CSCI 136
Data Structures & Advanced Programming

Graph Applications:

Minimum Cost Spanning Trees

Video Outline

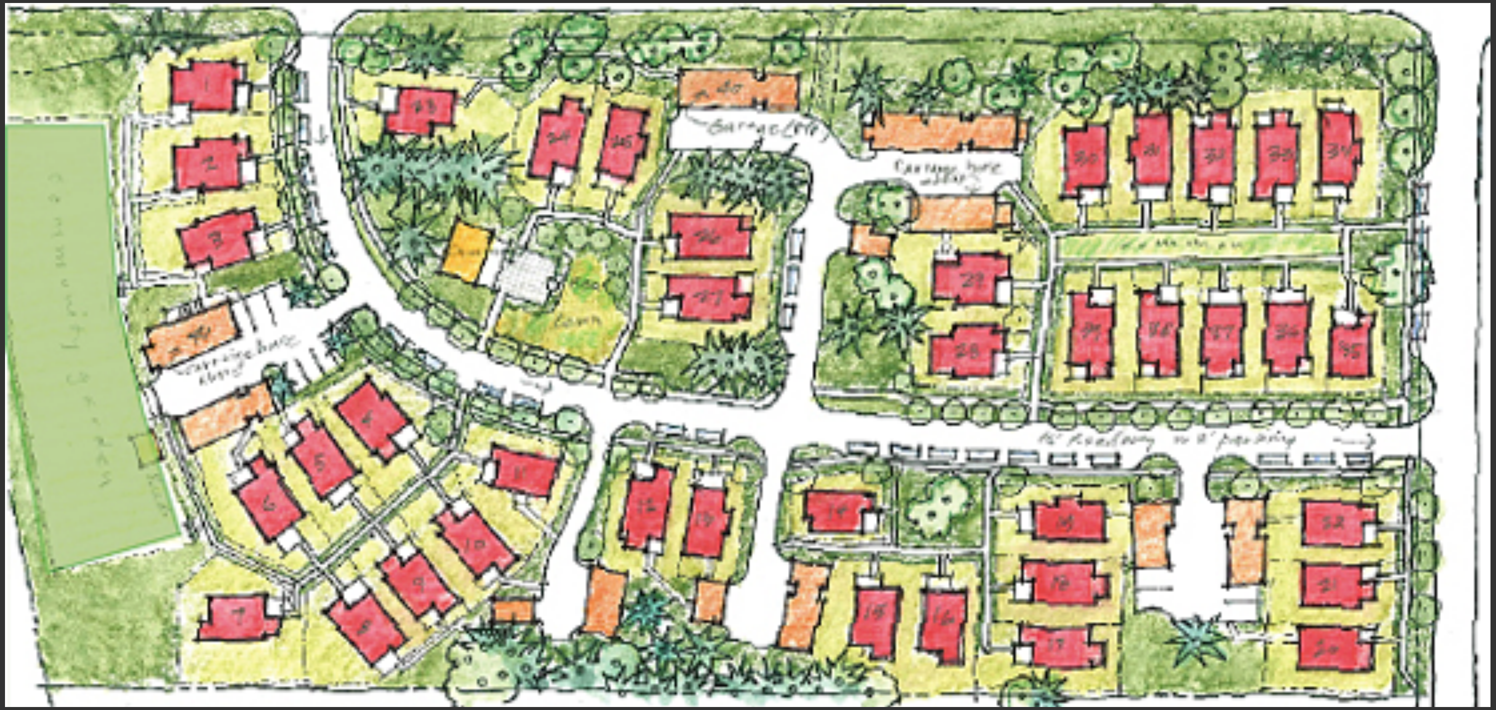
- Spanning subgraphs
- Spanning trees
- Prim’s algorithm to calculate spanning trees with the minimum cost
  - Description
  - Proof
  - Pseudocode
  - Implementation in structure5

Motivation

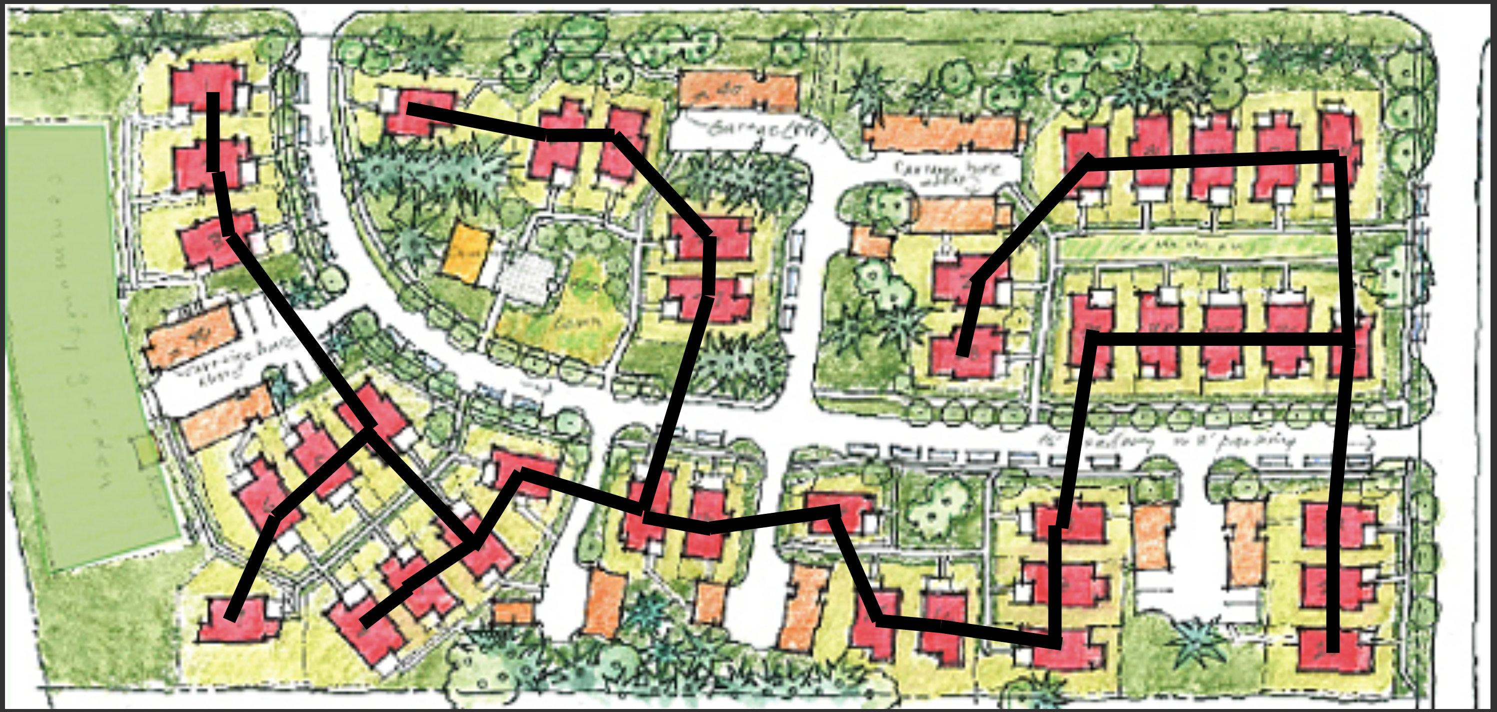
- Let’s say we have a neighborhood of houses
- Want to create an electrical grid
- Goal: each house needs to be connected to a single network
- (In other words, there is a path along the electrical wires between any two houses)

- Also works for creating ethernet networks, etc.

Motivation



Motivation



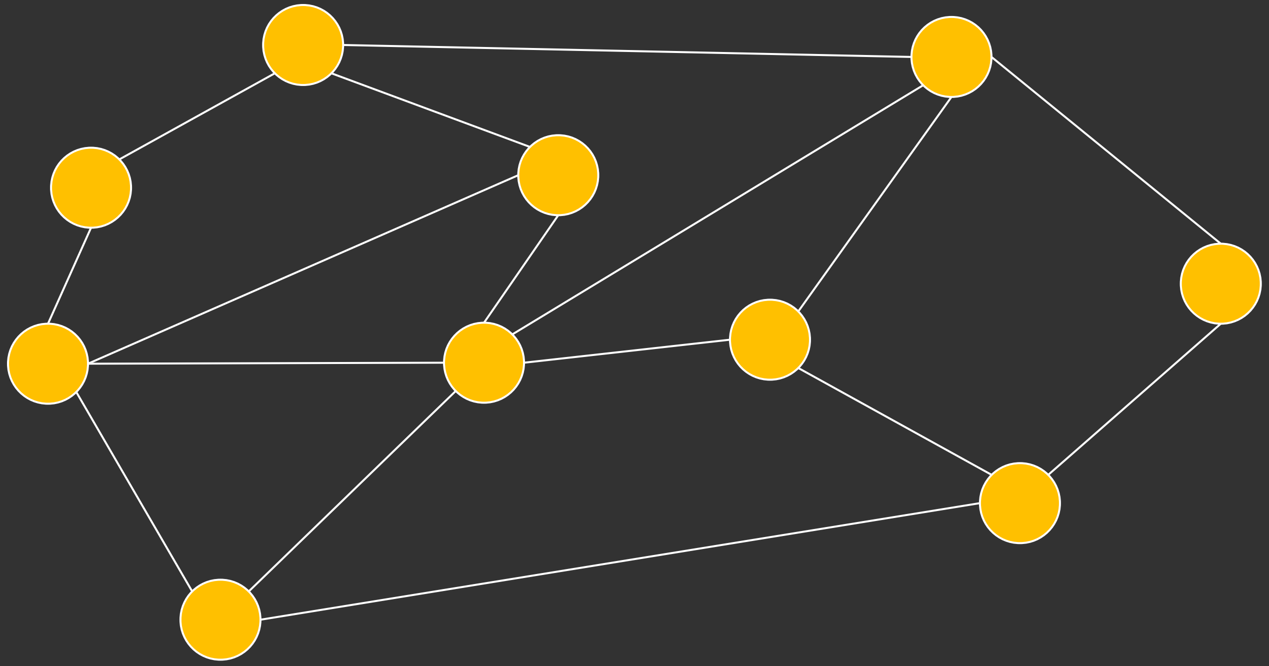
Goal

- Graph problem!
- Select edges to connect all vertices using a single tree
- Can only select edges *in the original graph*
- Want to select the minimum cost:
  - Minimum *total* of edge weights in the tree

