CS340 - Analysis of Algorithms

Greedy Algorithms
Minimum Spanning Trees

Announcements:

HW3 due today

Upcoming deadlines:

- Lab3 due Thursday (9/25)
- Project checkpoint 1 (10/5)
- HW4 posted - due next monday (10/6)
- Midterm (10/8)

Agenda

1. Warmup - a new data structure

2. Minimum Spanning Trees
    a. Greedy algorithm to find them

Warmup: A New Data Structure

Disjoint Set: Stores a collection of disjoint (non-overlapping) sets

- also called Union-Find

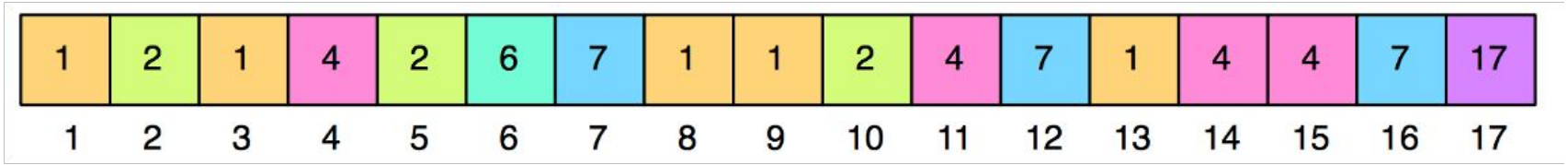
Operations:

1. `create(u)` : creates a set containing a single item u
2. `find(u)` : find the set that contains u
3. `union`(u, v) : merge the set containing u and the set containing v

How can we implement this efficiently?

Array Based Disjoint Set

Each entry contains the name of the set that the element belongs to



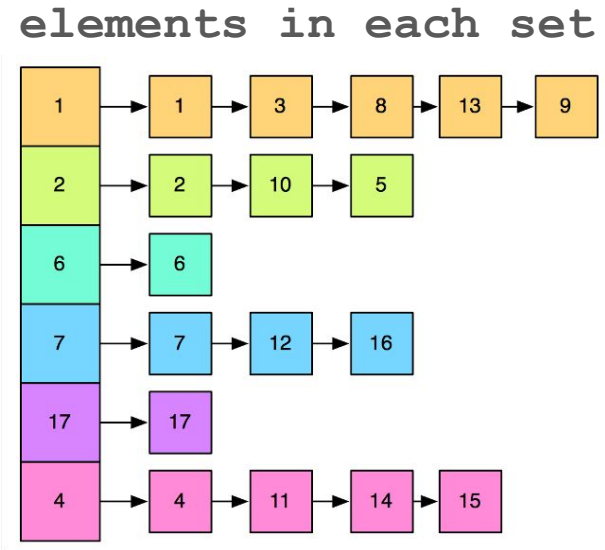
Runtime complexity?

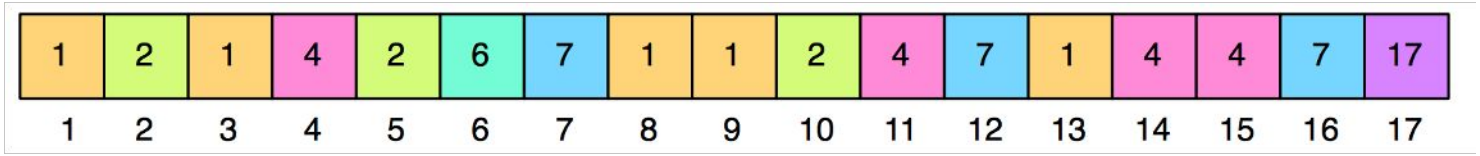
1. `find(u)`
   a. O(1)

2. `union`(u, v) - update all of the entries for each elem in u's set and each elem in v's set
   a. O(n)

Array Based Disjoint Set with Optimizations

1. Can we avoid looping through the entire array to know which ones need to be updated?
   a. Maintain a list of elements in each set
   b. Now we don't need to look through the whole array to find elements that need updating





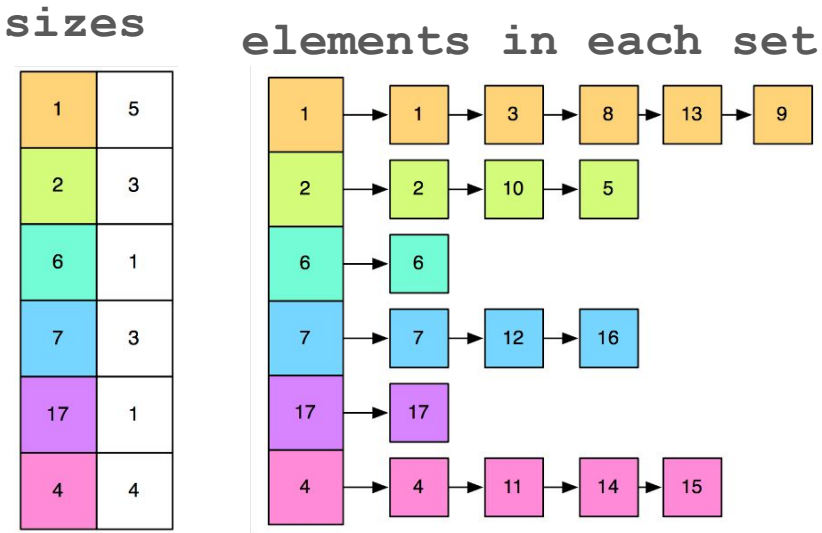
Array Based Disjoint Set with Optimizations

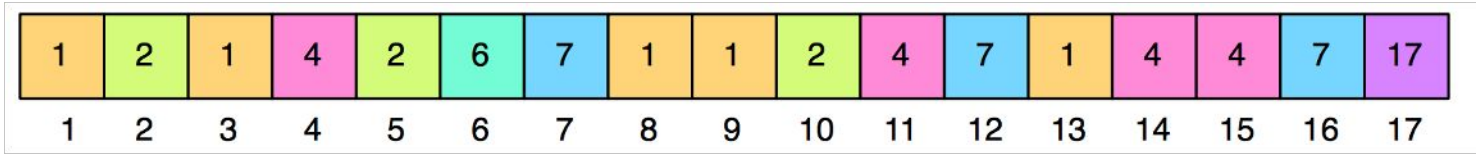
2. If we're merging sets A and B, instead of creating a new set C and modifying the entries for all elements in A and B, let's just put all of A into B

If set A is large, this doesn't help very much..

