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

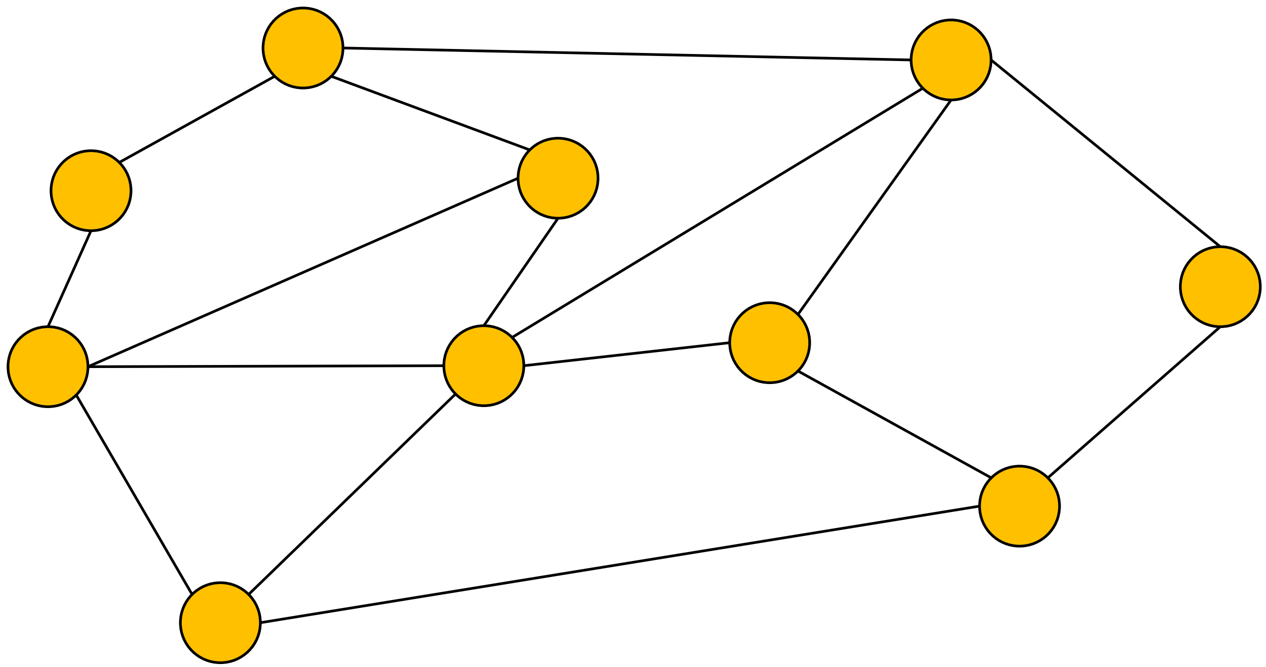
Graph Definitions

- A *subgraph* of a graph G=(V, E) is a graph G'=(V',E') where:
  - $V' \subseteq V$ // the set of vertices in the subgraph is a subset
  - $E' \subseteq E$, // the set of edges in the subgraph is a subset
  - If $e \in E'$ where $e = \{u,v\}$, then $u, v \in V'$ // edges in the subgraph connect vertices that are also in the subgraph

- If V' = V, then G' is called a *spanning subgraph of G*
  - In other words, a spanning subgraph must contain all the vertices of the original graph, but it is not required to contain all of the edges