Spanning Trees

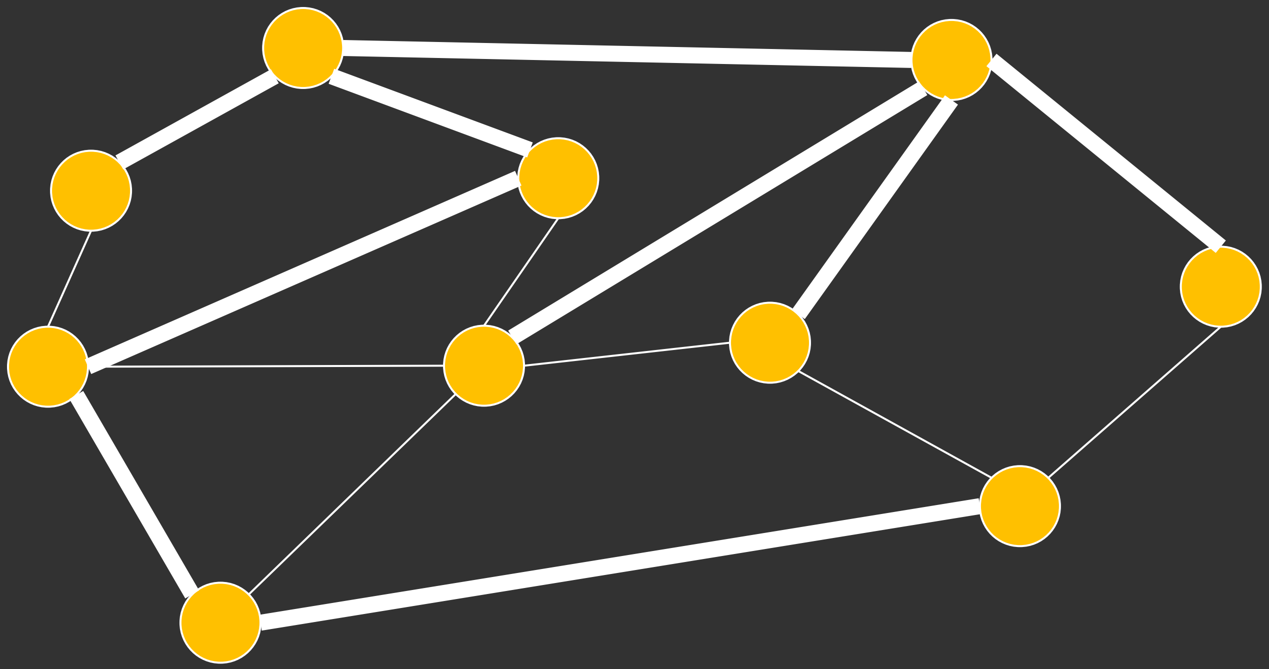
Collection of vertices and edges from the original graph

- A spanning tree is a subgraph that covers all the vertices using the *minimum number of edges*



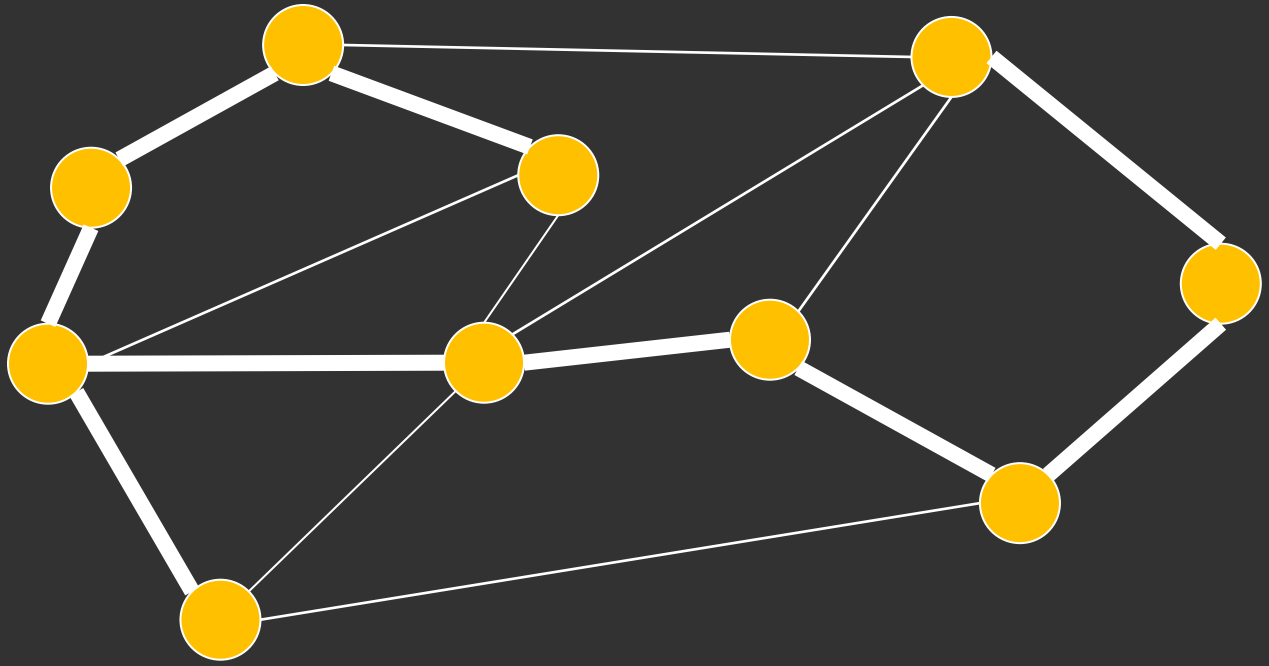
Spanning Trees

- A spanning tree is a subgraph that covers all the vertices using the *minimum number of edges*

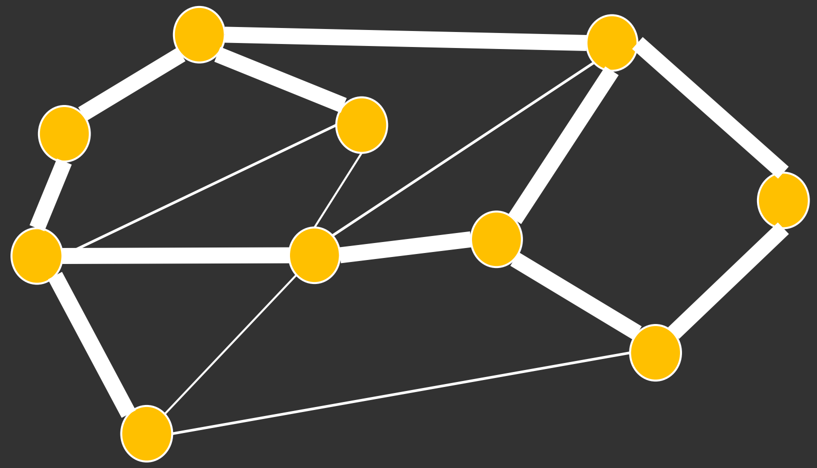


Spanning Trees

- A spanning tree is a subgraph that covers all the vertices using the *minimum number of edges*



Why Trees?

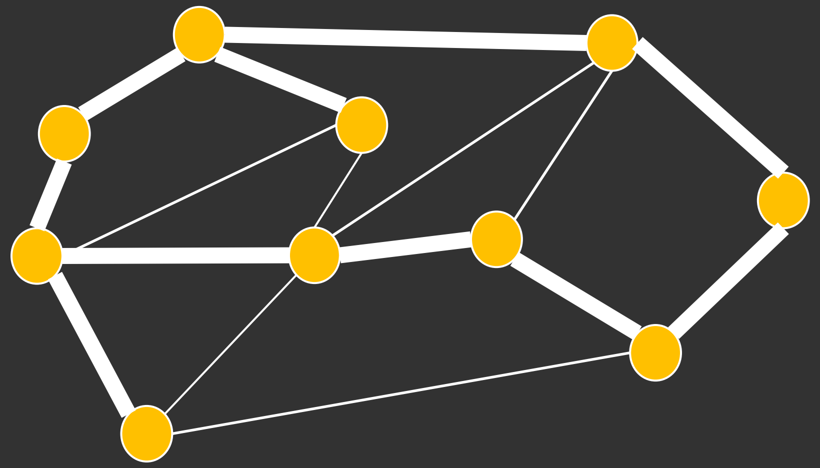


Theorem: The minimum collection of edges that connects all vertices is a tree

Proof idea:

- If it’s is not a tree, then it contains some cycle C
- If we remove an edge from C, the resulting collection of edges still connects all vertices in the tree
- Repeat this process of removing edges until no more cycles remain
- Now we are left with a tree

Why Trees?

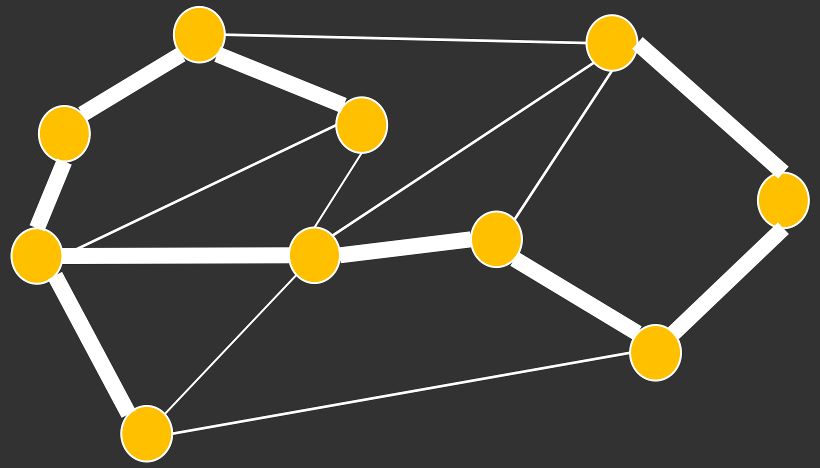


Theorem: The minimum collection of edges that connects all vertices is a tree

Proof idea:

- If it's is not a tree, then it contains some cycle C
- If we remove an edge from C, the resulting collection of edges still connects all vertices in the tree
- Repeat this process of removing edges until no more cycles remain
- Now we are left with a tree

Why Trees?