Let's maintain an additional array **`sizes`**

On **`union`**, merge into the larger set



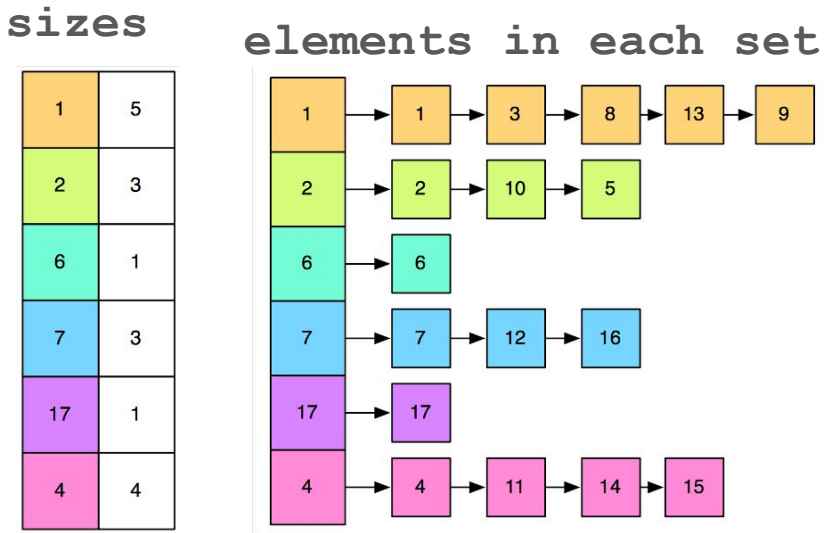


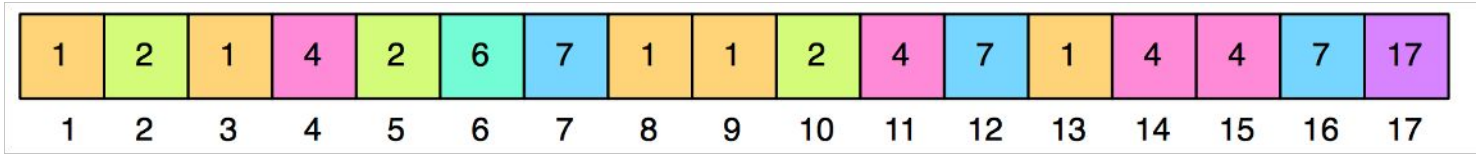
Array Based Disjoint Set with Optimizations

With these optimizations, what is the runtime complexity?

`union(u,v)`:

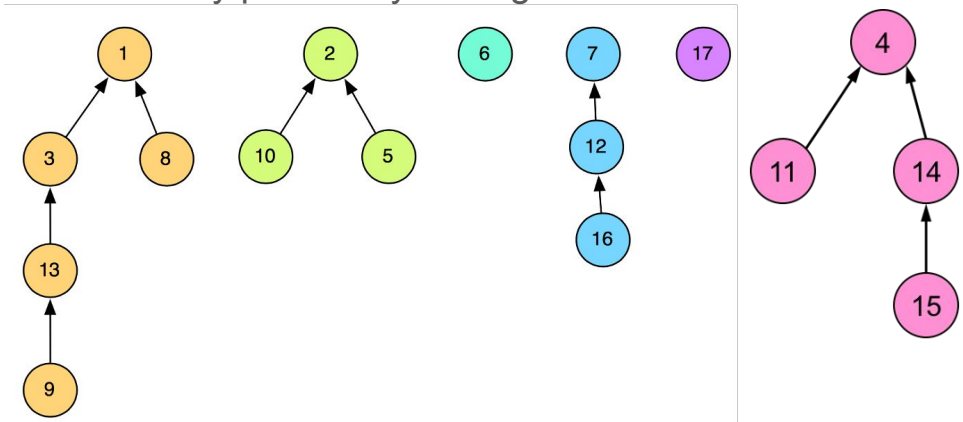
1. Retrieve the set A s.t. u ∈ A
   a. O(1)
2. Retrieve the set B s.t. v ∈ B
   a. O(1)
3. Let S be the larger set of A and B and T be the smaller set
   a. O(1)
4. Foreach elem in T, update the entry to be S
   a. O(|T|)





Pointer Based Disjoint Set

- Name each set after one of its elements
- Each node will contain a pointer to the name of its set (root)
- `union(u, v)` where u ∈ A and v ∈ B, update u's pointer to point to v
  - We do not need to update any other pointers
  - As a result, the name of the set an element belongs to, must be computed by following pointers
  - The parent relationships represent the sets they previously belonged to.



Pointer Based Disjoint Set

```
union(w, u)
union(s, u)
union(t, v)
union(z, v)
union(i, x)
union(y, j)
union(x, j)
union(u, v)
```

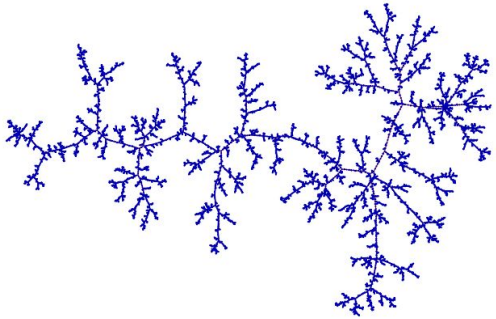
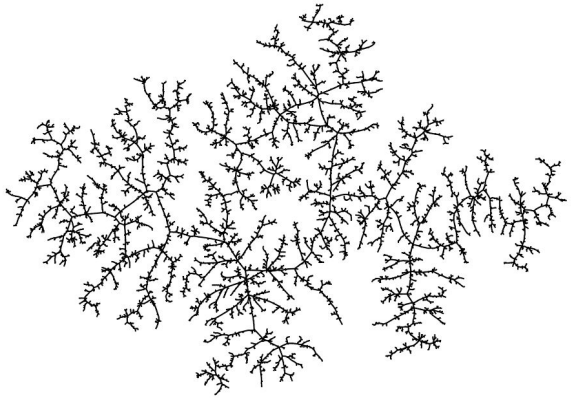
Draw the result of the following operations.

Assume we merge into the larger set.

https://visualgo.net/en/ufds

Pointer Based Disjoint Set

- Runtime complexity?
  - `find(u): logn where n = |S containing u|`
    - Path to root is equivalent to the number of times u has changed sets
    - Each time the name of the set changes, the size of the set *at most doubles*
      - This is because we merge into the larger set
    - Since the size of the set is bounded by n, the path to the root can be at most **logn**
    - O(logn)
  - `union(u,v):`
    - Union is bounded by find
    - O(logn)


Minimum Spanning Trees

and a greedy algorithm to find them

Minimum Spanning Tree

Given a connected undirected graph G = (V, E), a *spanning tree* is an acyclic subset of edges $T \subseteq E$ that connects all $v \in V$.

A minimum spanning tree is a spanning tree such that the cost $\sum_{(u, v) \in T} w(u, v)$ is minimized.

