

# COMP 322: Parallel and Concurrent Programming

## Lecture 30: Parallel Graph Algorithms

Mack Joyner  
mjoyner@rice.edu

<http://comp322.rice.edu>

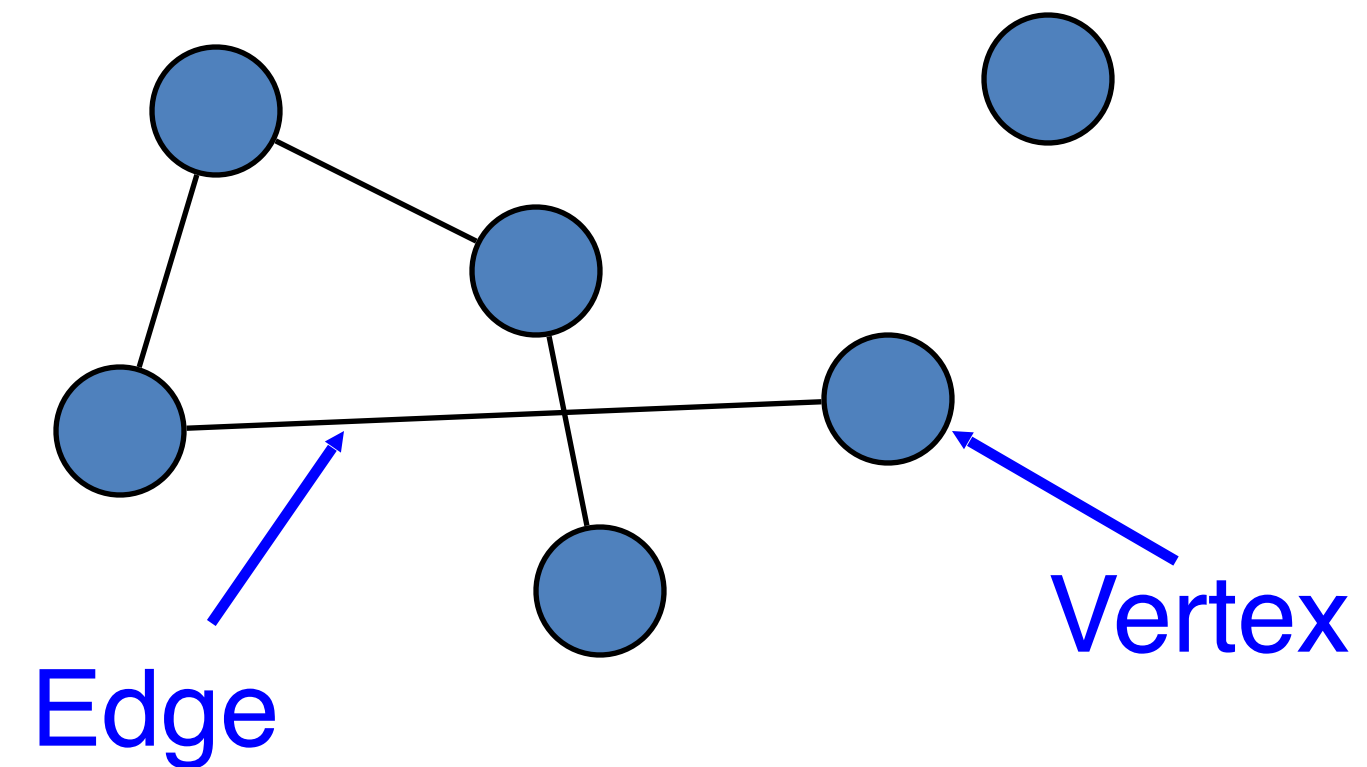
Some slides in this presentation are adopted from Aydin Buluç: “Parallel Graph Algorithms”, LBNL, CS267, Spring 2016, Hall Perkins, “Data Structures”, CSE 374, University of Washington



# Graphs

Graph  $G = (V, E)$

- a set of **vertices** and a set of **edges** between vertices



$n=|V|$  (number of vertices)

$m=|E|$  (number of edges)

$D$ =diameter (max #hops between any pair of vertices)


- Edges can be directed or undirected, weighted or not.
- They can even have attributes (i.e. semantic graphs)
- Sequences of edges  $\langle u_1, u_2 \rangle, \langle u_2, u_3 \rangle, \dots, \langle u_{n-1}, u_n \rangle$  is a **path** from  $u_1$  to  $u_n$ . Its **length** is the sum of its weights.









# Routing in transportation networks

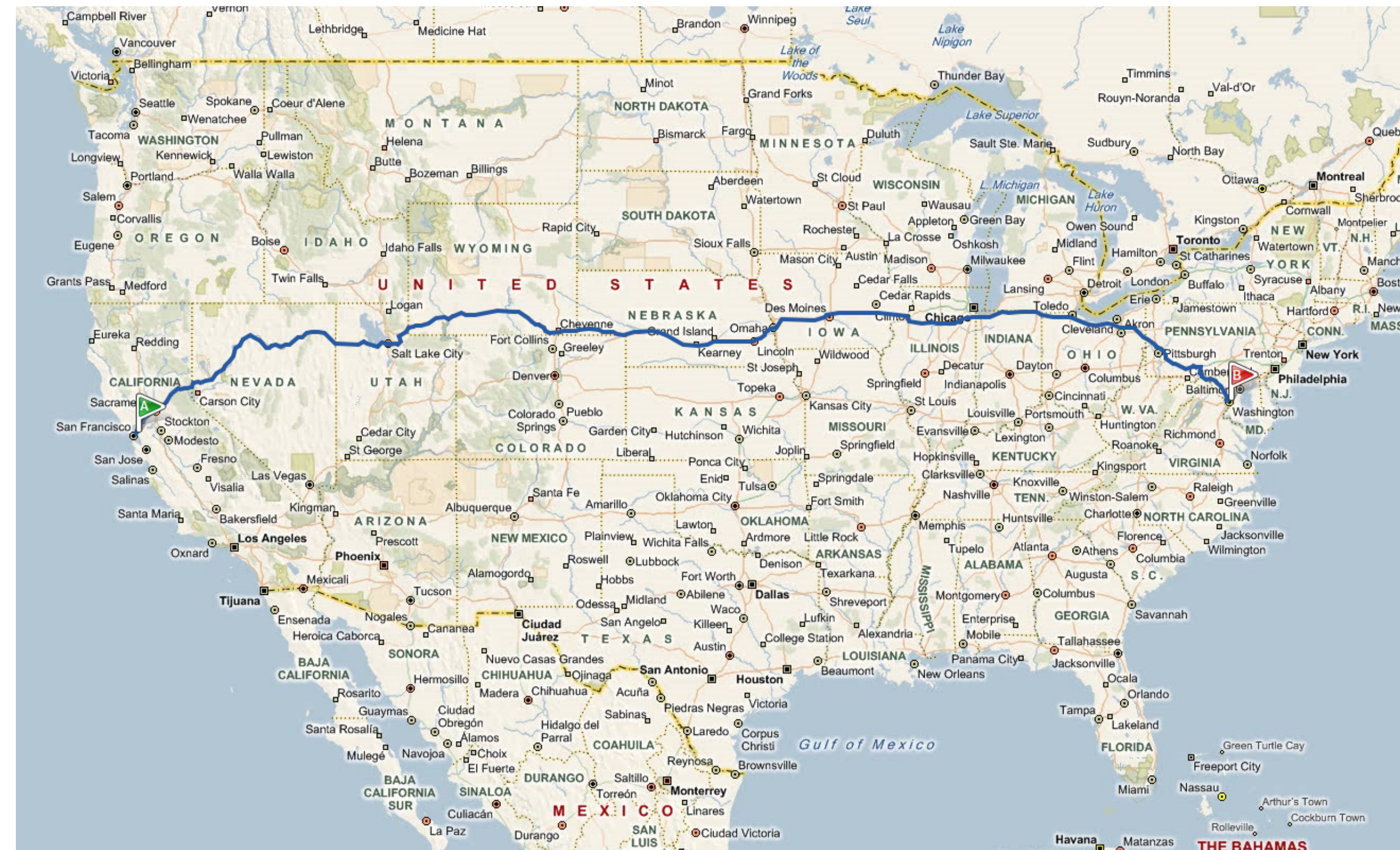
## Driving Directions

To: Washington, D.C.

 **Berkeley, CA**  
[Edit or drag the route](#) • [Save this location](#)

↗ **A-B: 2809.3 miles, 40 hr 10 min** + [Add to route](#)

- 1 Depart Milvia St 0.2 miles
- 2 Turn left onto University Ave 1.8 miles  
*Pass 76 in 0.6 mi*
- 3 Take ramp right for I-80 West / I-580 East / Eastshore Fwy toward Richmond / Sacramento 1.3 miles
- 4 Keep left to stay on I-80 East / Eastshore Fwy 69.3 miles  
 Stop for toll booth
- 5 Take ramp right for I-80 East toward Airport / Reno 651.8 miles  
 Entering Nevada  
 Entering Utah
- 6 Take ramp for I-15 South / I-80 East toward Las Vegas / Cheyenne 2.8 miles
- 7 At exit 304, take ramp right for I-80 East toward Cheyenne 935.0 miles  
 Entering Wyoming  
 Entering Nebraska  
 Entering Iowa