Theorem: The minimum collection of edges that connects all vertices is a tree

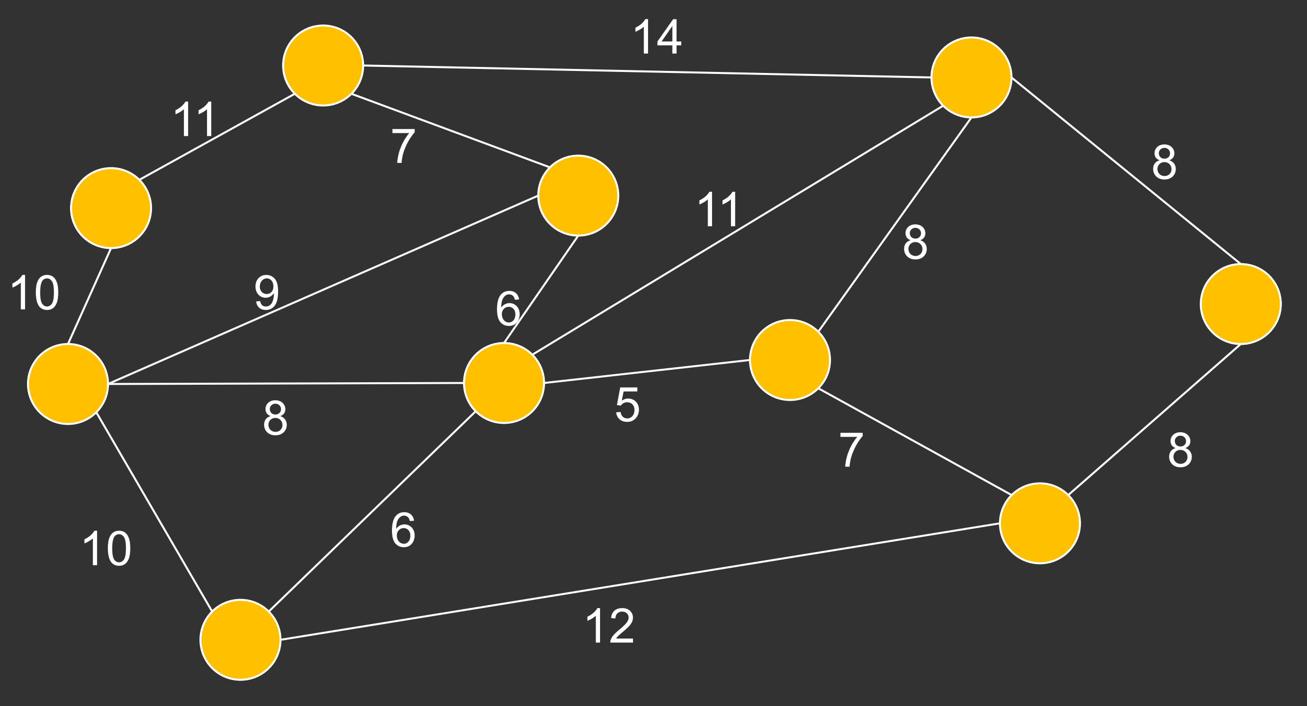
Proof idea:

- If it's is not a tree, then it contains some cycle C
- If we remove an edge from C, the resulting collection of edges still connects all vertices in the tree
- Repeat this process of removing edges until no more cycles remain
- Now we are left with a tree

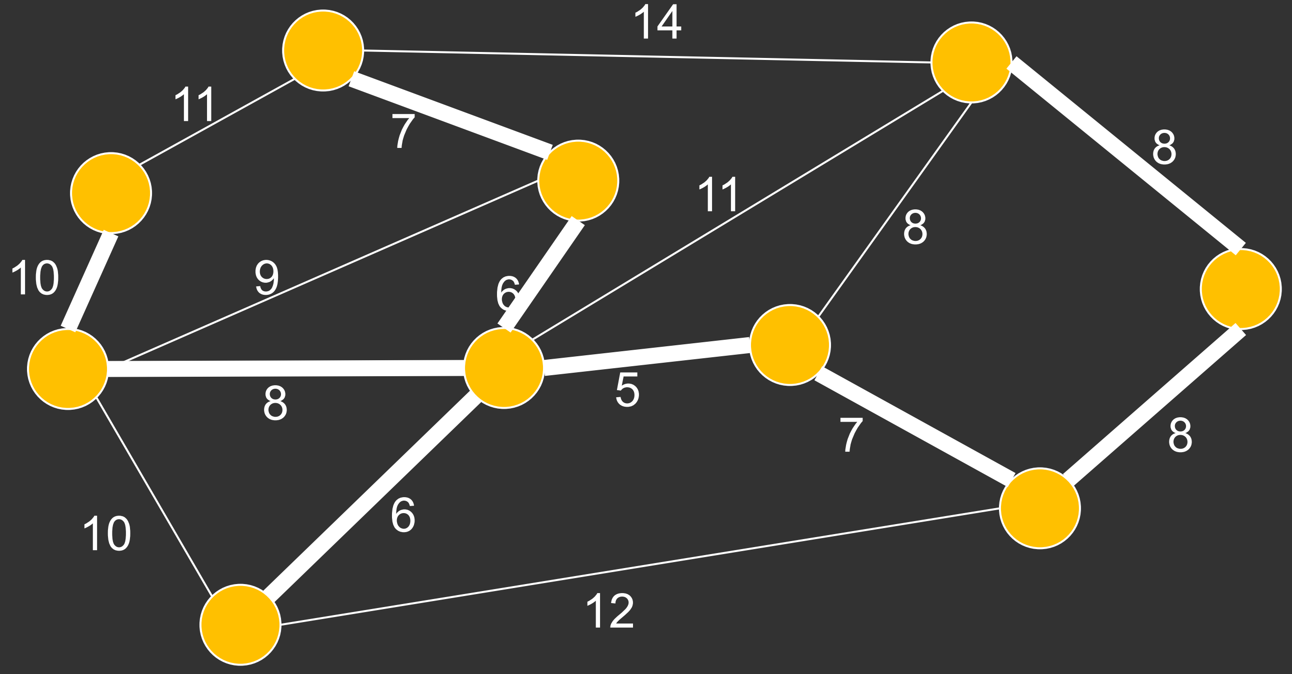
Minimum-Cost Spanning Trees

- Suppose we're given a graph that is:
  - connected, and
  - has weighted edges (integer, float, double, etc.)
- A minimum cost spanning tree is a spanning tree where the *sum of all the edge weights* is the smallest possible

Minimum-Cost Spanning Trees



Minimum-Cost Spanning Trees

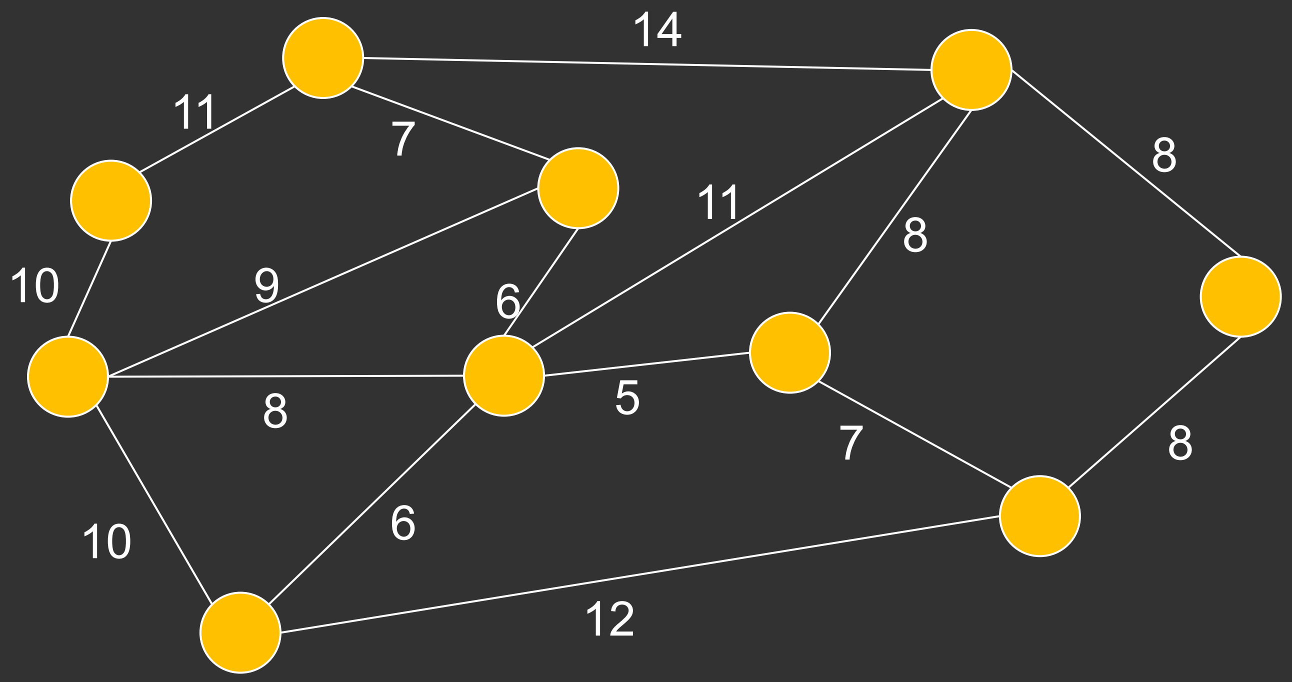


How can we find an MCST?

- Let’s start with some node in our tree
- We need to hook it up to the network…how?
- Let’s use the cheapest edge
- Now we have a two-node network. Need to expand this network again…how?
- Take the cheapest edge connecting either of the two nodes currently in the network to an outside node
- Repeat until n nodes in the tree!

