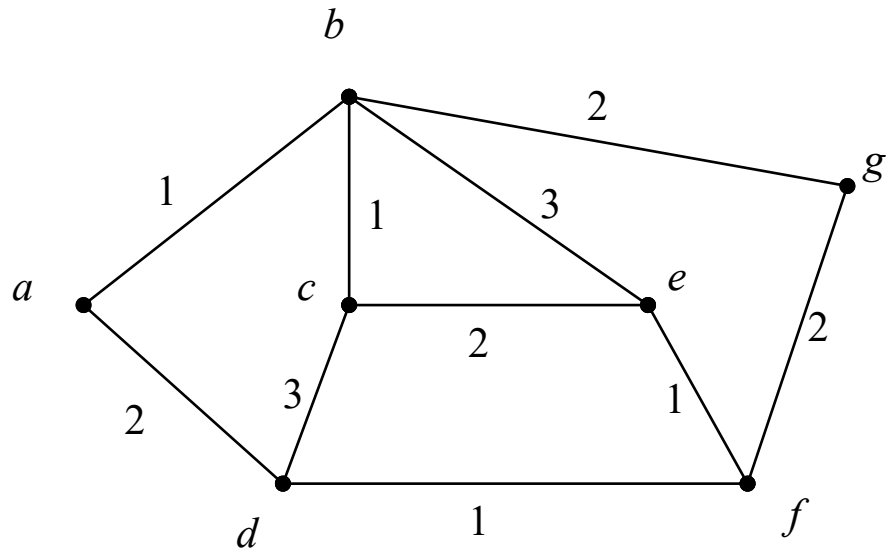
L = ~~{(B,D,2)~~, ~~(D,T,2)~~, ~~(A,C,3)~~, ~~(C,D,3)~~, ~~(C,B,4)~~, ~~(B,T,5)~~, ~~(A,B,6)~~, ~~(S,A,7)~~, (S,C,8)}

- All the vertices are now part of the MST, so we are done. Our final MST using Kruskal's algorithm has total weight = 17.



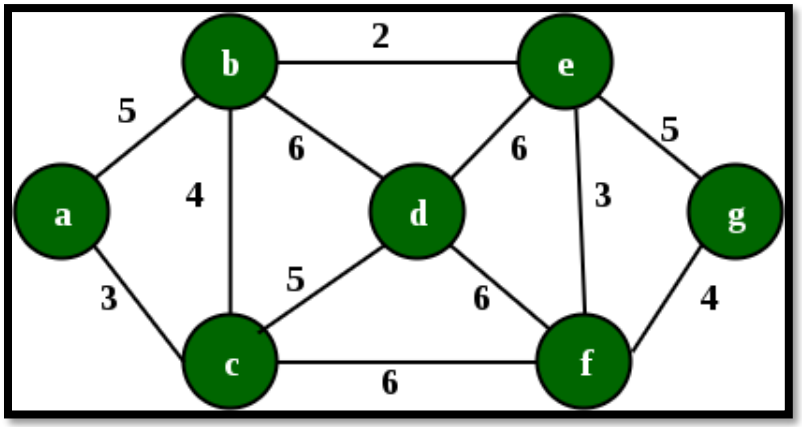
# On Your Own: Kruskal's

- **Practice On Your Own:** Walk through Kruskal's algorithm and create an MST for the graph pictured below.




# Kruskal's MST Algorithm

**Question:** Out of sequences below, which one is **not** a possible sequence of edges added to the MST for the graph below using Kruskal's algorithm. Explain.




(A) (b,e), (e,f), (a,c), (b,c), (f,g), (c,d)

(B) (b,e), (e,f), (a,c), (f,g), (b,c), (c,d)

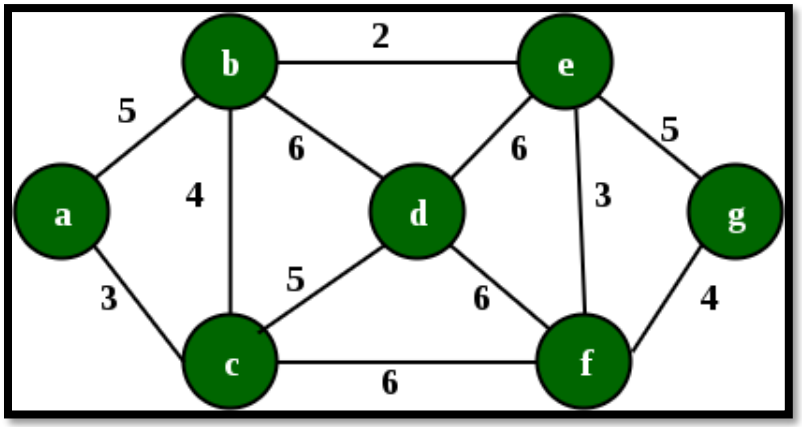
(C) (b,e), (a,c), (e,f), (b,c), (f,g), (c,d)