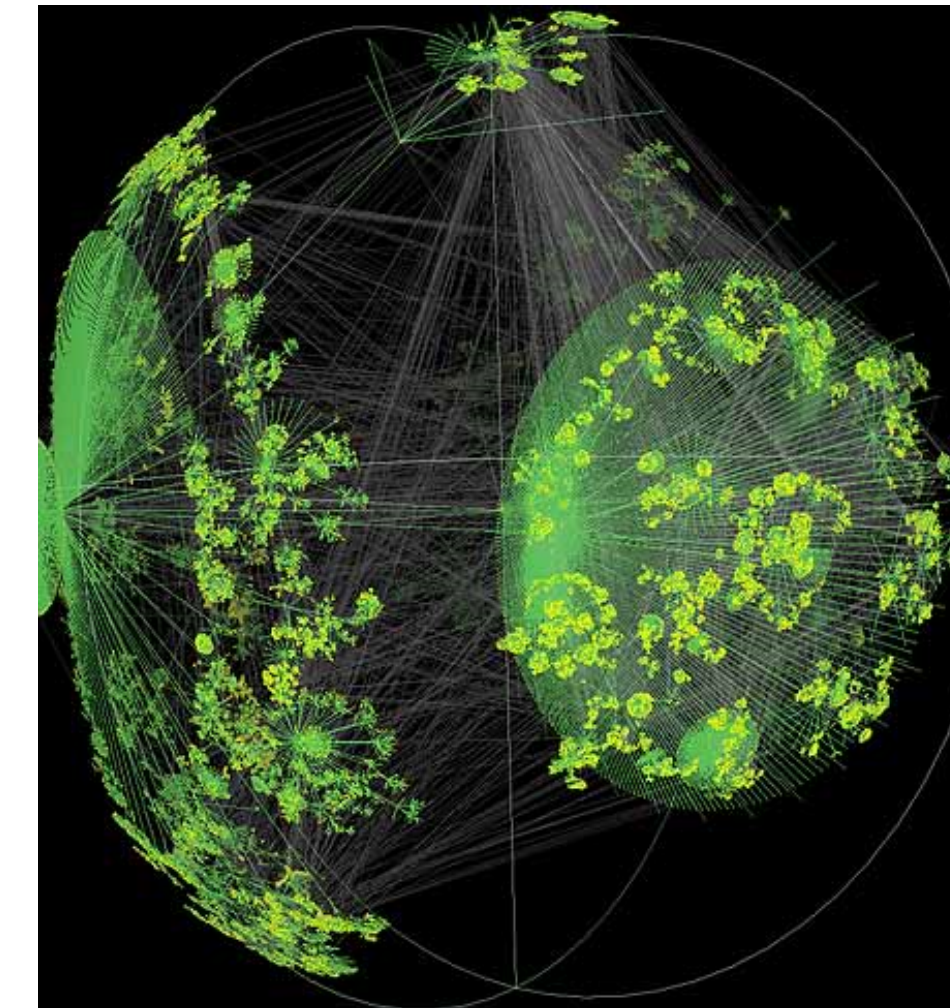
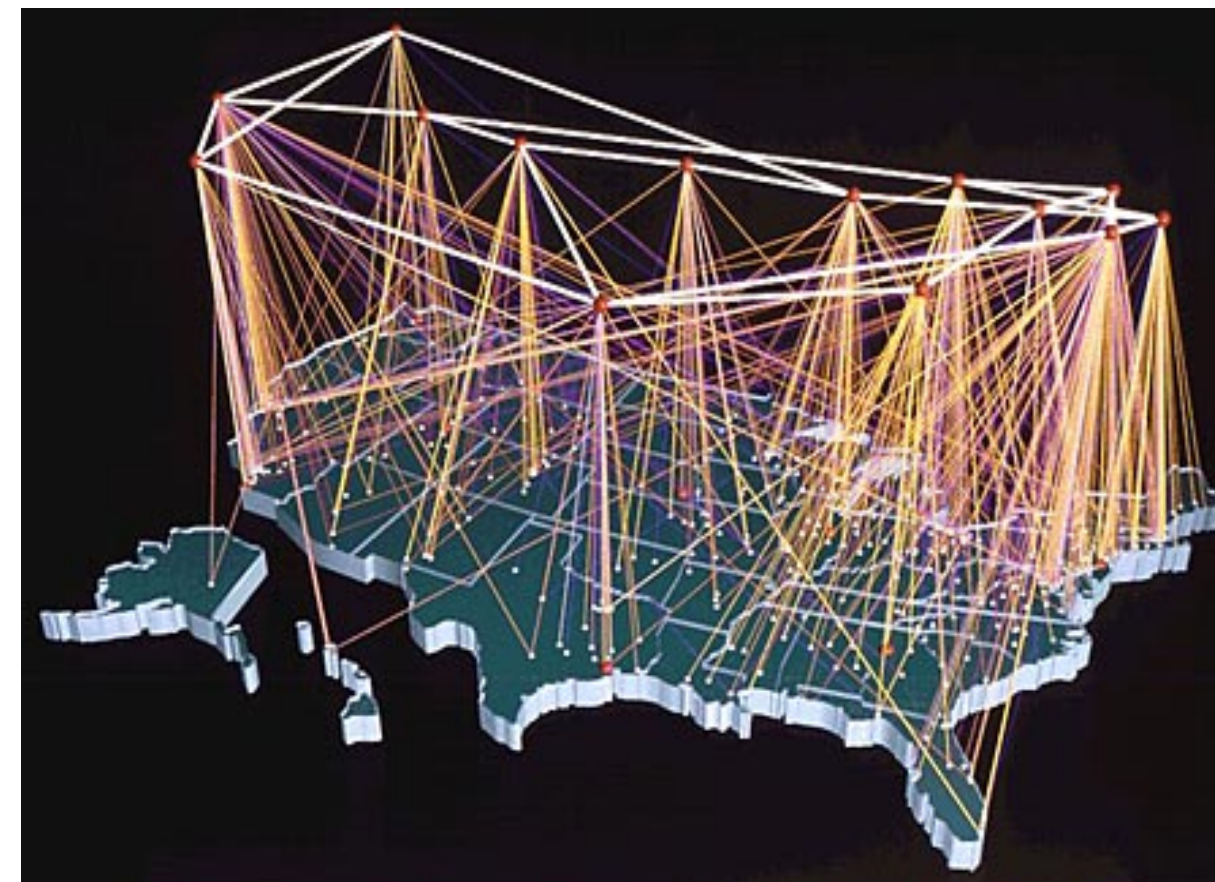
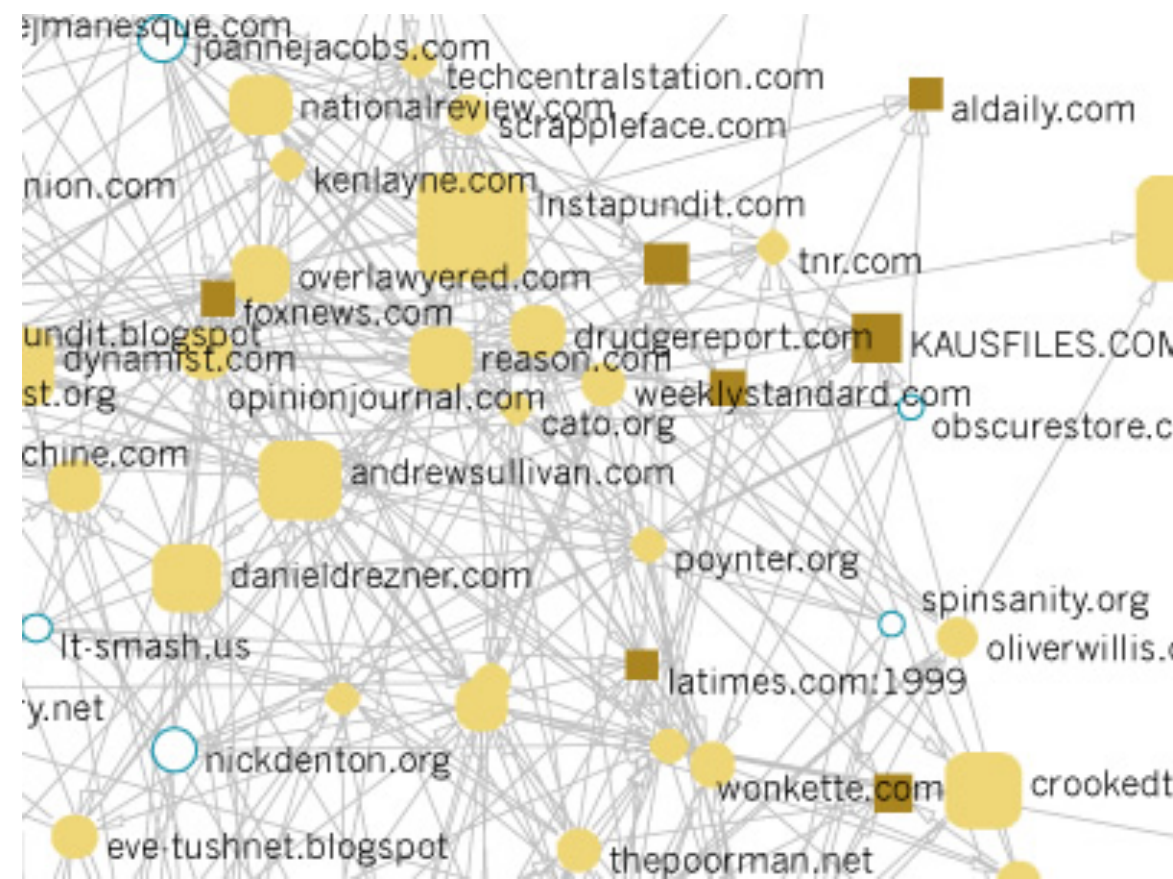
Road networks, Point-to-point **shortest paths: 15 seconds** (naïve) → **10 microseconds**

