Minimum Cost Spanning Trees

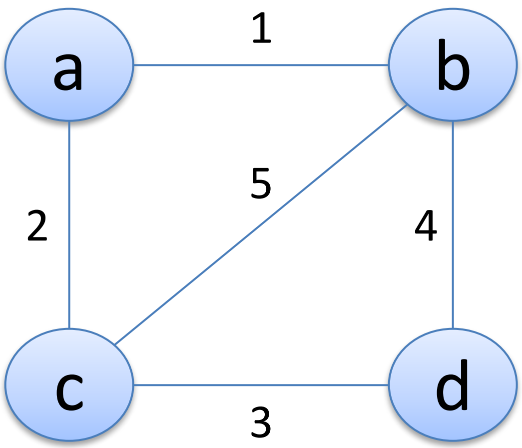
CSC263 Tutorial 10

Minimum cost spanning tree (MCST)

- What is a minimum cost spanning tree?
  - Tree
    - No cycles; equivalently, for each pair of nodes u and v, there is only one path from u to v
  - Spanning
    - Contains every node in the graph
  - Minimum cost
    - Smallest possible total weight of any spanning tree

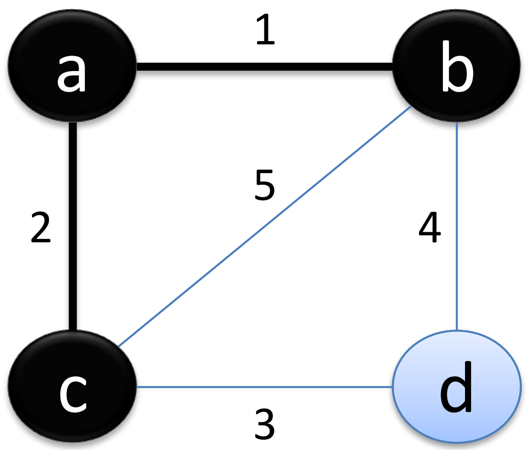
Minimum cost spanning tree (MCST)

- Let’s think about simple MCSTs on this graph:



Minimum cost spanning tree (MCST)

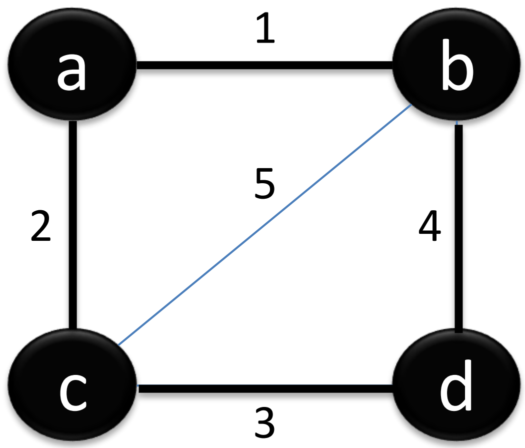
- Black edges and nodes are in T
- Is T a minimum cost spanning tree?



- Not spanning; d is not in T.

Minimum cost spanning tree (MCST)

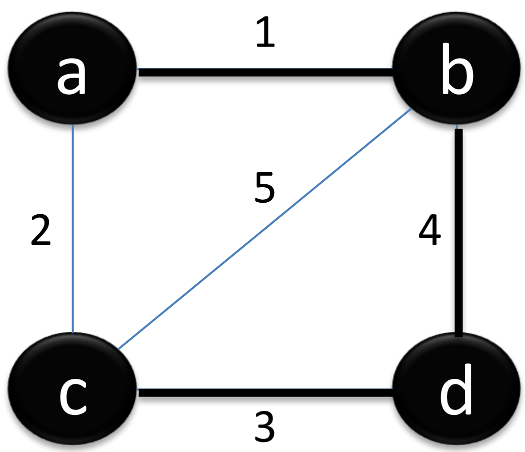
- Black edges and nodes are in T
- Is T a minimum cost spanning tree?



- Not a tree; has a cycle.

Minimum cost spanning tree (MCST)

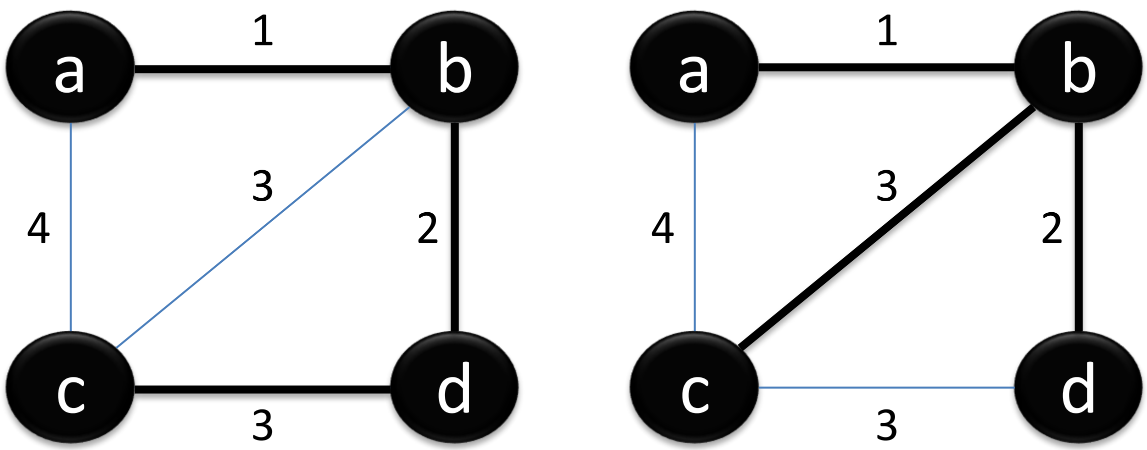
- Black edges and nodes are in T
- Is T a minimum cost spanning tree?



- Not minimum cost; can swap edges 4 and 2.

Minimum cost spanning tree (MCST)

- Which edges form a MCST?



Quick Quiz

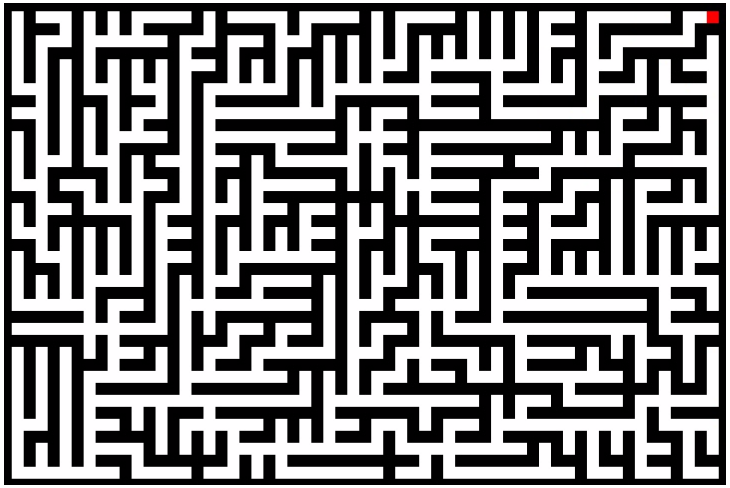
- If we build a MCST from a graph G = (V, E), how may edges does the MCST have?
- When can we find a MCST for a graph?

An application of MCSTs

- Electronic circuit designs (from Cormen et al.)
  - Circuits often need to wire together the pins of several components to make them electrically equivalent.
  - To connect *n* pins, we can use *n* - 1 wires, each connecting two pins.
  - Want to use the minimum amount of wire.
  - Model problem with a graph where each pin is a node, and every possible wire between a pair of pins is an edge.

A few other applications of MCSTs

- Planning how to lay network cable to connect several locations to the internet
- Planning how to efficiently bounce data from router to router to reach its internet destination
- Creating a 2D maze (to print on cereal boxes, etc.)