(D) (b,e), (e,f), (b,c), (a,c), (f,g), (c,d)


VANDERBILT UNIVERSITY

# Kruskal's MST Algorithm

**Question:** Out of sequences below, which one is **not** a possible sequence of edges added to the MST for the graph below using Kruskal's algorithm. Explain.




(A) (b,e), (e,f), (a,c), (b,c), (f,g), (c,d)

(B) (b,e), (e,f), (a,c), (f,g), (b,c), (c,d)

(C) (b,e), (a,c), (e,f), (b,c), (f,g), (c,d)

**(D) (b,e), (e,f), (b,c), (a,c), (f,g), (c,d)**


VANDERBILT UNIVERSITY

# Kruskal's Algorithm: Analysis

- **Question:** Kruskal's performs better than Prim's because it eliminates the bottleneck.
  A. Absolutely
  B. No way
  C. I'm not sure yet

# Kruskal's Algorithm: Analysis

- **Question:** Kruskal's performs better than Prim's because it eliminates the bottleneck.

  A. Absolutely

  **B. No way**

  C. I'm not sure yet

# Kruskal's MST Algorithm

- Like Prim's algorithm, **Kruskal's** is an example of a greedy algorithms.
- A **greedy algorithm** is any algorithm that follows the problem-solving heuristic of making the best choice at each stage with the intent of finding the overall global best choice.
- It is the philosophy of "No Regrets"

VANDERBILT UNIVERSITY

# Kruskal's MST Algorithm: Analysis

We'll need a little light math for our Kruskal's analysis:

1. A good sorting algorithm works in O(n log n). This means sorting the edges will be O(E log E). Can we bound this better?
2. We also know $E \leq V^2$. If we take the log of both sides, that means $\log(E) \leq \log(V^2)$
3. Basic log properties tells us $\log(E) \leq 2\log(V)$. So, this means log(E) is O(log V).
4. A better bound for sorting edges is O(E log V).

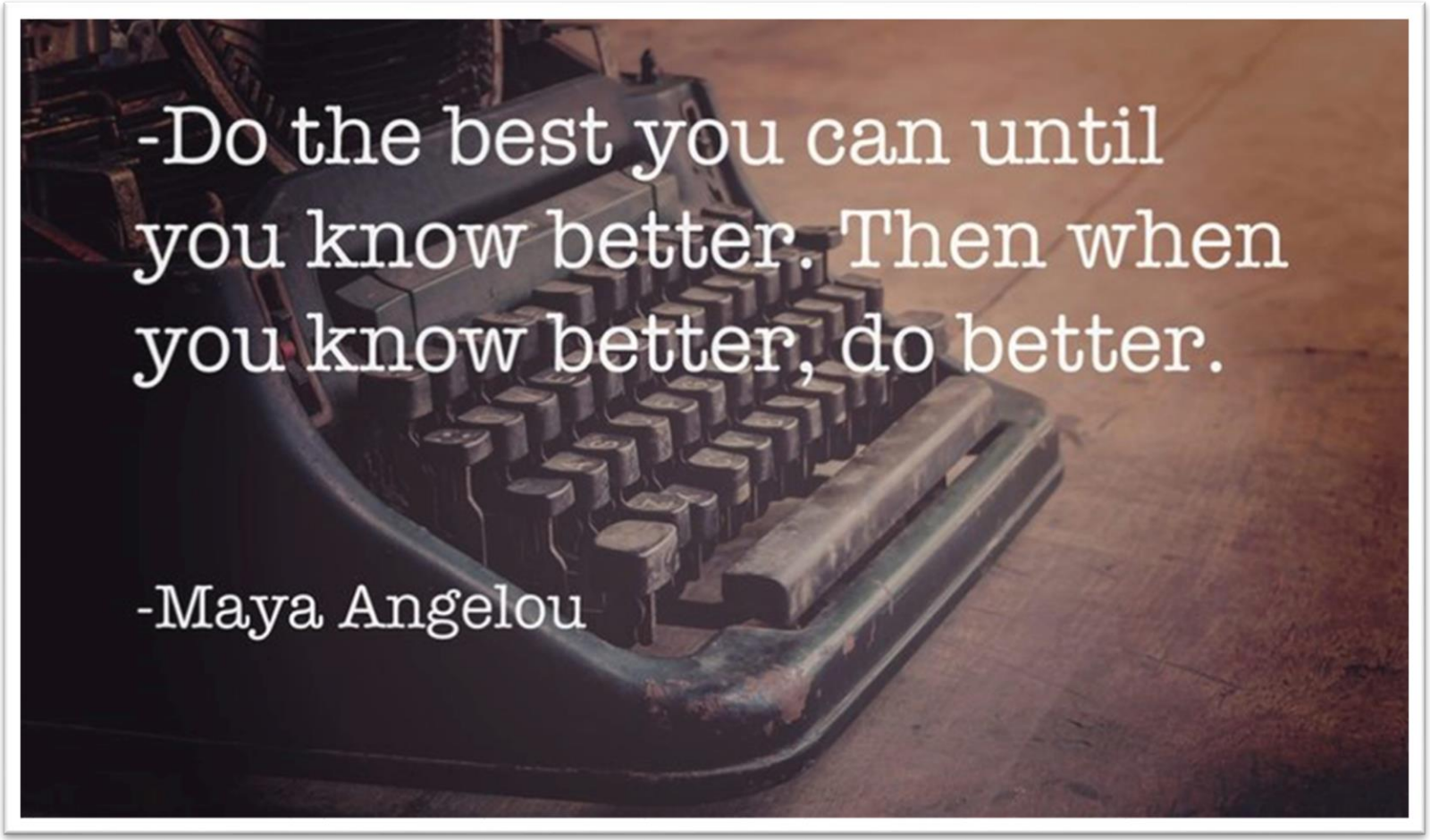
# Kruskal's MST Algorithm: Rough Analysis