H. Bast et al., "Fast Routing in Road Networks with Transit Nodes", Science 27, 2007.

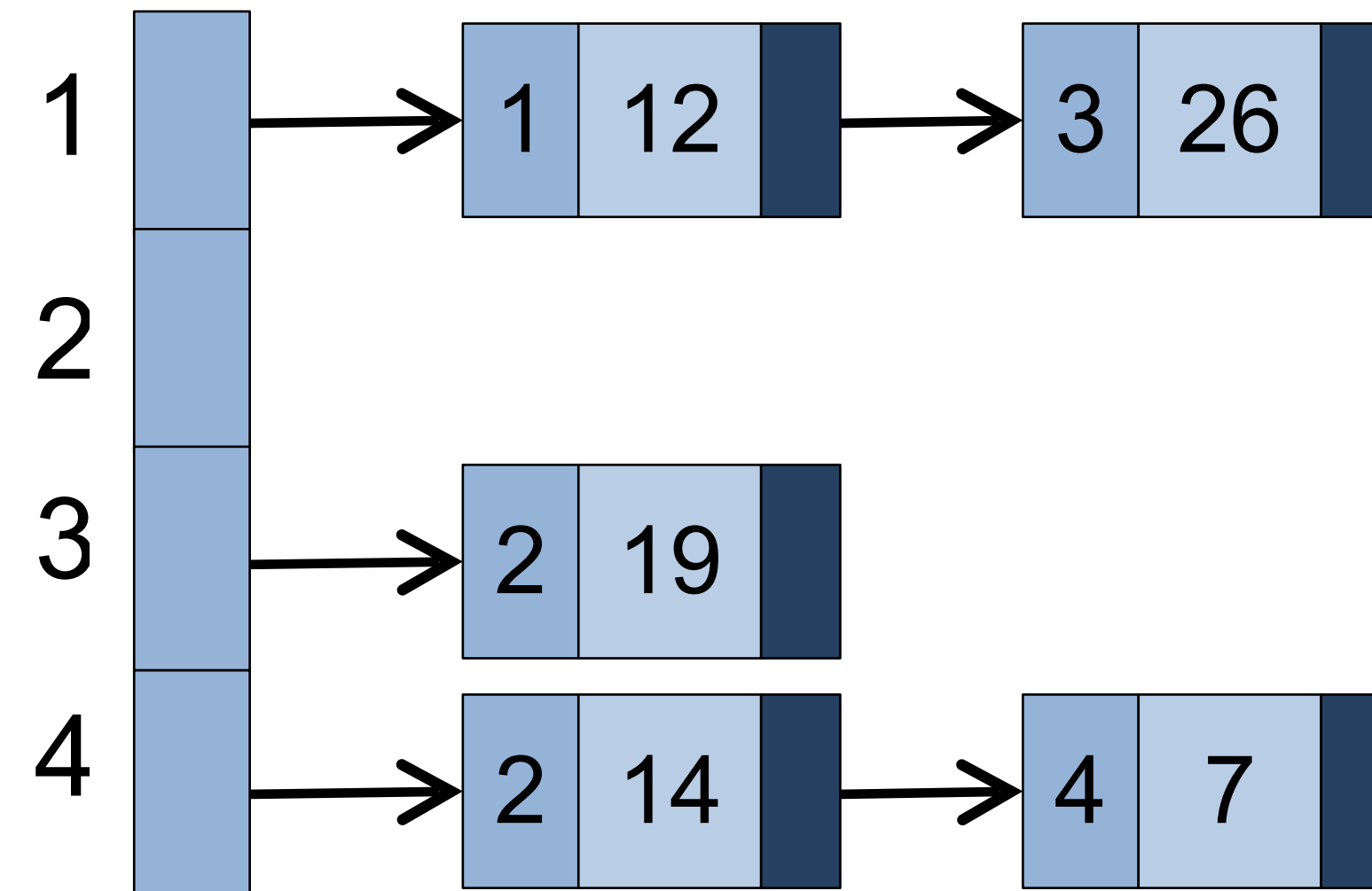
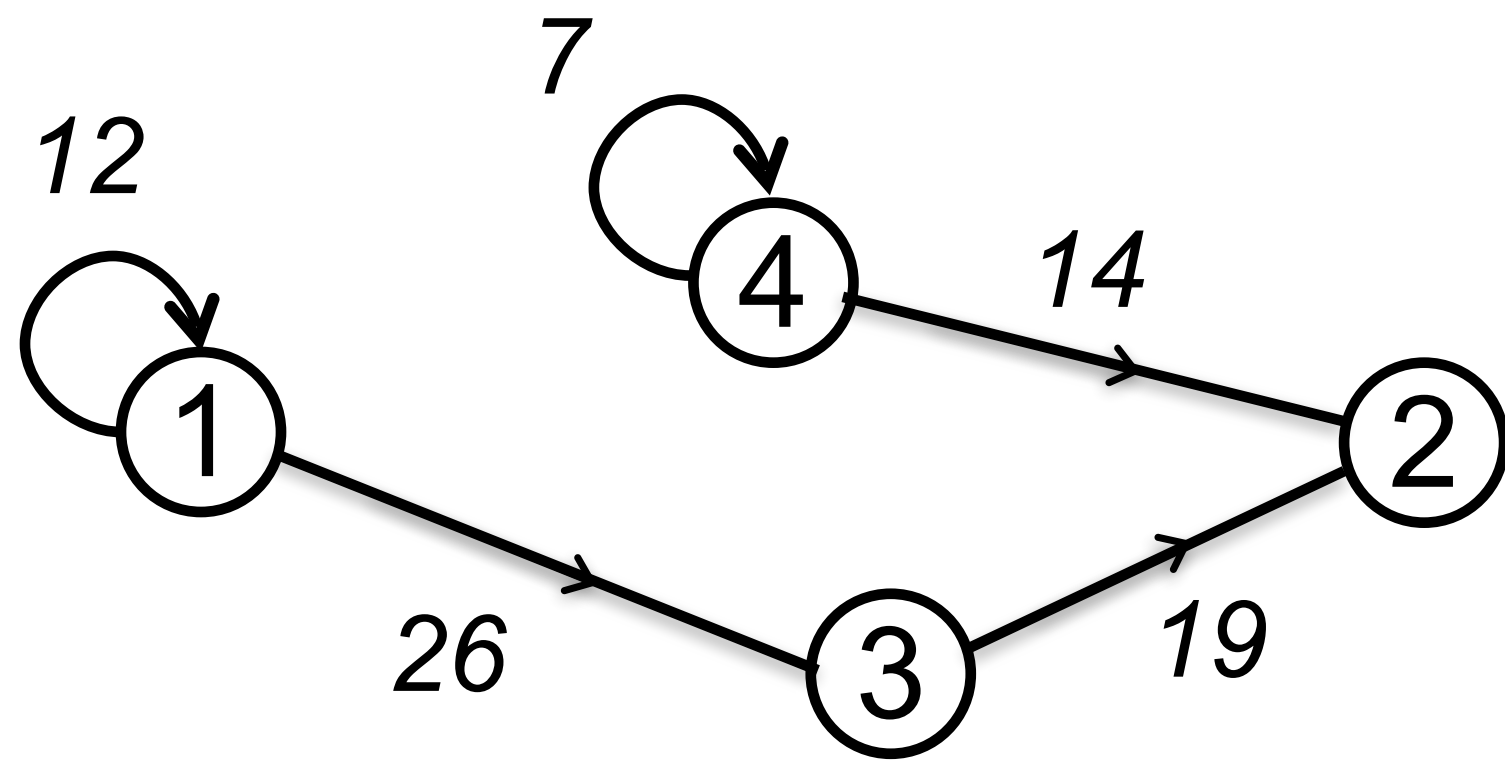


# Internet and the WWW

- The world-wide web can be represented as a directed graph
  - Web search and crawl: **traversal**
  - Link analysis, ranking: **Page rank and HITS**
  - Document classification and **clustering**
- Internet topologies (router networks) are naturally modeled as graphs



# Adjacency List graph representation



# Graph Algorithms

---

- Traversals
  - DFS, BFS
- Finding paths
  - Single-source shortest paths (Dijkstra, Bellman-Ford)
  - All-pairs shortest-paths (Floyd-Warshall)
- Maximal independent sets
- Decomposition (connected components, strongly connected components)
- Maximum cardinality matching
- Connecting
  - Minimum spanning tree



# Spanning Tree Definition

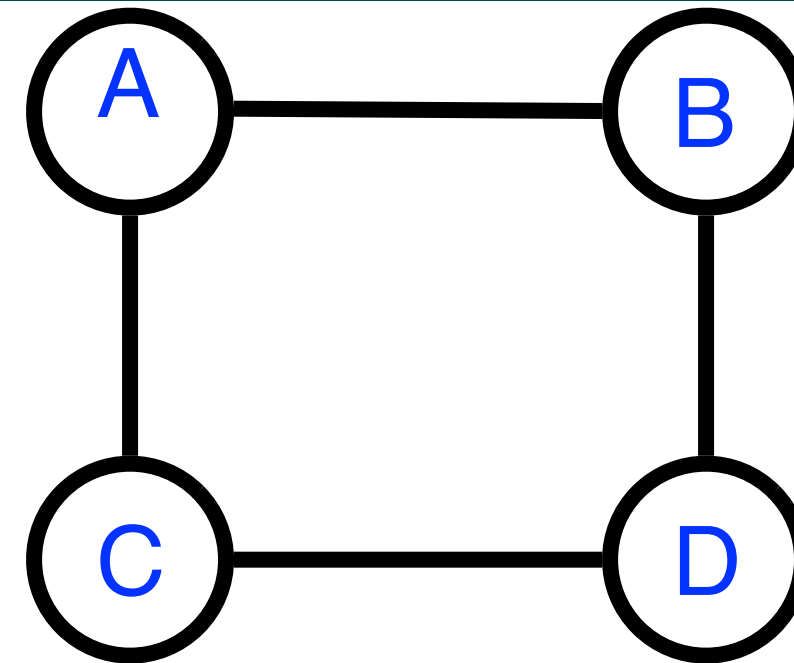
---

- A spanning tree,  $T$ , of a connected undirected graph  $G$  is
  - rooted at some vertex of  $G$
  - defined by a parent map for each vertex
  - contains all the vertices of  $G$ , i.e. spans all vertices
  - contains exactly  $|V| - 1$  edges
    - adding any other edge will create a cycle
  - contains no cycles (a tree!)
- The edges involved in  $T$  are a subset of the edges in  $G$