Minimum Spanning Tree: a real world example

Suppose we have a set of locations: $V = \{v_1, v_2, ..., v_n\}$

We want to build a communication network on top of them

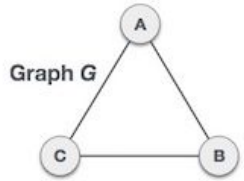
- The network should be connected - there should be a path between every pair of nodes
- We want to build this network as cheaply as possible
- For certain pairs (v_i, v_j), we may build a direct link between them for a certain cost $w(v_i, v_j) > 0$
- We can represent the set of possible links that may be built using a graph G = (V,E) with a positive cost associated with each edge

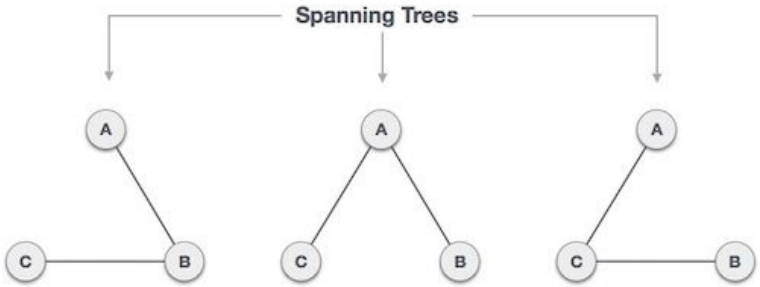
Problem: find a subset of the edges $T \subseteq E$ such that the graph (V, T) is connected,

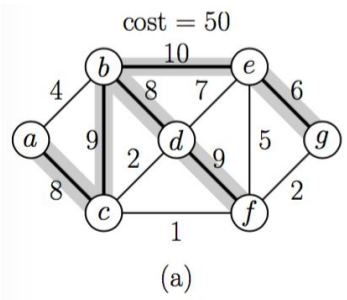
and the cost $\sum_{(u, v) \in T} w(u, v)$ is minimized.

Spanning Trees

Unless G is a very simple graph, it has exponentially many spanning trees.



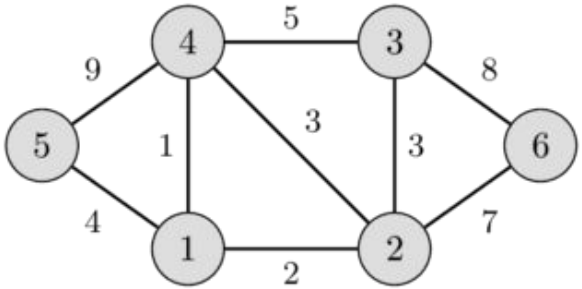


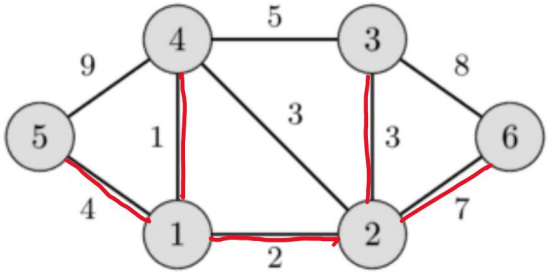


Minimum Spanning Tree

Greedy approaches:

1. Kruskal's Algorithm: MST = {}. Iteratively insert edges from E in order of increasing cost. If an edge will cause a cycle, do not add it.

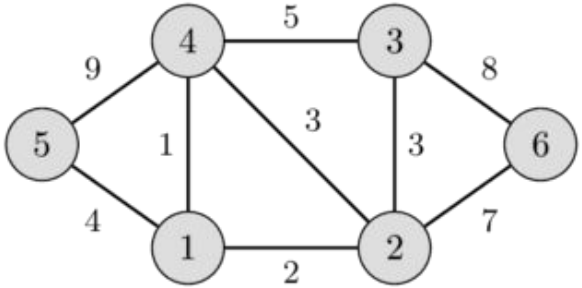




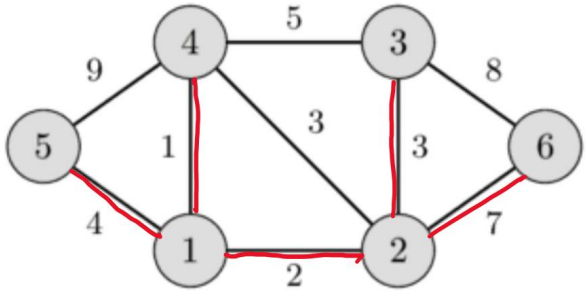
Minimum Spanning Tree

Greedy approaches:

2. Prim's Algorithm: start with a root node s and try to greedily grow the tree outward. At each step, add the node that can be attached as cheaply as possible.

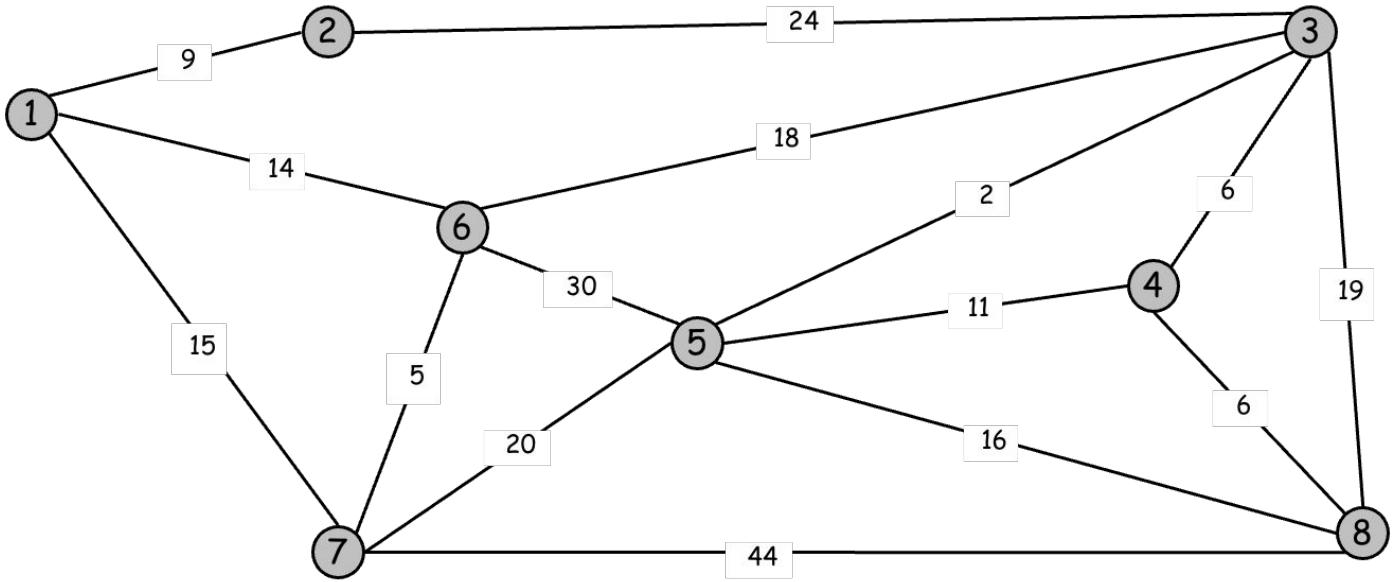


Maintains a set of S ⊆ V on which a spanning tree has been constructed so far. At each iteration, we grow S by one node, adding the node with the minimum "attachment cost"