Building a MCST

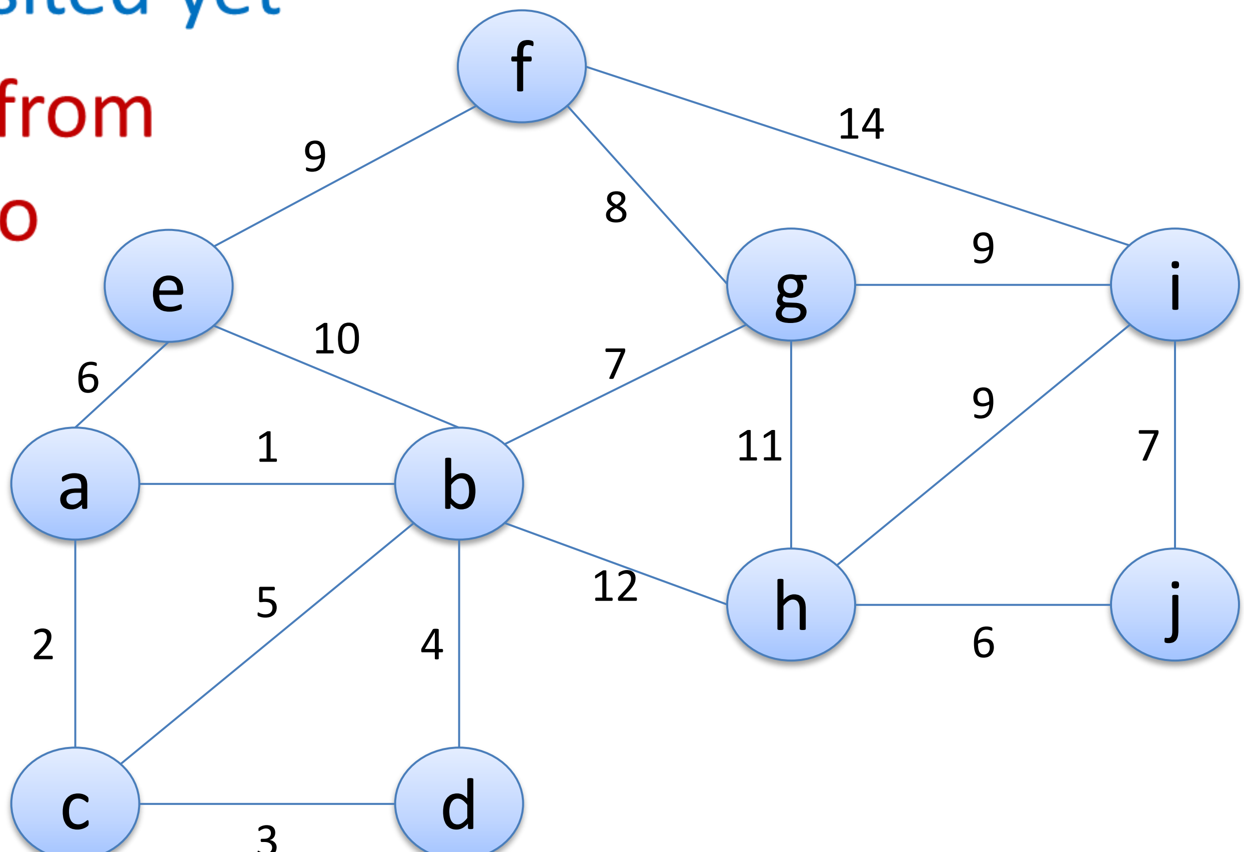
- Prim's algorithm takes a graph G = (V, E) and builds an MCST ***T***
- PrimMCST(V, E)
  - Pick an arbitrary node **r** from V
  - Add **r** to *T*
  - While *T* contains < |V| nodes
    - Find a **minimum weight edge (u, v)** where $\mathbf{u} \in \boldsymbol{T}$ and $\mathbf{v} \notin \boldsymbol{T}$
    - Add node v to *T*

In the book's terminology, we find a **light edge crossing the cut (T, V-T)**

The book proves that adding |V|-1 such edges will create a MCST

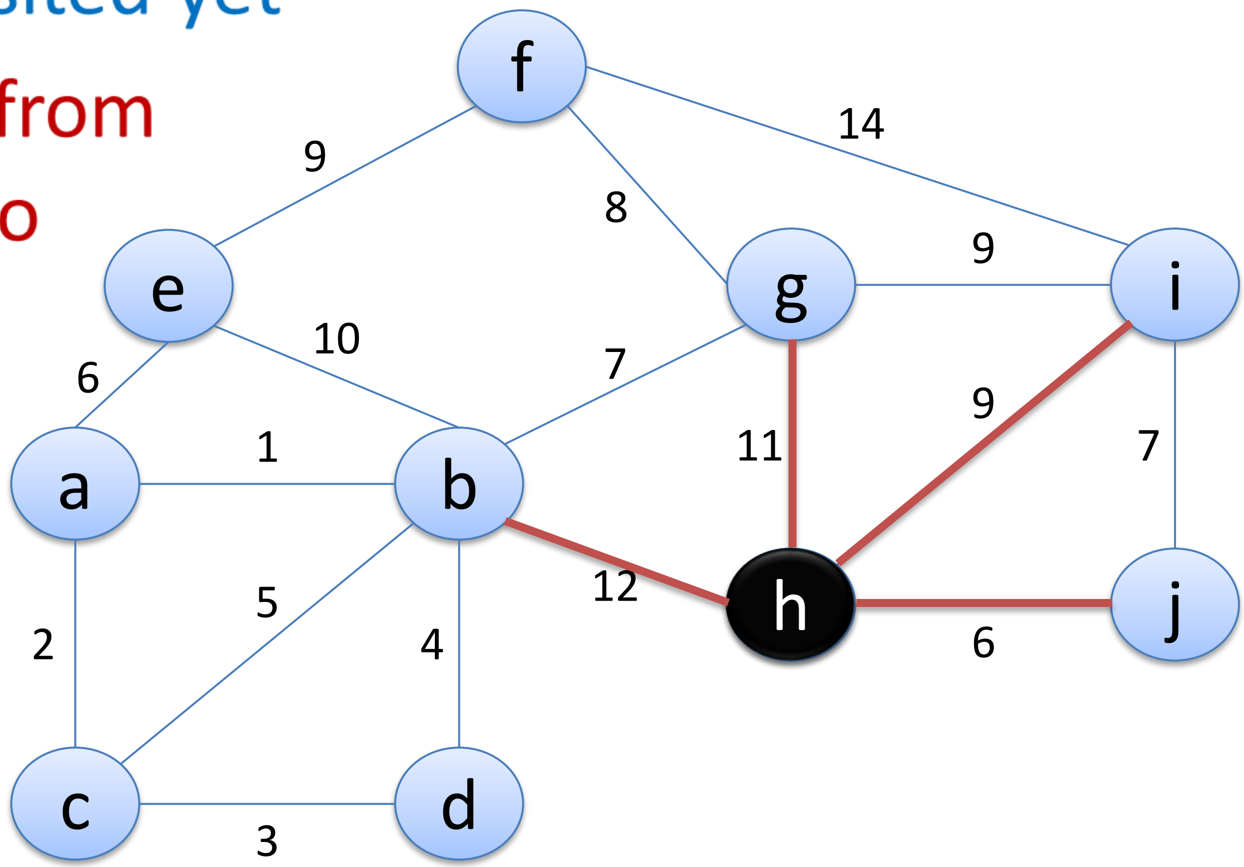
Running Prim's algorithm

- Start at an arbitrary node, say, h.
- **Blue:** not visited yet
- **Red:** edges from nodes $\in T$ to nodes $\notin T$
- **Black:** in T