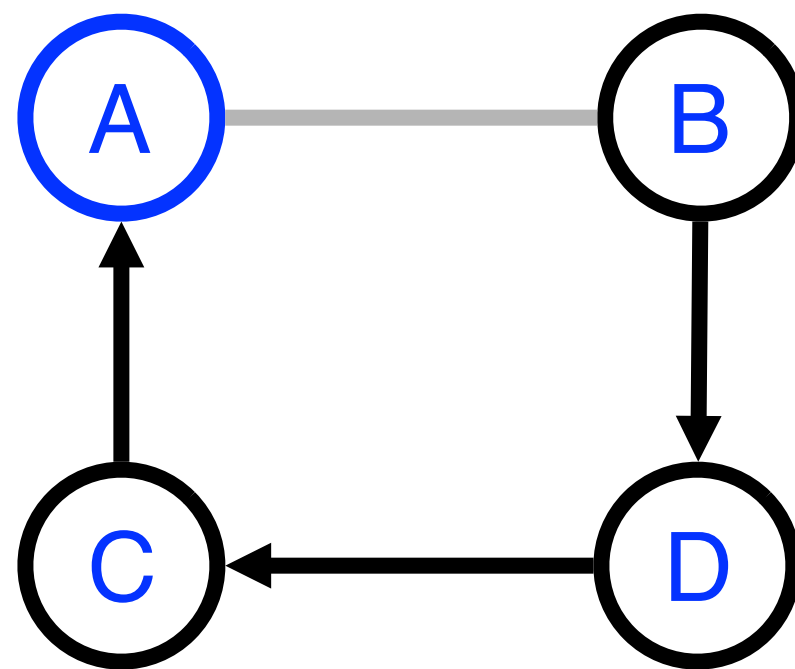


# An Example Graph with 4 possible spanning trees rooted at vertex A

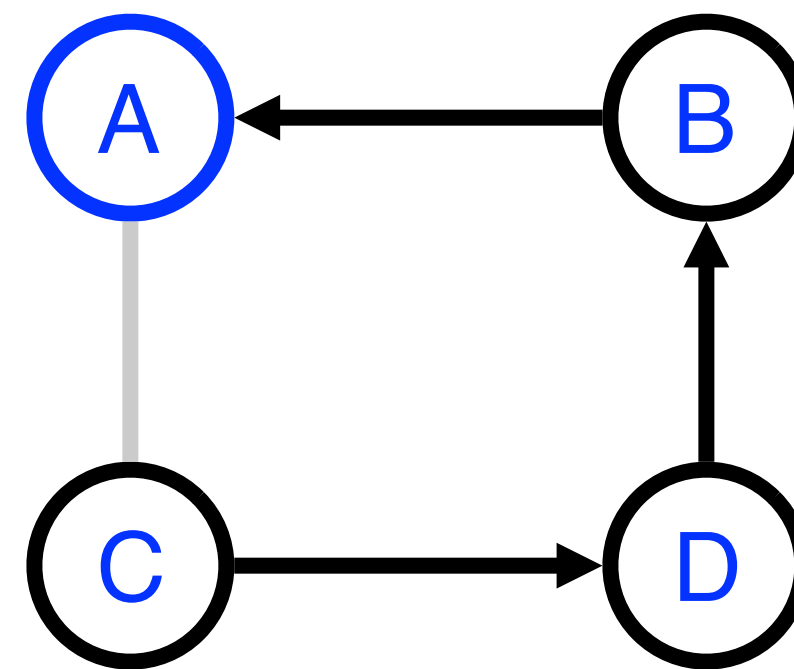
Example Undirected Graph:



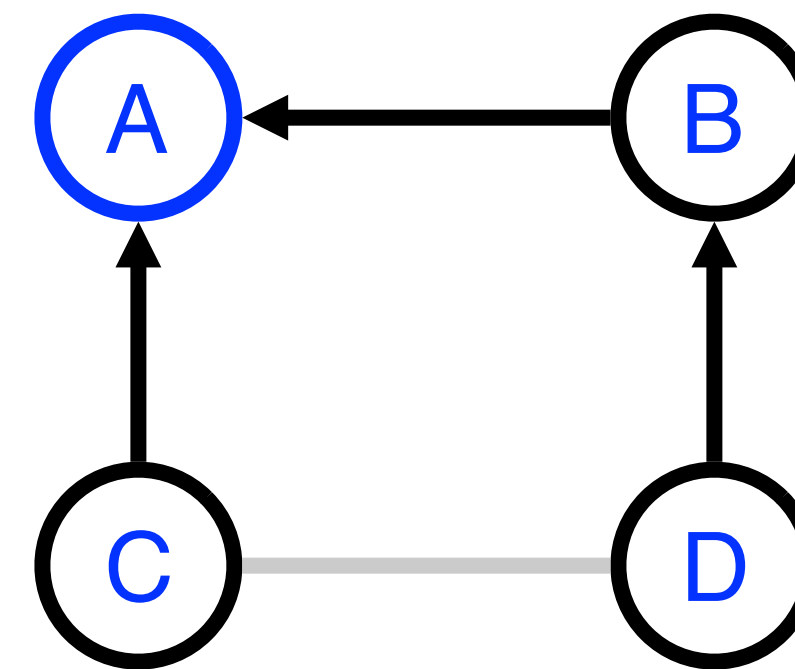
Spanning Trees (edges are directed from child to parent):



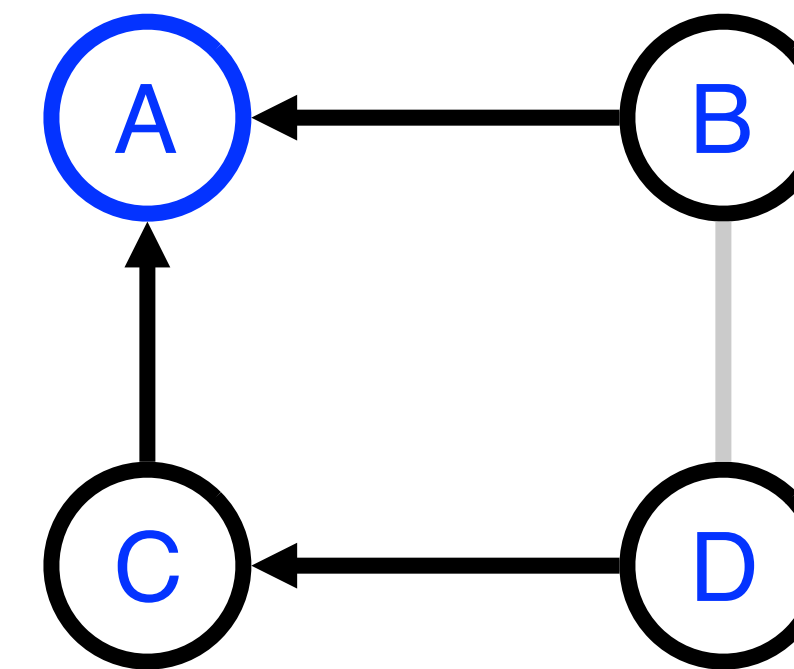
Vertex	Parent
A	null
B	D
C	A
D	C



Vertex	Parent
A	null
B	A
C	D
D	B



Vertex	Parent
A	null
B	A
C	A
D	B



Vertex	Parent
A	null
B	A
C	A
D	C





# Sequential Spanning Tree Algorithm

```
1. class V {
2.   V [] neighbors; // adjacency list for input graph
3.   V parent; // output value of parent in spanning tree

4.   boolean makeParent(V n) {
5.     if (parent == null) { parent = n; return true; }
6.     else return false; // return true if n became parent
7.   } // makeParent

8.   void compute() {
9.     for (int i=0; i<neighbors.length; i++) {
10.      final V child = neighbors[i];
11.      if (child.makeParent(this))
12.        child.compute(); // recursive call
13.    }
14.  } // compute
15. } // class V
16. ... // main program
17. root.parent = root; // Use self-cycle to identify root
18. root.compute();
19. ...
```



# Minimum Spanning Tree

---

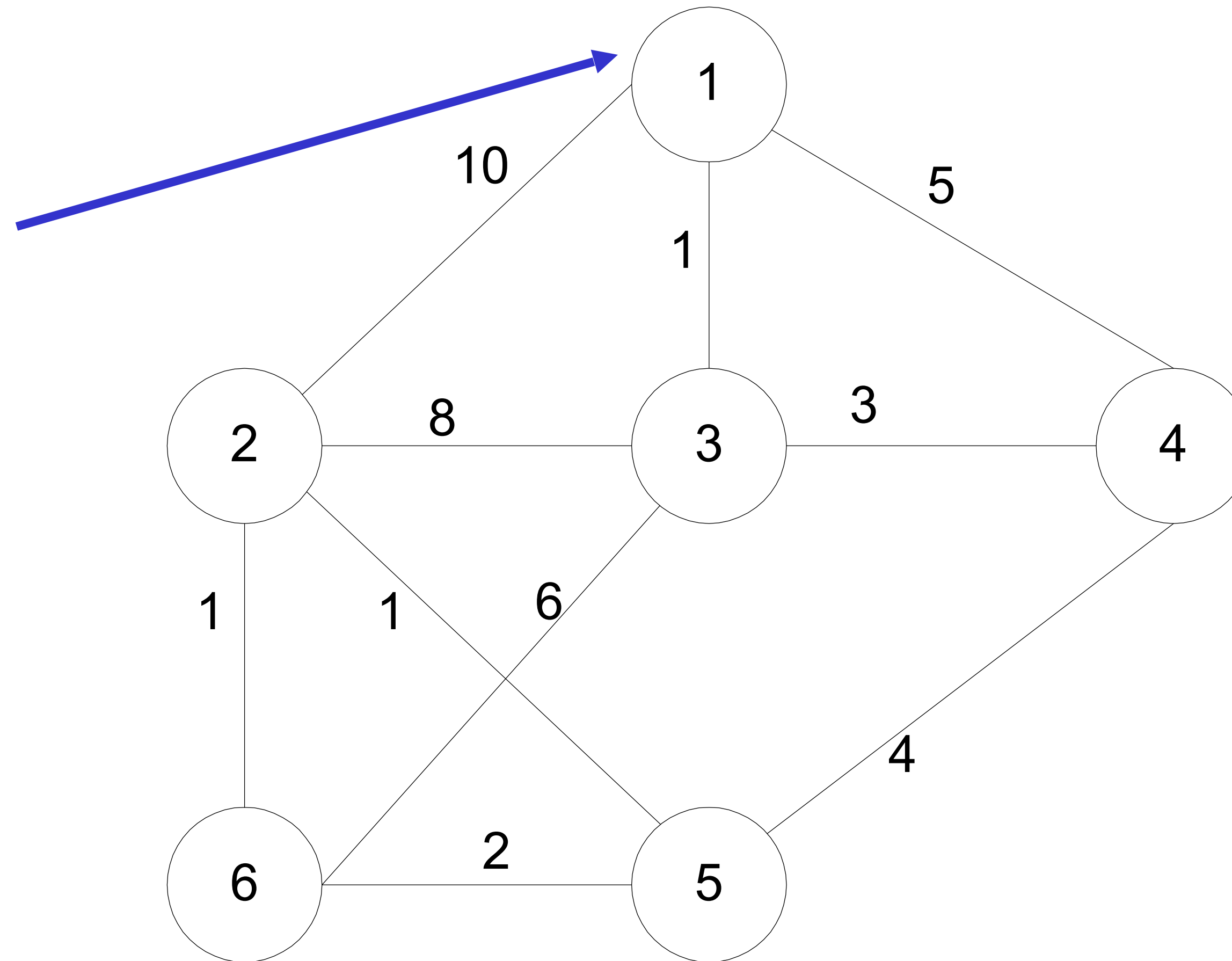
- For graphs that have edge weights
- Spanning tree with a minimum weight
- Sequential algorithms:
  - Prim's algorithm: greedy, grow a single tree by adding nodes closest to it
  - Kruskal's algorithm: greedy, add lightest edges that don't create a cycle
  - Boruvka's algorithm: combination of Prim's and Kruskal's
    - Can be parallelized