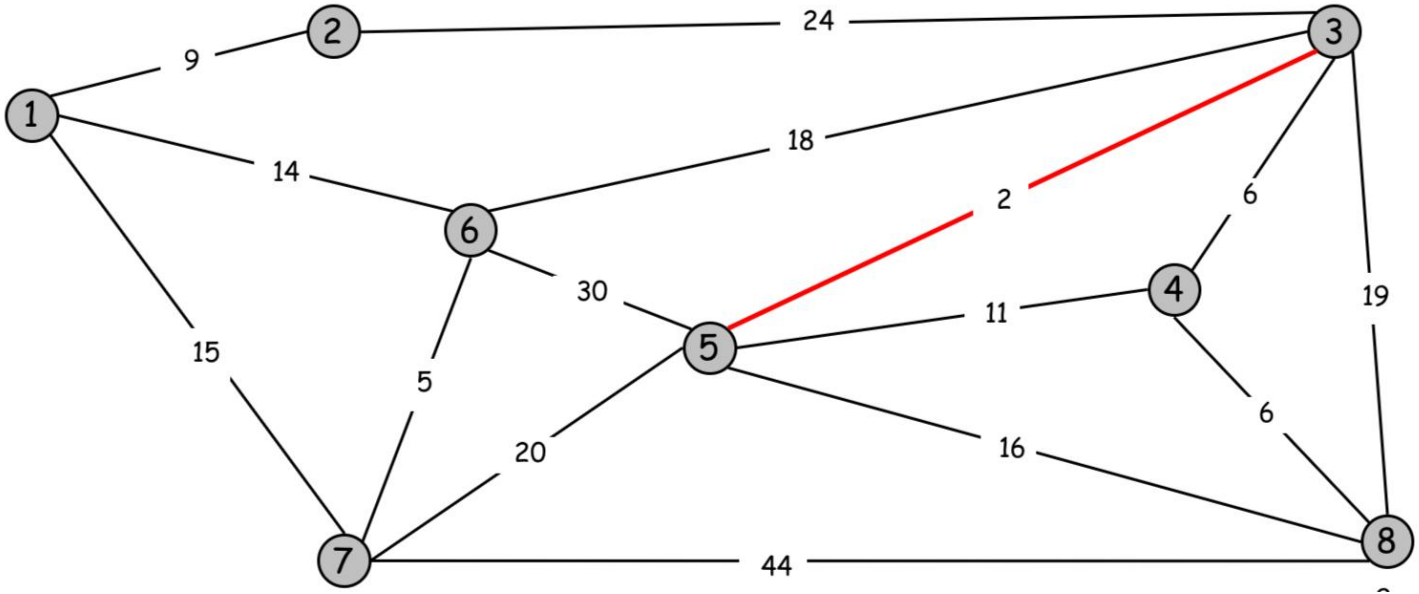
Kruskal’s Algorithm

T = {}



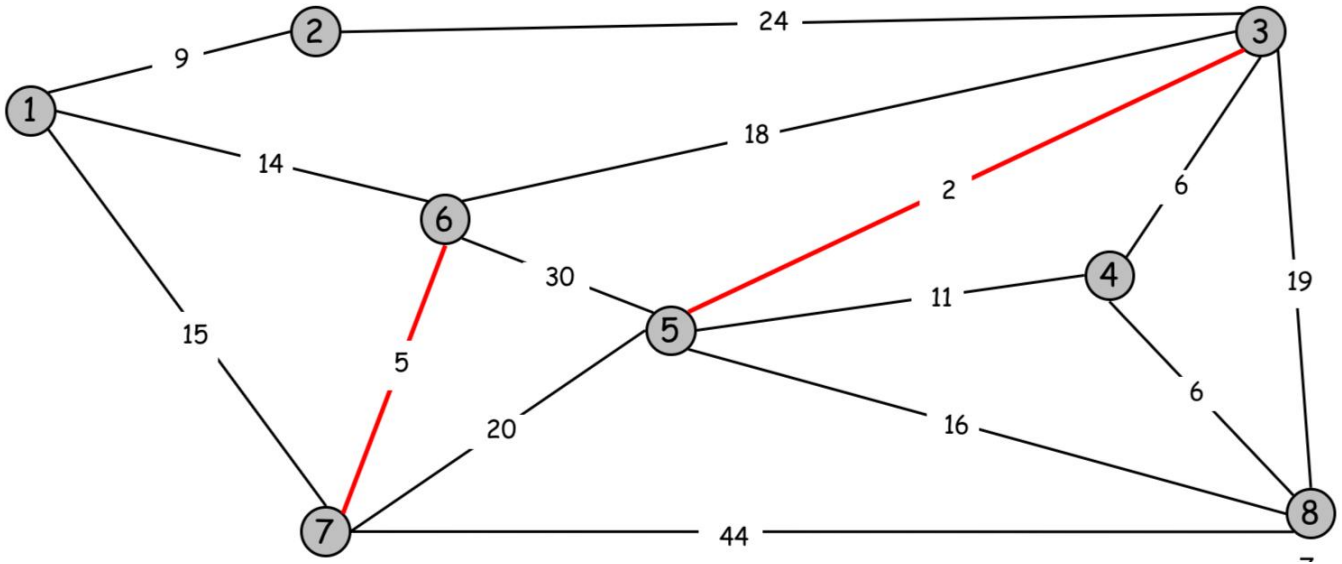
Kruskal’s Algorithm

T = {(5, 3)}



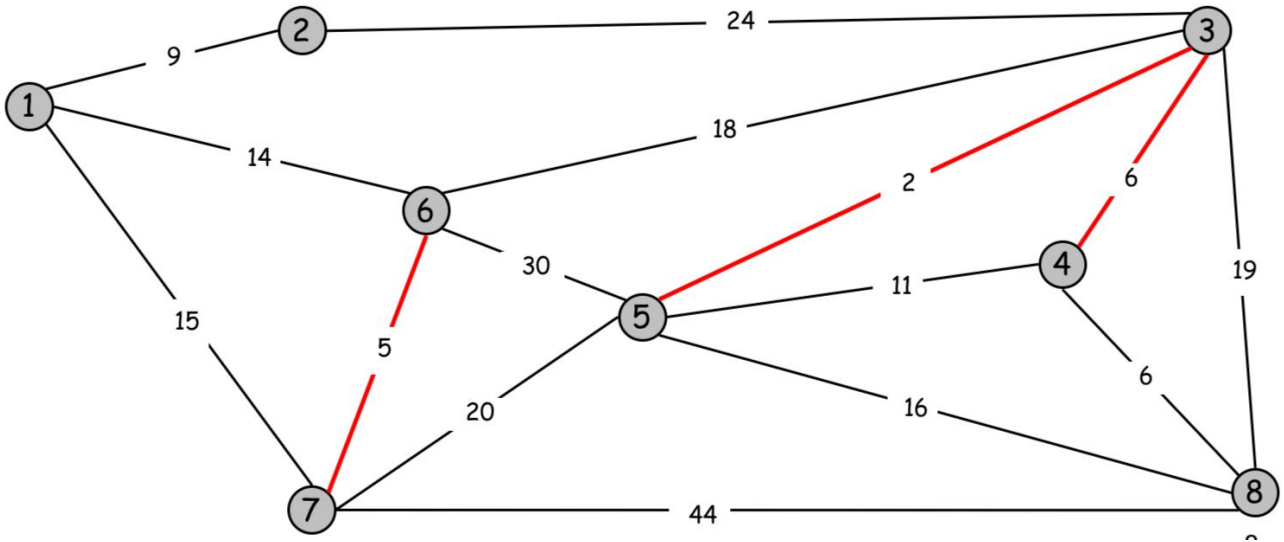
Kruskal’s Algorithm

T = {(5, 3), (6,7)}



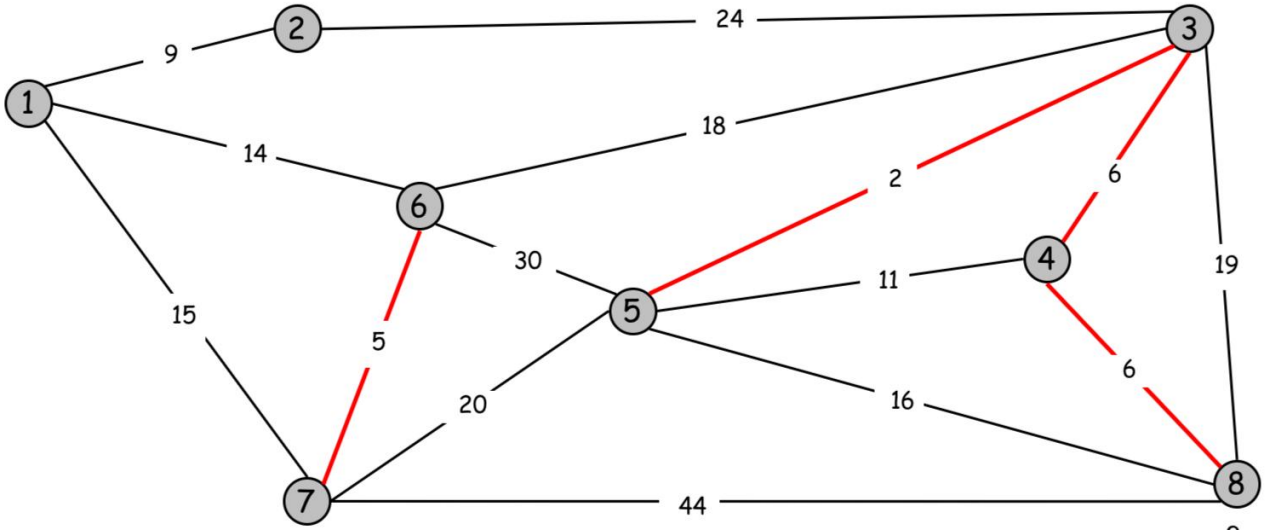