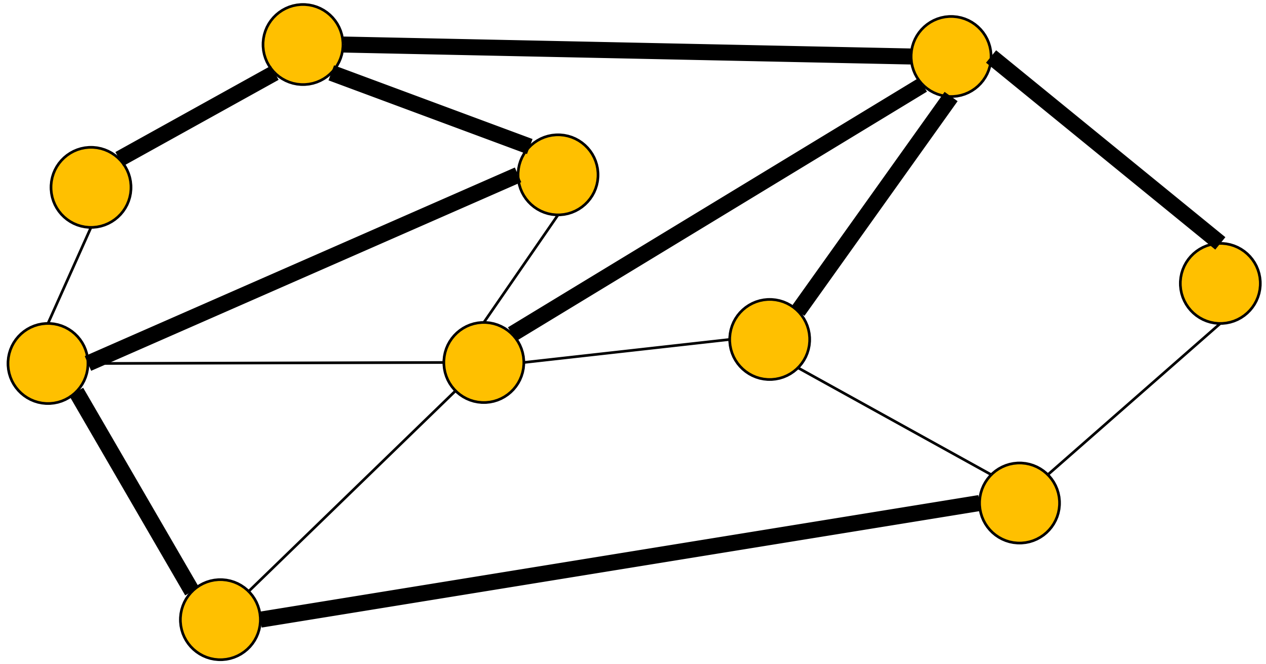
Spanning Trees

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



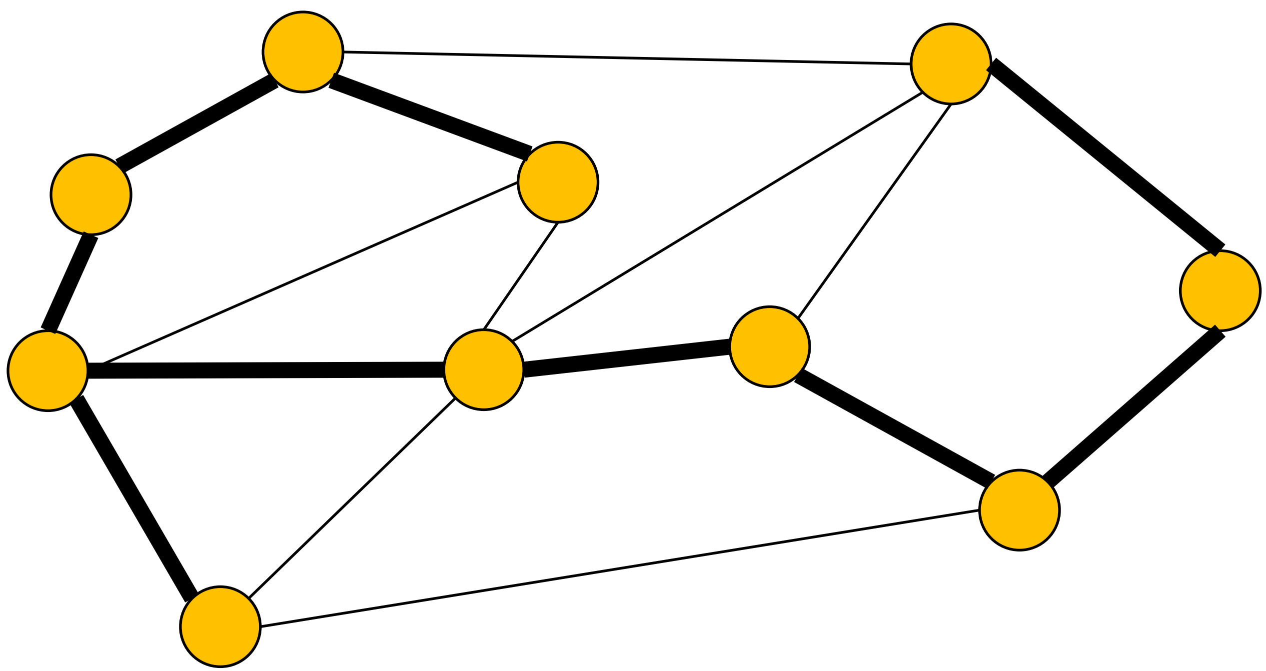
Spanning Trees

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



Spanning Trees

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



Spanning Trees

Theorem: Every connected graph G=(V,E) contains a spanning subgraph G'=(V,E') that is a tree

Proof idea:

- If G' is not a tree, then it contains some cycle C
- Removing an edge from C leaves G' connected (why?)
- Repeat this process of removing edges until no more cycles remain
- Now we are left with a tree

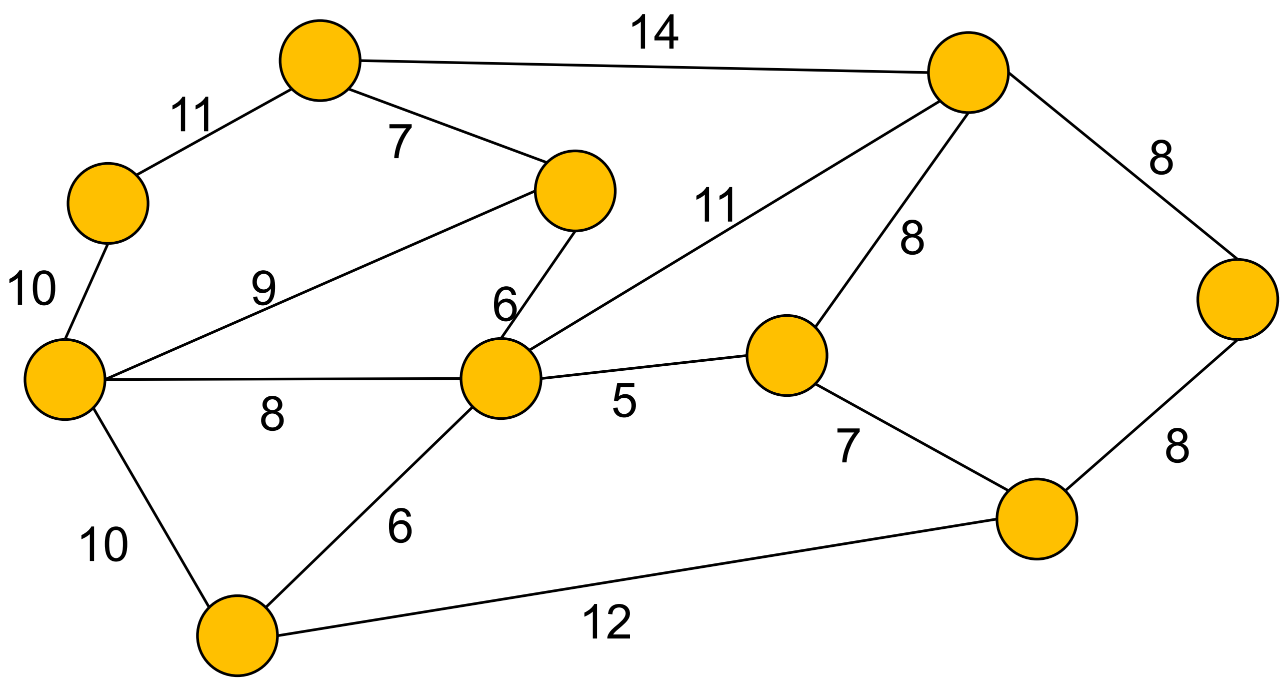
Minimum Cost Spanning Tree

- If a spanning tree is a subgraph that covers all the vertices using the minimum number of edges then,

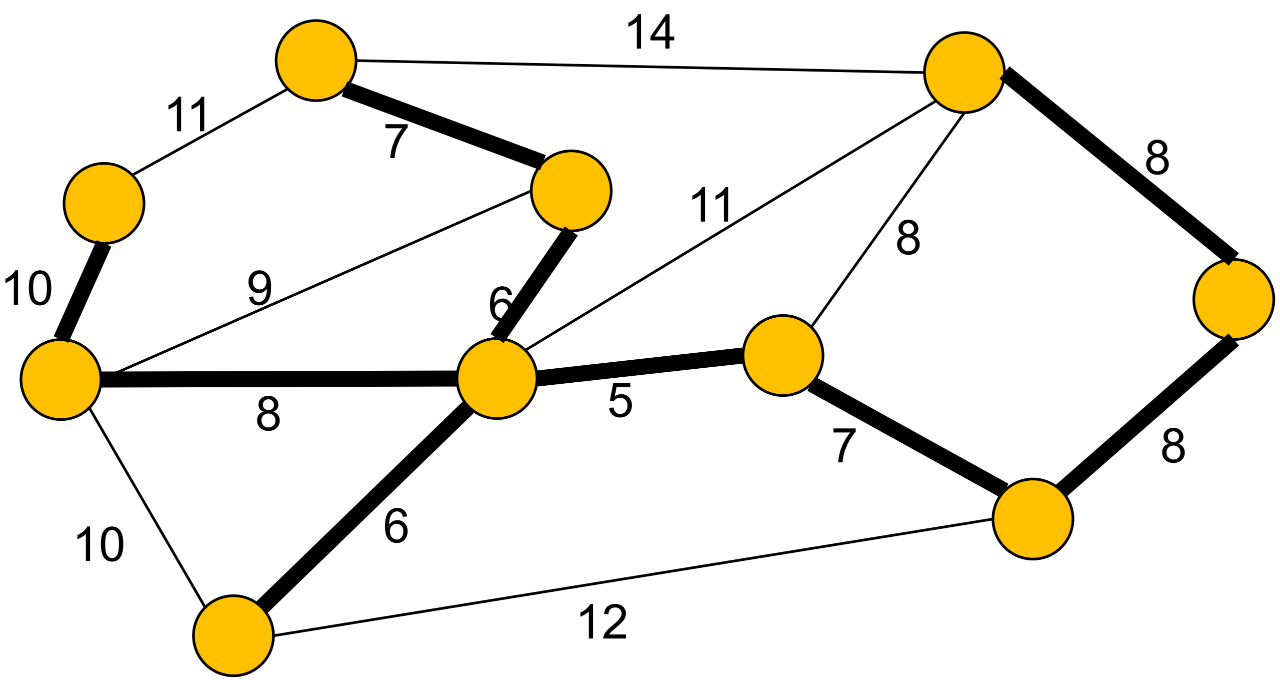
- Suppose we're given a graph that is:
  - connected, and
  - has weighted edges (integer, float, double, etc.)