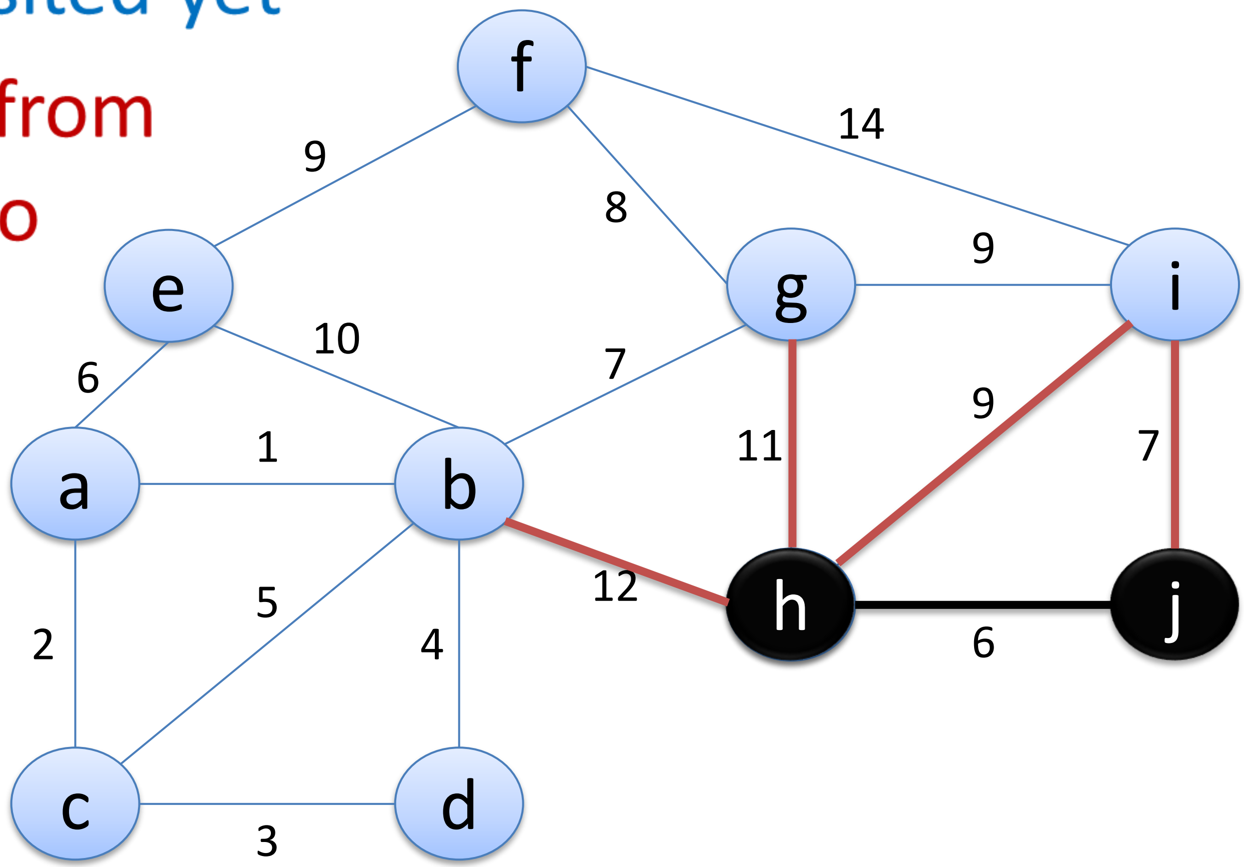
Running Prim's algorithm

- Start at an arbitrary node, say, h.
- **Blue:** not visited yet
- **Red:** edges from nodes $\in T$ to nodes $\notin T$
- **Black:** in T



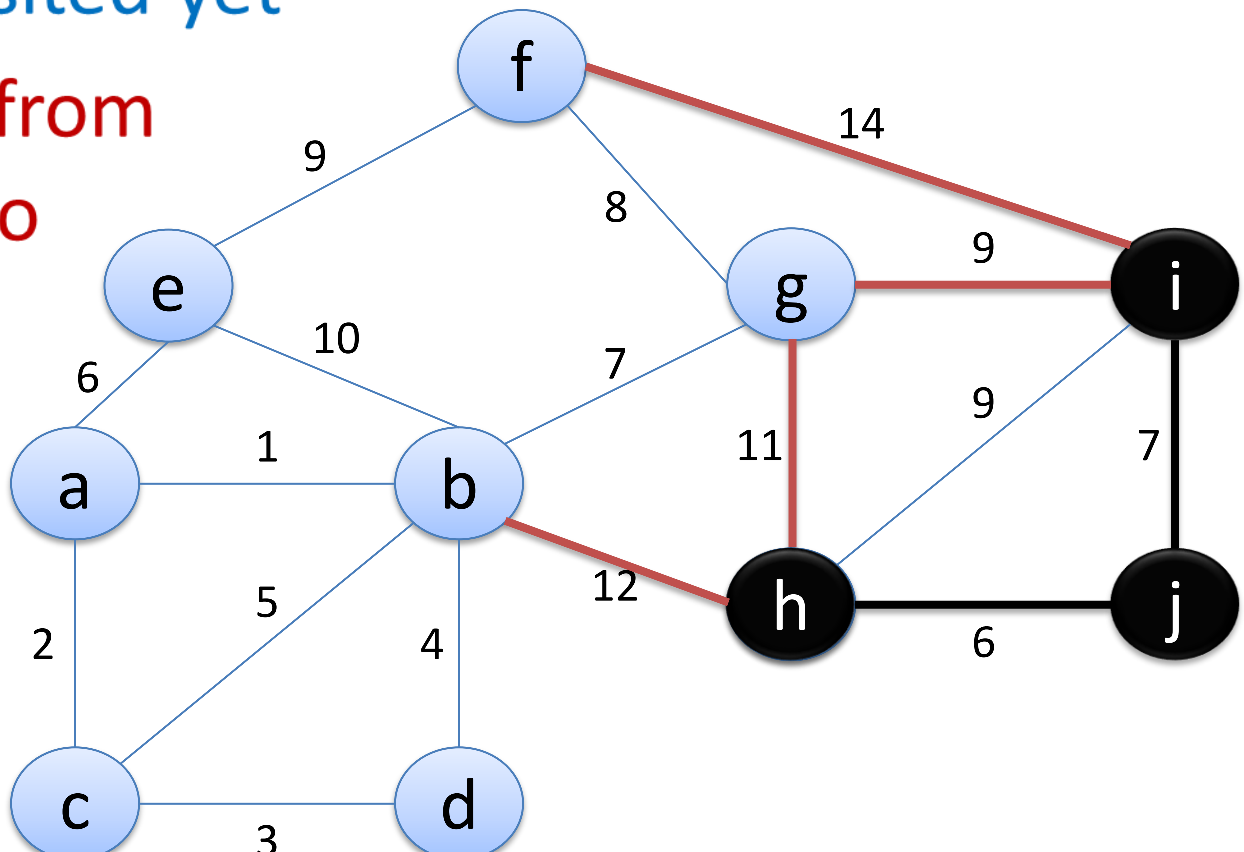
Running Prim's algorithm

- Start at an arbitrary node, say, h.
- **Blue:** not visited yet
- **Red:** edges from nodes $\in T$ to nodes $\notin T$
- **Black:** in T