Kruskal’s Algorithm

T = {(5, 3), (6,7), (3, 4)}



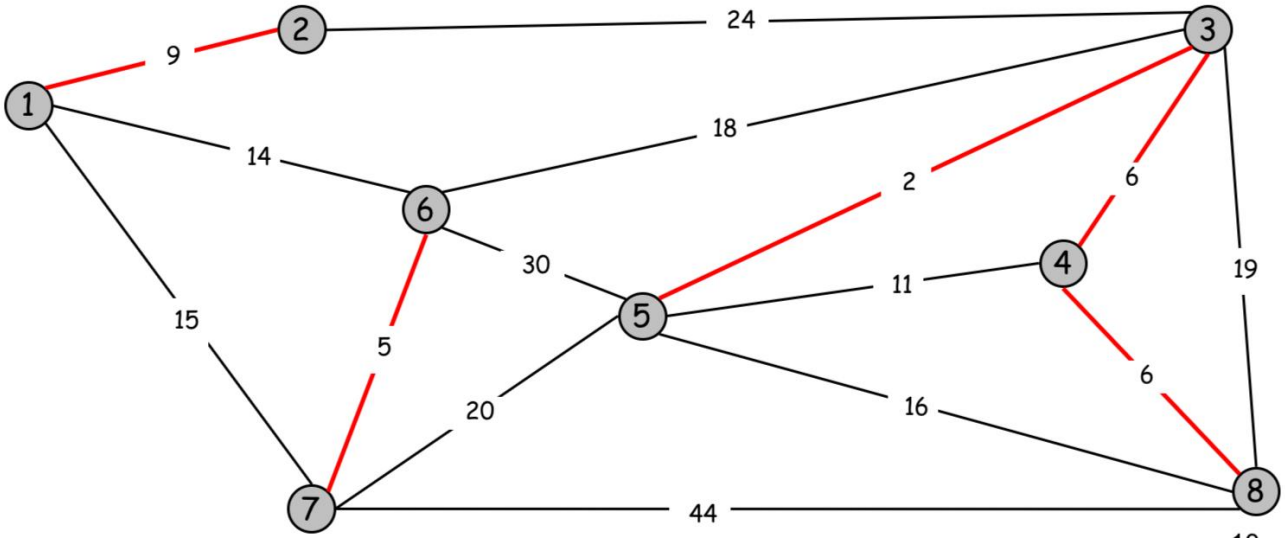
Kruskal’s Algorithm

T = {(5, 3), (6,7), (3,4), (4,8)}



Kruskal’s Algorithm

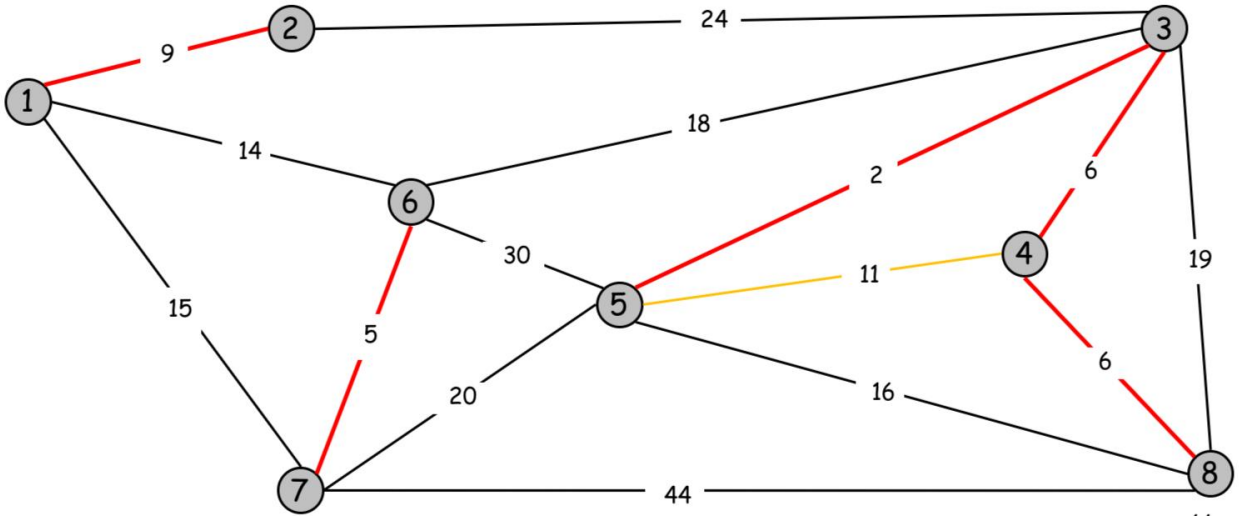
T = {(5, 3), (6,7), (3,4), (4,8), (1,2)}