- 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



MCST Applications?

- Suppose Williamstown builds a municipal broadband network. Let:
  - vertices be homes,
  - edges be places where cables can be laid (roads?),
  - weights be distance (cable priced $/meter)
- If we identify the minimum cost spanning tree, we can build a network that connects every home for the minimum cost.
- Thus, finding a MCST is both theoretically and practically interesting!

First Attempt at Finding a MCST

Instead of finding a minimum cost spanning tree, suppose we just wanted to find to find one that is "pretty good"

Idea: we could try to grow it greedily!

- Pick a vertex and choose its cheapest incident edge. Now we have a (small) tree
- Repeatedly add the cheapest edge to our tree that still keeps it a tree (i.e., connected and no cycles)
- Once every vertex is connected, we have a 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)!
- Is this surprising?
  - Each step makes the best choice it can in the moment, but it lacks the "global" state of the problem.
  - Yet the solution is in fact globally optimal. Cool!

The Key to Prim's Algorithm

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

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.)

Using The Key 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 key!

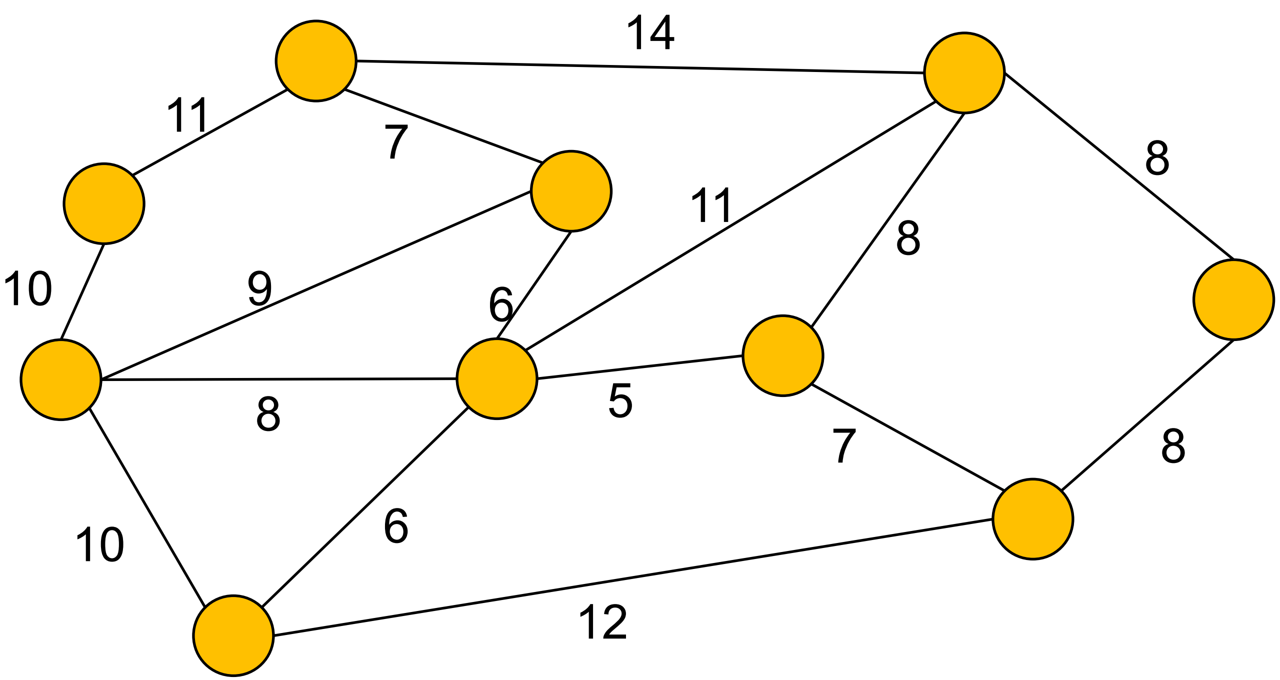
Using The Key

Now use induction!

- Suppose that, for some $k \geq 1$, 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*