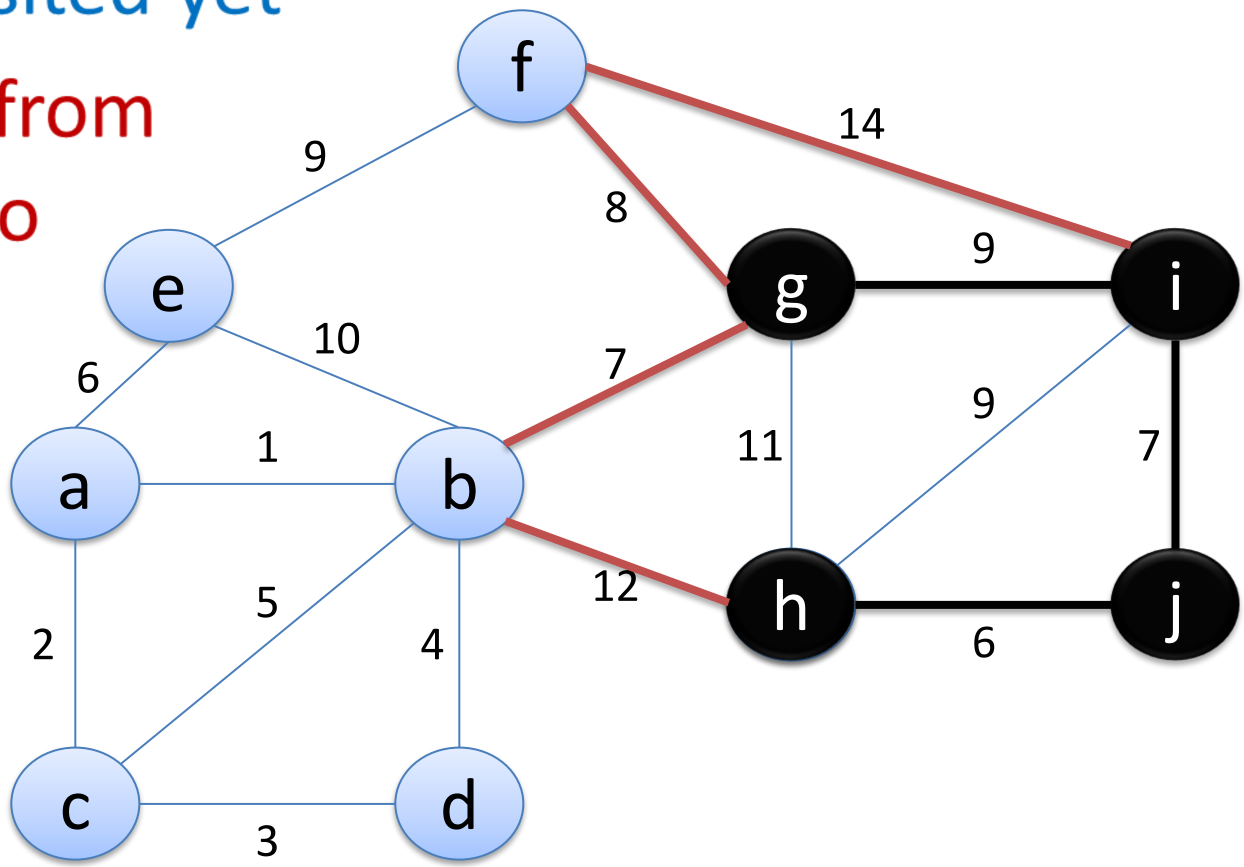
Running Prim’s algorithm

- Start at an arbitrary node, say, h.
- **Blue:** not visited yet
- **Red:** edges from nodes $\in T$ to nodes $\notin T$
- **Black:** in T



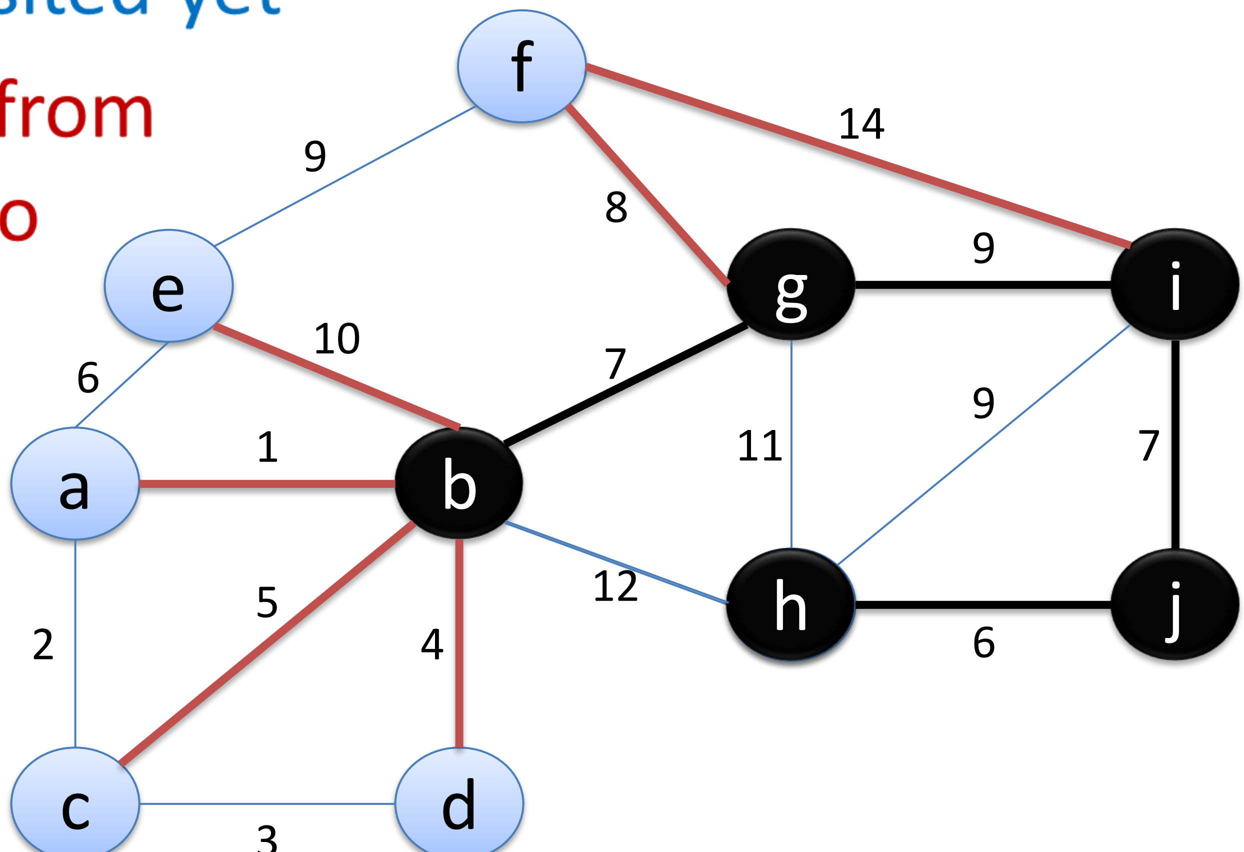
Running Prim's algorithm

- Start at an arbitrary node, say, h.
- **Blue:** not visited yet
- **Red:** edges from nodes $\in T$ to nodes $\notin T$
- **Black:** in T



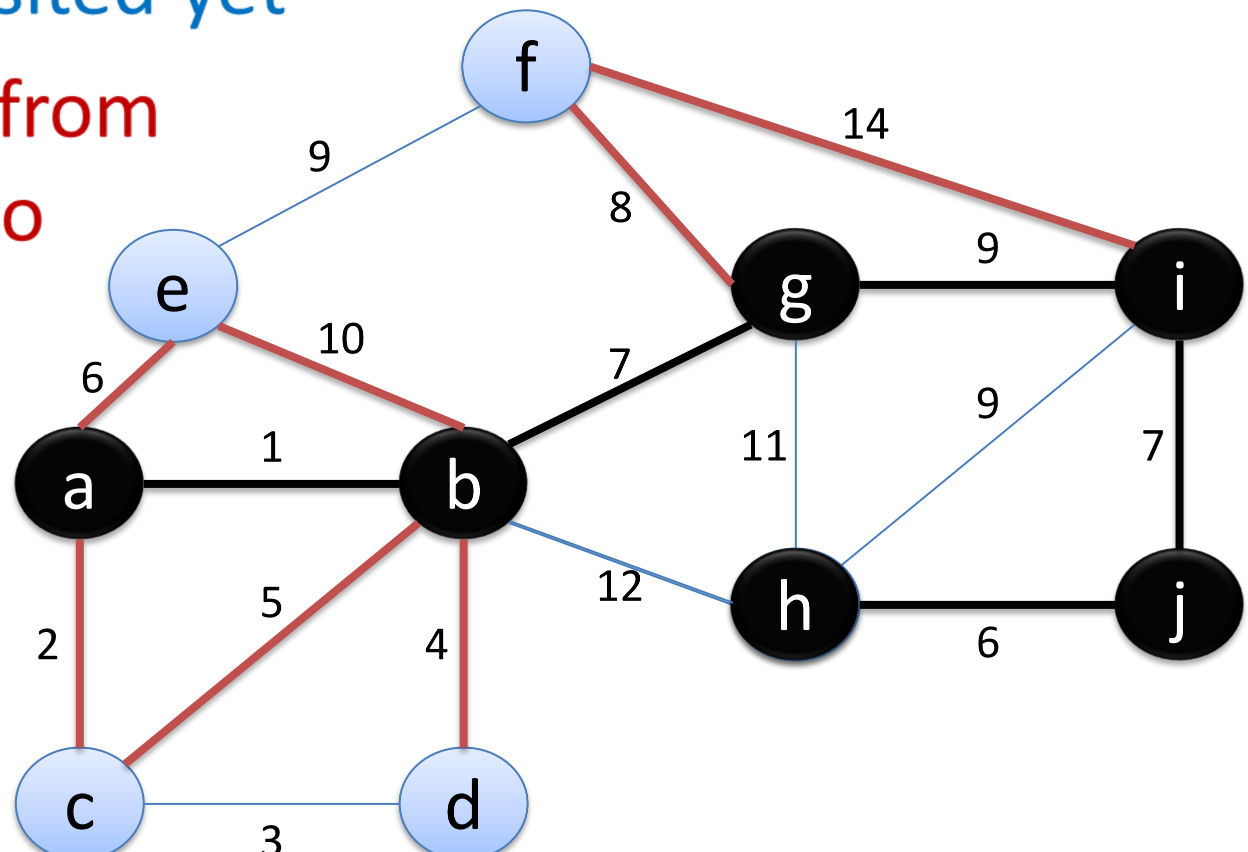
Running Prim's algorithm

- Start at an arbitrary node, say, h.
- **Blue:** not visited yet
- **Red:** edges from nodes $\in T$ to nodes $\notin T$
- **Black:** in T