# Prim's Algorithm

Starting from empty  $T$ ,  
choose a vertex at  
random and initialize

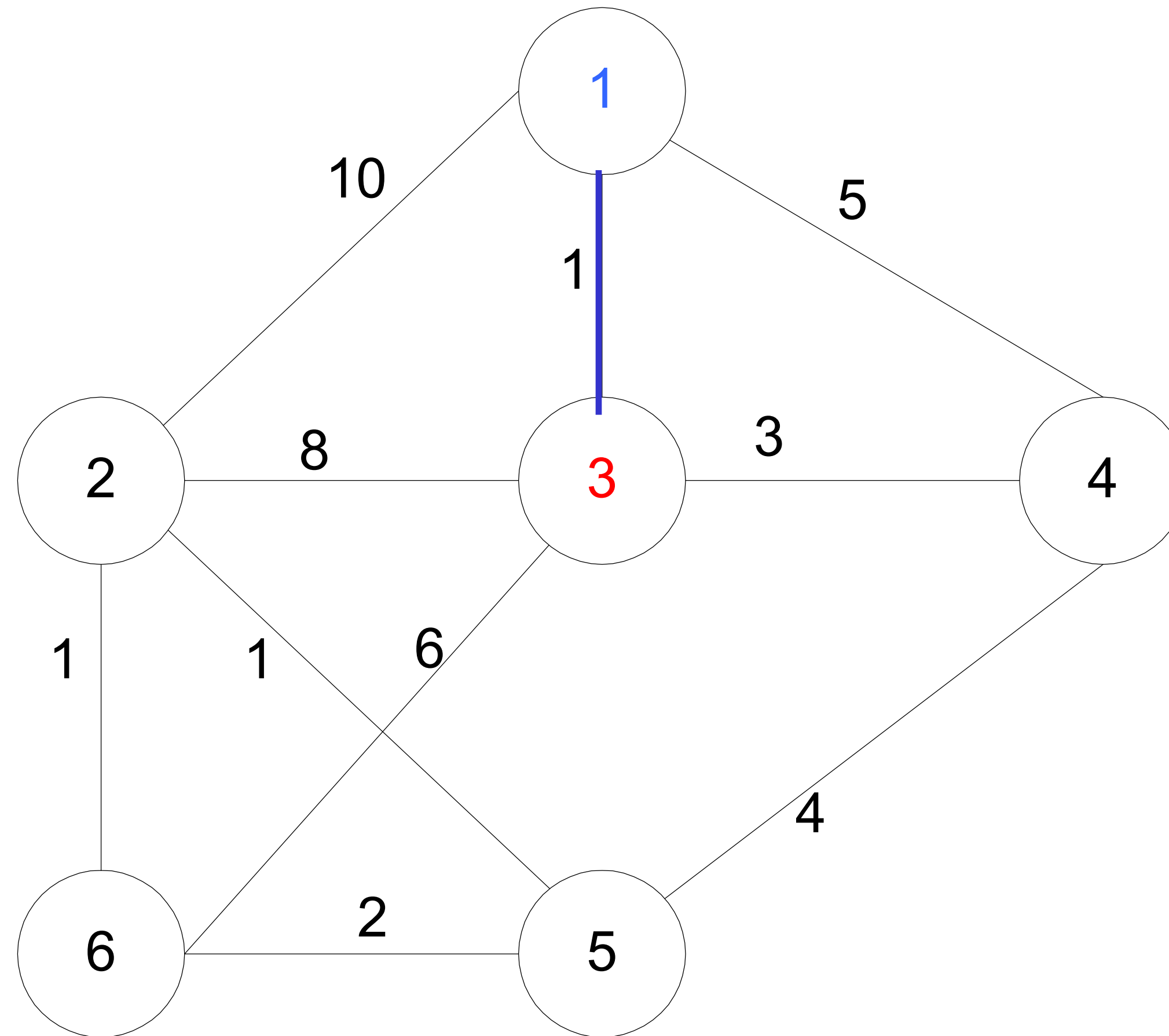
$V = \{1\}$ ,  $E' = \{\}$



# Prim's Algorithm

Choose the vertex **u** not in **V** such that edge weight from **u** to a vertex in **V** is minimal (greedy!)

$V = \{1, 3\}$   $E' = \{(1, 3)\}$



# Prim's Algorithm

Repeat until all vertices have been chosen

Choose the vertex **u** not in **V** such that edge weight from **v** to a vertex in **V** is minimal (**greedy!**)

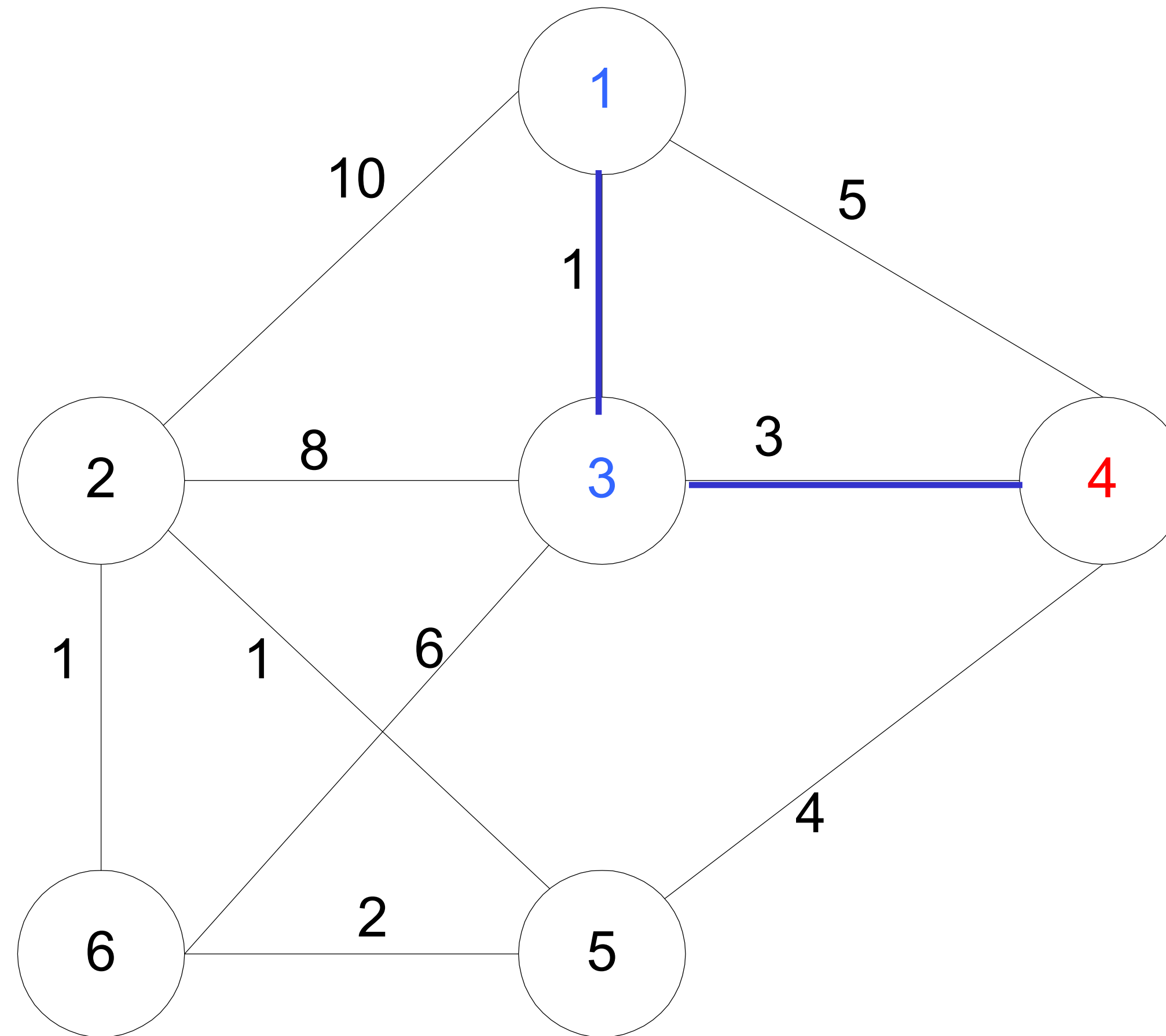
$V = \{1, 3, 4\}$   $E' = \{(1, 3), (3, 4)\}$

$V = \{1, 3, 4, 5\}$   $E' = \{(1, 3), (3, 4), (4, 5)\}$

....