Kruskal's Algorithm

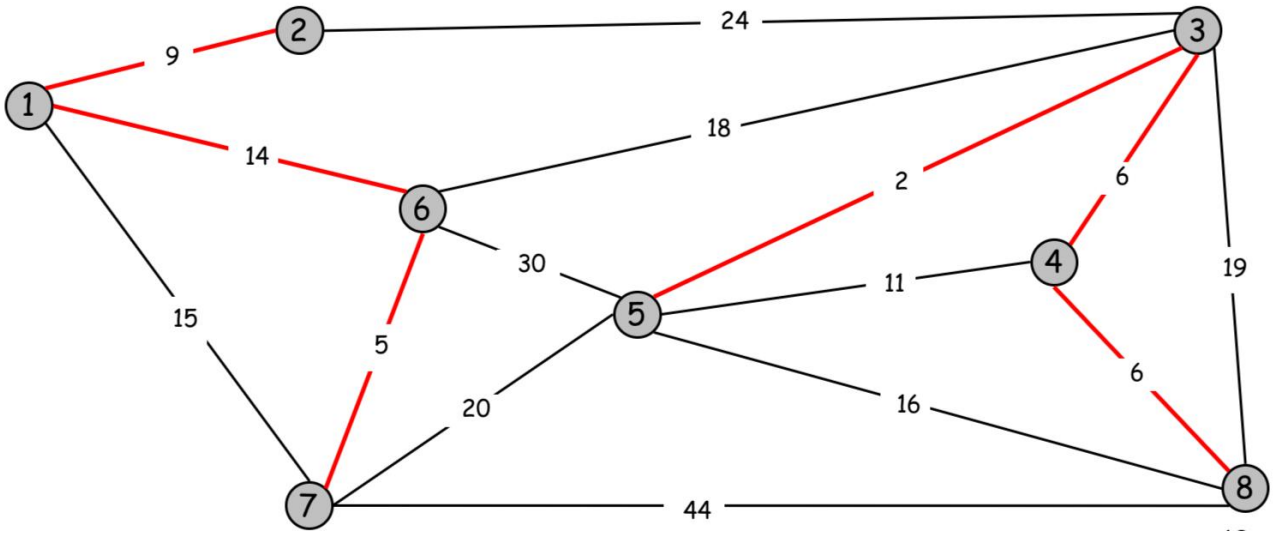
Don't add edges which will create a cycle!

T = {(5, 3), (6,7), (3,4), (4,8), (1,2)}



Kruskal’s Algorithm

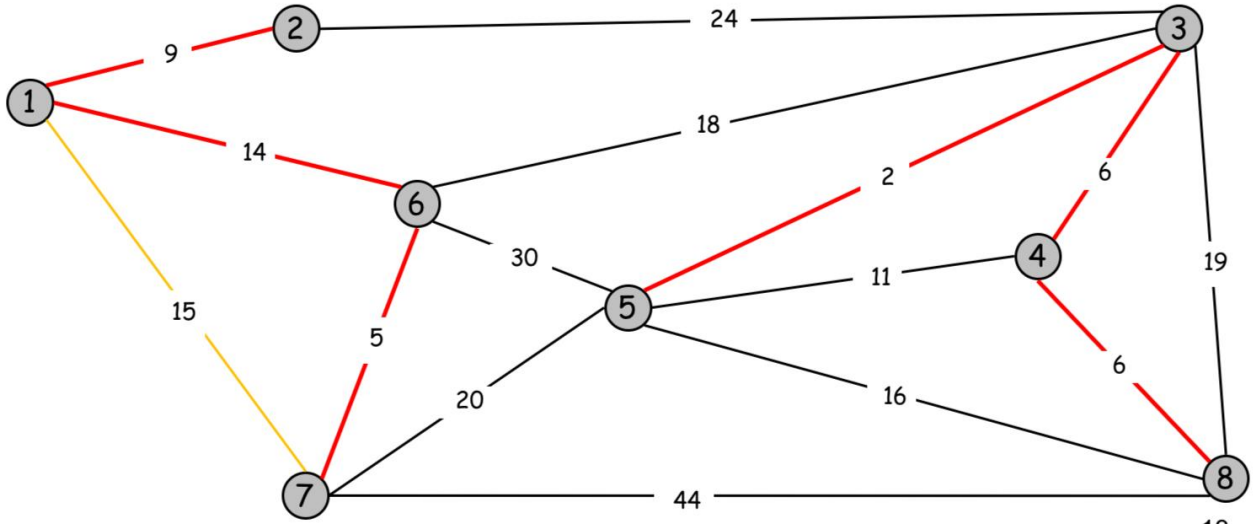
T = {(5, 3), (6,7), (3,4), (4,8), (1,2), (1,6)}



Kruskal's Algorithm

Don't add edges which will create a cycle!

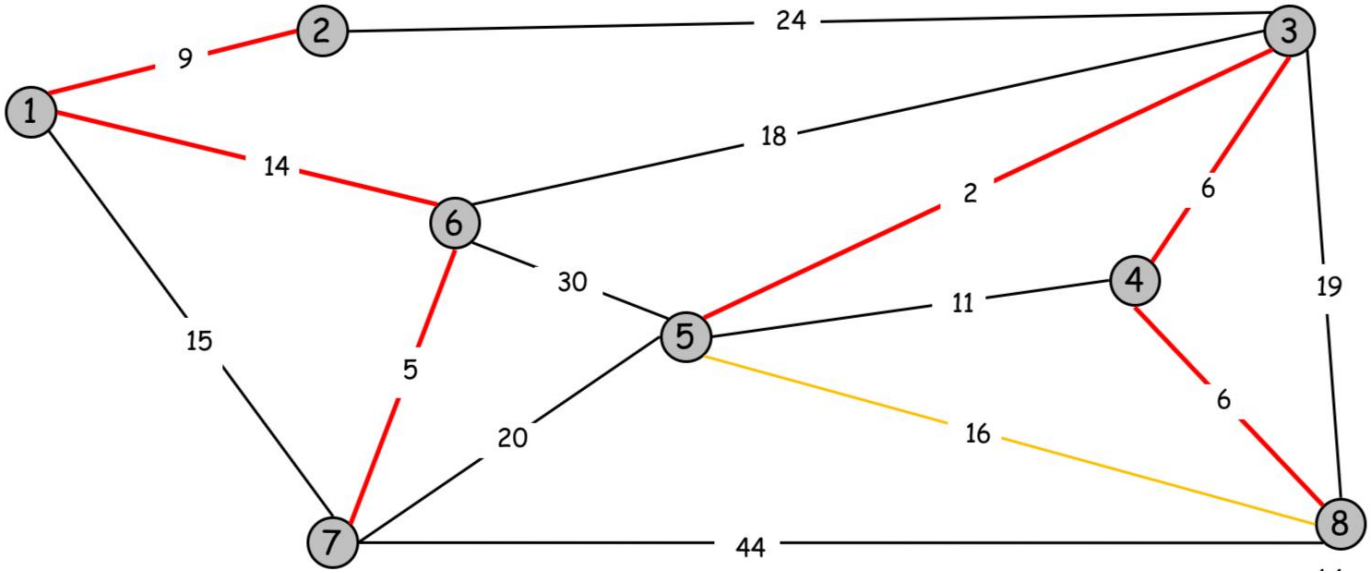
T = {(5, 3), (6,7), (3,4), (4,8), (1,2), (1,6)}