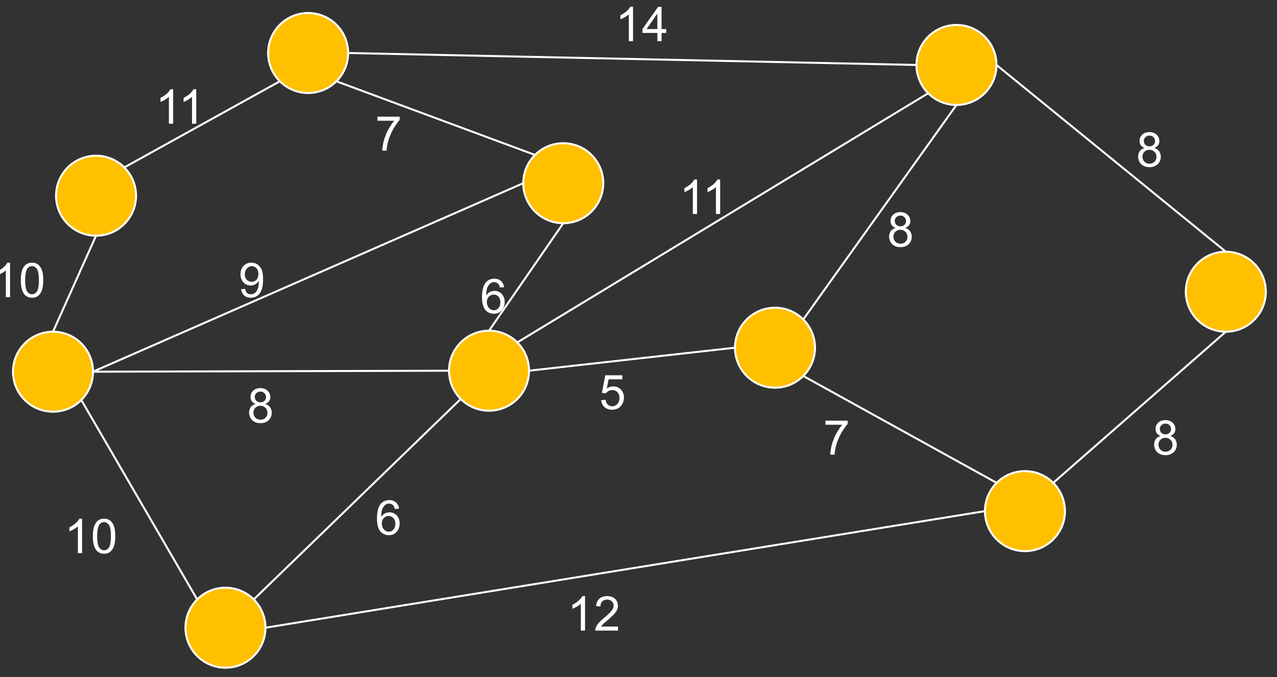
Prim’s Algorithm for MCSTs

- Let’s walk through an example to solidify the algorithm. In this example, not all edge weights are unique.



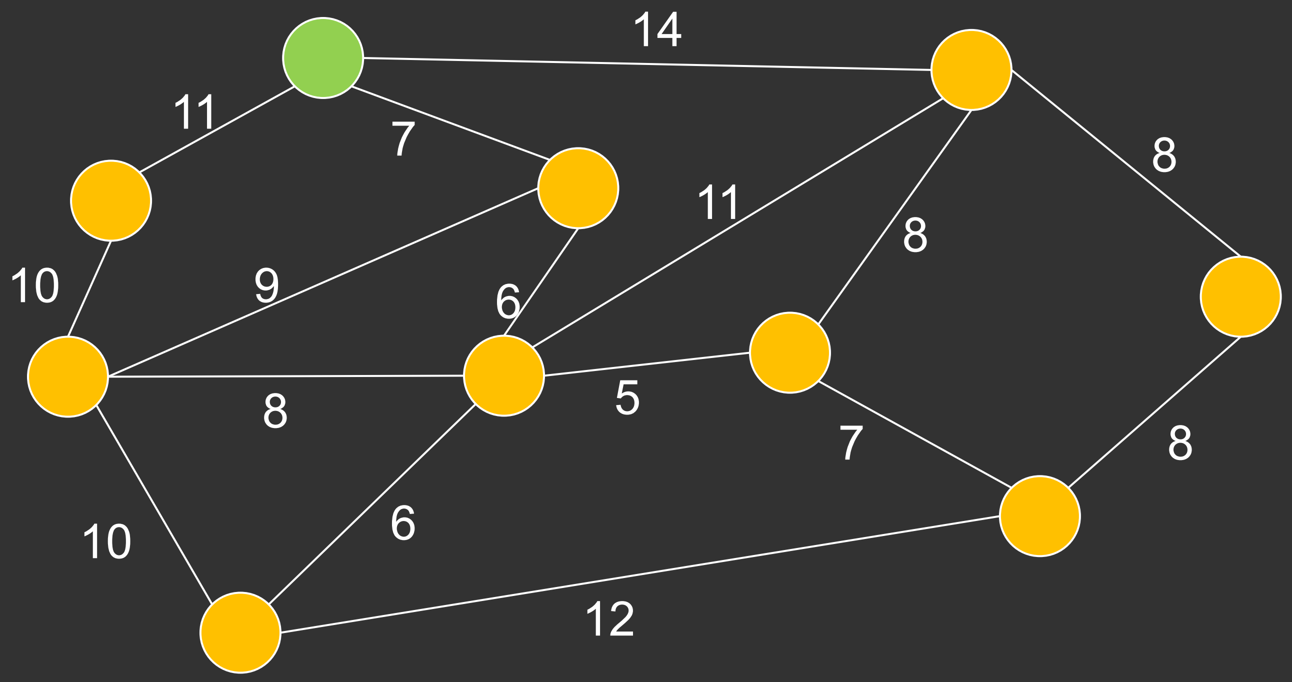
Prim’s Algorithm for MCSTs

- Start by picking some vertex



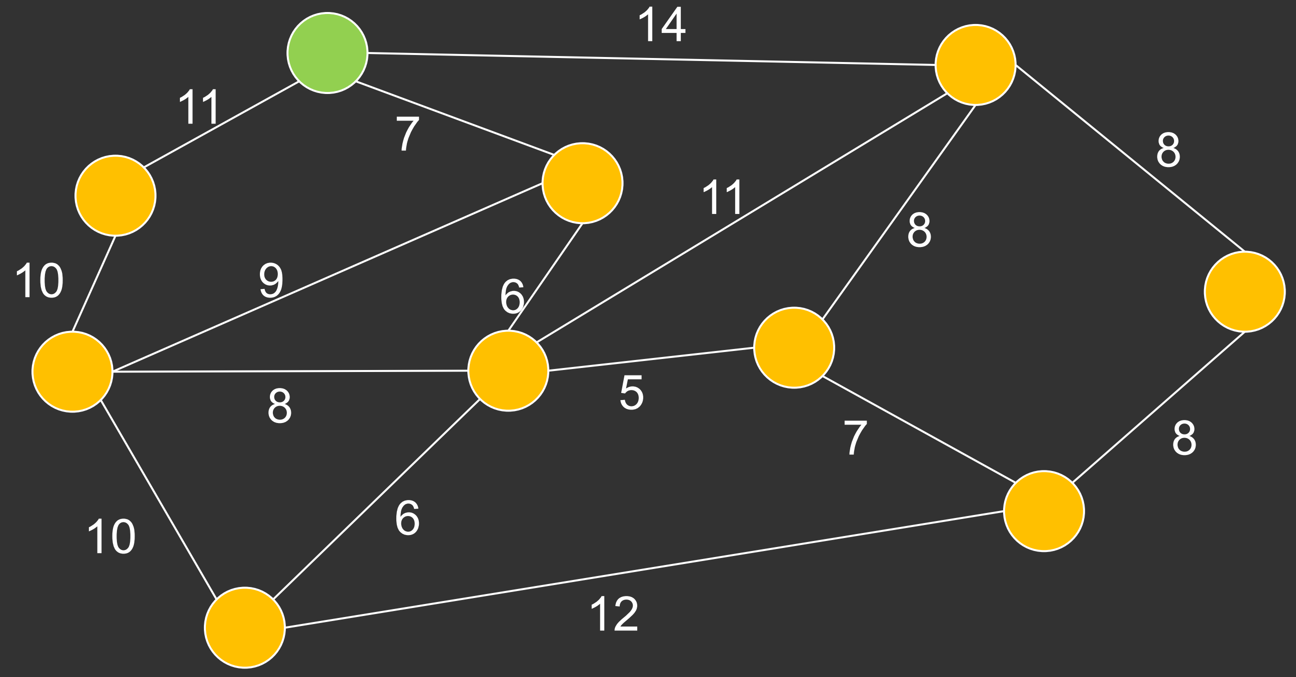
Prim’s Algorithm for MCSTs

- Start by picking some vertex



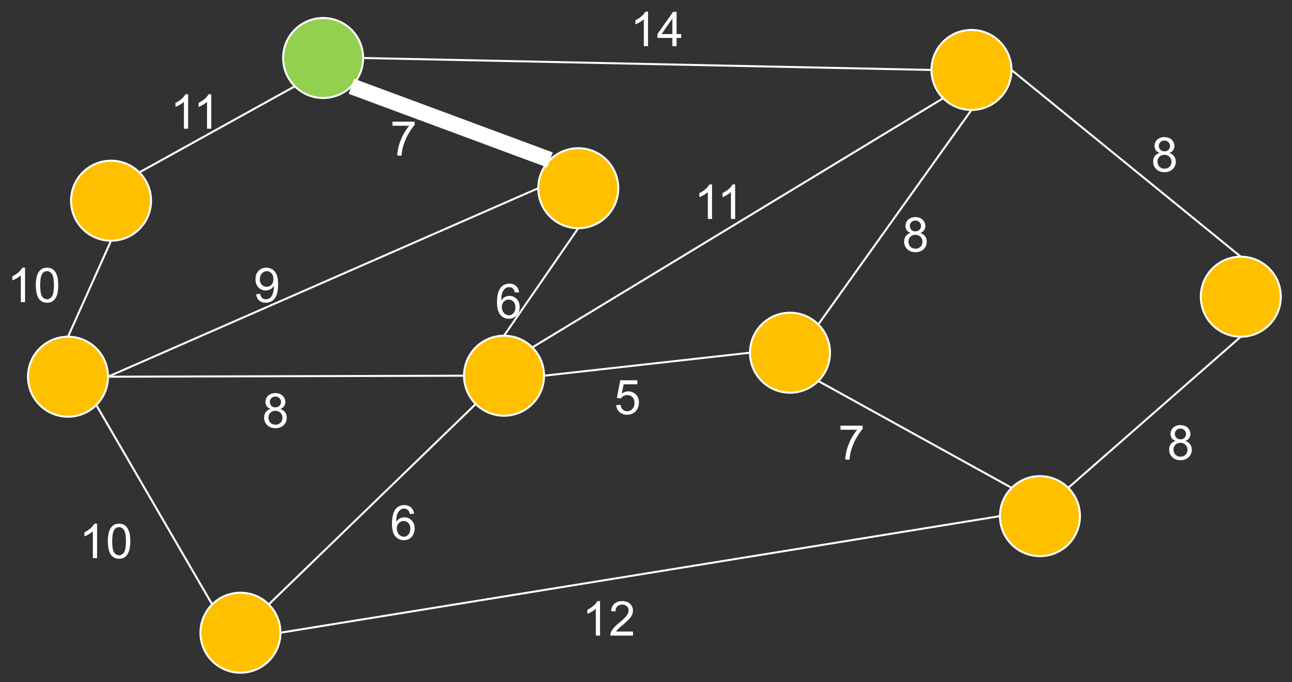
Prim’s Algorithm for MCSTs

- We’ll note our current tree in green, and other vertices in orange. Select an edge with the cheapest cost that connects a green vertex to an orange vertex.