1. Sort the edges by weight. ← O(ElogV)
2. Examine the edge to see if it creates a cycle.
   - Assume a sparse graph with V vertices and V edges.
   - Using a simple DFS/BFS to detect cycles in MST (linear in number of vertices). Do this for every edge.

O(V²)

- Hmmm…this is no better than Prim's. We fixed one bottleneck and created a different one.
- Can we be smarter about this?

# Kruskal's Algorithm via Union/Find




VANDERBILT UNIVERSITY

# Kruskal's Algorithm via Union/Find

- **Observation**: Adding edge (u, v) creates a cycle if vertices u and v are in the same component.
- Let's start with each node being in its own component. When we add the next edge…
  1. If u and v are in same component, adding the edge will create a cycle.
  2. Otherwise, we can merge the two components from u and v into one bigger component.
- This is the idea behind the **Union/Find data structure**.

# Kruskal's Algorithm via Union/Find

- More formally…each vertex u is initially in its own **equivalence class**.
- An edge between two vertices, u and w is the joining or union of two equivalence classes.
  - If u and w already belong to the same equivalence class, adding the edge to the MST would create a cycle. Don't add it.
  - Otherwise, there is no such path between u and w in the MST. The edge can be added, and the equivalence classes merged.

# Kruskal's Algorithm via Union/Find

- Many variations of Union/Find.
- It's important when working on a union/find problem to make sure everyone is using the same approach.
  - Lazy/Eager Union
  - Smart Union by Rank
    - Rank by count
    - Rank by tree height
  - Simple Find
  - Find with path compression

VANDERBILT UNIVERSITY

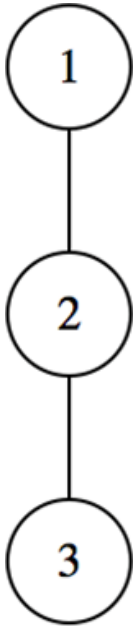
# Union by Rank via Height

**The Basics of How Union by Rank Works:**

1. For each set or "tree," maintain a rank that is an 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.


# Union by Rank via Height

- **Overall Objective:** Try to keep trees "short" so we can determine if a cycle exists quickly.
- **Why?**
- If we don't choose the root with the smaller rank (shorter tree), during a union, our "trees" might grow into a long list, slowing us down.