Kruskal's Algorithm

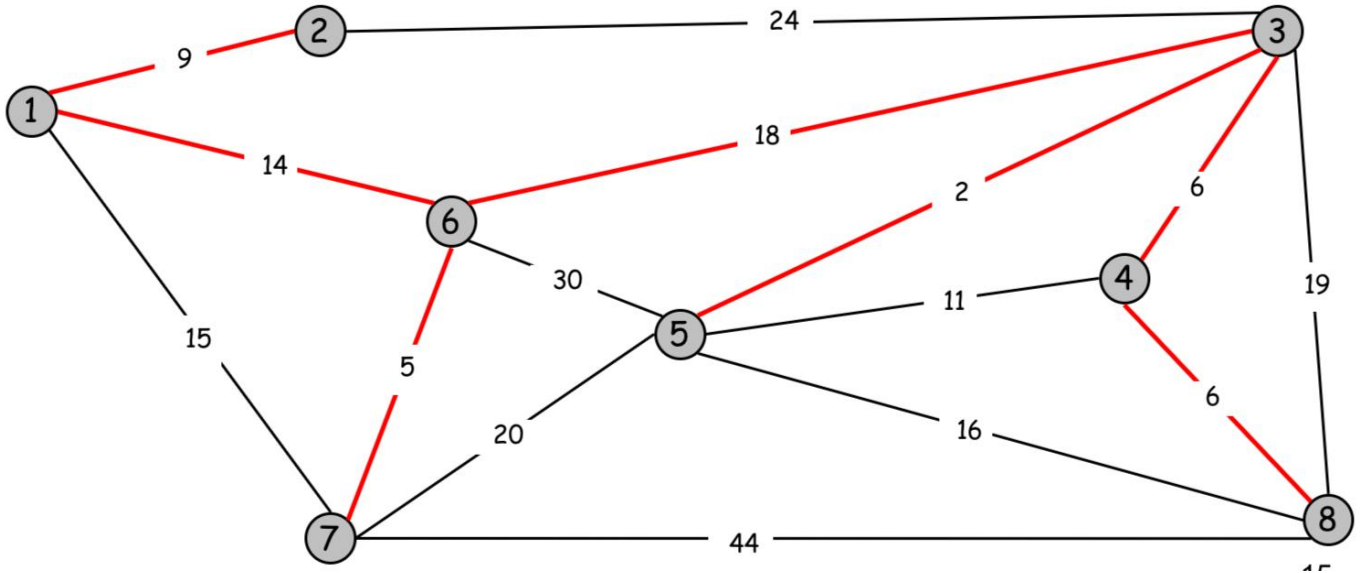
Don't add edges which will create a cycle!

T = {(5, 3), (6,7), (3,4), (4,8), (1,2), (1,6)}

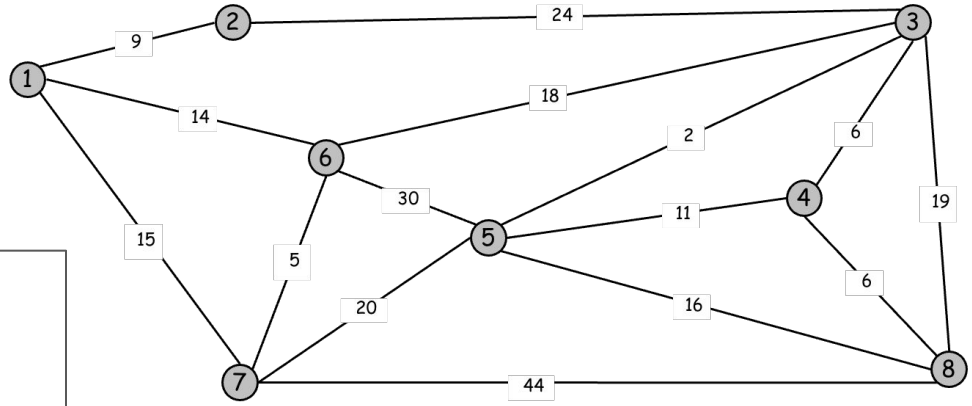


Kruskal’s Algorithm

T = {(5, 3), (6,7), (3,4), (4,8), (1,2), (1,6), (6, 3)}



Kruskal's Algorithm




```
kMST(G=(V, E)){
  T = {}
  place each vertex in a set by itself
  sort E in increasing order by weight
  for each ((u, v) in E in sorted order){
    // u and v in different sets
    if (find(u) != find(v)) {
      add (u, v) to T
      union(u, v)
    }
  }
}
```

Runtime Analysis

```
kMST(G=(V, E)){
  T = {}
  place each vertex in a set by itself
  sort E in increasing order by weight
  for each ((u, v) in E in sorted order){
    // u and v in different sets
    if (find(u) != find(v)) {
      add (u, v) to T
      union(u, v)
    }
  }
}
```

O(n)

O(mlogm)

m iterations

O(logn)

O(logn)

`Total: O(mlogm) + O(mlogn)`

Which term is larger?

For each node, what is max number of edges?