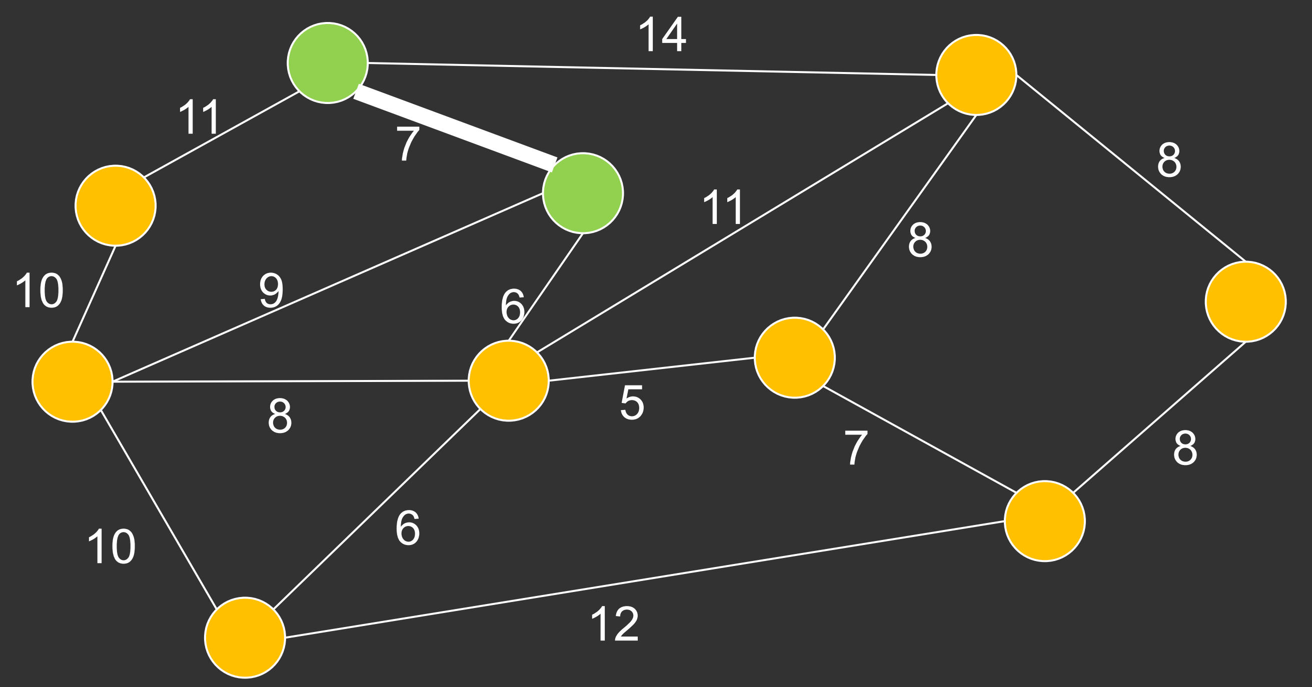
Prim’s Algorithm for MCSTs

- We’ll note our current tree in green, and other vertices in orange. Select an edge with the cheapest cost that connects a green vertex to an orange vertex.



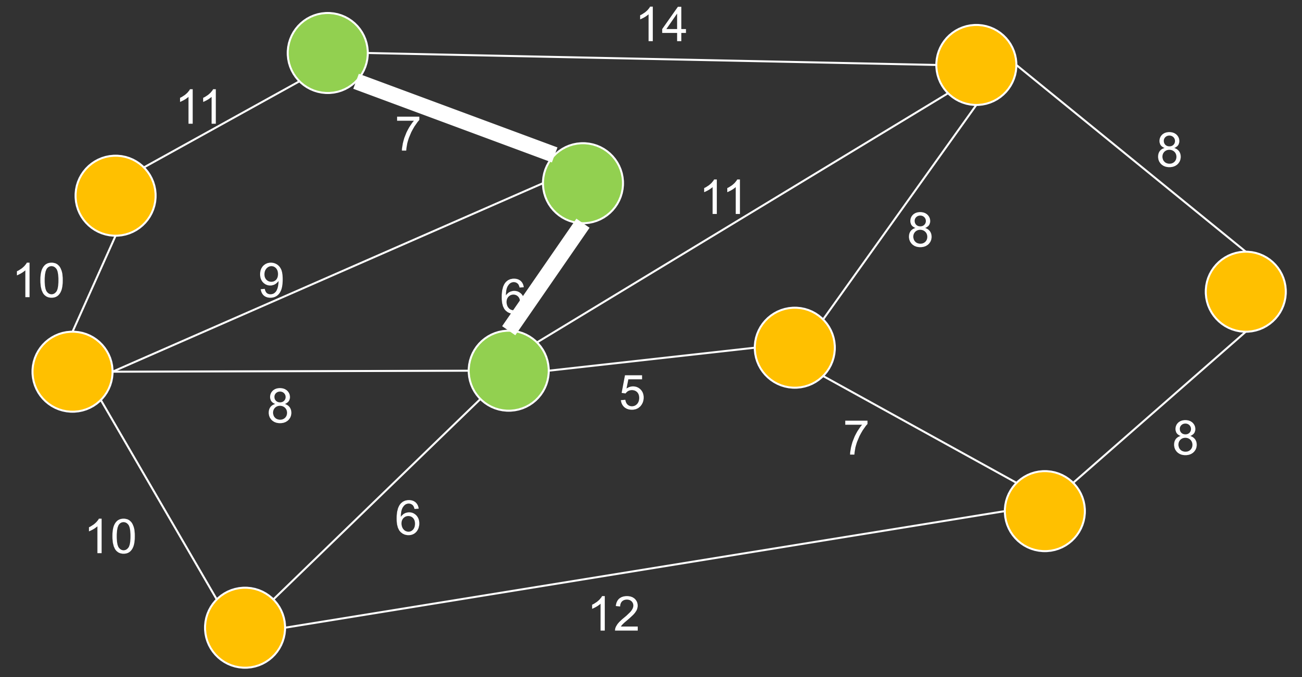
Prim’s Algorithm for MCSTs

- Continue this process of adding an edge with the cheapest cost connecting a green vertex to an orange vertex until all vertices are green.



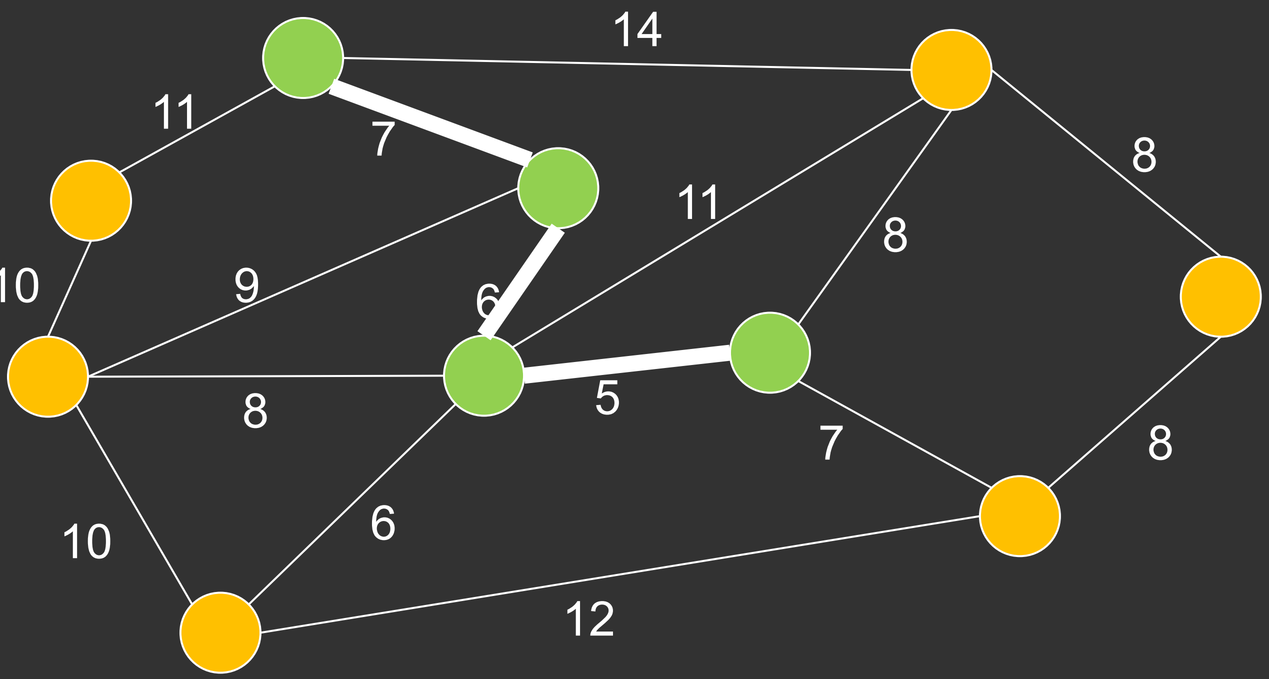
Prim’s Algorithm for MCSTs

- Continue this process of adding an edge with the cheapest cost connecting a green vertex to an orange vertex until all vertices are green.