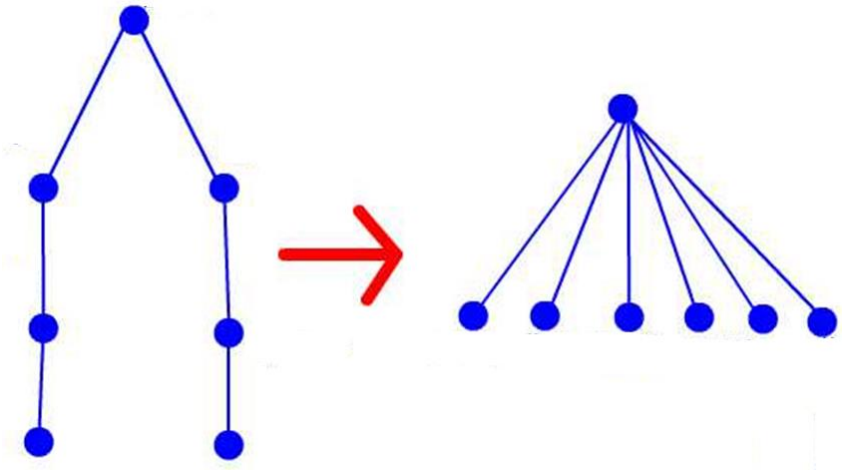
# FindSet – Who's My Leader?

- 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?


VANDERBILT UNIVERSITY

# 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.



# 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**.
  - ✓This will keep our **find** nearly constant time!


VANDERBILT UNIVERSITY

# FindSet – Who's My Leader?

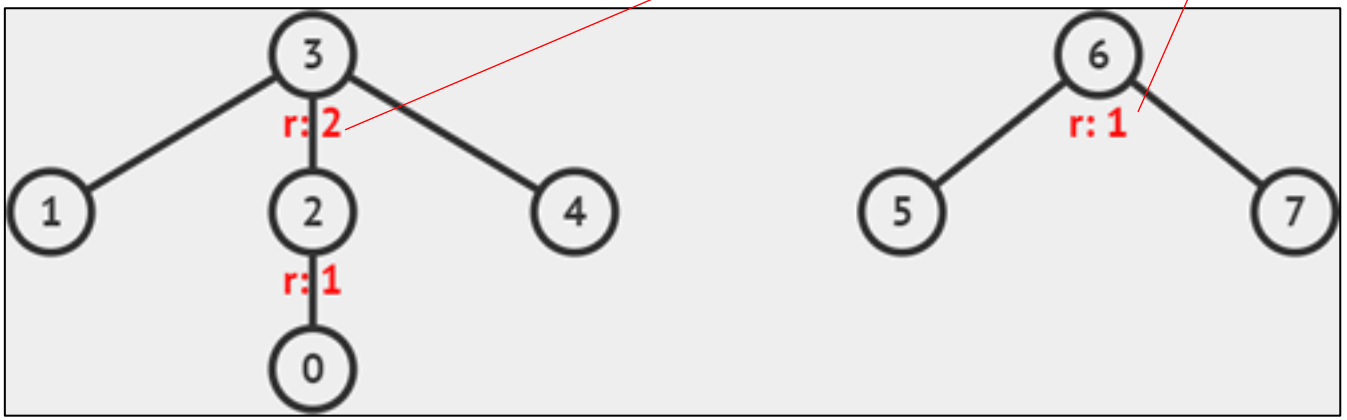
- A couple of additional notes about our FindSet:
  1. We represent union/find visually as a "tree."
  2. **Plot Twist!** Under the hood, we use an array of parent indices instead of pointers. This allows us to go instantly to any node and follow its parent "up" to determine the set leader.
  3. We do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