$V = \{1, 3, 4, 5, 2, 6\}$

$E' = \{(1, 3), (3, 4), (4, 5), (5, 2), (2, 6)\}$



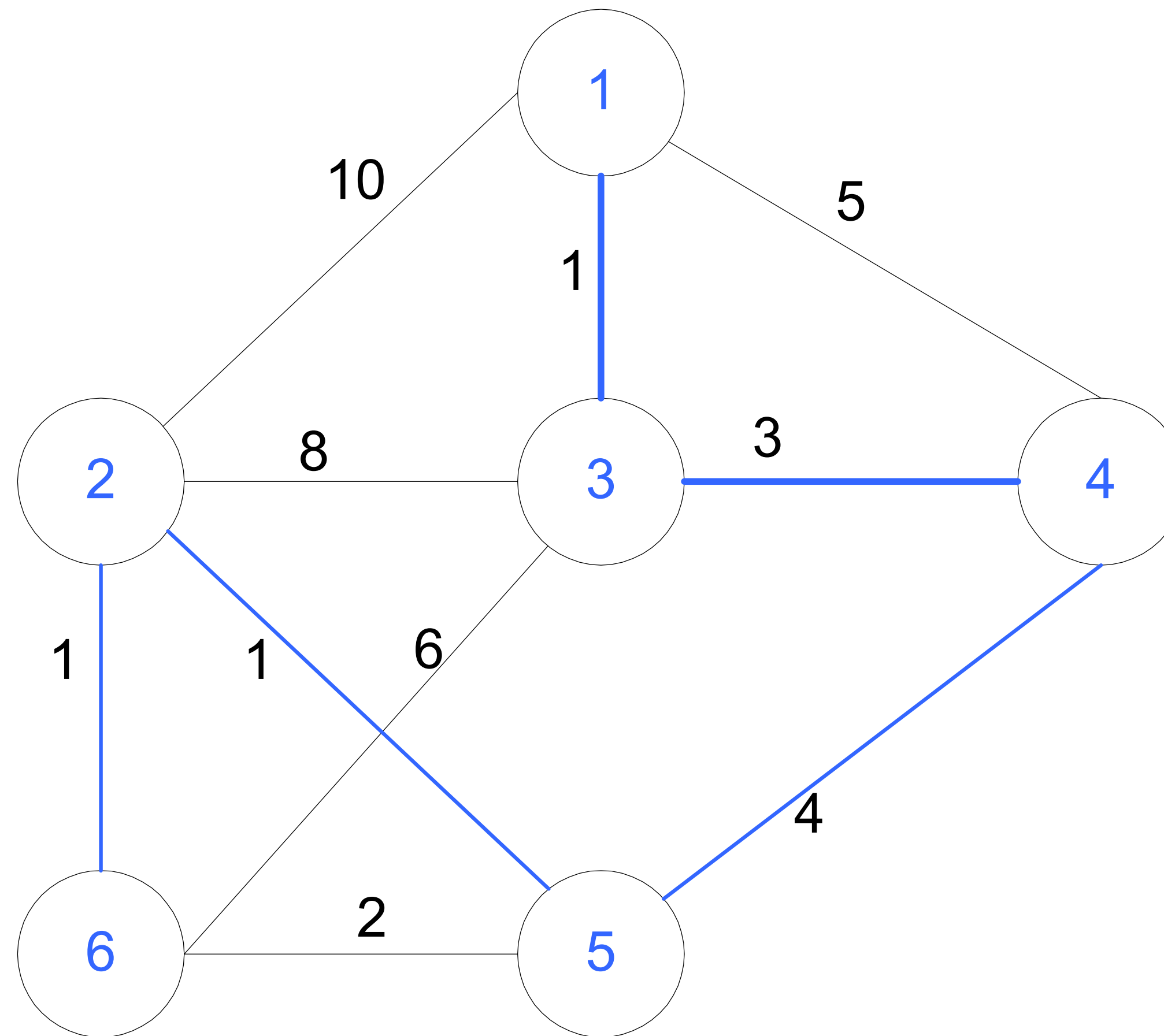
# Prim's Algorithm

Repeat until all vertices have been chosen

$V = \{1, 3, 4, 5, 2, 6\}$

$E' = \{(1, 3), (3, 4), (4, 5), (5, 2), (2, 6)\}$

Final Cost:  $1 + 3 + 4 + 1 + 1 = 10$



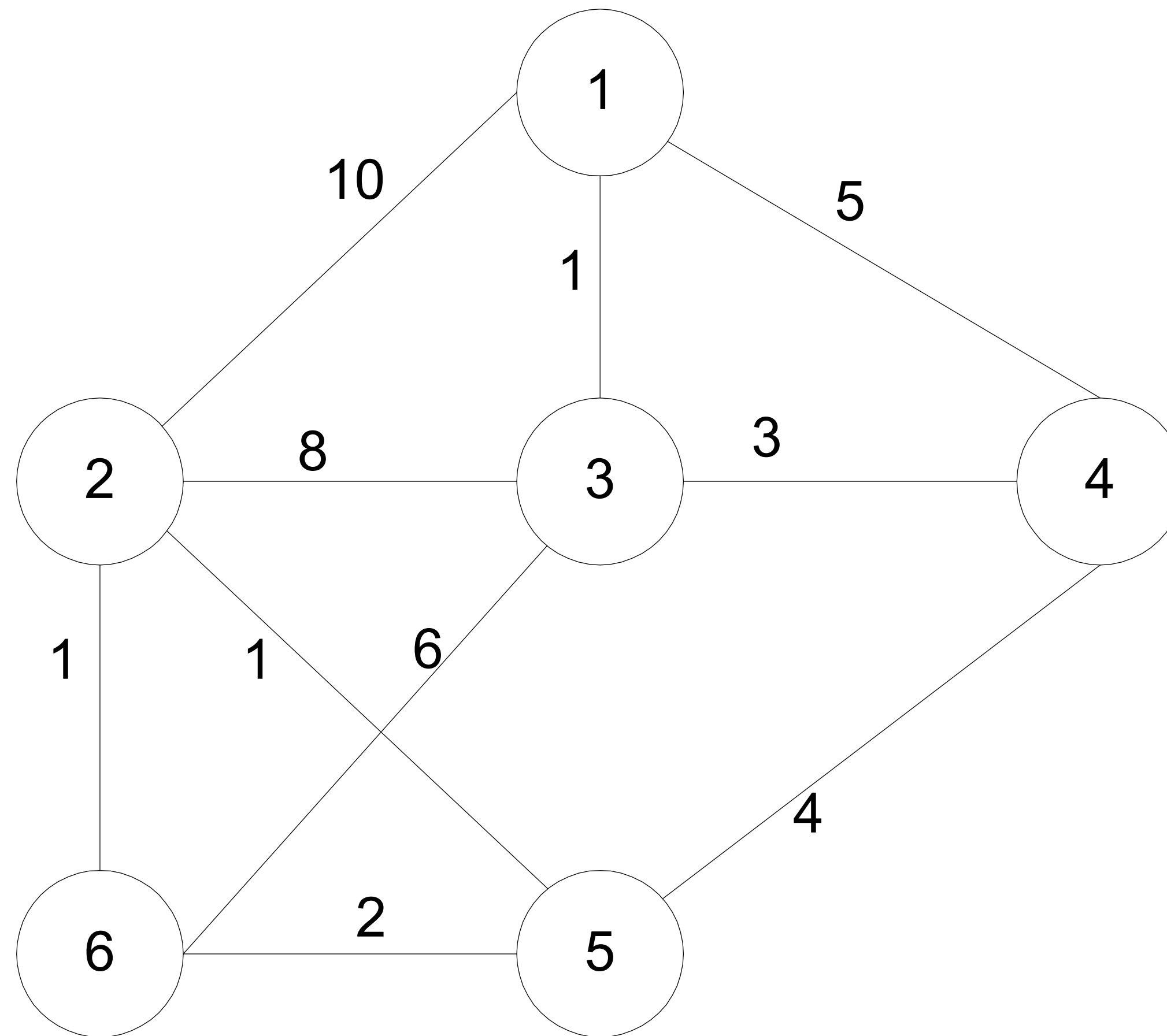
# Kruskal's Algorithm

---

- Select edges in order of increasing cost
- Accept an edge to expand tree or forest only if it does not cause a cycle
- Implementation using adjacency list, priority queues and disjoint sets