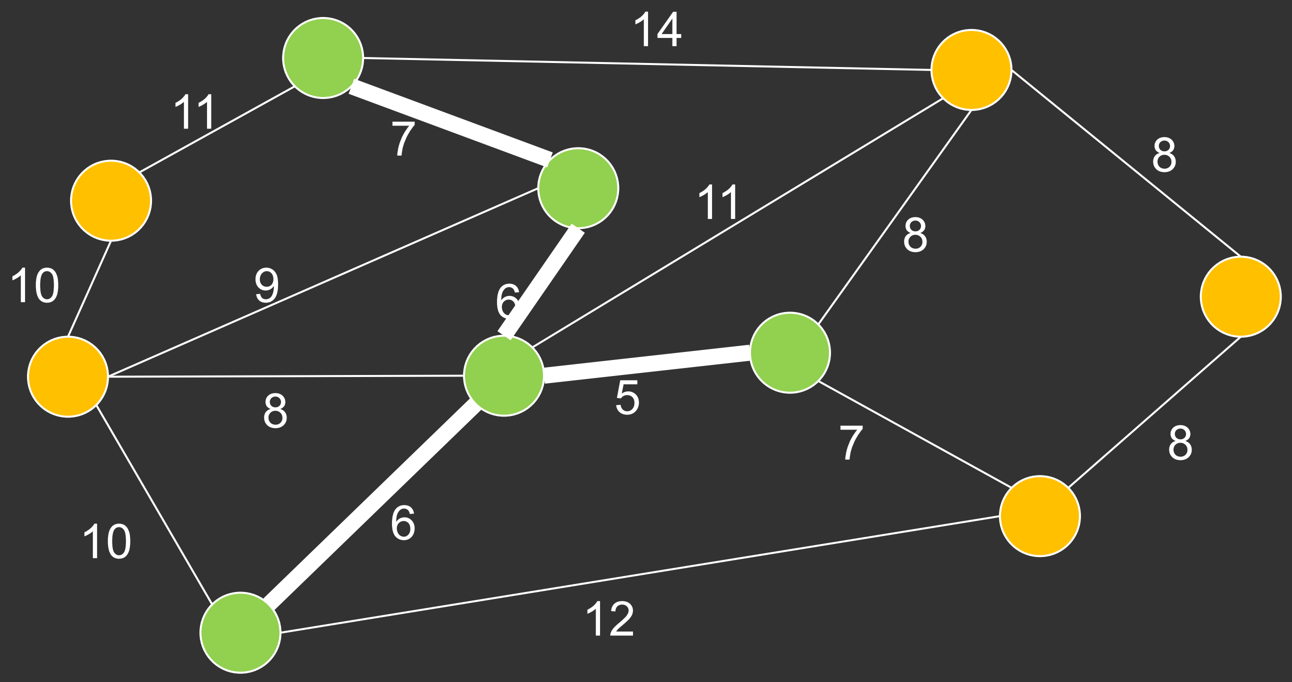
Prim’s Algorithm for MCSTs

- Continue this process of adding an edge with the cheapest cost connecting a green vertex to an orange vertex until all vertices are green.



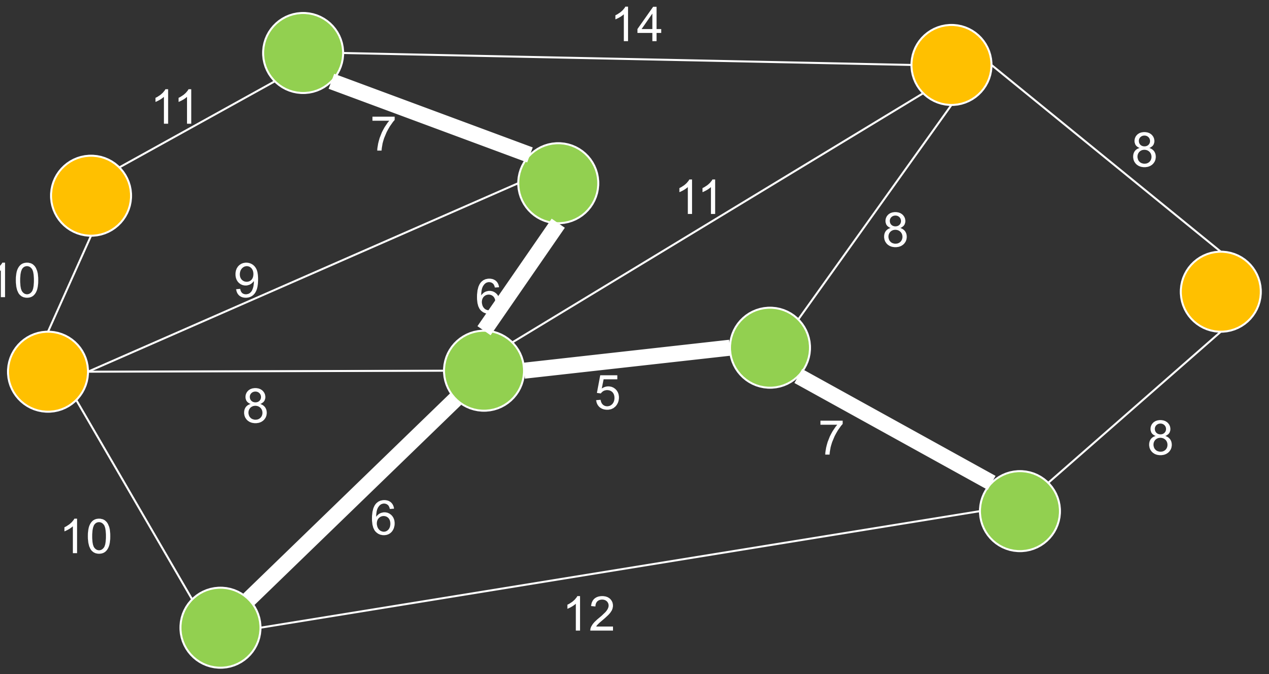
Prim's Algorithm for MCSTs

- Continue this process of adding an edge with the cheapest cost connecting a green vertex to an orange vertex until all vertices are green.



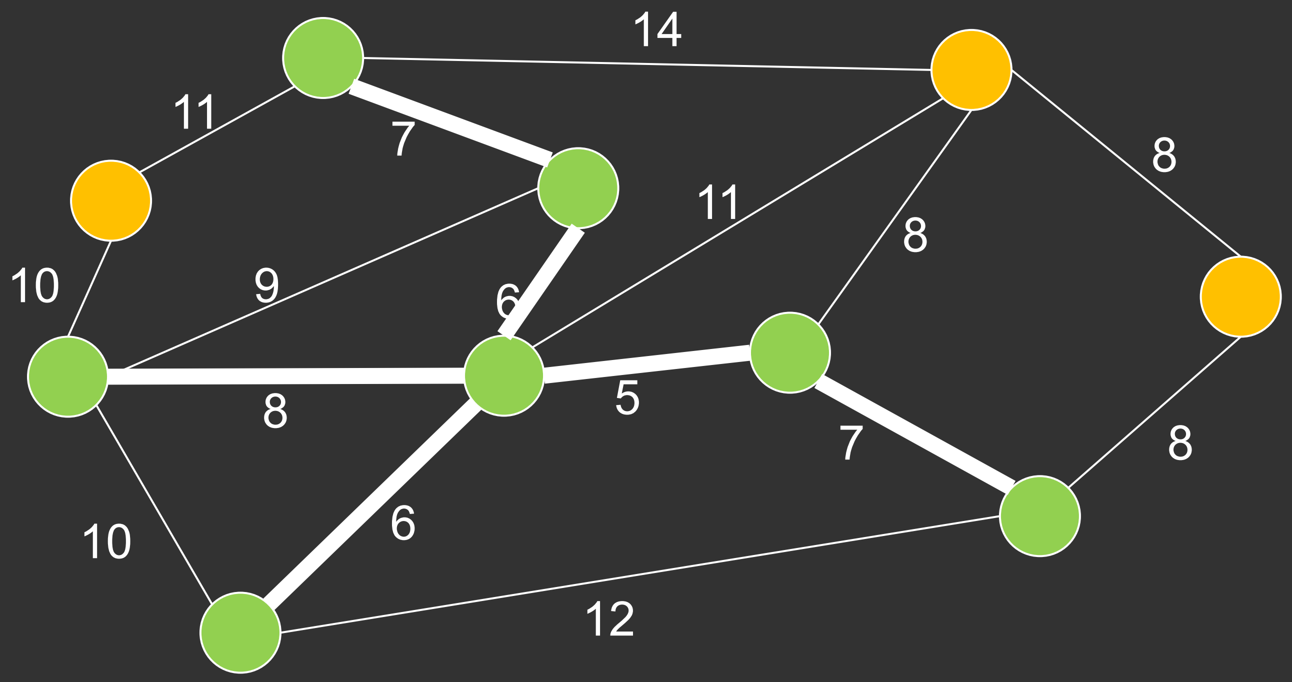
Prim’s Algorithm for MCSTs

- Continue this process of adding an edge with the cheapest cost connecting a green vertex to an orange vertex until all vertices are green.



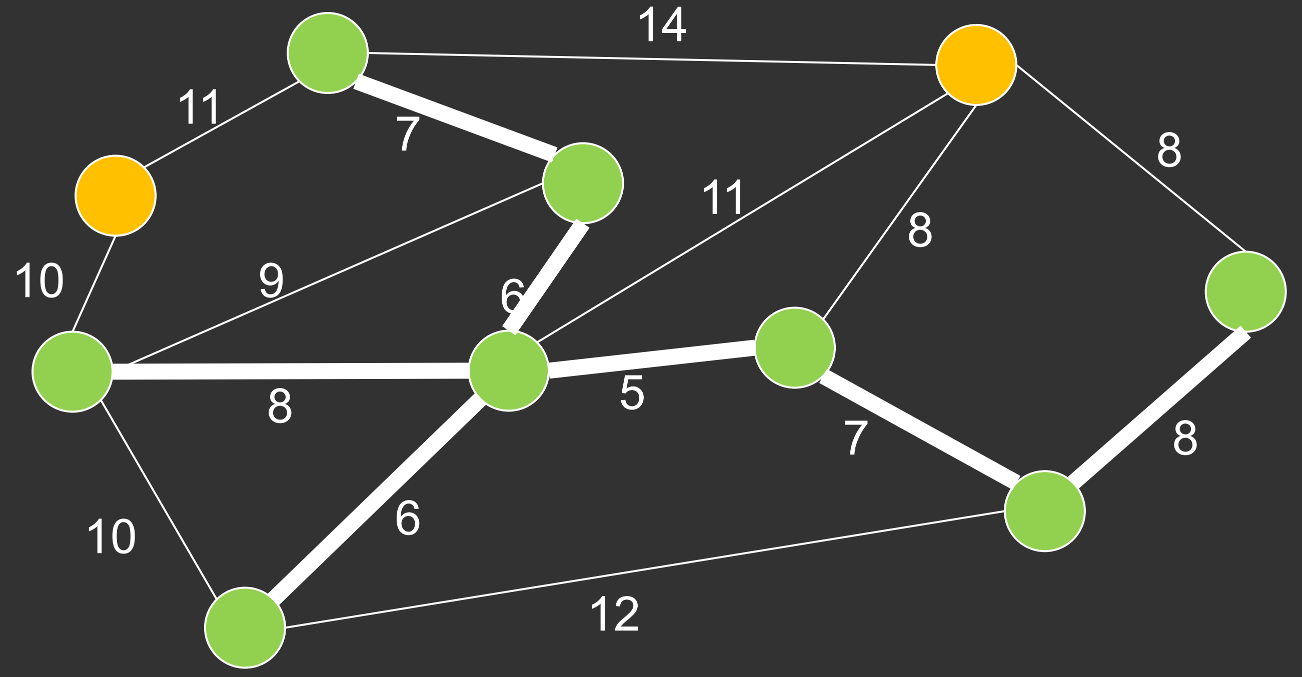
Prim’s Algorithm for MCSTs

- What if we have multiple cheapest edges? Ties can be broken arbitrarily. There may be multiple valid minimum cost spanning trees!