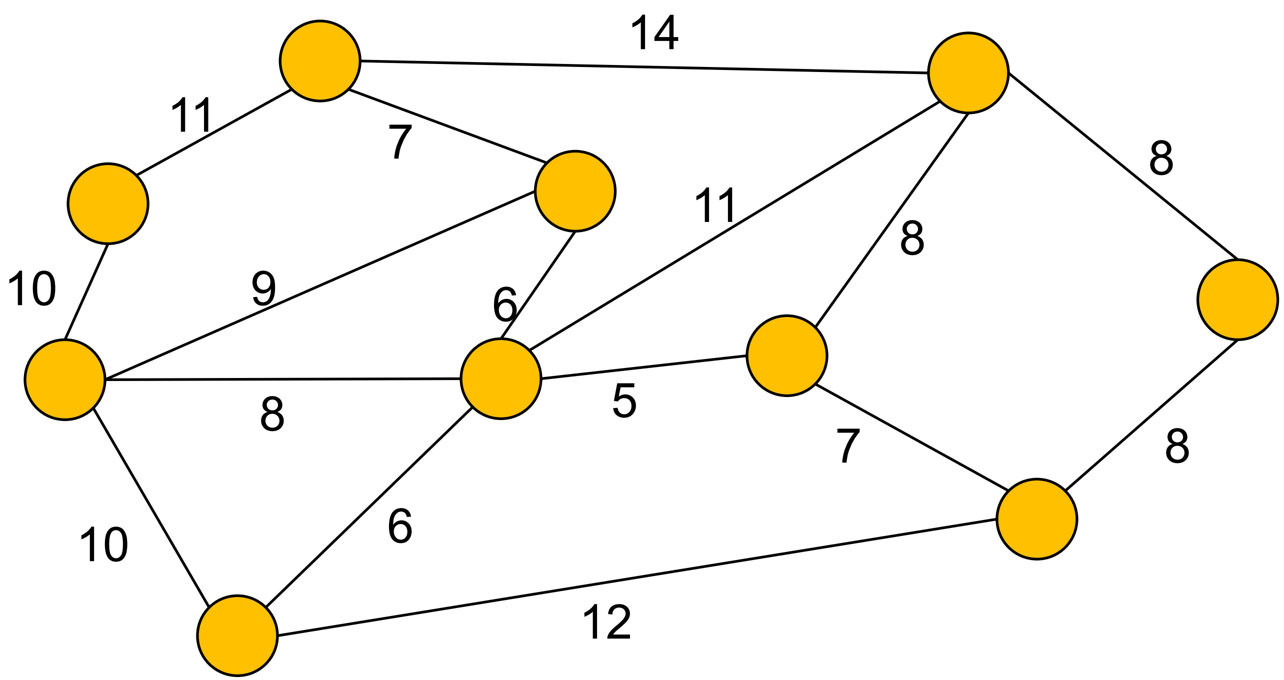
Prim's Algorithm for MCSTs

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



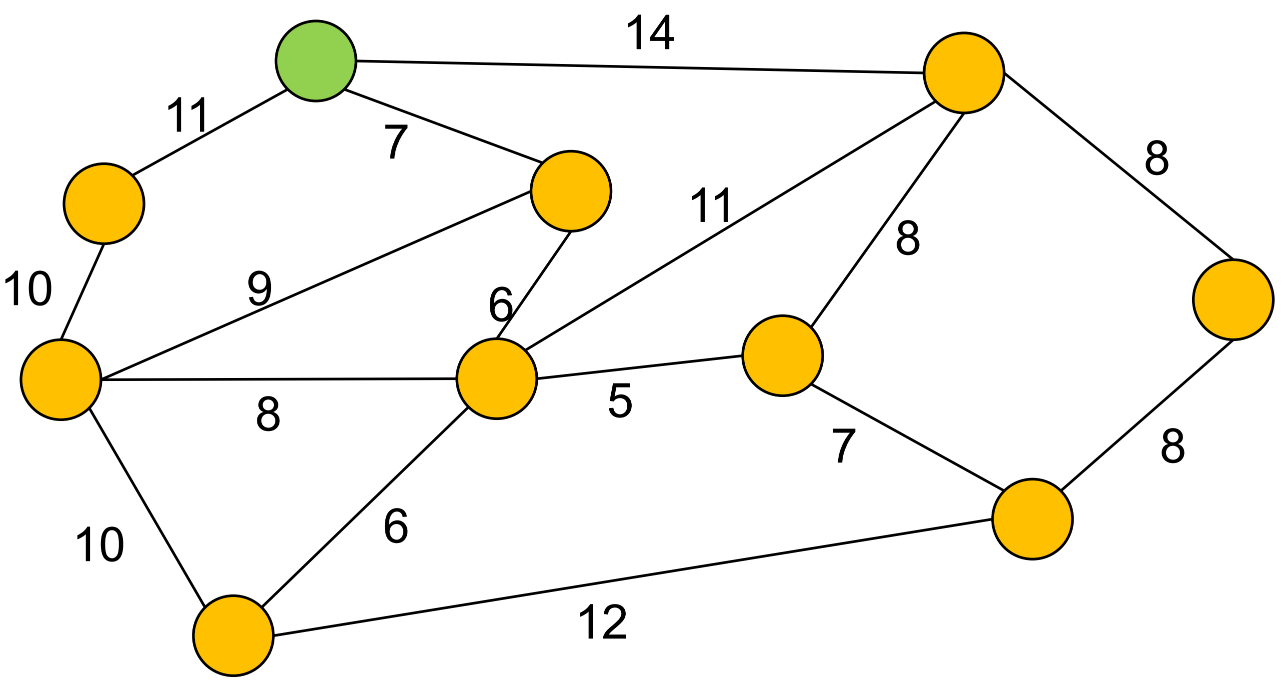
Prim's Algorithm for MCSTs

- Start by picking some vertex



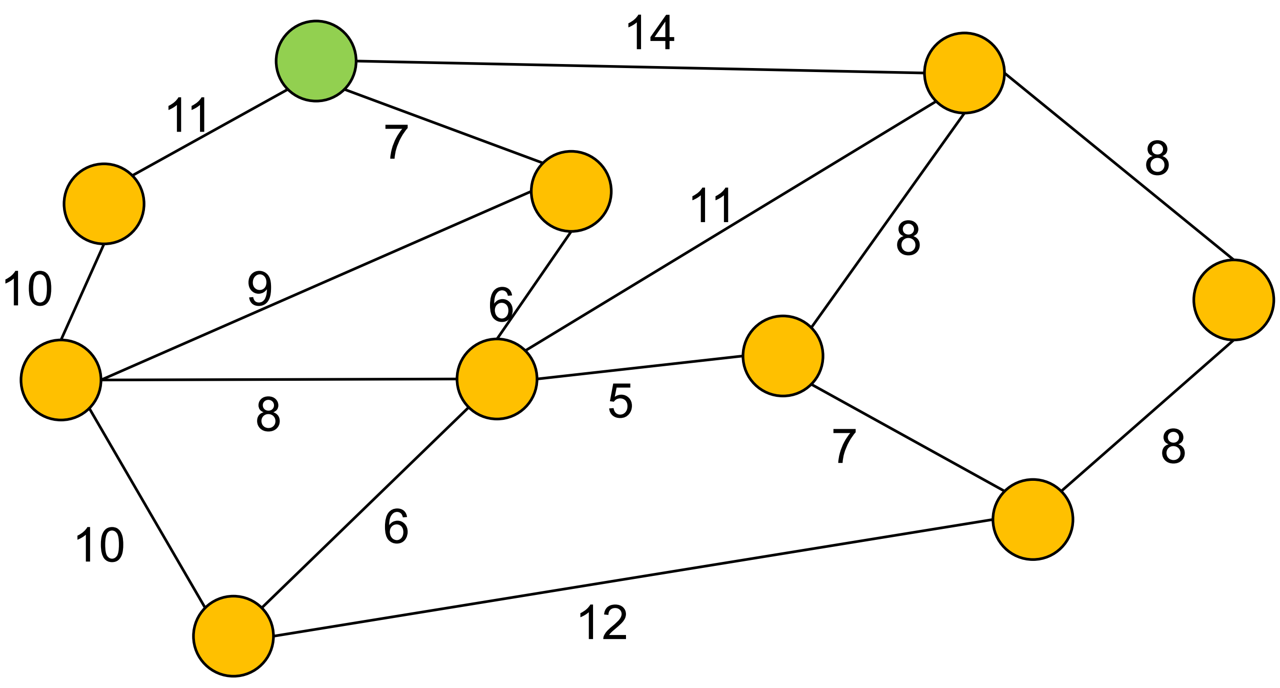
Prim's Algorithm for MCSTs

- Start by picking some vertex



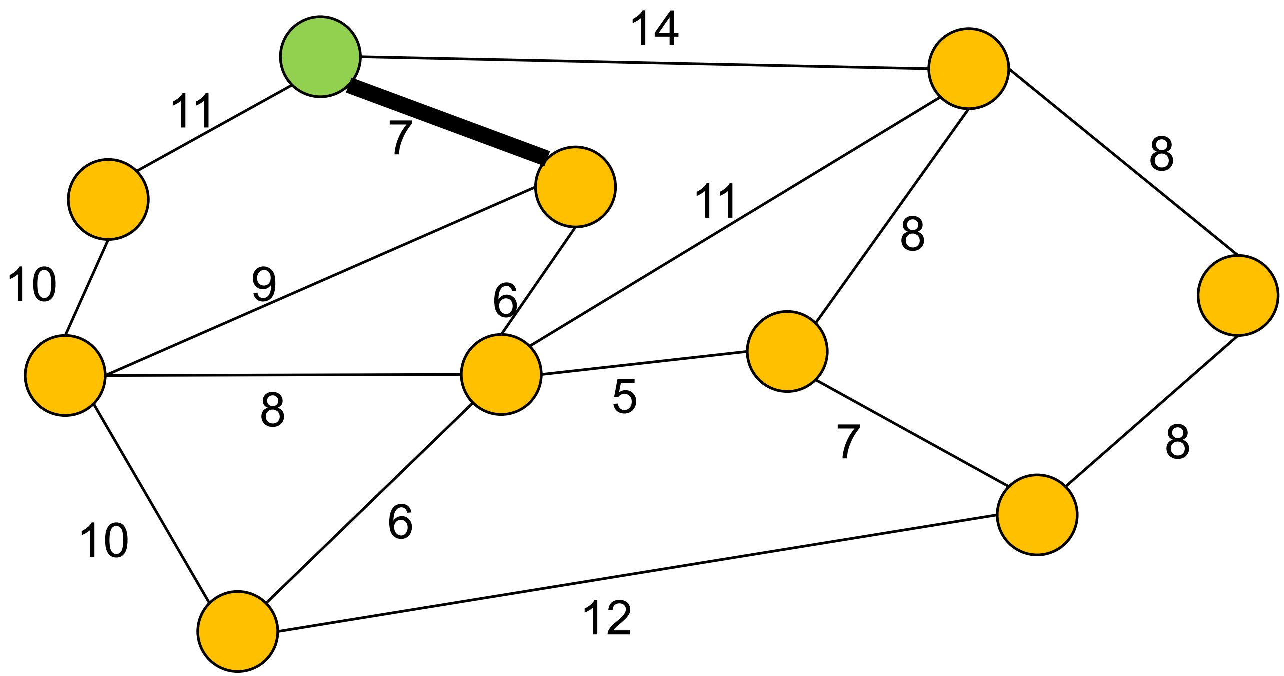
Prim's Algorithm for MCSTs