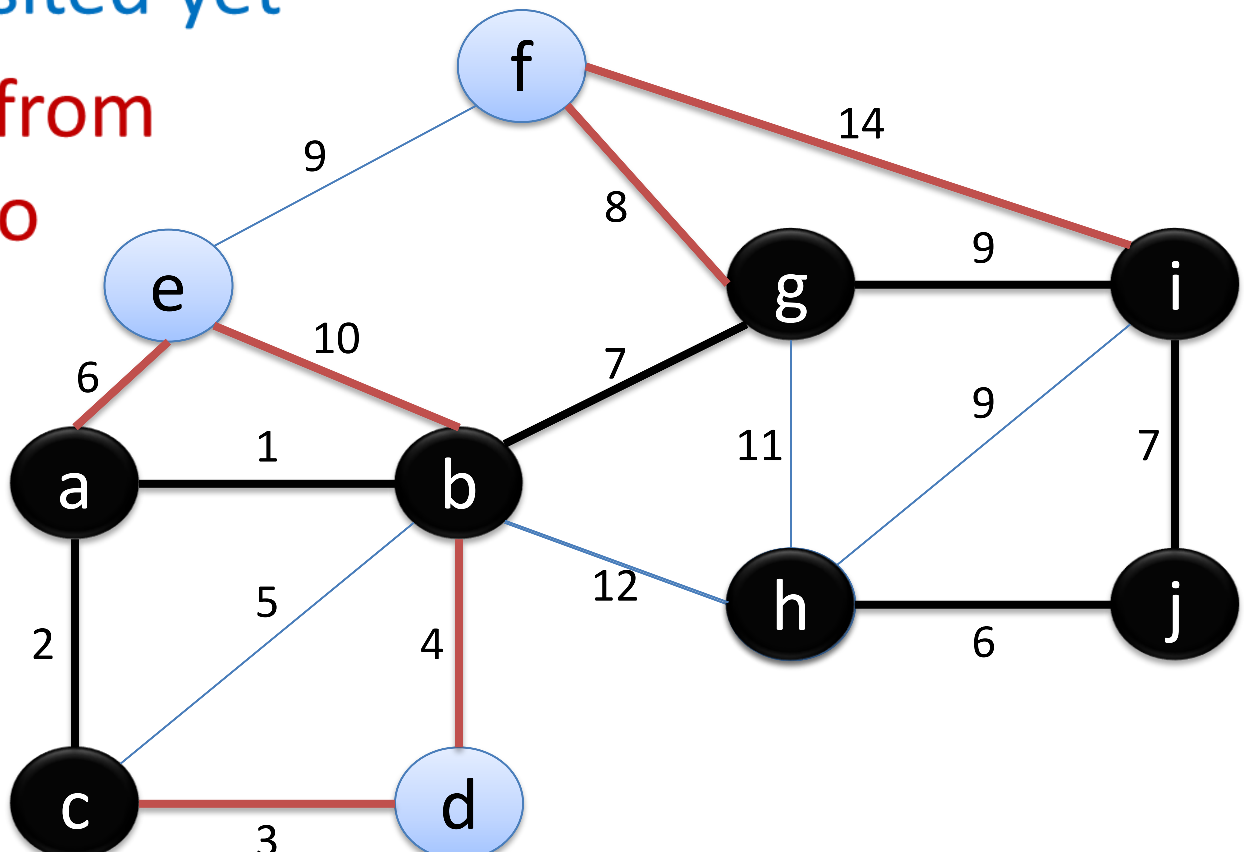
Running Prim's algorithm

- Start at an arbitrary node, say, h.
- **Blue:** not visited yet
- **Red:** edges from nodes $\in T$ to nodes $\notin T$
- **Black:** in T



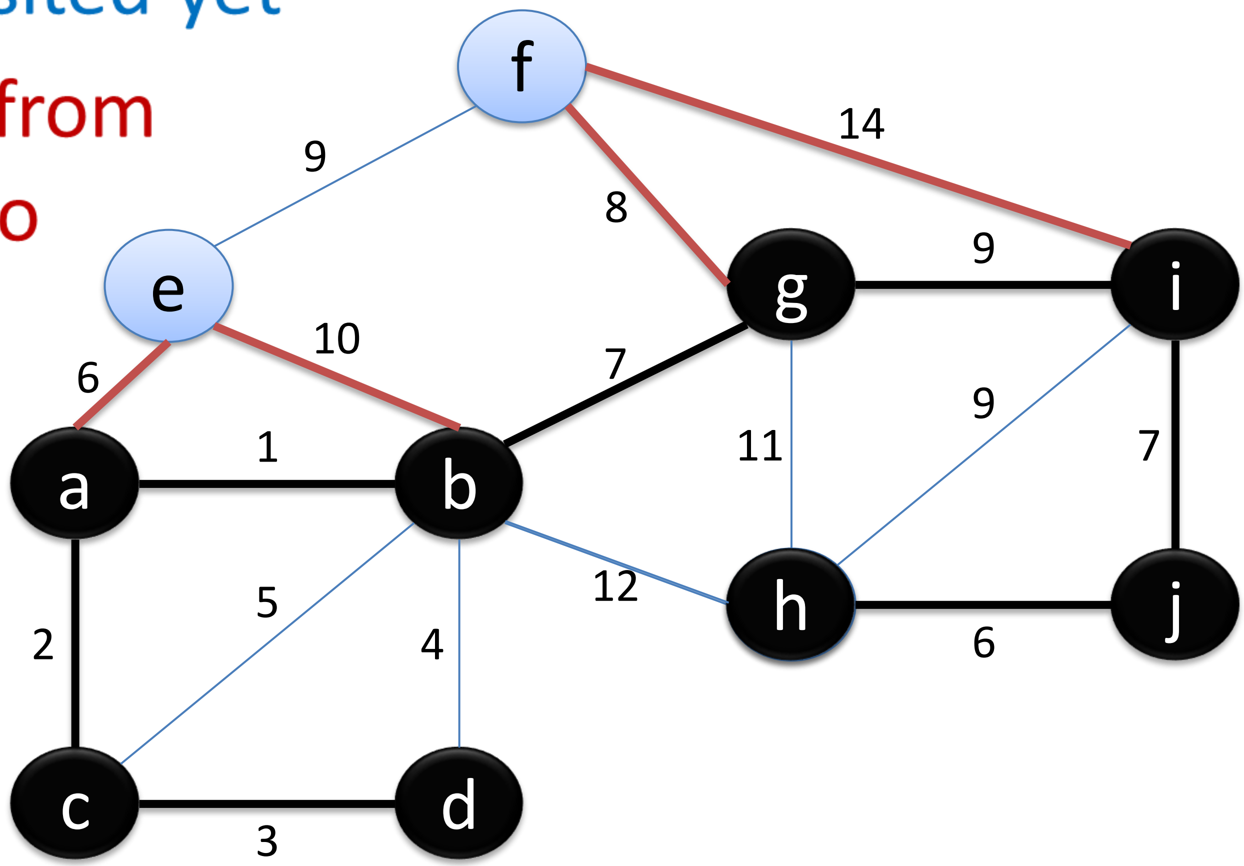
Running Prim's algorithm

- Start at an arbitrary node, say, h.
- **Blue:** not visited yet
- **Red:** edges from nodes $\in T$ to nodes $\notin T$
- **Black:** in T