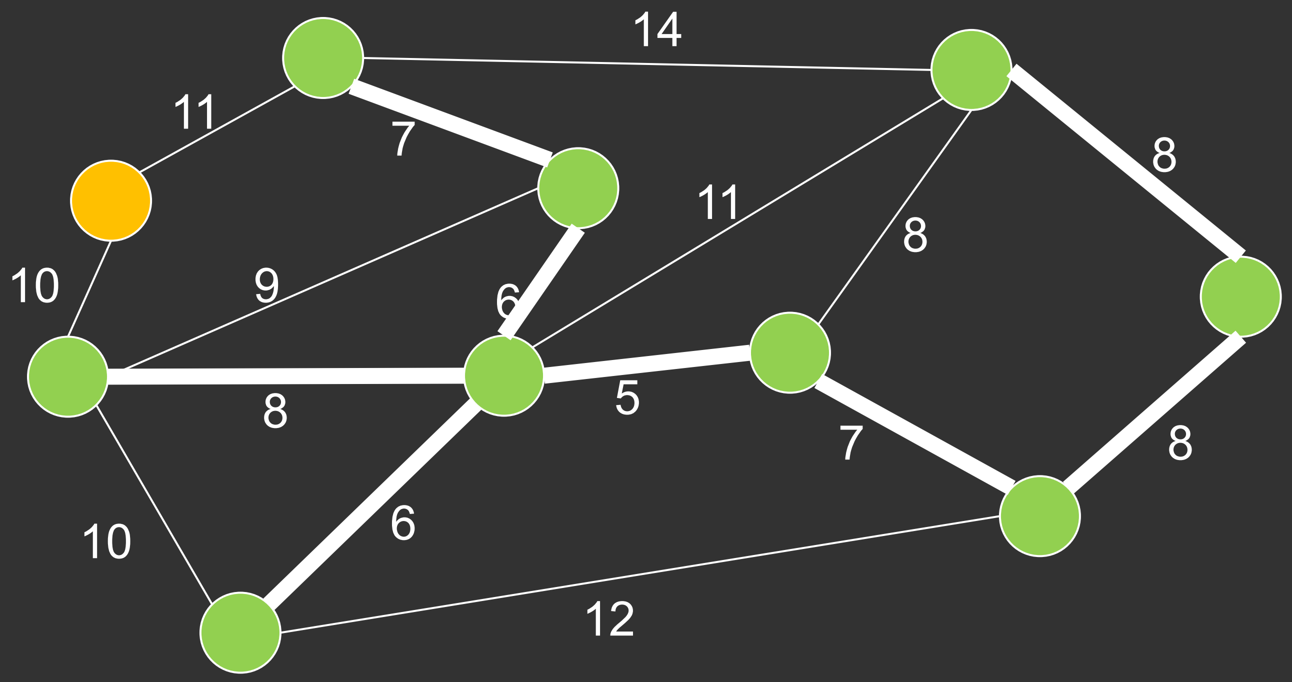
Prim’s Algorithm for MCSTs

- What if we have multiple cheapest edges? Ties can be broken arbitrarily. There may be multiple valid minimum cost spanning trees!



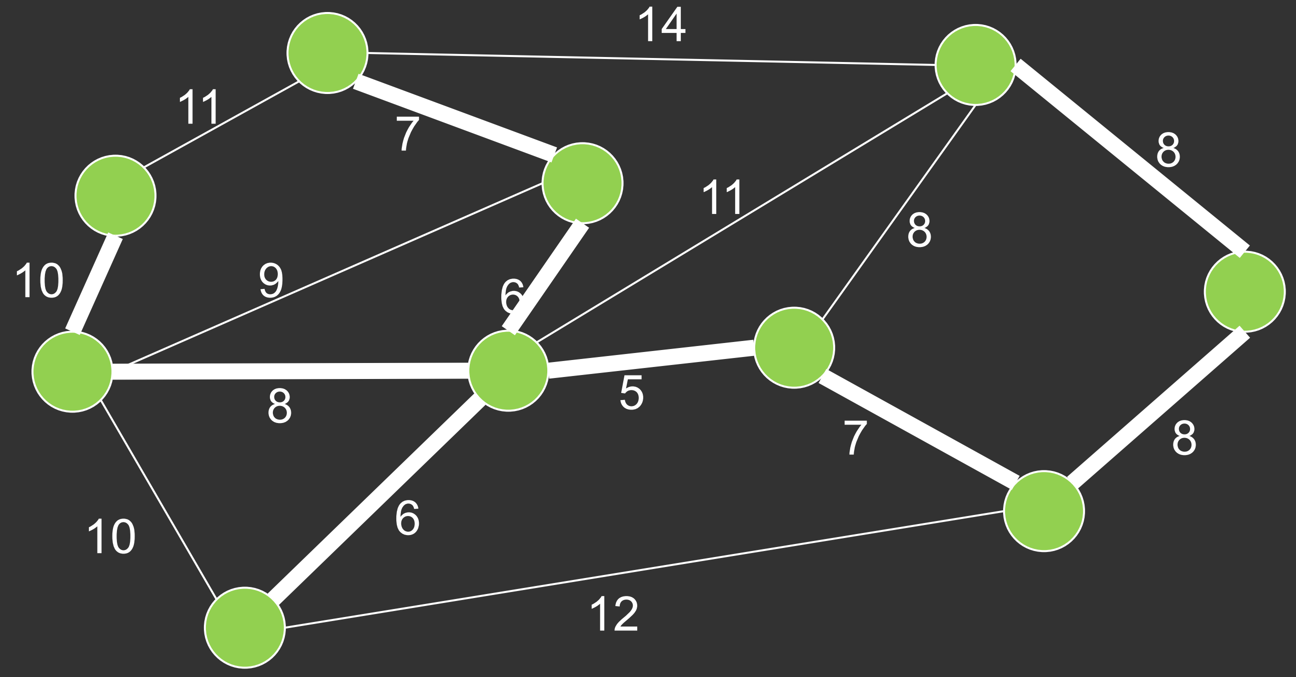
Prim’s Algorithm for MCSTs

- What if we have multiple cheapest edges? Ties can be broken arbitrarily. There may be multiple valid minimum cost spanning trees!



Prim’s Algorithm for MCSTs

- Once all vertices are green, we have constructed a minimum cost spanning tree.



Prim’s Algorithm

- The greedy algorithm we just described is called Prim’s algorithm
  - It always find a minimum-cost spanning tree for any connected graph (even if the weights are negative)!
- How can we argue that Prim’s algorithm is optimal?
- Why is it always a good idea to connect the current network to the cheapest node outside of the network?

The Key to Prim's Algorithm: "Cut Property"

Def: Sets V_1 and V_2 form a *partition* of a set V if

$$V_1 \cup V_2 = V \text{ and } V_1 \cap V_2 = \emptyset$$

- In other words, V_1 and V_2 together contain all of the vertices in V, but no vertex is in both V_1 and V_2.

Lemma: Let G=(V,E) be a connected graph and let V_1 and V_2 be a partition of V. *Every* MCST of G contains *a* cheapest edge between V_1 and V_2

Proof Sketch: Cut Property

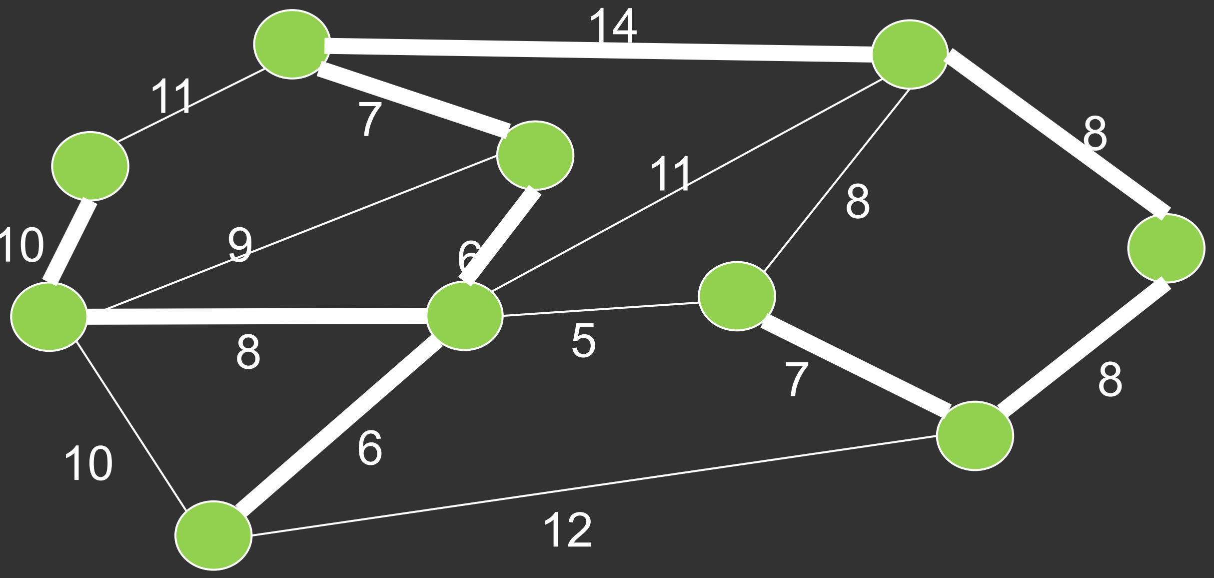
If there's one cheapest edge, it's in the MCST. If there's a tie, one of them is in the MCST

Lemma: Let G=(V,E) be a connected graph and let V_1 and V_2 be a partition of V. *Every* MCST of G contains *a* cheapest edge between V_1 and V_2

- Let e be *a* cheapest edge between V_1 and V_2
- Let T be *a* MCST of G.
  - If $e \notin T$, then $T \cup \{e\}$ contains a cycle C and e is an edge of C
  - Some other edge e' of C must also be between V_1 and V_2; since e is *a* cheapest edge, so w(e') = w(e)
    - (If it weren't, we could replace e with e' and T's cost would be cheaper, but that's impossible because T was a MCST.)