- We'll note our partitions V_1 and V_2 using green and orange sets. Select an edge with the cheapest cost that connects a green vertex to an orange vertex.



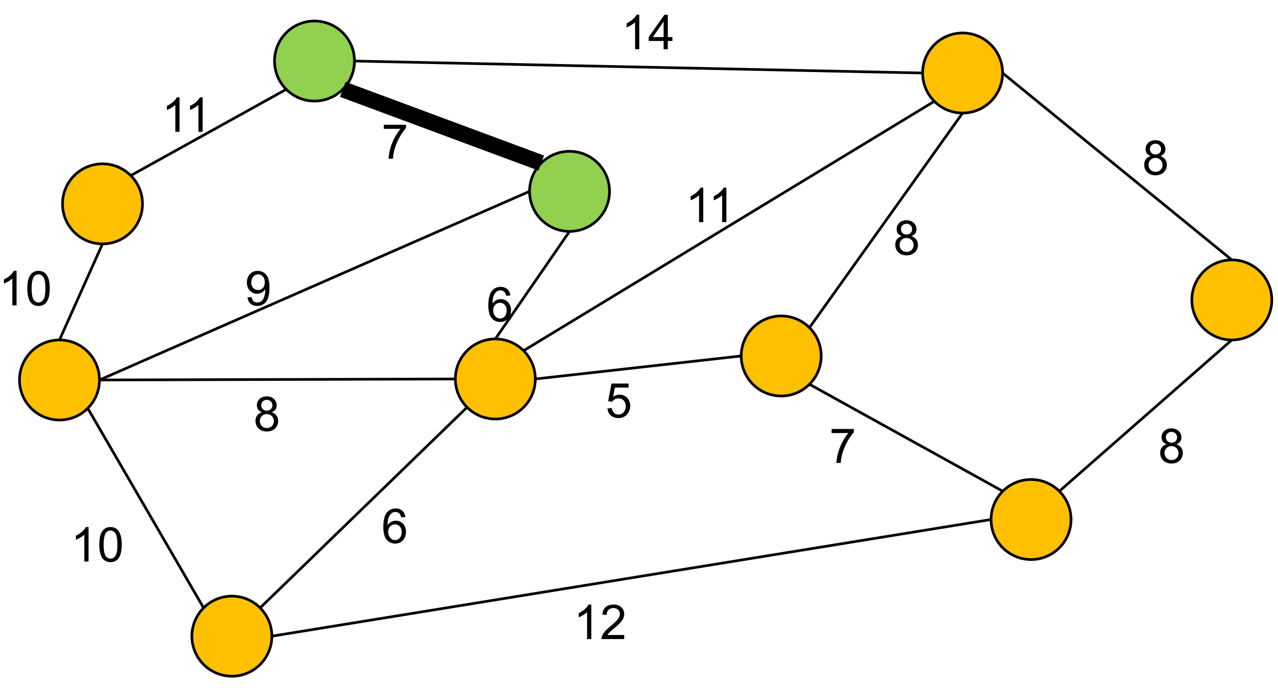
Prim's Algorithm for MCSTs

- We'll note our partitions V_1 and V_2 using green and orange sets. Select an edge with the cheapest cost that connects a green vertex to an orange vertex.