VANDERBILT UNIVERSITY

# Example: How Union/Find Works

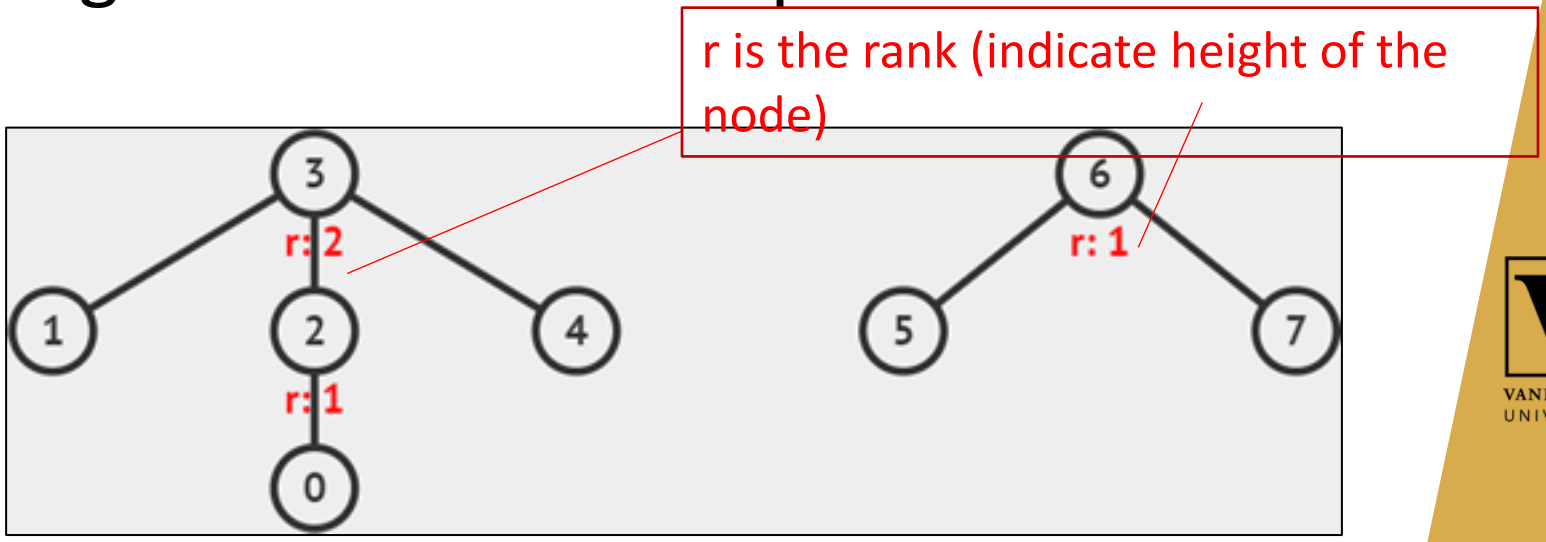
- Let's walkthrough a Union(2, 5).
  - We first need a FindSet(2).
  - We go directly to 2 (via array access).
  - Discover its parent is 3 which is the set leader.

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



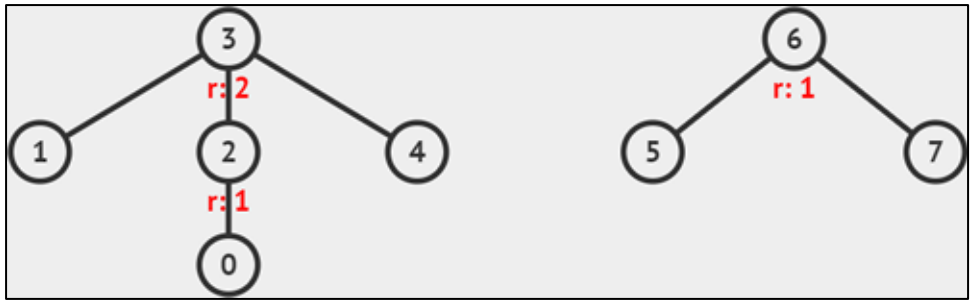
# Example: How Union/Find Works

- 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.

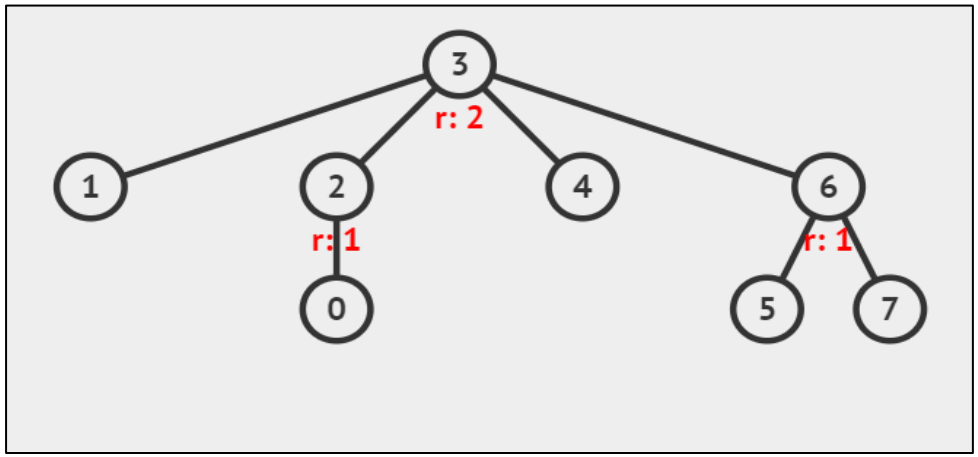


# Example: How Union/Find Works

- 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)

# That’s All For Now…

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

# That’s All For Now