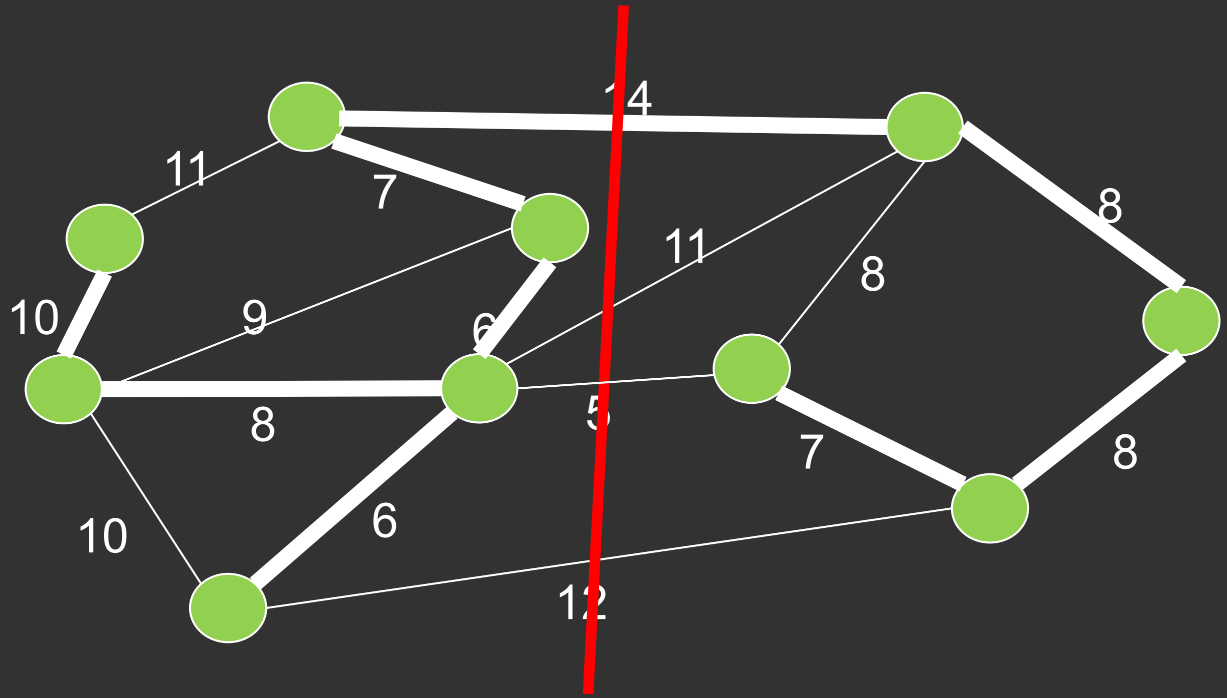
Proof Sketch: Cut Property

Lemma: Let G=(V,E) be a connected graph and let V_1 and V_2 be a partition of V. *Every* MCST of G contains *a* cheapest edge between V_1 and V_2



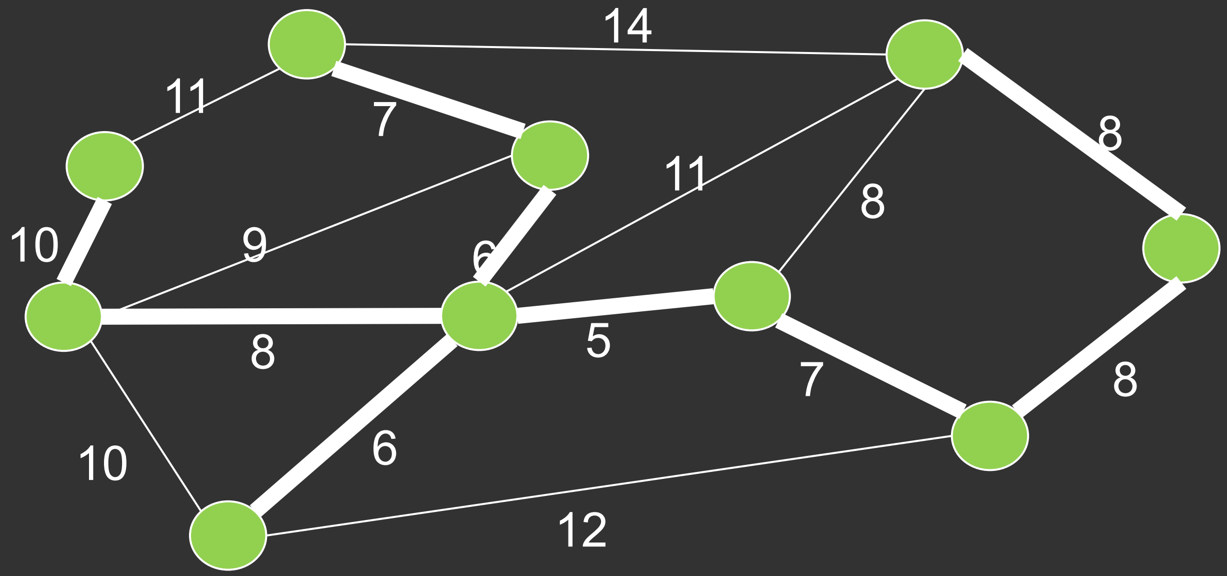
Proof Sketch: Cut Property

Lemma: Let G=(V,E) be a connected graph and let V_1 and V_2 be a partition of V. *Every* MCST of G contains *a* cheapest edge between V_1 and V_2



Proof Sketch: Cut Property

Lemma: Let G=(V,E) be a connected graph and let V_1 and V_2 be a partition of V. *Every* MCST of G contains *a* cheapest edge between V_1 and V_2



Using The Cut Property to Prove Prim

We’ll assume all edge costs are distinct

(Not necessary but otherwise proof is slightly less elegant)

Let T be a tree produced by the greedy algorithm, and suppose T* is a MCST for G.

Claim: T = T*

Idea of Proof: Show that every edge added to the tree T by the greedy algorithm is in T*

Clearly the first edge added to T is in T*

Why? Use the cut property!

Using The Cut Property to Prove Prim

Now use induction!

- Suppose that, for some k ≥ 1, the first k edges added to T are in T*. These form a tree T_k
- Let V_1 be the vertices of T_k and let $V_2 = V-V_1$
- Now, the greedy algorithm will add to T the cheapest edge e between V_1 and V_2
- But any MCST contains the (only!) cheapest edge between V_1 and V_2, so e is in T*
- Thus the first k+1 edges of T are in T*

Where we are

- Prim’s works!
  - Sometimes greedy choices don’t work well, but we proved that they are always optimal in this case.
- How can we implement Prim’s?
- First: write pseudocode
  - What methods do we need our data structures to support? Which ones must be fast?
- Then: decide on specifics

Prim's Algorithm

```
let v be a vertex of G;
set $V_1 \leftarrow \{v\}$, and $V_2 \leftarrow V - \{v\}$
let $A \leftarrow \emptyset$ // A will contain ALL edges between $V_1$ and $V_2$
while ($|V_1| < |V|$) :
    add to A all edges incident to v
    // note: A now may have edges with both ends in $V_1$
    repeat :
        remove cheapest edge e from A
    until e is an edge between $V_1$ and $V_2$

    add e to MCST

    let v $\leftarrow$ the vertex of e that is in $V_2$
    move v from $V_2$ to $V_1$
```

Implementing Prim's Algorithm

- We'll "build" the MCST by marking its edges as "visited"
- We'll "build" V_1 by marking its vertices visited
- Question: How should we represent A?
  - What operations are important to A?
    - Add all edges that are incident to some vertex
    - Remove a cheapest edge
  - We'll use a priority queue!
- When we remove an edge from A, we must verify it has one end in each of V_1 and V_2

ComparableEdge Class

- Values in a `PriorityQueue` need to implement `Comparable`
- We wrap edges of the PQ in a class called `ComparableEdge`
  - It requires the label used by graph edges to be of a `Comparable` type (e.g., `Integer`)

MCST: The Code

```
PriorityQueue<ComparableEdge<String,Integer>> q =
      new VectorHeap<ComparableEdge<String,Integer>>();

String v;                  // current vertex
Edge<String,Integer> e; // current edge
boolean searching;         // still building tree?

g.reset();                 // clear visited flags

// select a node from the graph, if any
Iterator<String> vi = g.iterator();
if (!vi.hasNext())
    return; // graph is empty!
v = vi.next();
```

MCST: The Code

```
do {
    // Add vertex to MCST and add all outgoing edges
    // to the priority queue

    g.visit(v); // all $V_1$ are visited

    for (String neighbor : g.neighbors(v)) {
        // turn it into outgoing edge
        e = g.getEdge(v, neighbor);
        // add the edge to the priority queue
        q.add(new ComparableEdge<String,Integer>(e));
    }

    ...
```

MCST: The Code

```
      ...
   searching = true; // looking for an edge btwn $V_1$&$V_2$
   while (searching && !q.isEmpty()) {
       // grab next shortest edge
       e = q.remove();
       // Is e between $V_1$ and $V_2$?
       v = e.there();
       if (g.isVisited(v)) v = e.here();
       if (!g.isVisited(v)) {
          searching = false;
             g.visitEdge(g.getEdge(e.here(),
                                e.there()));
            }
   }
} while (!searching);
```

Prim : Space Complexity

- Graph: O(|V| + |E|)
  - Each vertex and edge uses a constant amount of space
- Priority Queue O(|E|)
  - Each edge takes up constant amount of space
- Every other object (including the neighbor iterator) uses a constant amount of space
- Result: O(|V| + |E|)
  - Optimal in Big-O sense!

Prim : Time Complexity

Assume Map ops are O(1) time

For each iteration of do ... while loop

- Add neighbors to queue: O(deg(v) log |E|)
  - Iterator operations are O(1) [Why?]
  - Adding an edge to the queue is O(log |E|)
- Find next edge: O(# edges checked * log |E|)
  - Removing an edge from queue is O(log |E|) time
  - All other operations are O(1) time

Prim : Time Complexity

Over *all* iterations of do ... while loop

Step I: Add neighbors to queue:

- For each vertex, it’s O(deg(v) log |E|) time
- Adding over all vertices gives

$$\sum_{v \in V} \deg(v) \log |E| = \log |E| \sum_{v \in V} \deg(v) = \log |E| * 2 |E|$$

Prim : Time Complexity

Over *all* iterations of do ... while loop

Step 2: Find next edge: O(# edges checked * log |E|)

- Each edge is checked at most once
- Adding over all edges gives O(|E| log |E|) again

Thus, overall time complexity (worst case) of Prim's Algorithm is O(|E| log |E|)

Summary

- Prim's algorithm finds a MCST for a single connected component of any graph G=(V,E)

- It is a greedy algorithm, but
- it finds a globally optimal solution!

- Careful analysis of the required operations helps us choose the best data structures to maximize performance.