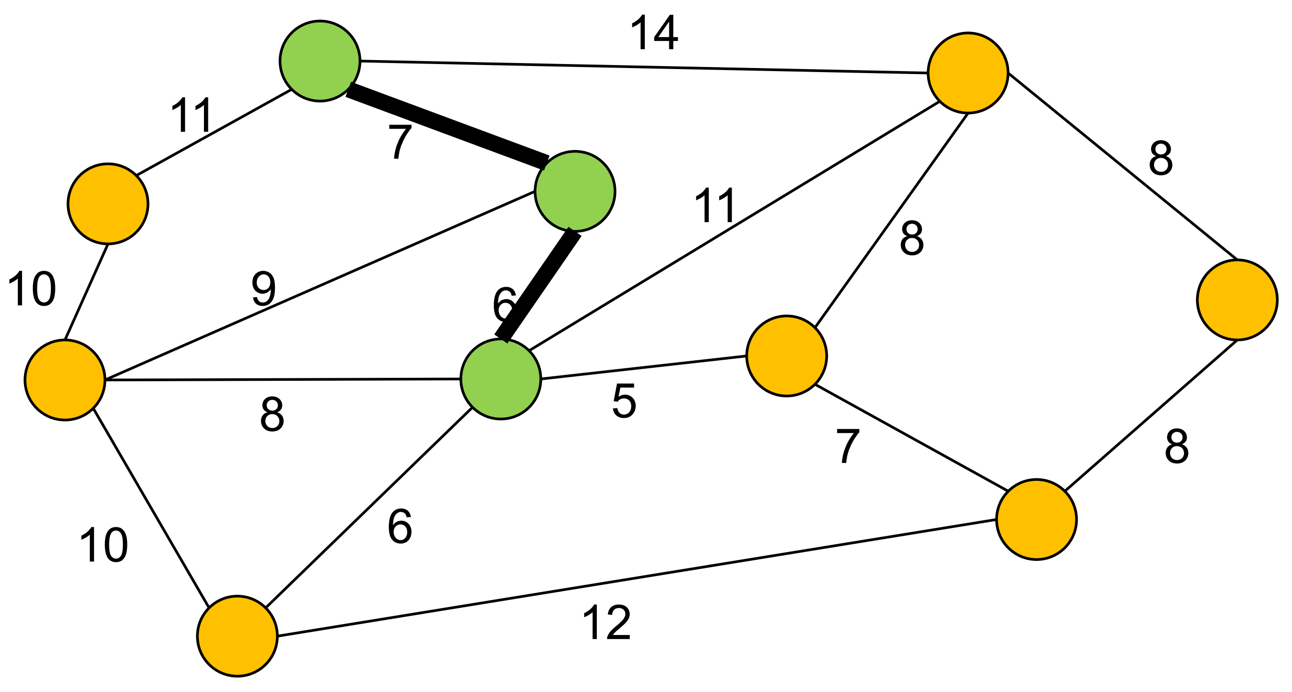
Prim's Algorithm for MCSTs

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



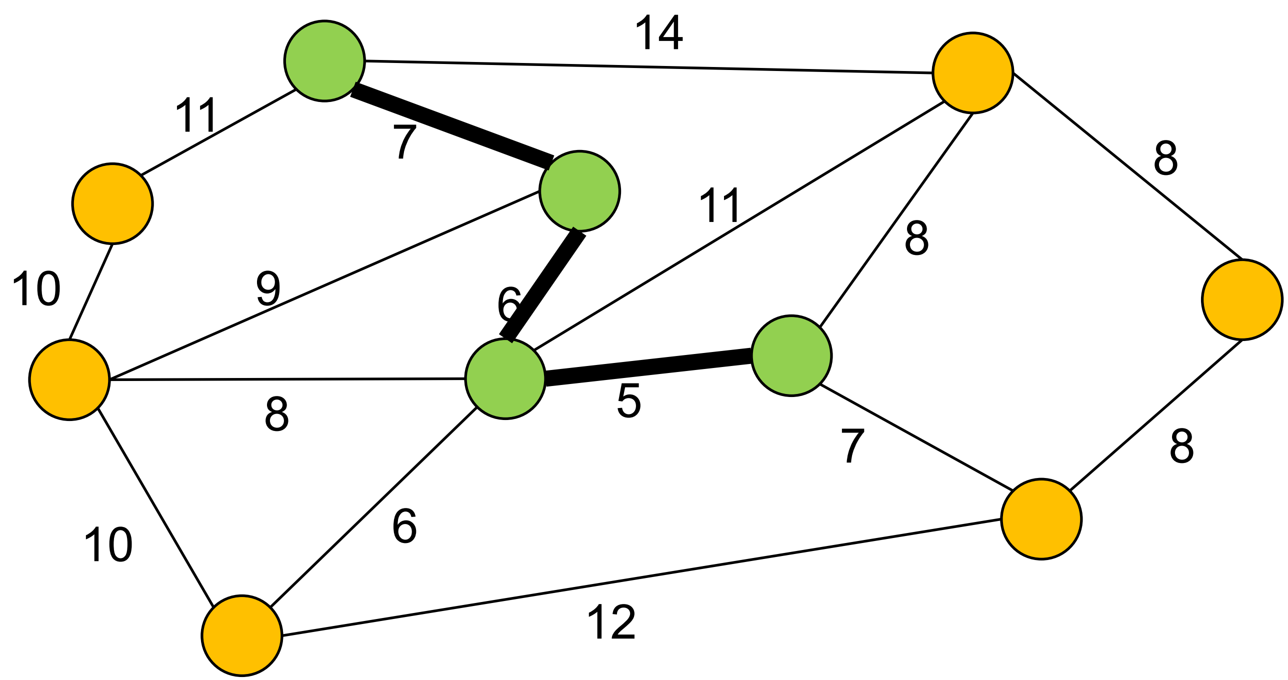
Prim's Algorithm for MCSTs

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



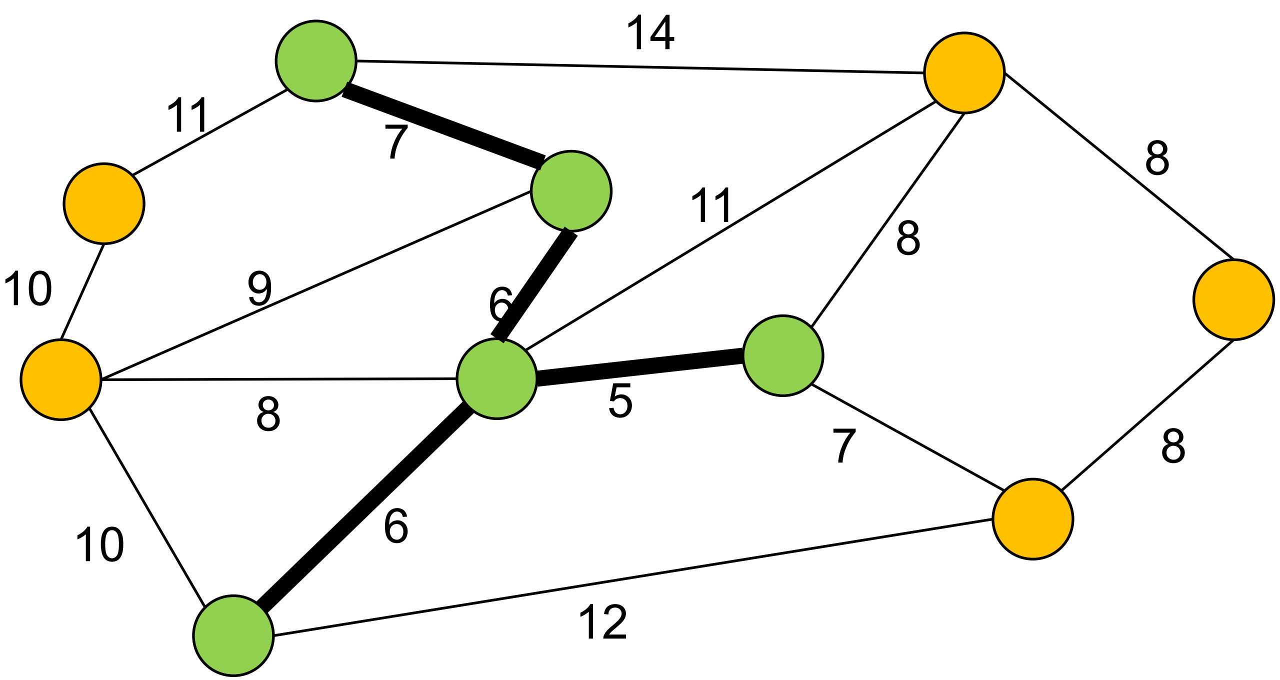
Prim's Algorithm for MCSTs

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



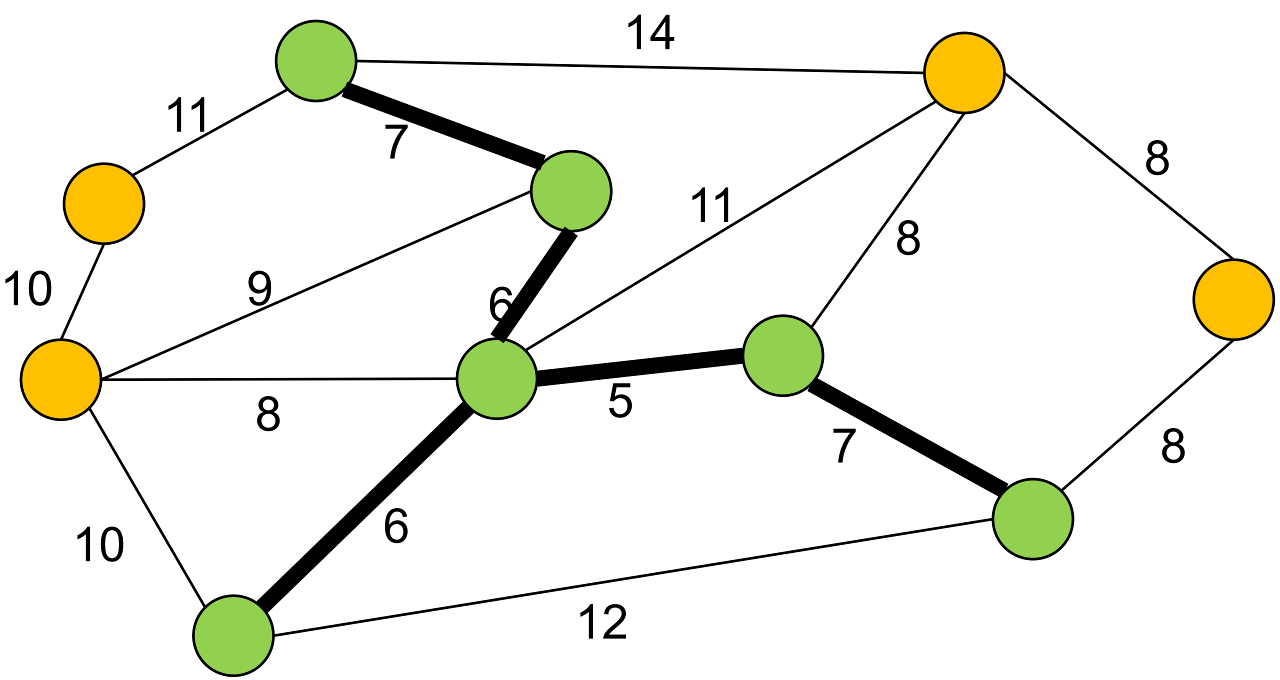
Prim's Algorithm for MCSTs

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



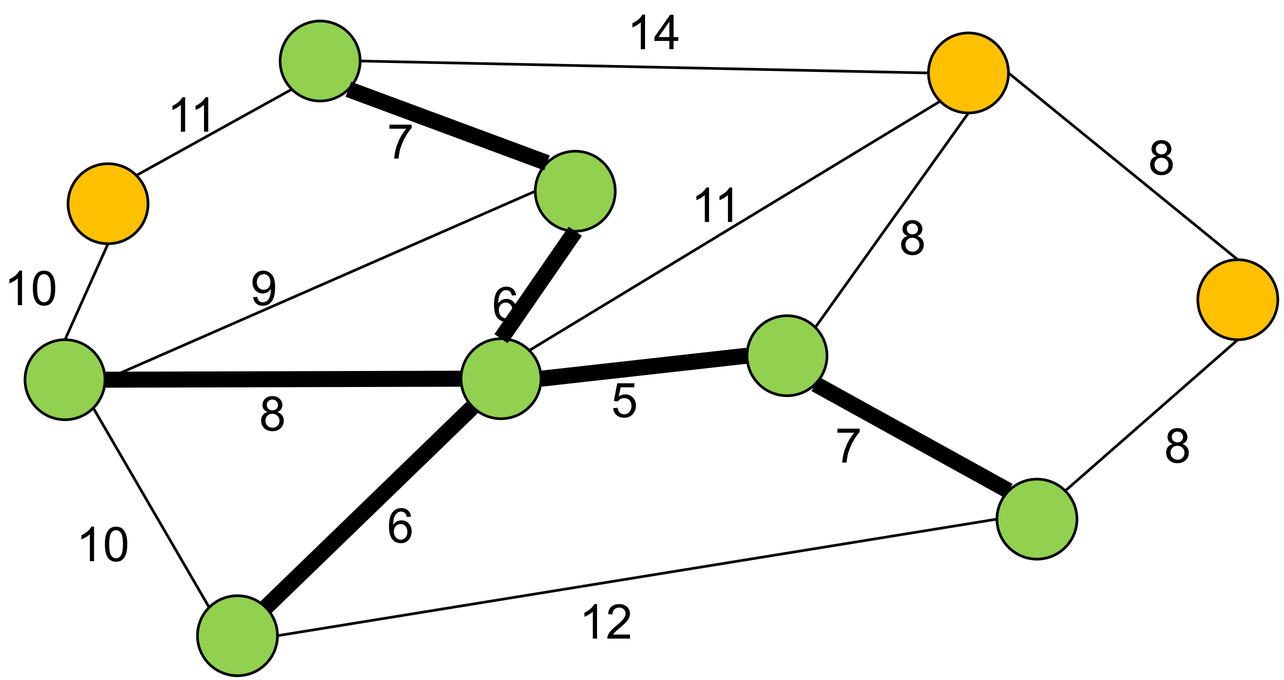
Prim's Algorithm for MCSTs

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



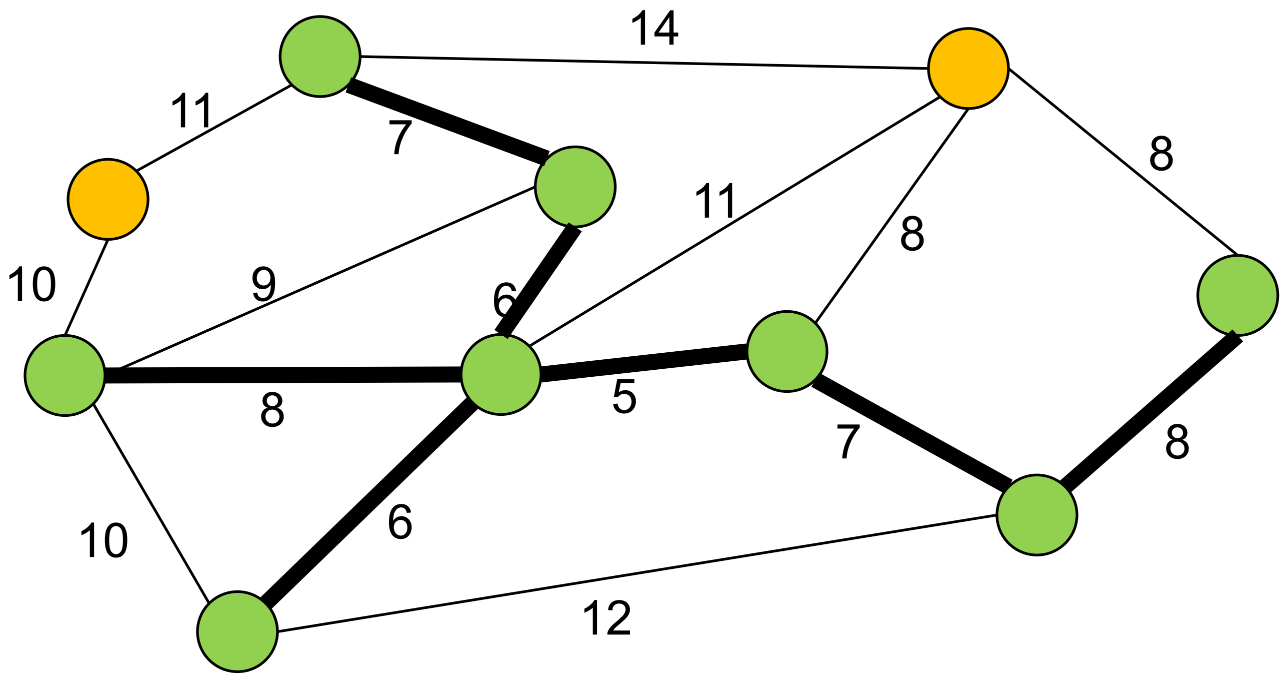
Prim's Algorithm for MCSTs

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



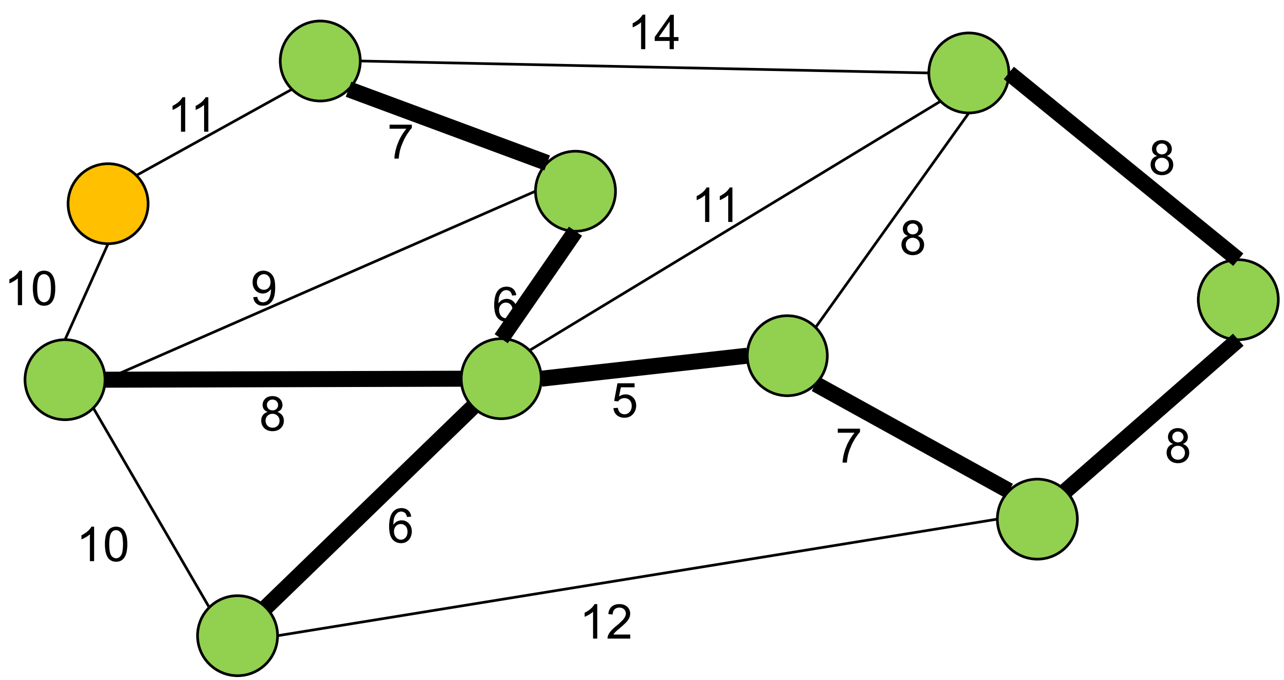
Prim's Algorithm for MCSTs

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



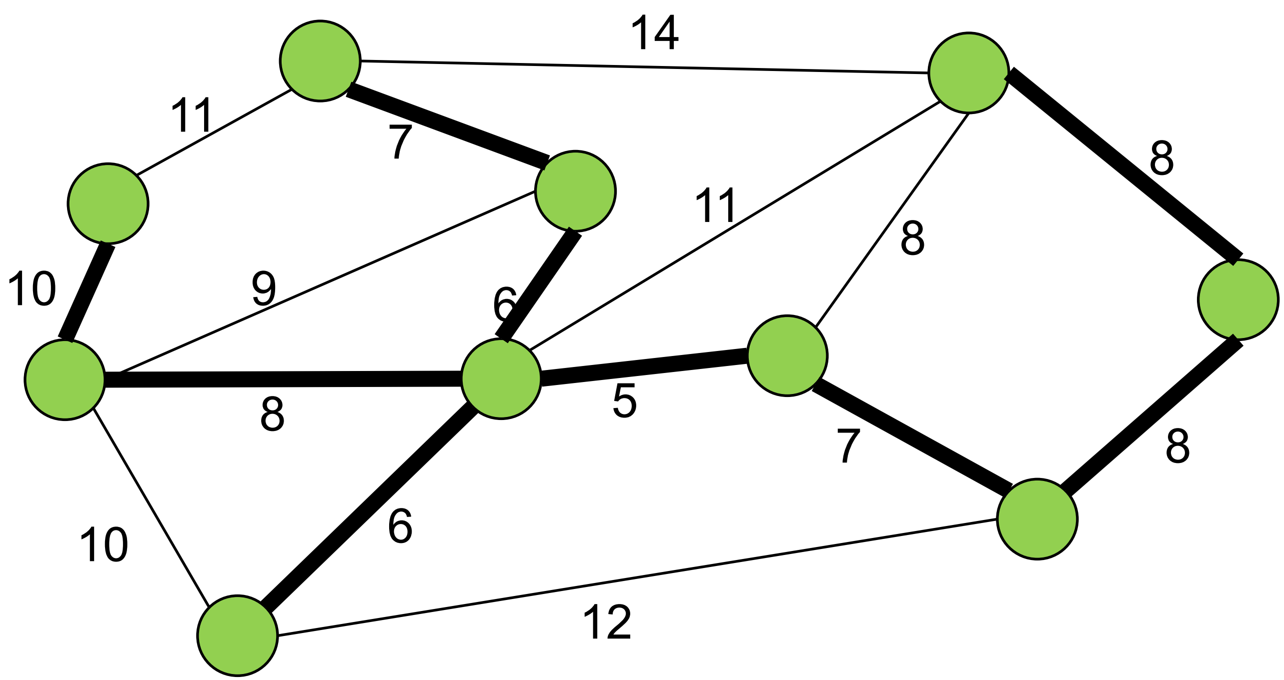
Prim's Algorithm for MCSTs

- 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

```
let v be a vertex of G;