- n-1
- For n nodes, n*(n-1)

$m < n^2$

$\log m < \log n^2 = 2\log n$

O(m logn)

Proof of Correctness

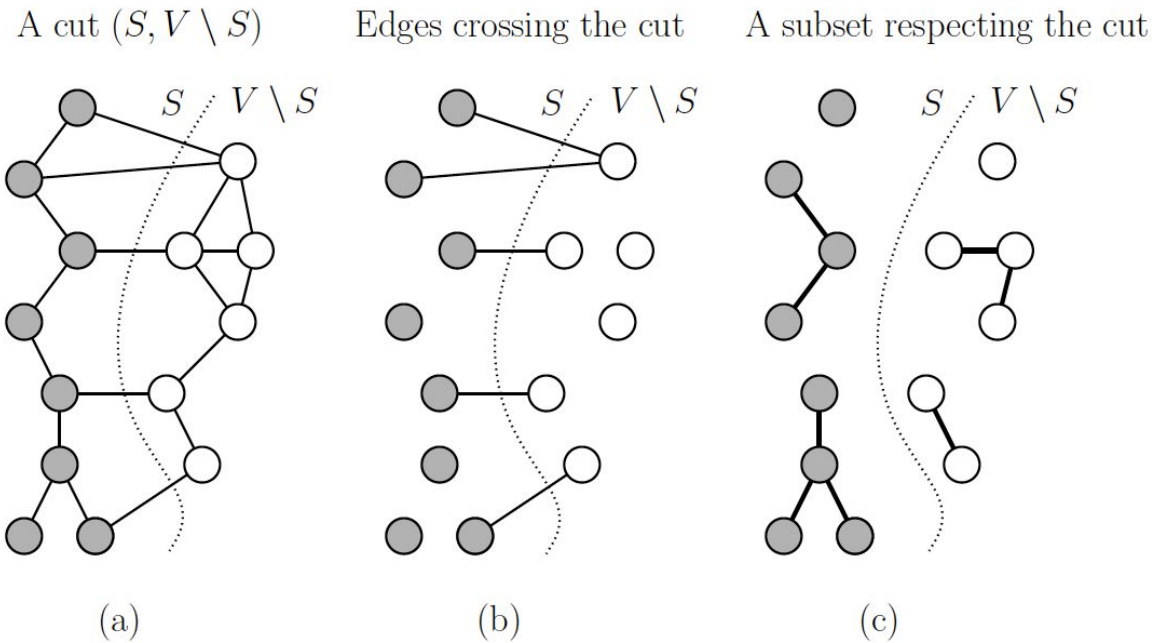
When is it safe to include an edge in an MST?

When is it safe to eliminate an edge on the grounds that it couldn’t possibly be in the MST?

Cut Property: Assume that all edge costs are distinct. Let S be any subset of nodes is neither empty nor equal to all V, and let edge e = (u, v) be the minimum cost edge with one end in S and the other in V - S. Then every MST contains the edge e.

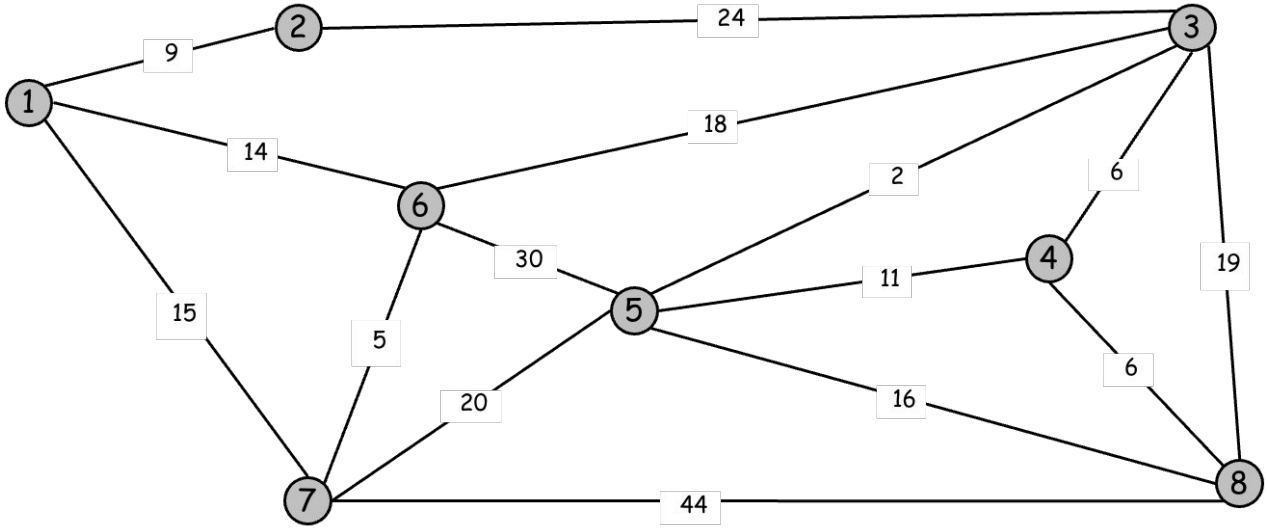
Proof of Correctness

Cut Property: S ⊆ V, and let edge e = (u, v) be the minimum cost edge with one end in S and the other in V - S. Then every MST contains the edge e.



Check for Understanding: cuts

1. For S = {1, 2, 7}, what edges cross the cut?
   a. (2, 3), (1, 6), (7, 6), (7, 5)
   b. What is the minimum edge that crosses this cut?
      i. (6, 7)
      ii. So this must be in the MST (we will prove this)



Proof of Correctness

Lemma: S ⊆ V, and let edge e = (u, v) be the minimum cost edge with one end in S and the other in V - S. Then every MST contains the edge e.

Proof by contradiction: Let T an MST that does not contain (u, v).

It must have some other edge which crosses the cut: (x, y)

By swapping (x, y) for (u, v), we have the MST T'.

w(T') = W(T) - w(x, y) + w(u, v)

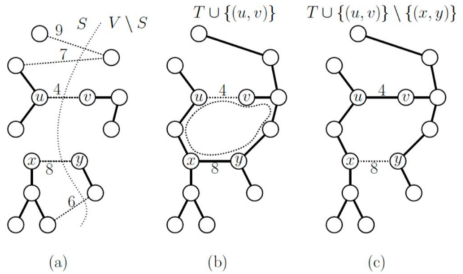


Figure 1: (u,v) is the lightest edge crossing the cut $(S,V\setminus S)$

Since (u, v) is the minimum cost edge crossing the cut, w(u, v) < w(x, y) and thus w(T') < w(T) which contradicts our claim that T is a MST.

Proof of Correctness

Proof: Consider any edge e = (v, w) selected by Kruskal's.

Let S be the set of all nodes to which v has a path to before e is added.

Because it was selected, e does not introduce a cycle: v $\in$ S and w $\notin$ S

Thus, e is the cheapest edge with one end in S and one end in V - S. By our lemma, this must be in our MST.

So, if we can show that the output T of Kruskal's is in fact a spanning tree, then we will have proven its correctness.

If T was not a spanning tree, there must be some set S such that there is no edge from S to V - S, because otherwise, the algorithm would have added it.

This contradicts our assumption that the input graph is connected.

Full Proof Posted on Webpage

Summary

1. Union Find
    a. Data structure with O(logn) find and union operations
2. Minimum Spanning Tree
    a. Subset of edges that connect all nodes with minimum cost
3. Kruskal's
    a. Greedy: insert edges in increasing order that do not cause a cycle
4. Upcoming deadlines:
    a. Lab 3 due Thursday
    b. Project checkpoint 1 (10/5)
    c. HW4 posted - due next monday (10/6)
    d. Midterm (10/8)
5. Next Class: Huffman Coding
    a. our last algorithm before the midterm