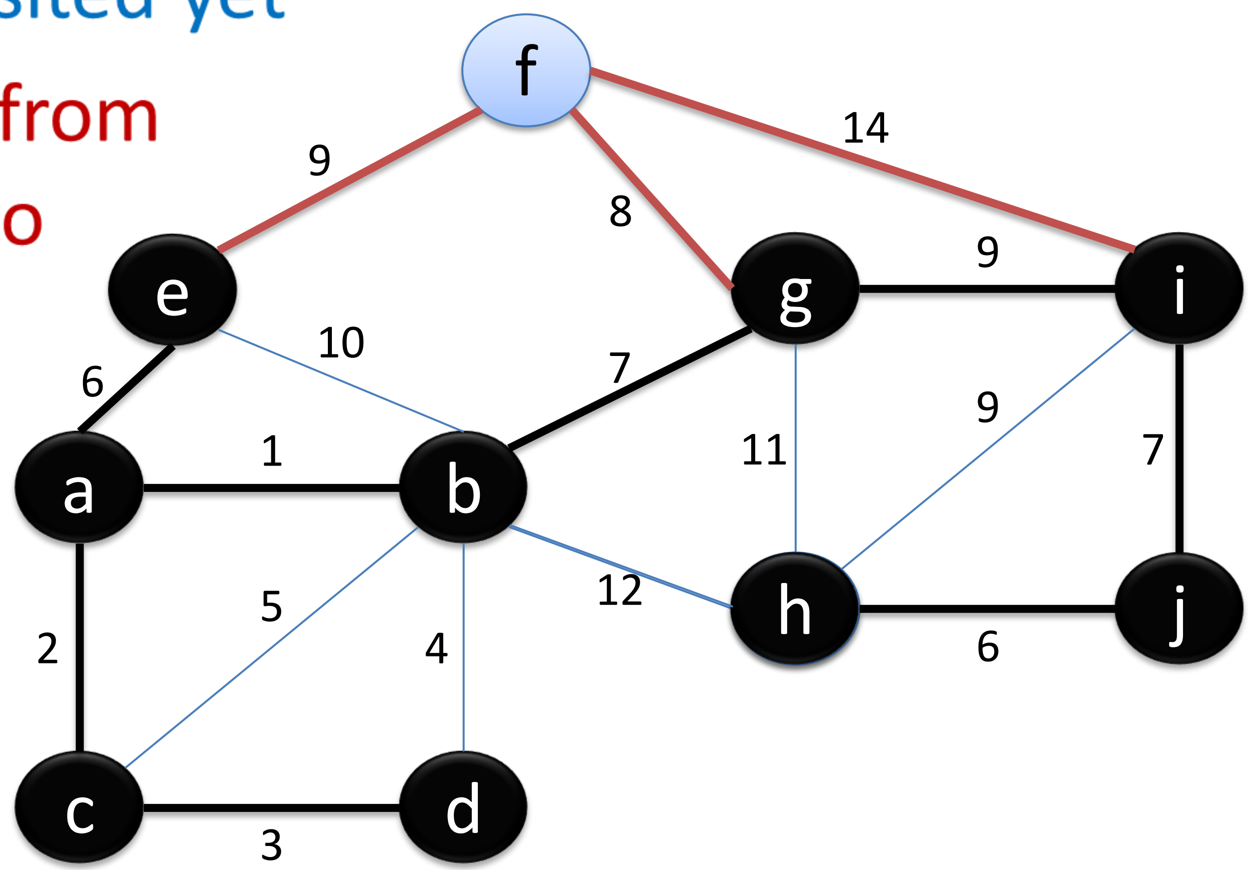
Running Prim's algorithm

- Start at an arbitrary node, say, h.
- **Blue:** not visited yet
- **Red:** edges from nodes $\in T$ to nodes $\notin T$
- **Black:** in T



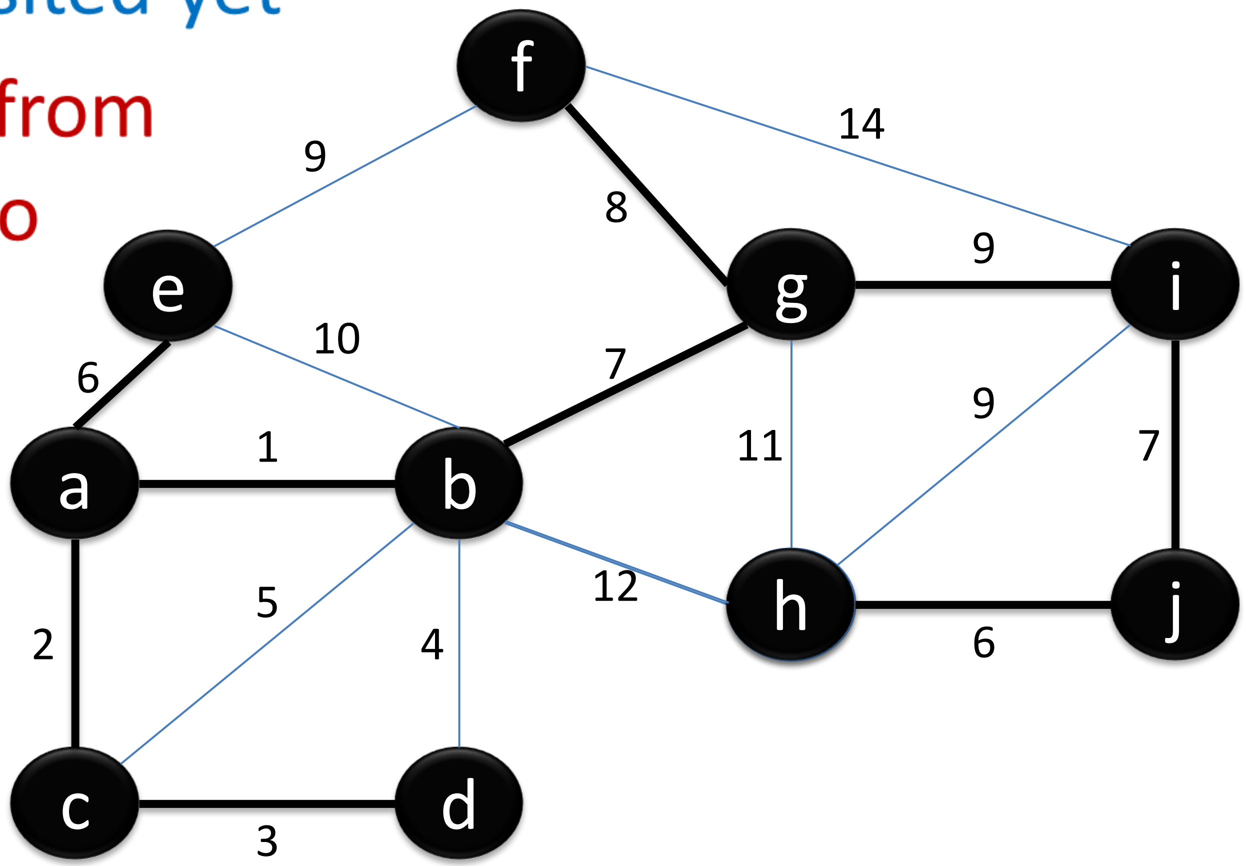
Running Prim's algorithm

- Start at an arbitrary node, say, h.
- **Blue:** not visited yet
- **Red:** edges from nodes $\in T$ to nodes $\notin T$
- **Black:** in T



Running Prim's algorithm

- Start at an arbitrary node, say, h.
- **Blue:** not visited yet
- **Red:** edges from nodes $\in T$ to nodes $\notin T$
- **Black:** in T
- Minimum Cost: 47



Implementing Prim's Algorithm

- Recall the high-level algorithm:

- PrimMCST(V, E)
  - Pick an arbitrary node **r** from V
  - Add **r** to *T*
  - While *T* contains < |V| nodes
    - Find a **minimum weight edge (u, v)** where $\mathbf{u} \in \boldsymbol{T}$ and $\mathbf{v} \notin \boldsymbol{T}$
    - Add node v to *T*

How can we do this **efficiently**?

Finding lots of minimums?
Use a priority queue!