set V₁ ← {v}, and V₂ ← V - {v}
let A be the set of all edges between V₁ and V₂
while (|V₁| < |V|) :
      let e ← min edge in A between V₁ and V₂

      add e to MCST

      let u ← the vertex of e that is in V₂
      move u from V₂ to V₁;


      add to A all edges incident to u
      // note: A now may have edges with both ends in V₁
```

Prim's Algorithm (Variant)

- Let's look at an equivalent version of the algorithm that more closely matches the Java code we will use…

- It replaces the "`let e ← min edge in A between` V_1 `and` V_2" with explicit steps to:
  - Find the cheapest edge not yet in the MCST
  - Verify that it connects a vertex in V_1 with one in V_2
  - (And if not, continue checking the next cheapest edge)

Prim's Algorithm (Variant)

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

Prim's Algorithm (Variant)

- Note: If G is not connected, A will eventually be empty even though $|V_1| < |V|$

- We fix this by:
  - Replacing `while(|V₁| < |V|)` with `while(|V₁| < |V|) && A≠∅)`
  - Replacing `until e is an edge btwn V₁ and V₂` with `until A≠∅ or e is an edge btwn V₁ and V₂`

- Then Prim will find the MCST for the component containing v

Prim's Algorithm (Variant)

```
let v be a vertex of G;
set V₁ ← {v}, and V₂ ← V - {v}
let A ← Ø // A will contain ALL edges between V₁ and V₂
while (|V₁| < |V| && |A| > 0) :
      add to A all edges incident to v
      // note: A now may have edges with both ends in V₁
      repeat :
            remove cheapest edge e from A
      until (A is empty ||
                e is an edge between V₁ and V₂)

      add e to MCST

      let v ← the vertex of e that is in V₂
      move v from V₂ to V₁
```

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 V1 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 V1&V2
    while (searching && !q.isEmpty()) {
          // grab next shortest edge
          e = q.remove();
          // Is e between V1 and V2?
          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);
```

Summary

- Prim's algrogithm 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.