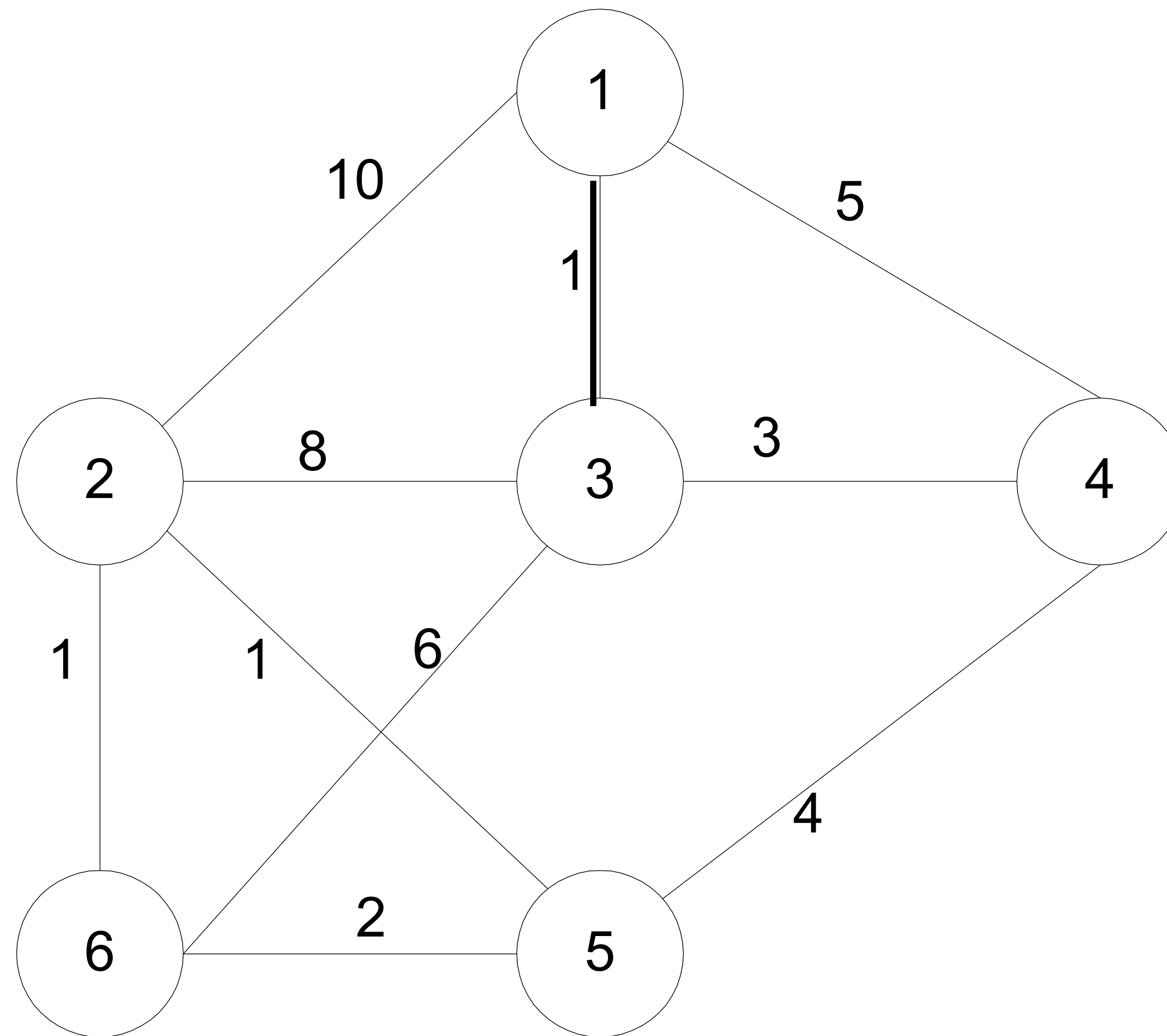


# Kruskal's Algorithm

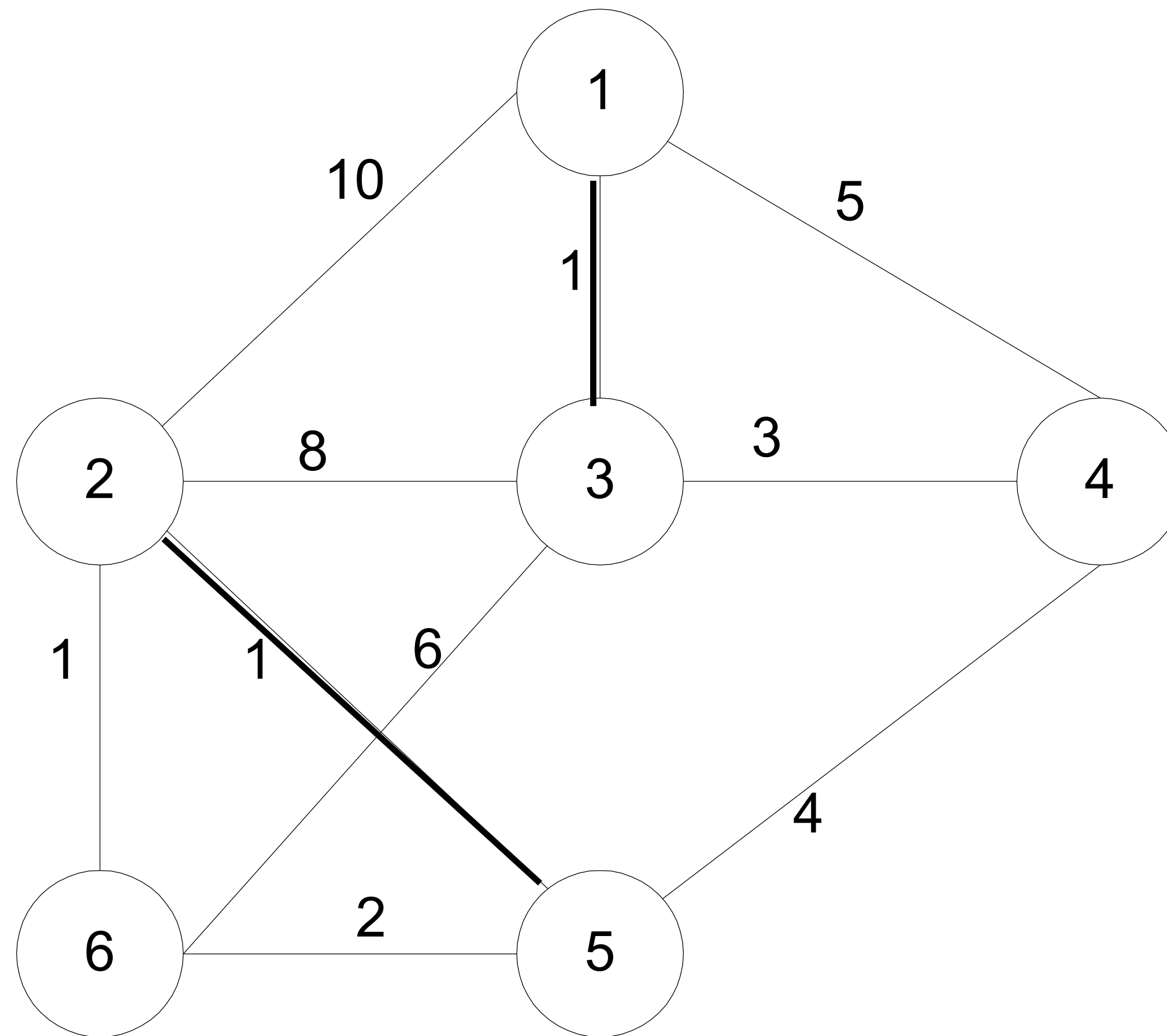




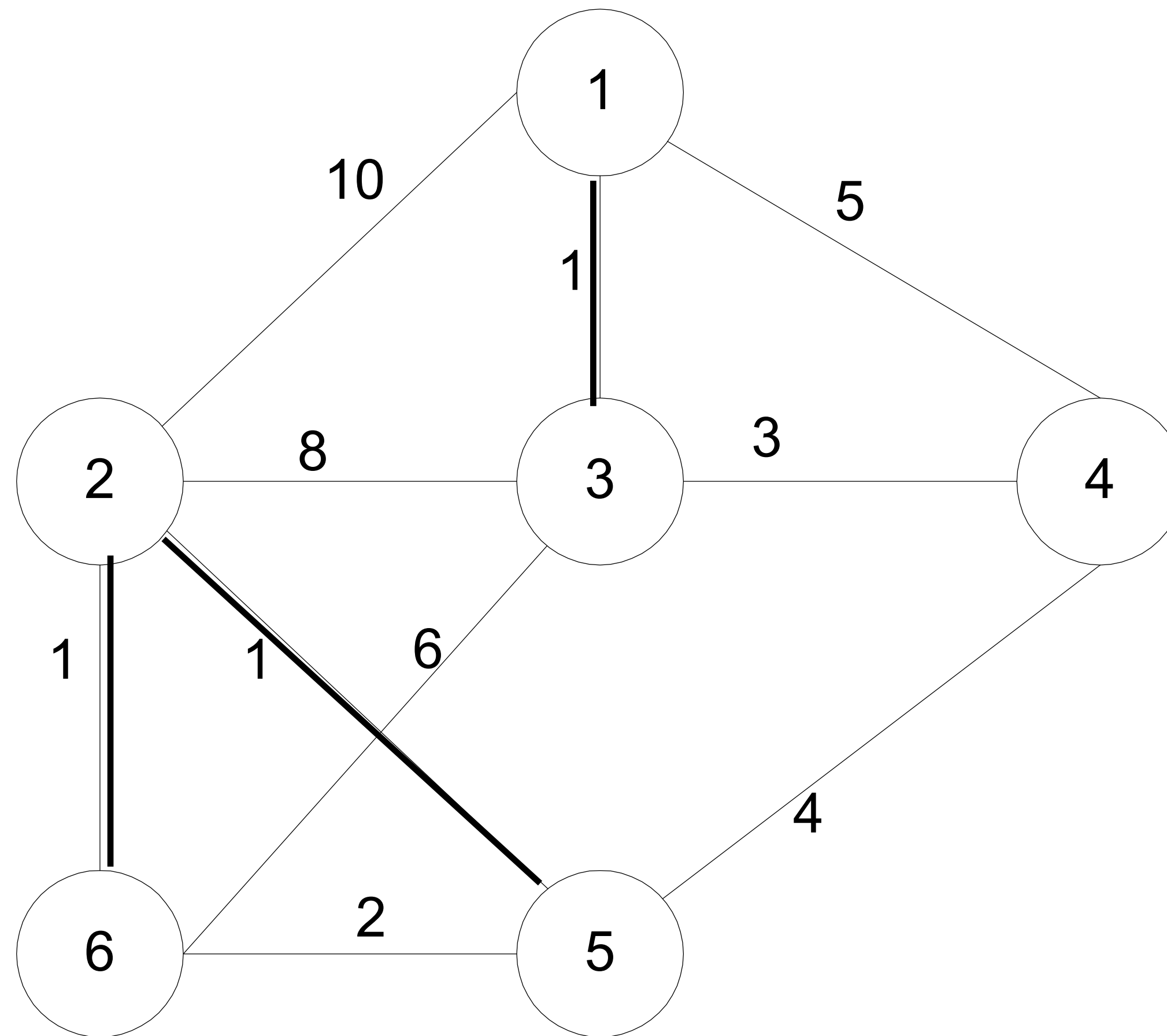
# Kruskal's Algorithm



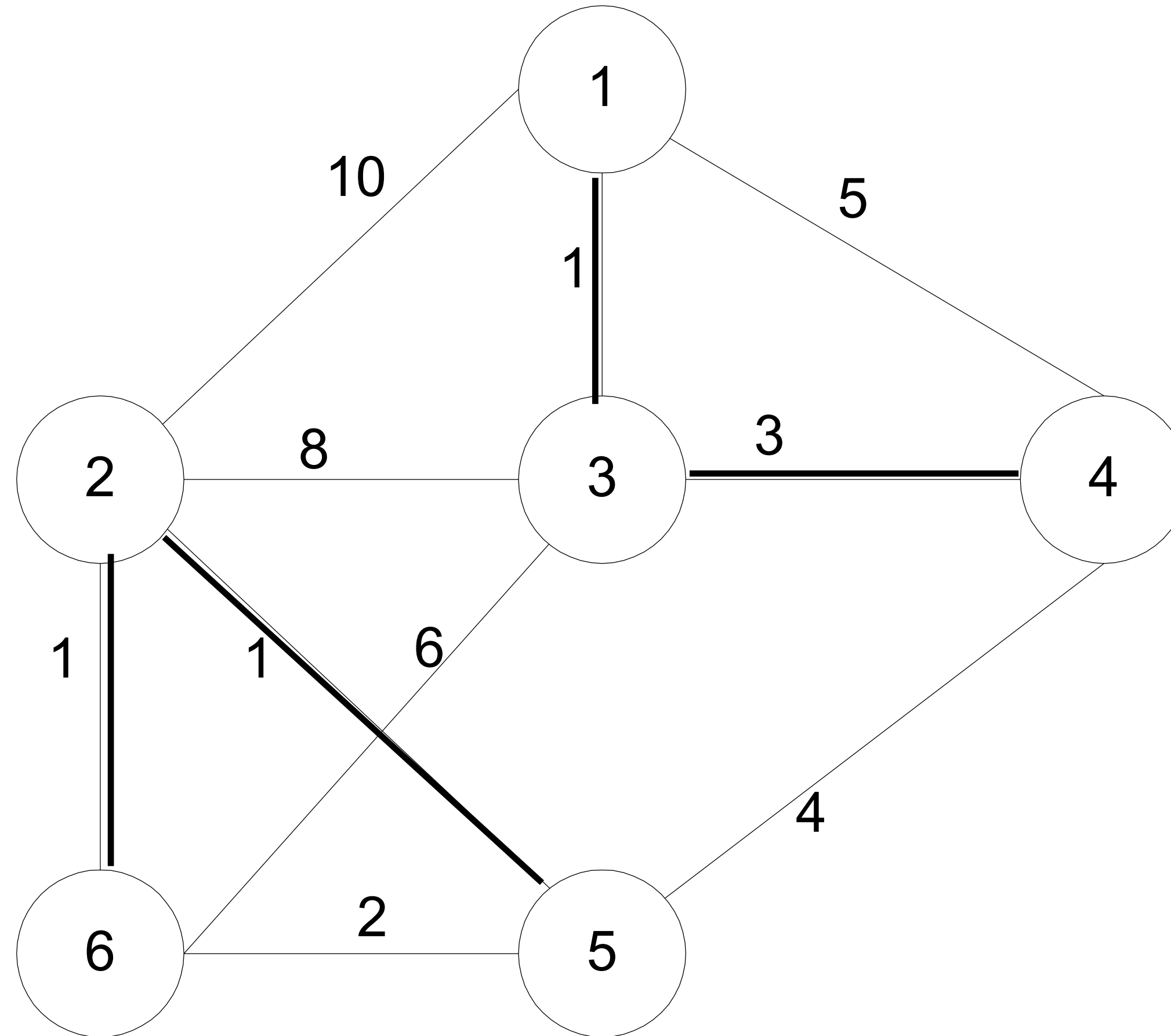
# Kruskal's Algorithm



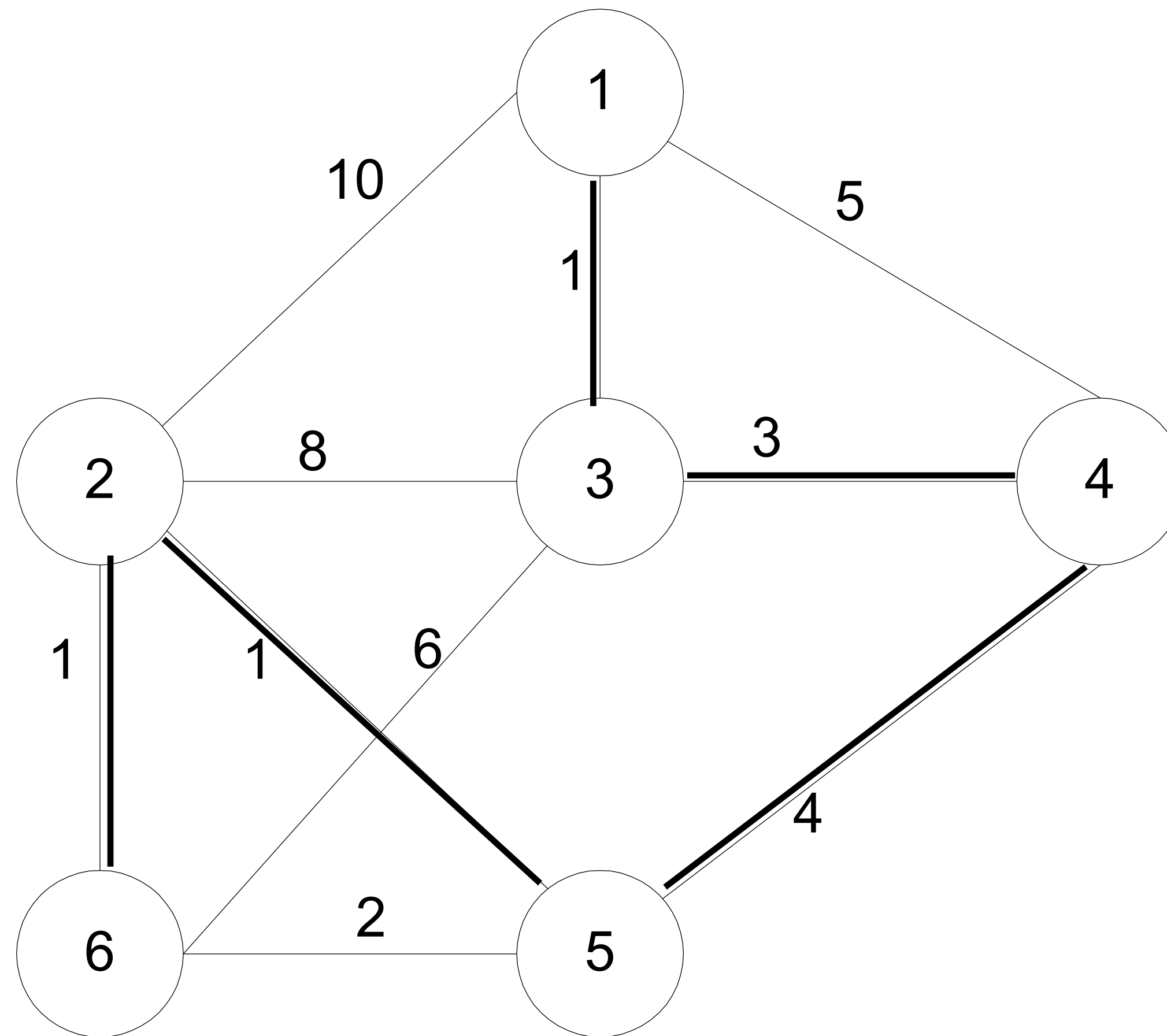
# Kruskal's Algorithm



# Kruskal's Algorithm



# Kruskal's Algorithm

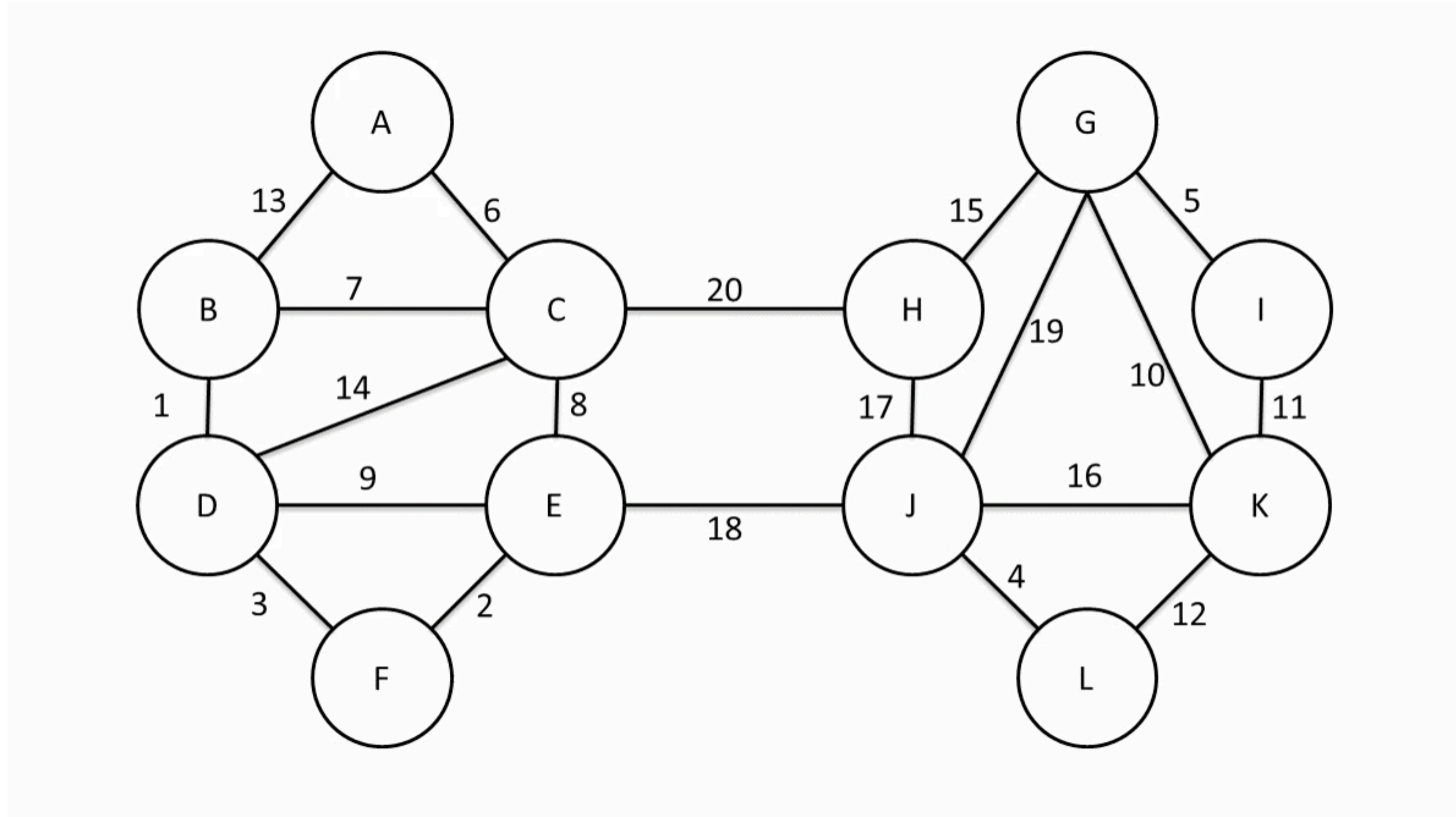


# Boruvka's Algorithm

- Combination of Prim's and Kruskal's
- Grow a tree (component) by picking the lightest edge connected to it, just like Prim
- Connect the trees when the lightest edge is between them, just like Kruskal
- Growing of each tree can be done in parallel
- Component contraction
  - Each component represented by a single node
  - When connecting two components, contract the edge and make a single node to represent the two



# Boruvka's Algorithm



Animation: Randy Cornell, Texas State University



# Parallel Boruvka's Algorithm

- Java threads or async tasks picking up components off the worklist
  - You don't want too many threads of tasks, tune for the machine
  - Worklist has to allow concurrent access
- Grow components in parallel
- When inspecting the closest node to expand the component, have to synchronize
  - Other thread or task could be also accessing it
  - Careful not to introduce deadlock
- When contracting an edge, have to synchronize
- When there's only a single component left, you are done