Adding a priority queue

- What should we store in the priority queue?
  - Edges
  - From nodes in T to nodes not in T
- What should we use as the key of an edge?
  - Weight of the edge

PrimMCST(V, E)

- Pick an arbitrary node **r** from V
- Add **r** to *T*
- While *T* contains < |V| nodes
  - Find a **minimum weight edge (u, v)** where $\mathbf{u} \in \boldsymbol{T}$ and $\mathbf{v} \notin \boldsymbol{T}$
  - Add node v to *T*

Prim’s Algorithm with a priority queue

- PrimMCST(V, E, r) where r is any arbitrary starting node
  - Q := new priority queue
  - For each u in V: inTree[u] = false, parent[u] = nil
  - inTree[r] = true, parent[r] = r
  - Add every edge that touches r to Q
  - While Q is not empty
    - Do Q.Extract-Min to get edge e = (u, v)
    - If not inTree[v] then
      - inTree[v] = true, parent[v] = u
      - Add every edge that touches v to Q

Small optimization

- PrimMCST(V, E, r)
  - Q := new priority queue
  - For each u in V: ~~inTree[u] = false,~~ parent[u] = nil

  ~~– inTree[r] = true,~~ parent[r] = r
  - Add every edge that touches r to Q

  - While Q is not empty
    - Do Q.Extract-Min to get edge e = (u, v)
    - If ~~not inTree[v]~~ **parent[v] = nil** then
      - ~~inTree[v] = true,~~ parent[v] = u
      - Add every edge that touches v to Q

Analysis of running time

PrimMCST(V, E, r)

$\Theta(|V|)$
- Q := new priority queue
- For each u in V: inTree[u] = false

$\Theta(|adj(r)| \log |E|)$
- inTree[r] = true, parent[r] = nil
- Add every edge that touches r to Q

$\Theta(|E| \log |E|)$
- While Q is not empty
  - $\Theta(\log |E|)$
    - Do Q.Extract-Min to get edge e = (u, v)
  - $\Theta(|adj(v)| \log |E|)$
    - If not inTree[v] then
      - inTree[v] = true, parent[v] = u
      - Add every edge that touches v to Q

- $O(|E| \log |E|) = O(|E| \log (|V|^2))$
- $= O(|E|\ 2 \log |V|)$
- $= O(|E| \log |V|)$

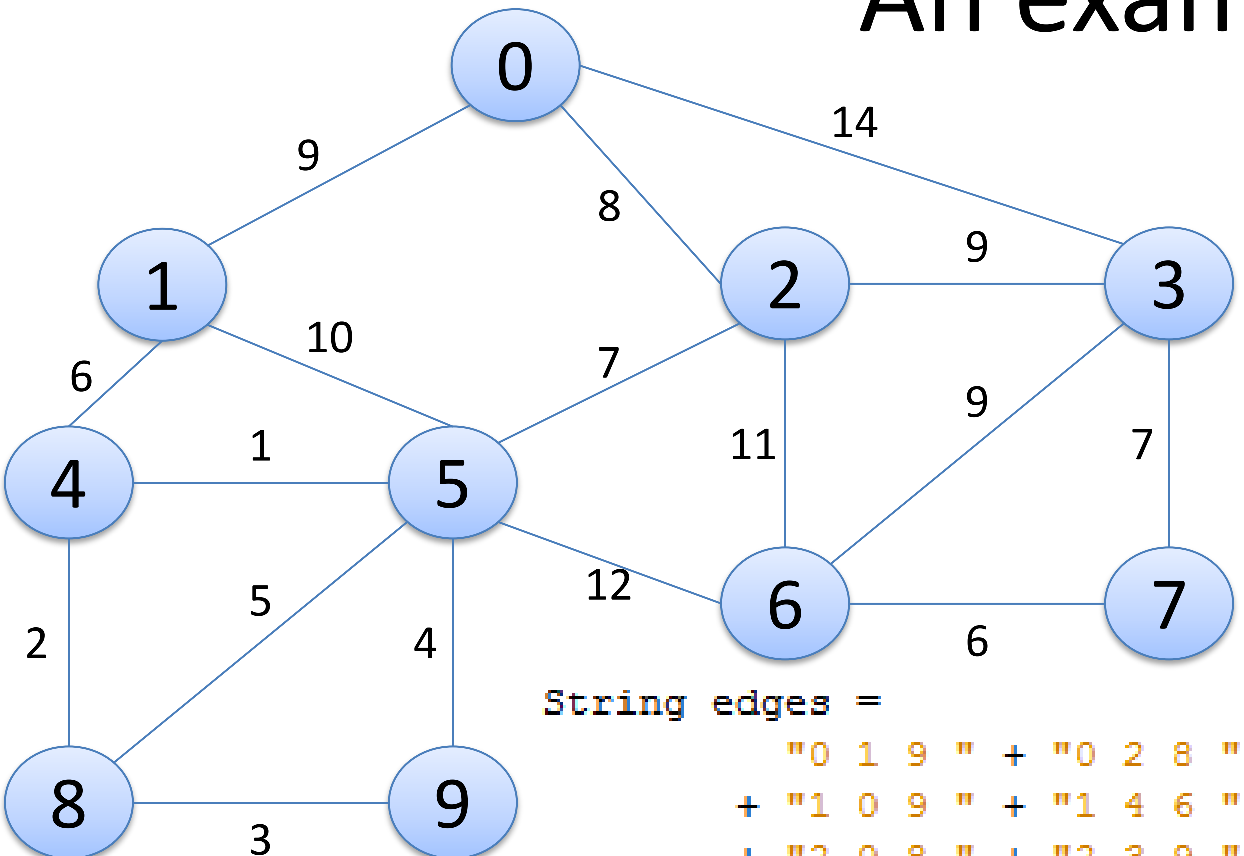
Java Implementation - 1

```
static int[] prim(int n, ArrayList<ArrayList<Edge>> adj, int start) {
    TreeSet<Edge> q = new TreeSet<>();
    int[] parent = new int[n];
    for (int i=0;i<parent.length;++i) parent[i] = -1;

    parent[start] = start;
    for (int i=0;i<adj.get(start).size();++i) {
        q.add(adj.get(start).get(i));
    }

    while (!q.isEmpty()) {
        Edge e = q.pollFirst();
        if (parent[e.b] == -1) {
            parent[e.b] = e.a;
            for (int i=0;i<adj.get(e.b).size();++i) {
                q.add(adj.get(e.b).get(i));
            }
        }
    }
    return parent;
}
```

Java Implementation - 2

```java
    static class Edge implements Comparable<Edge> {
        int a, b, w;
        Edge(final int a, final int b, final int w) {
            this.a = a; this.b = b; this.w = w;
        }
        public int compareTo(Edge o) {
            if (w < o.w) return -1;
            if (w > o.w) return 1;
            if (a < o.a) return -1;
            if (a > o.a) return 1;
            if (b < o.b) return -1;
            if (b > o.b) return 1;
            return 0;
        }
        public String toString() {
            return "(" + a + ", " + b + ", " + w + ")";
        }
    }
```

An example input




```
String edges =
    "0 1 9 " + "0 2 8 " + "0 3 14 "
  + "1 0 9 " + "1 4 6 " + "1 5 10 "
  + "2 0 8 " + "2 3 9 " + "2 5 7 " + "2 6 11 "
  + "3 0 14 " + "3 2 9 " + "3 6 9 " + "3 7 7 "
  + "4 1 6 " + "4 5 1 " + "4 8 2 "
  + "5 1 10 " + "5 2 7 " + "5 4 1 " + "5 6 12 "
              + "5 8 5 " + "5 9 4 "
  + "6 2 11 " + "6 3 9 " + "6 5 12 " + "6 7 6 "
  + "7 3 7 " + "7 6 6 "
  + "8 4 2 " + "8 5 5 " + "8 9 3 "
  + "9 5 4 " + "9 8 3";
```

Java Implementation - 3

```
    public static void main(String[] args) {
        int n = 10;
        String edges =
                "0 1 9 " + "0 2 8 " + "0 3 14 "
              + "1 0 9 " + "1 4 6 " + "1 5 10 "
              + "2 0 8 " + "2 3 9 " + "2 5 7 " + "2 6 11 "
              + "3 0 14 " + "3 2 9 " + "3 6 9 " + "3 7 7 "
              + "4 1 6 " + "4 5 1 " + "4 8 2 "
              + "5 1 10 " + "5 2 7 " + "5 4 1 " + "5 6 12 "
                          + "5 8 5 " + "5 9 4 "
              + "6 2 11 " + "6 3 9 " + "6 5 12 " + "6 7 6 "
              + "7 3 7 " + "7 6 6 "
              + "8 4 2 " + "8 5 5 " + "8 9 3 "
              + "9 5 4 " + "9 8 3";
        ArrayList<ArrayList<Edge>> adj = new ArrayList<>();
        for (int i=0;i<n;++i) adj.add(new ArrayList<Edge>());
        Scanner in = new Scanner(edges);
        while (in.hasNext()) {
            int a = in.nextInt();
            int b = in.nextInt();
            int w = in.nextInt();
            adj.get(a).add(new Edge(a, b, w));
        }
        int[] tree = prim(n, adj, 6);
```

Java Implementation - 4

- Outputting the answer:

```
        int[] tree = prim(n, adj, 6);
        for (int i=0;i<tree.length;++i) {
            System.out.print((i>0?", ":"") + "(" + i + ", " + tree[i] + ")");
        }
        System.out.println();
    }
```

- The answer:

```
run:
(0, 2), (1, 4), (2, 3), (3, 7), (4, 5), (5, 2), (6, 6), (7, 6), (8, 4), (9, 8)
BUILD SUCCESSFUL (total time: 0 seconds)
```

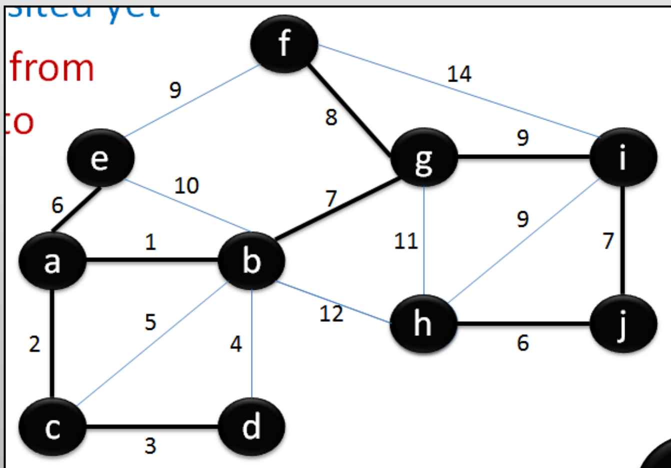
Recall: the root is its own parent.

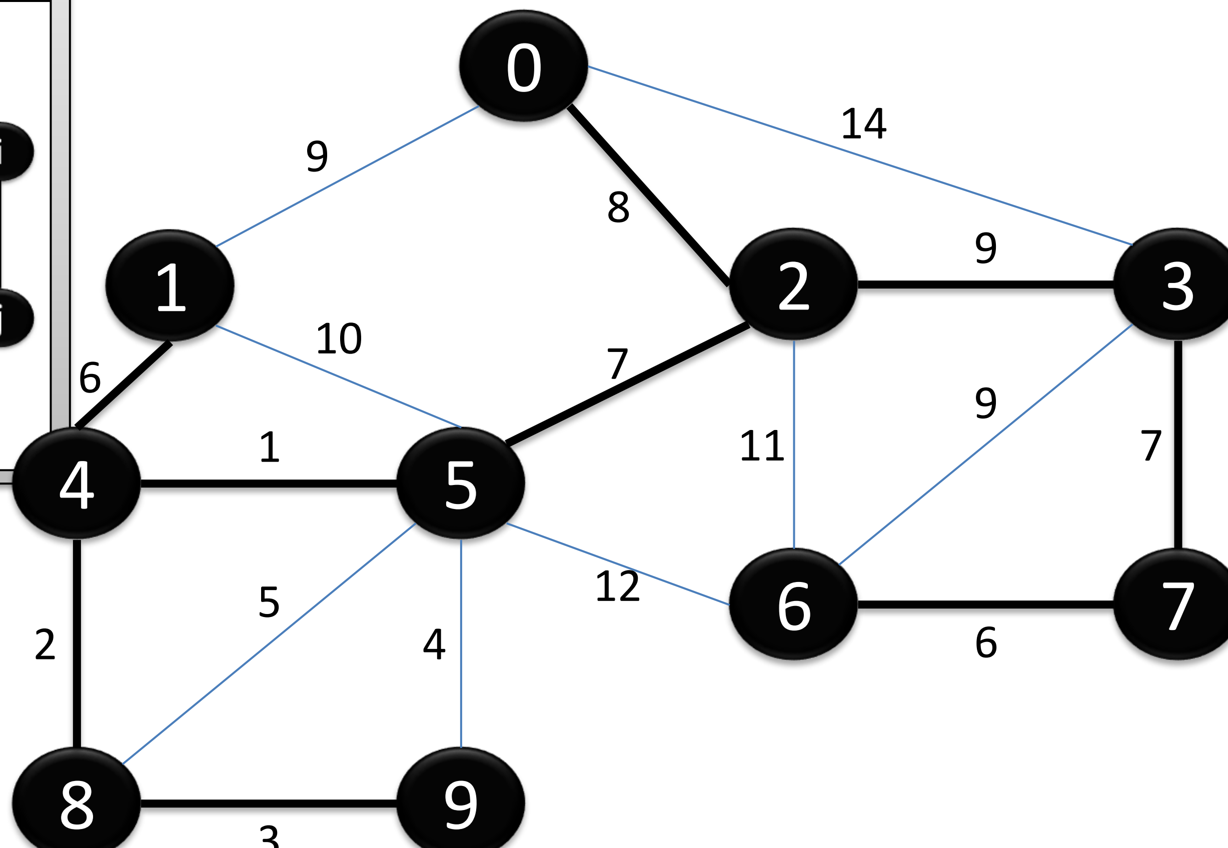
- What does this look like?

Drawing the answer

```
(0, 2), (1, 4), (2, 3), (3, 7), (4, 5), (5, 2), (6, 6), (7, 6), (8, 4), (9, 8)
```

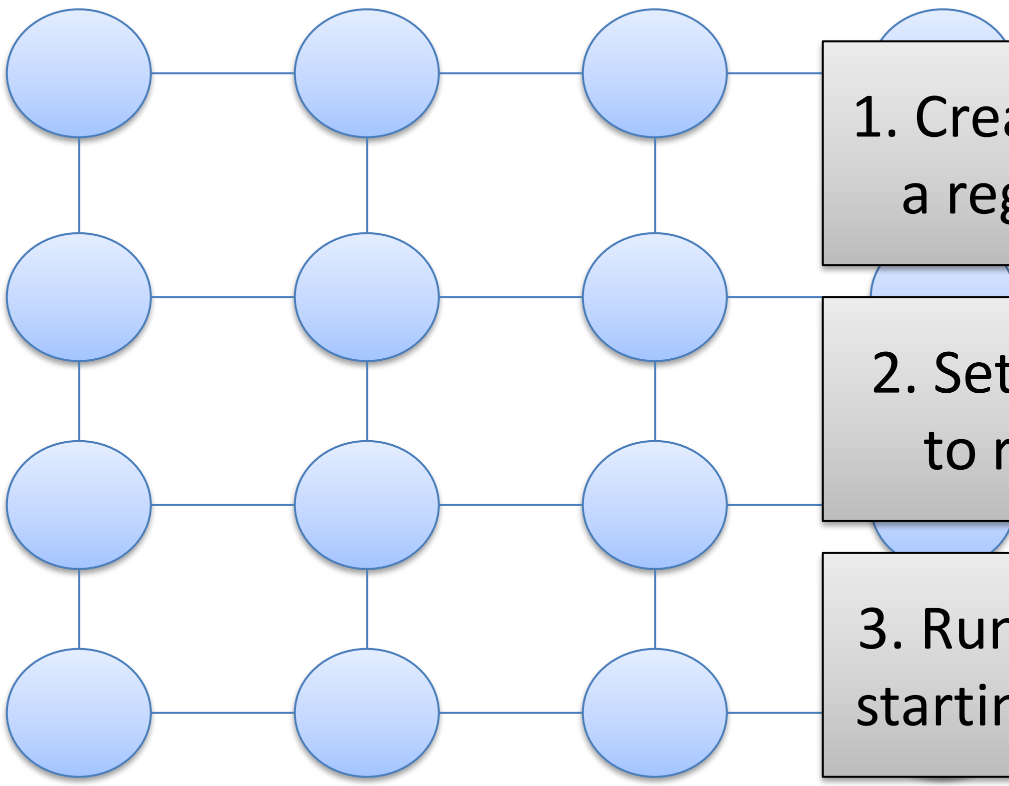
Recall our earlier solution by hand:





Fun example: generating 2D mazes

- Prim's algorithm maze building video
- How can we use Prim's algorithm to do this?



1. Create a graph that is a regular m x n grid.

2. Set all edge weights to random values!

3. Run Prim's algorithm starting from any node.

Fun example: generating 2D mazes

- After Prim's, we end up with something like: