Graph algorithms in their many shapes and sizes

Daniel Epstein, 5/8/14
depstein@cs.washington.edu
https://github.com/depstein/programming-competitions

Housekeeping

http://www.programmingcontest.net/

Amazon Programming Contest

Thanks for your interest in the programming contest on May 13th. This page has all the details you need to make the competition fun.

Prizes

First place gets a Kindle Fire HDX!

Second and third place get Kindle Paperwhites!

All competitors get awesome Amazon swag!

Rules

1. Any current UW student is eligible to participate. Registration is limited to the first 120 participants.
2. Individual competitors only.
3. Previous winners can compete but are ineligible for prizes.
4. Contest runs 2 hours and includes 4 different short problems.
5. Programming in Java 7.
6. The only reference material allowed is this website and Oracle's online Java API: http://docs.oracle.com/javase/7/docs/api/. No other materials, online or printed, are allowed.

The number of players is limited, so it is first come first served! You will receive an email confirmation of your spot within 72 hours of signing up.

Dates and Locations

We will have food, drinks, and the opportunity to meet some of our developers at the end of the competition.

Date: Tuesday, May 13, 2014
Time: 6:00 PM
Location: Paul Allen Center for CSE ATRIUM

https://github.com/depstein/programming-competitions

branch: **master** **programming-competitions** / problems / **dynamic programming** / History

Added solutions to Problem 2728 and 4123

ping128 authored 8 days ago — latest commit 41b420e535

..

2728 (A Spy on the Metro)	Added solutions to Problem 2728 and 4123	8 days ago
4123 (Glenbow Museum)	Added solutions to Problem 2728 and 4123	8 days ago
4131 (Currency Shopping)	Solution to Problem 4131	10 days ago
4213 (DNA Sequences)	Solutions to Problem 4213, 4877, 4905, 5945, 6088	10 days ago
4280 (Pencils from the Ninet...	Solution to Problem 4280	13 days ago
4794 (Sharing Chocolate)	Yeah, solved Problem 4794	9 days ago
4877 (Non-Decreasing Digits)	Solutions to Problem 4213, 4877, 4905, 5945, 6088	10 days ago
4905 (Pro-Test Voting)	Solutions to Problem 4213, 4877, 4905, 5945, 6088	10 days ago
5945 (Raggedy, Raggedy)	Solutions to Problem 4213, 4877, 4905, 5945, 6088	10 days ago
6088 (Approximate Sorting)	Solutions to Problem 4213, 4877, 4905, 5945, 6088	10 days ago

Practice Contest

- 5 hours (+setup time)
- Real problem set!
- Real submission system
- Teams of 3
- Food?

Practice Contest

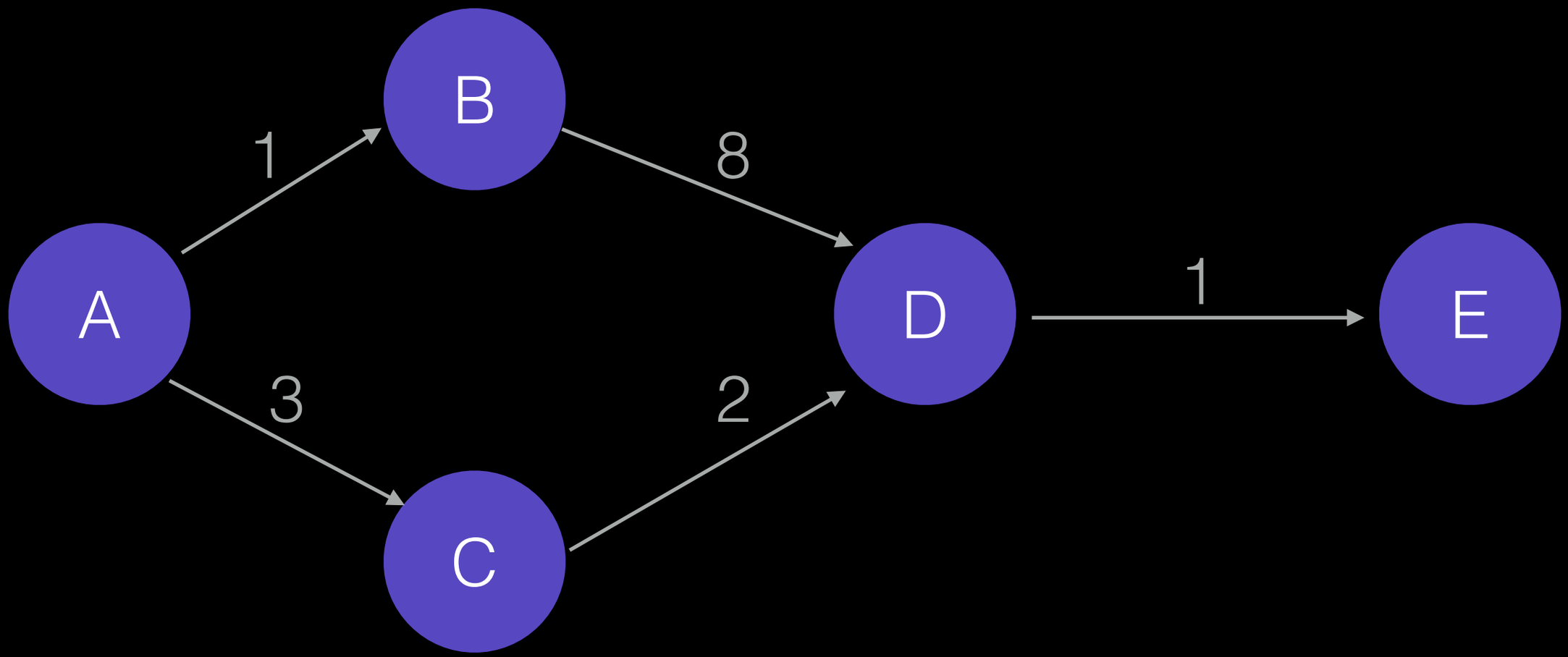
- 5/24, 5/25 (Memorial Day weekend)
- Monday 5/26 (Memorial Day)
- 5/31, 6/1
- 6/7, 6/8 (Weekend before finals)
- 6/9-6/13 (Finals week)

In mathematics and computer science, **graph theory** is the study of *graphs*, which are mathematical structures used to model pairwise relations between objects. A "graph" in this context is made up of "vertices" or "nodes" and lines called *edges* that connect them. A graph may be *undirected*,

In mathematics and computer science, **graph theory** is the study of *graphs*, which are mathematical structures used to model pairwise relations between objects. A "graph" in this context is made up of "vertices" or "nodes" and lines called *edges* that connect them. A graph may be *undirected*,

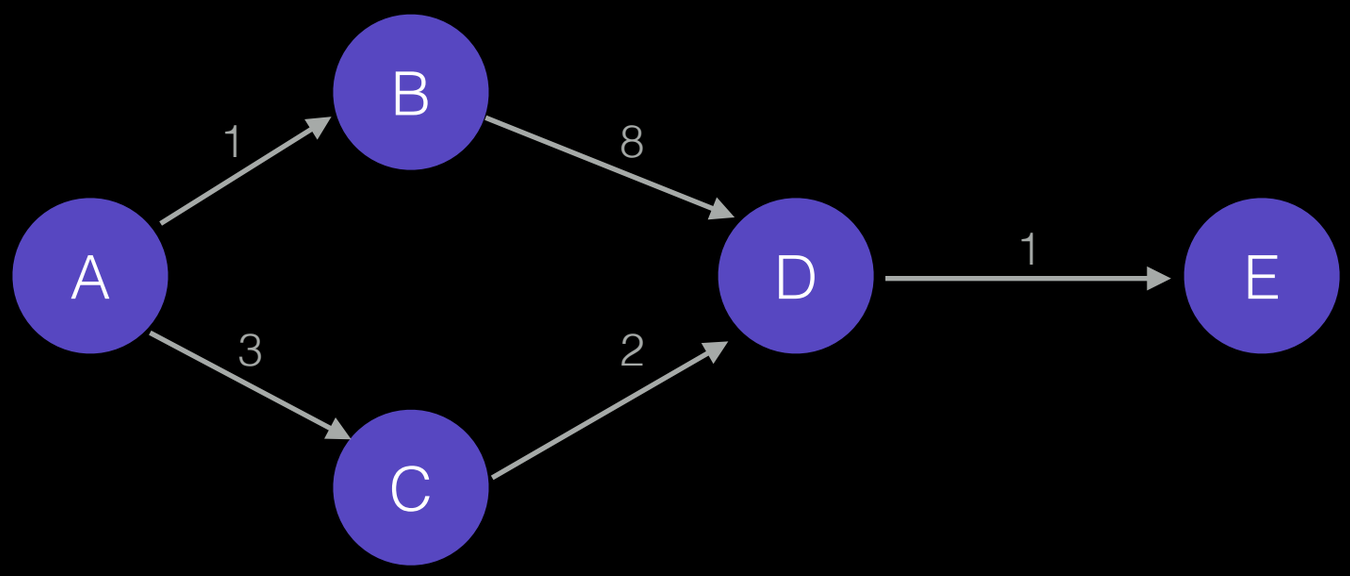
Nodes and Edges!

Graph Representation



Arrays

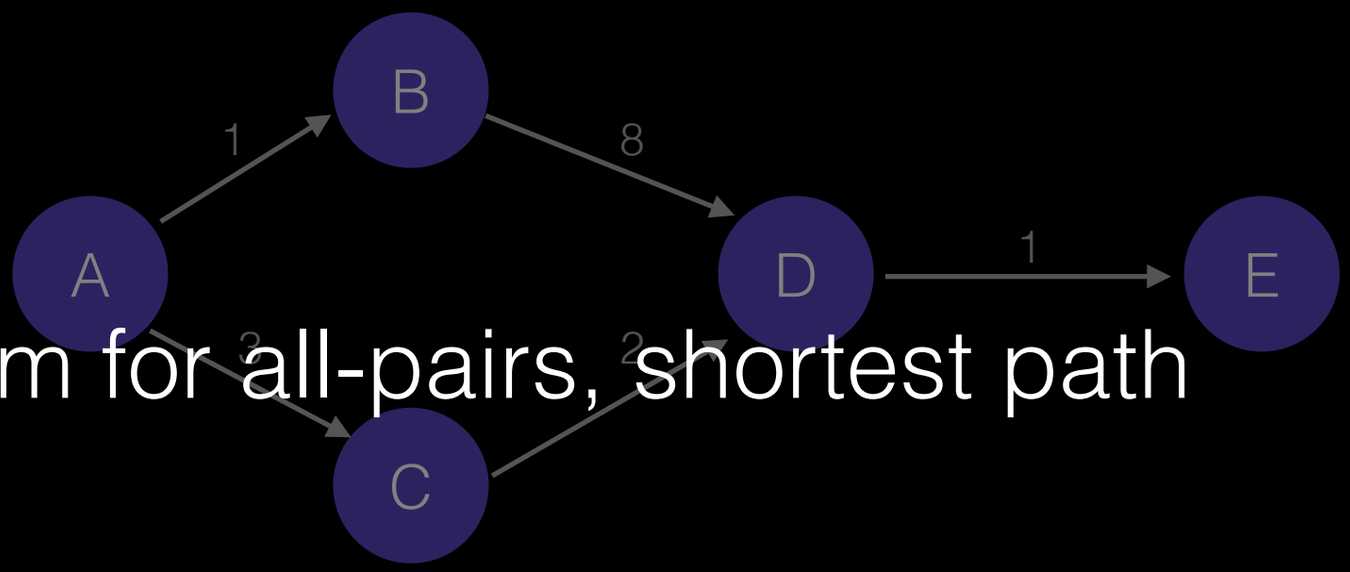
```java
import java.io.*;
import java.util.*;

public class arrays {
  public static int[][] dist = new int[5][5];

  public static void main(String[] args) {
    for(int i=0;i<5;i++)
      for(int j=0;j<5;j++) {
        if(i != j)
          dist[i][j] = 1000; // Not using Integer.MAX_VALUE to avoid integer overflowing
      }
    //Initialize graph as described
    dist[0][1] = 1;
    dist[0][2] = 3;
    dist[1][3] = 8;
    dist[2][3] = 2;
    dist[3][4] = 1;
  }
}
```



Arrays

```
import java.io.*;
import java.util.*;

public class Arrays {
  public static int[][] dist = new int[5][5];

  public static void main(String[] args) {
    for(int i=0;i<5;i++) {
      for(int j=0;j<5;j++) {
        if(i != j)
          dist[i][j] = 1000; // Not using Integer.MAX_VALUE to avoid integer overflowing
      }
    }
    //Initialize graph as described
    dist[0][1] = 1;
    dist[0][2] = 3;
    dist[1][3] = 8;
    dist[2][3] = 2;
    dist[3][4] = 1;
  }
}
```

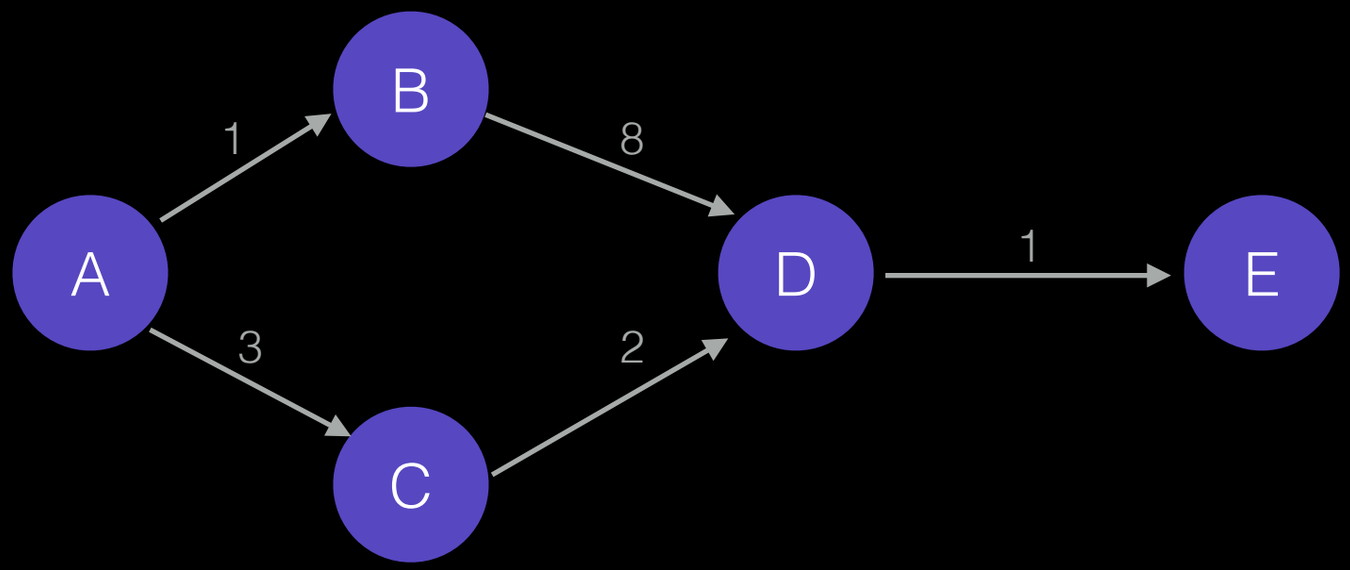
- Know the max number of nodes
- Adjacency Matrix
- Easy to look up edge weight/edge existence
- Slow to get all edges out of a node
- Uses N^2 memory
- I tend to only use them for all-pairs, shortest path



Dedicated classes

```
class Node {
    public ArrayList<Edge> neighbors = new ArrayList<Edge>();
}
```

```
class Edge {
    public Node dest;
    public int distance;

    public Edge(Node dest, Node source, int distance) {
        this.dest = dest;
        this.distance = distance;
        source.neighbors.add(this);
    }
}
```



Dedicated classes

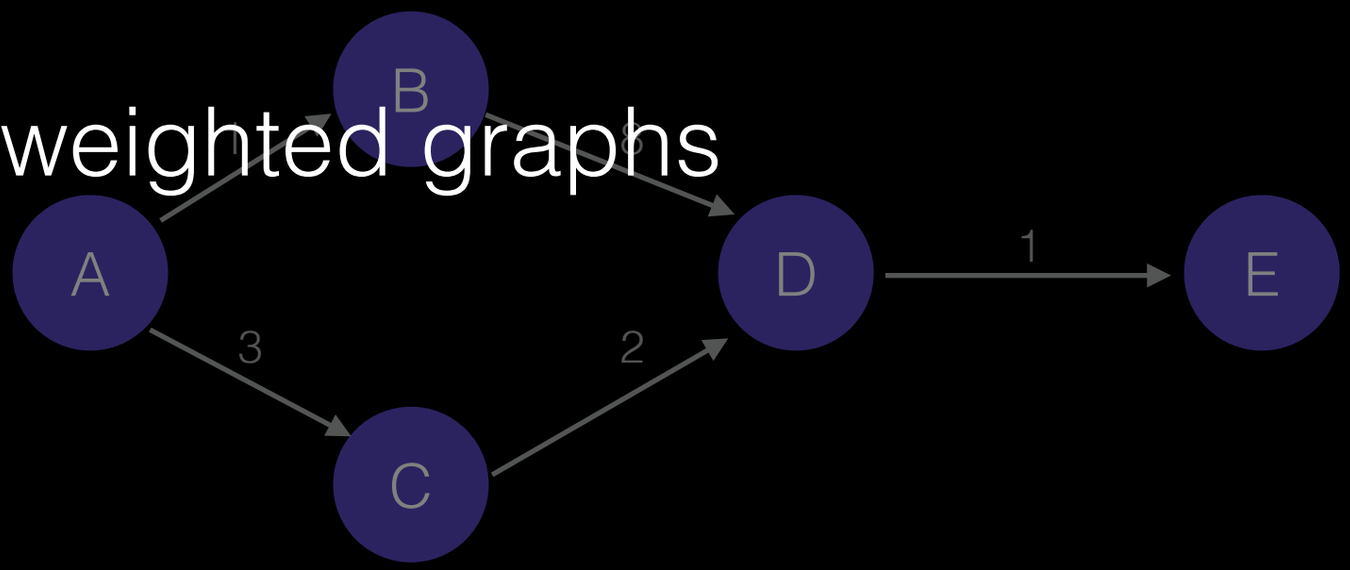
- Well-organized
- Adjacency Lists

```
class Node {
    public ArrayList<Edge> neighbors = new ArrayList<Edge>();
}
```

- Easy to get all edges out of a node
- Difficult to find a particular edge

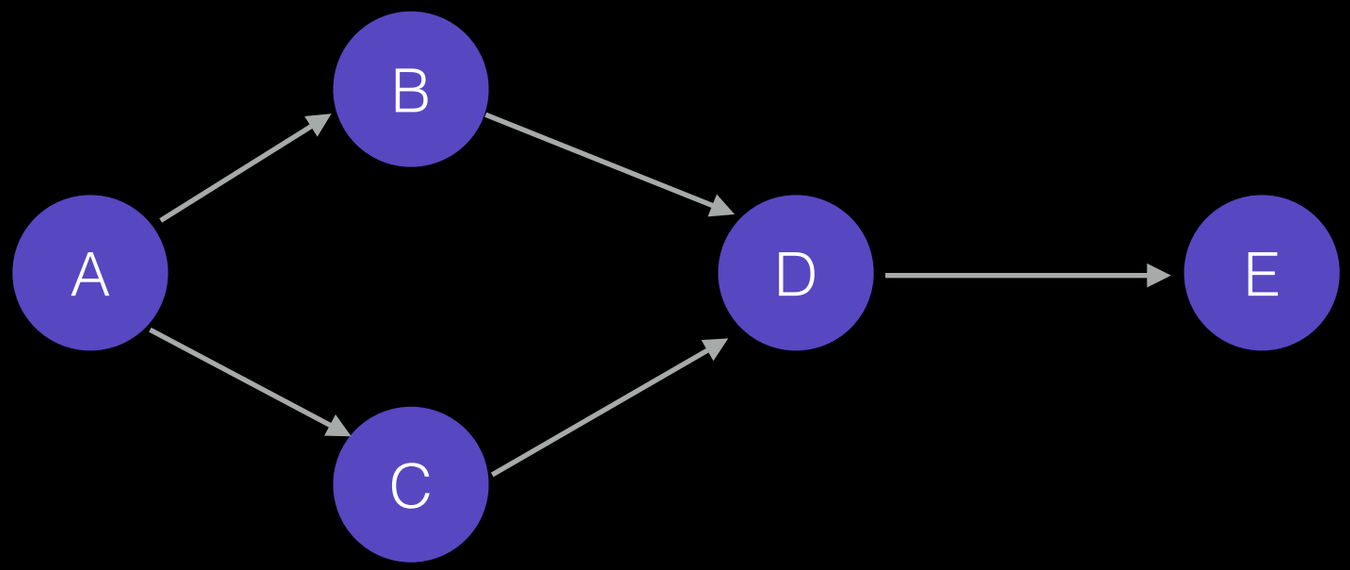
```
class Edge {
    public Node dest;
    public int distance;

    public Edge(Node dest, Node source, int distance) {
        this.dest = dest;
        this.distance = distance;
        source.neighbors.add(this);
    }
}
```

- This is what I use for weighted graphs



Unweighted Graphs

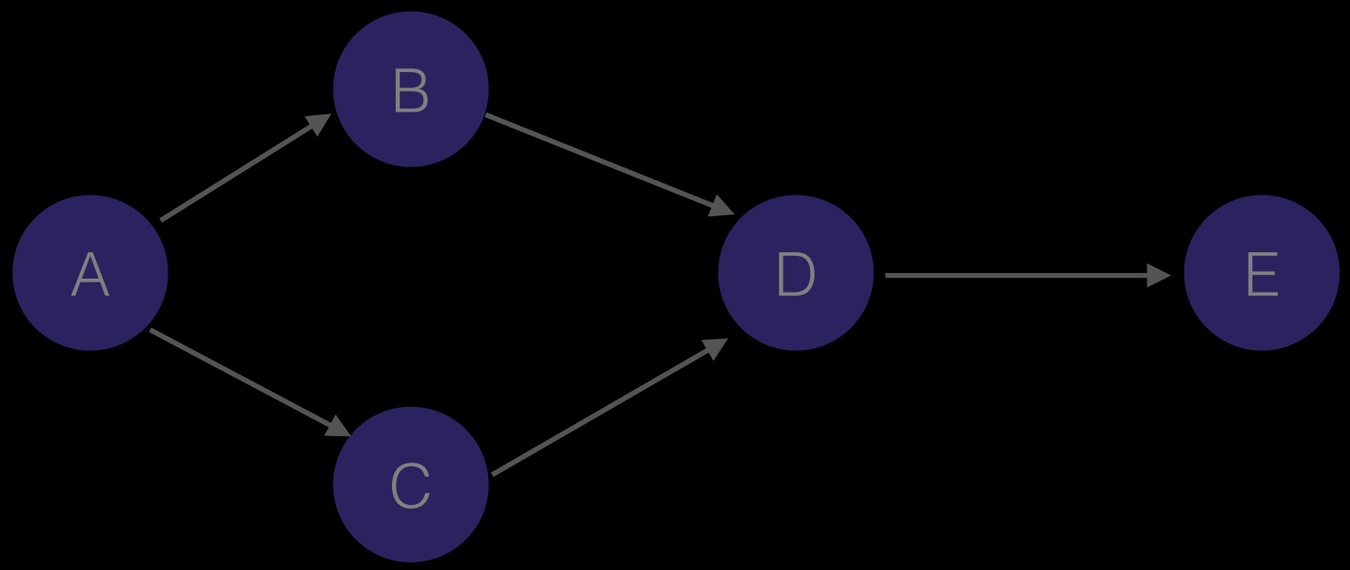
```
class Node {
    public ArrayList<Node> neighbors = new ArrayList<Node>();
}
```



Unweighted Graphs

- Easy. You don't need an edge class!

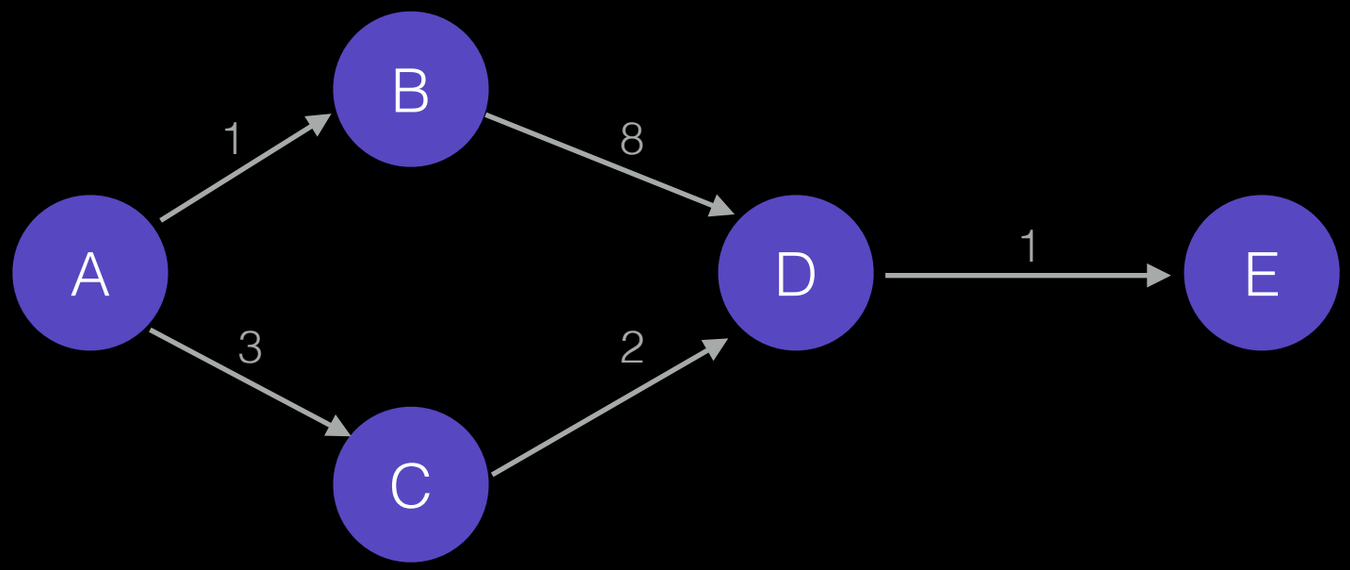
```
class Node {
    public ArrayList<Node> neighbors = new ArrayList<Node>();
}
```



HashMaps

```
//For a weighted graph
class Node {
    public HashMap<Node, ArrayList<Node, Integer>> neighbors = new HashMap<Node,
ArrayList<Node, Integer>>();
}
```

```
//For a weighted graph, naming the nodes with Strings
public HashMap<String, HashMap<String, Integer>> neighbors = new HashMap<String,
HashMap<String, Integer>>();

//For an unweighted graph, naming the nodes with Strings
public HashMap<String, ArrayList<String>> neighbors = new HashMap<String,
ArrayList<String>>();
```

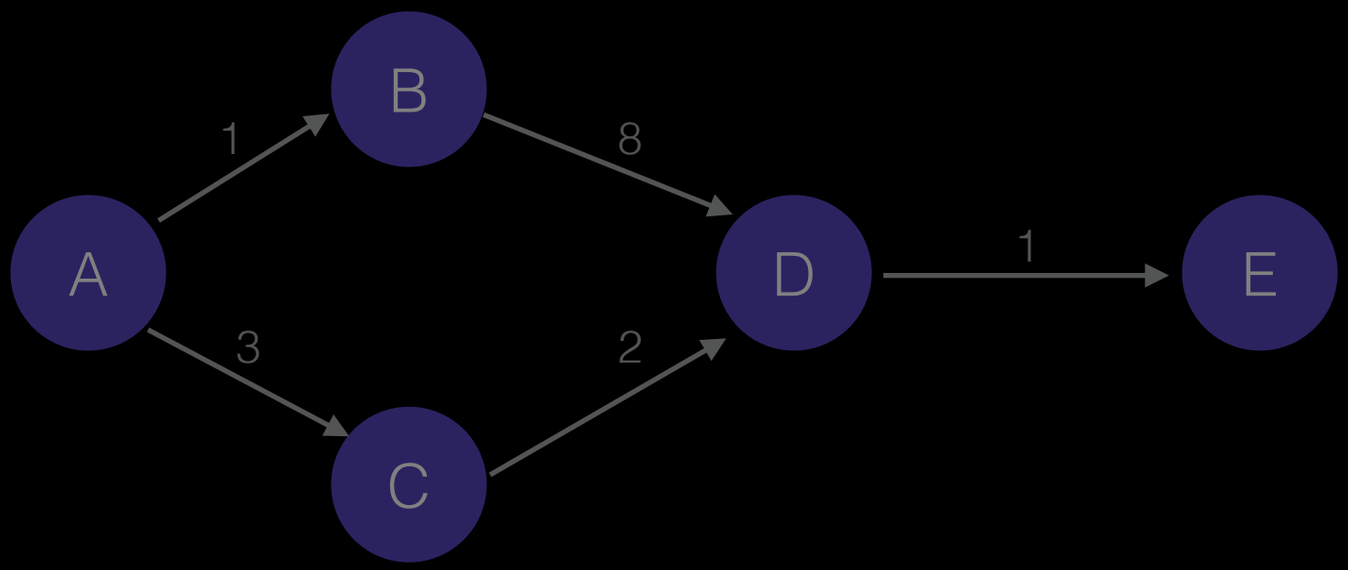


HashMaps

- A little harder to organize, debug
- But if you get used to it, can write shorter code

```
//For a weighted graph
class Node {
    public HashMap<Node, ArrayList<Node, Integer>> neighbors = new HashMap<Node,
ArrayList<Node, Integer>>();
}
```

```
//For a weighted graph, naming the nodes with Strings
public HashMap<String, HashMap<String, Integer>> neighbors = new HashMap<String,
HashMap<String, Integer>>();

//For an unweighted graph, naming the nodes with Strings
public HashMap<String, ArrayList<String>> neighbors = new HashMap<String,
ArrayList<String>>();
```

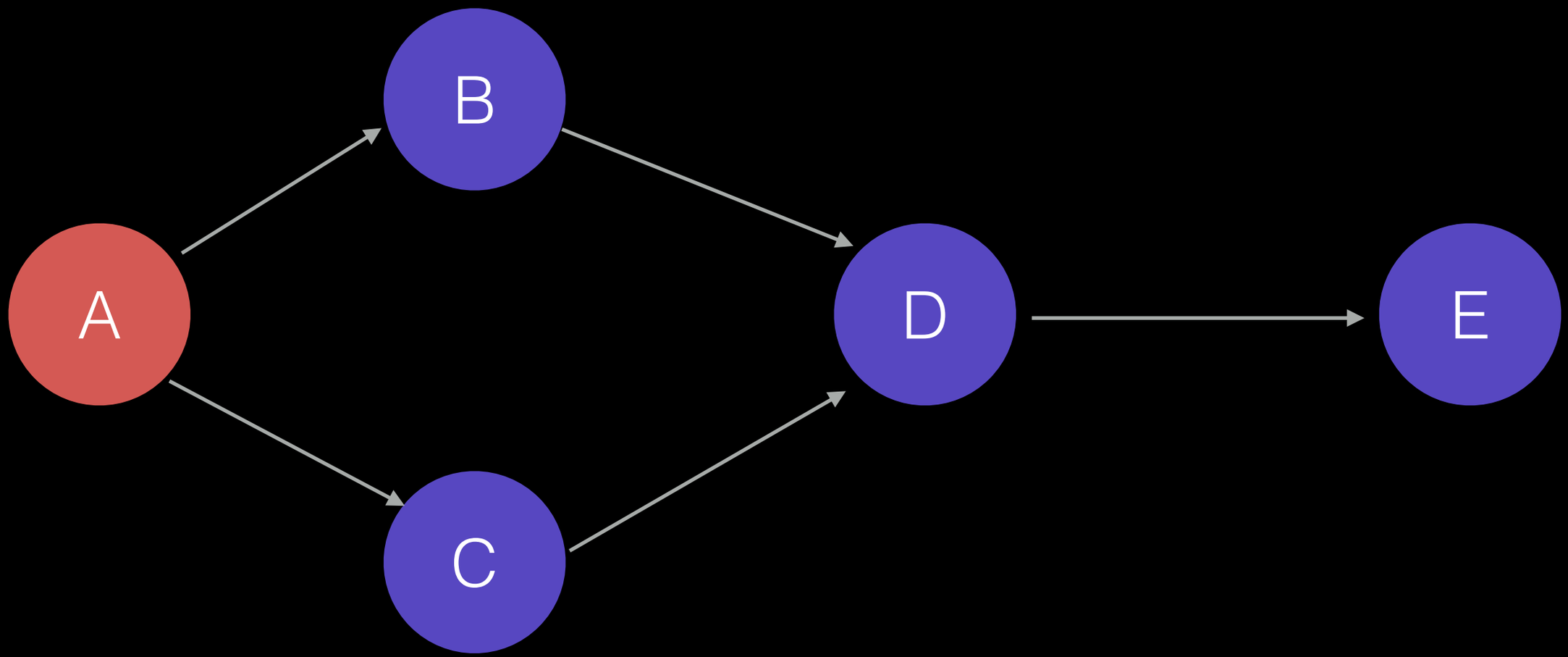


Graph Algorithms

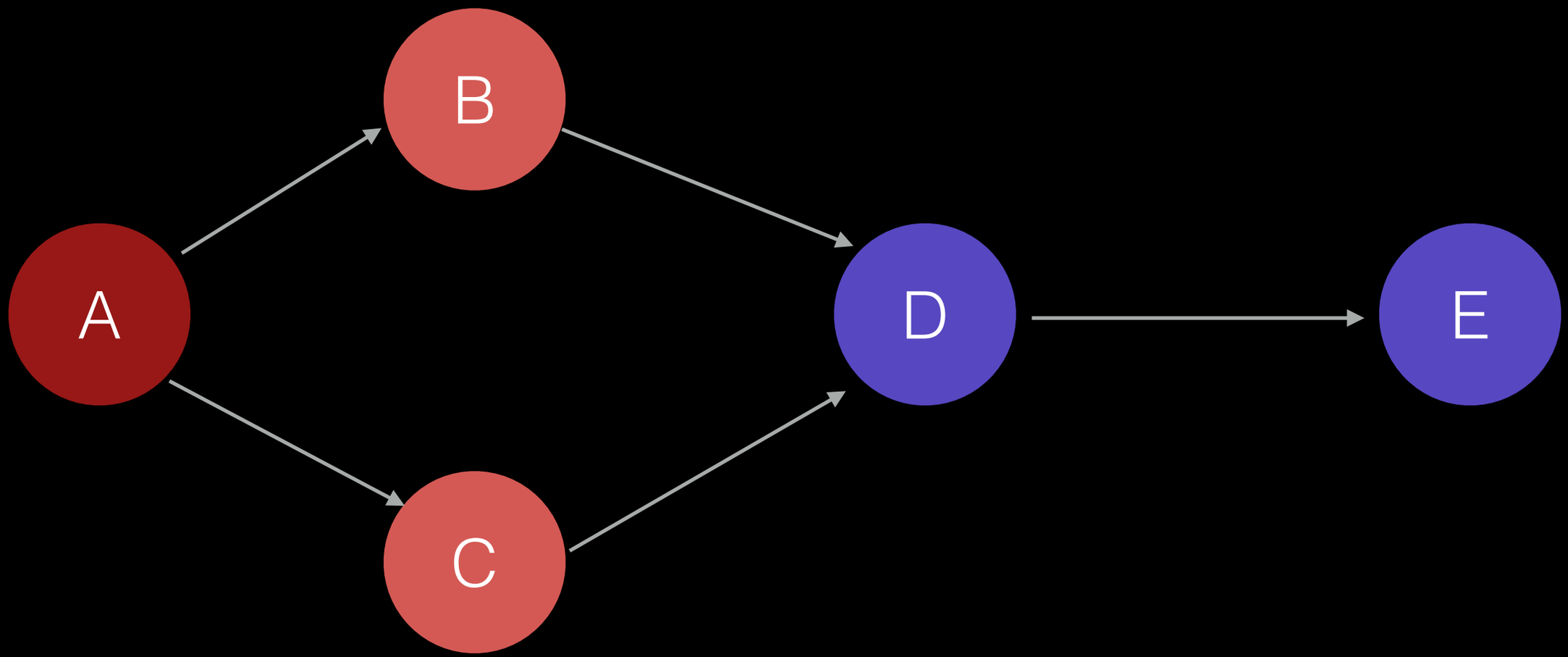
- Searches (Breadth-First, Depth-First)
- Shortest Path (Dijkstra's)
- Minimum Spanning Tree (Prim's, Kruskal's)
- Topological Sort
- Negative-Edge Shortest Path (Bellman-Ford)
- All-Pairs, Shorest Path (Floyd-Warshall)
- Max Flow (Ford-Fulkerson, Edmonds-Karp, Preflow Push)
- Minimum-Cost Flow

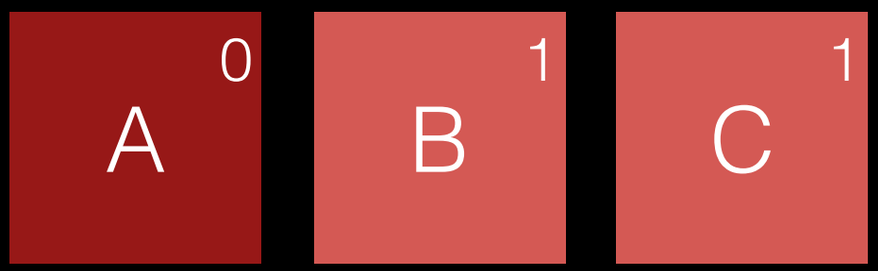
Graph Algorithms

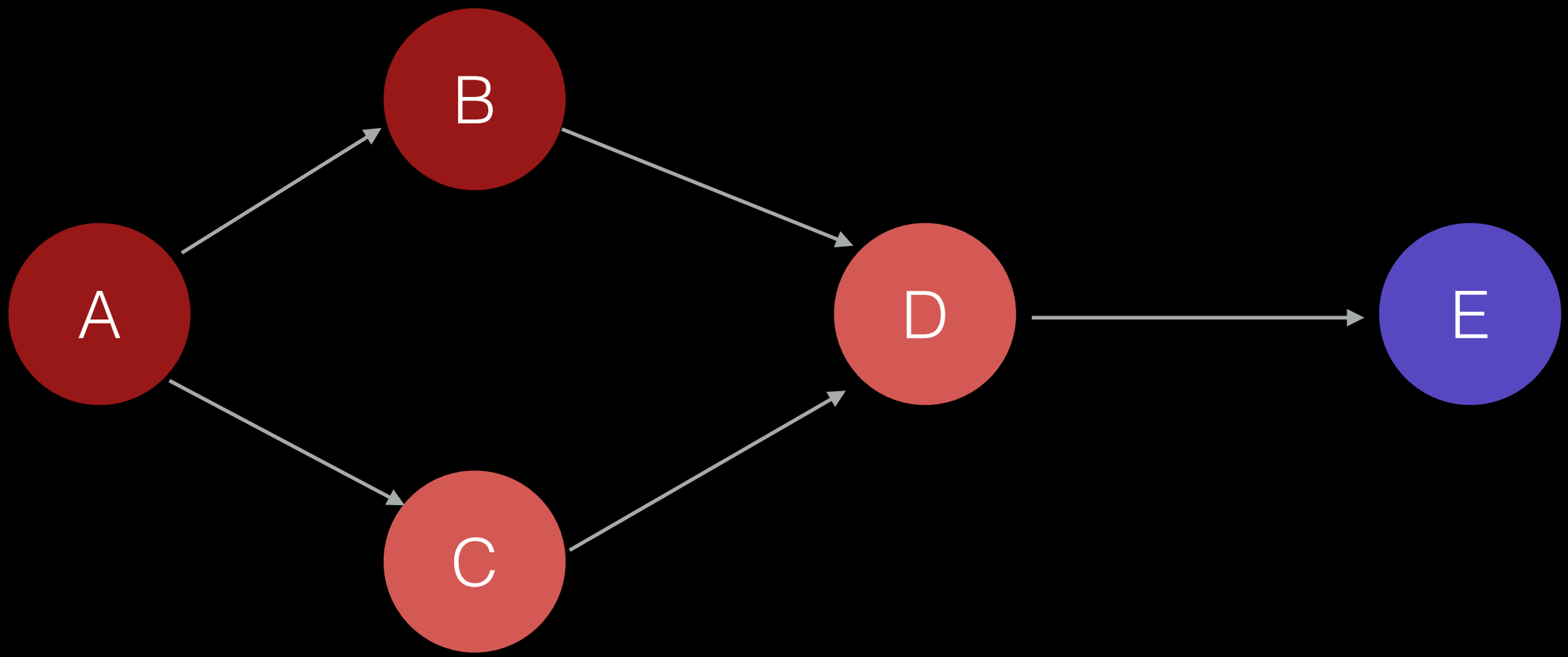
- Searches (Breadth-First, Depth-First)
- Shortest Path (Dijkstra's)
- Minimum Spanning Tree (Prim's, Kruskal's)
- Topological Sort
- Negative-Edge Shortest Path (Bellman-Ford)
- All-Pairs, Shorest Path (Floyd-Warshall)
- Max Flow (Ford-Fulkerson, Edmonds-Karp, Preflow Push)
- Minimum-Cost Flow



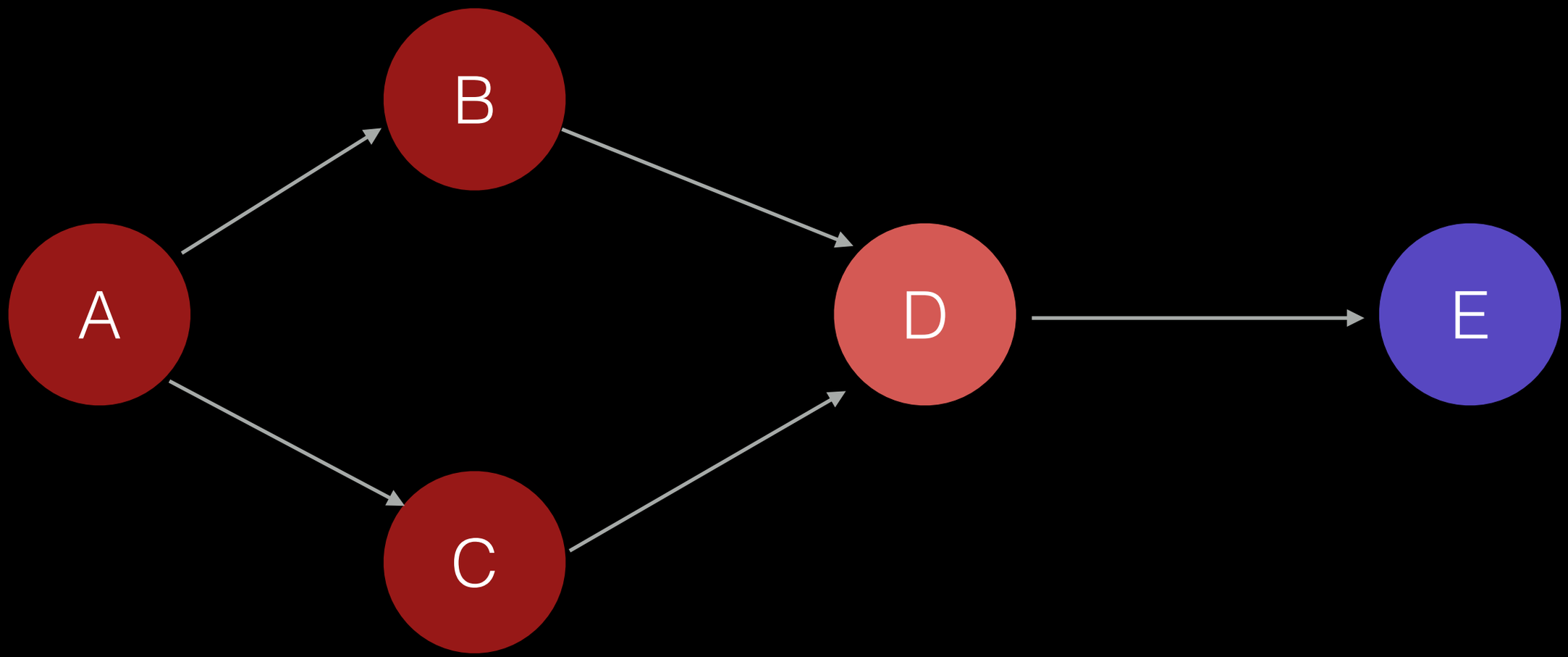
A 0

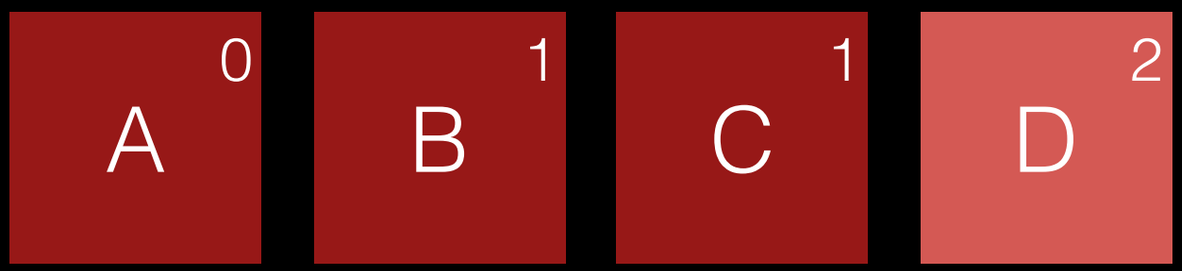

B
A
D
E
C

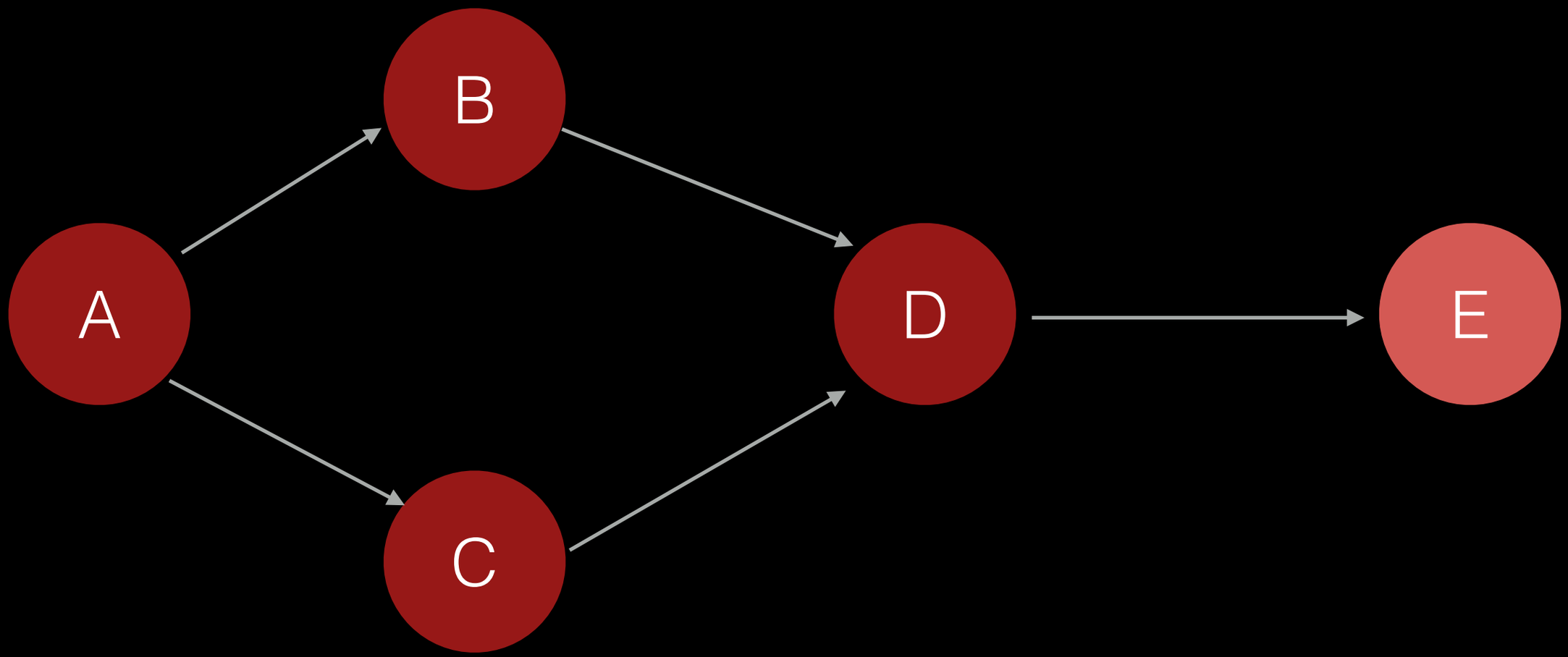

A
0
B
1
C
1



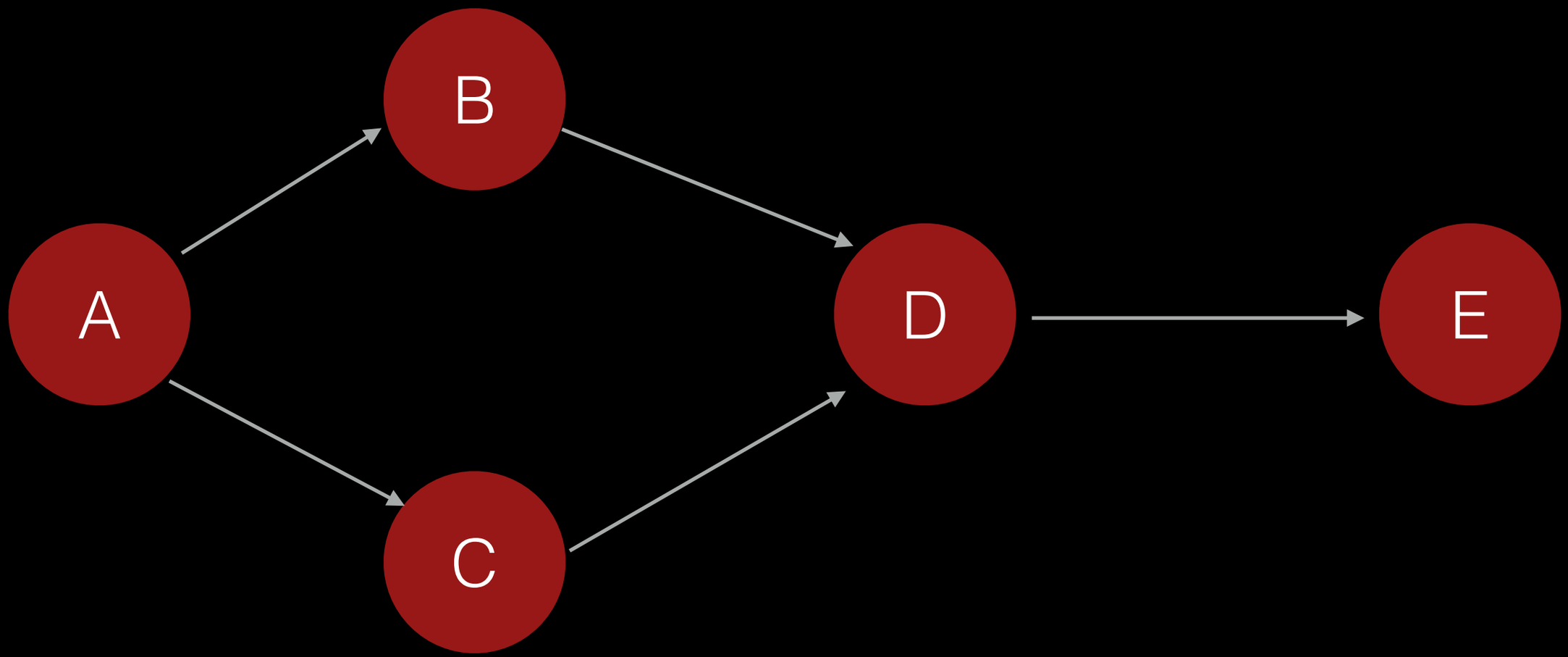
A	B	C	D
0	1	1	2


A
B
C
D
E


A
0
B
1
C
1
D
2



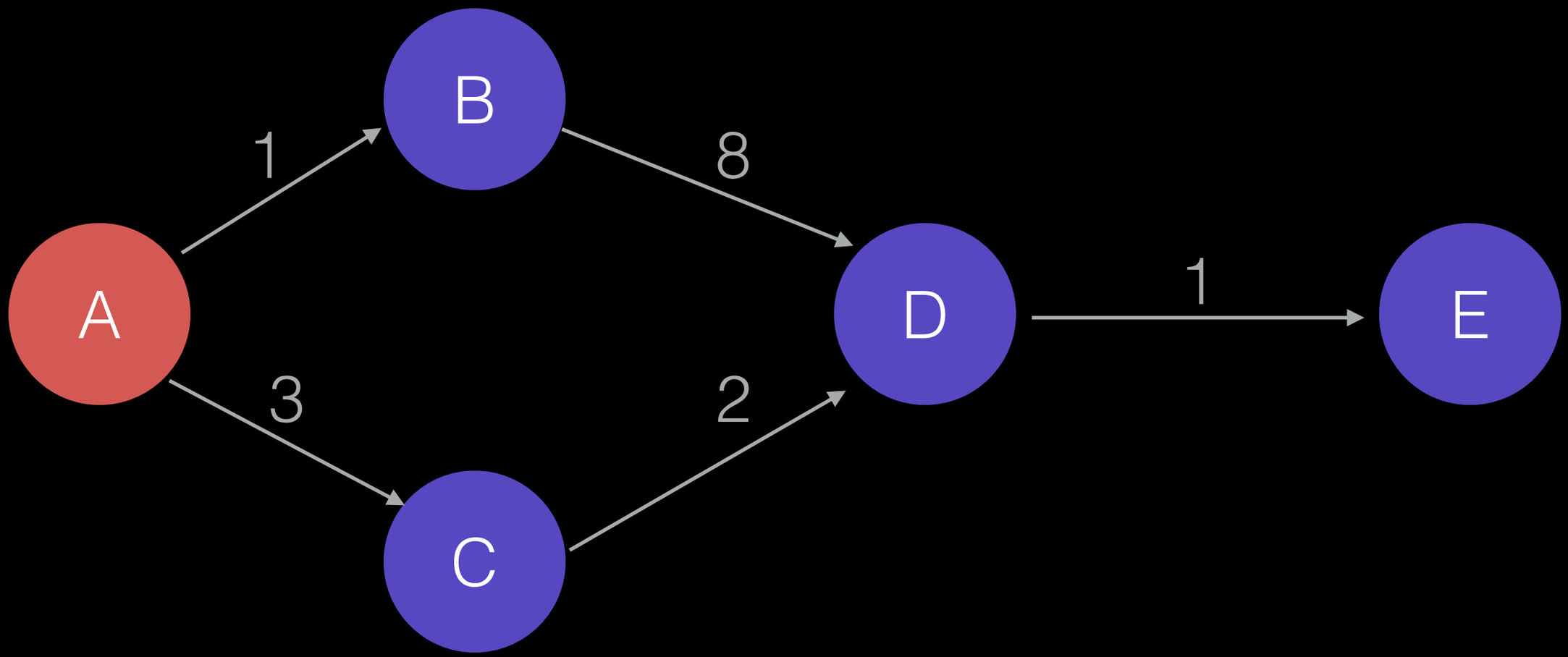
A	B	C	D	E
0	1	1	2	3



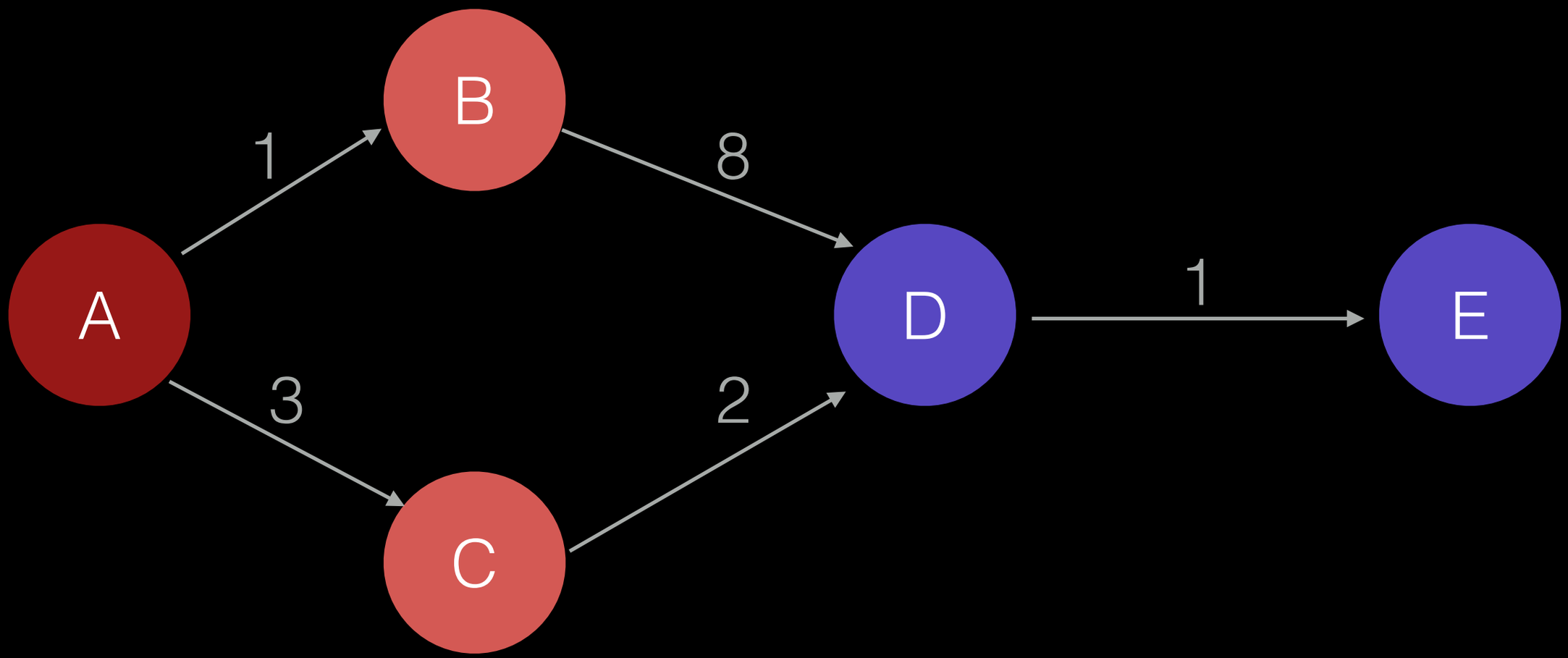
A	B	C	D	E
0	1	1	2	3

```java
import java.io.*;
import java.util.*;

public class bfs {

  public static void main(String[] args) {
    Node a = new Node();
    Node b = new Node();
    Node c = new Node();
    Node d = new Node();
    Node e = new Node();
    a.edges.add(b);
    a.edges.add(c);
    b.edges.add(d);
    c.edges.add(d);
    d.edges.add(e);
    ArrayList<Node> allNodes = new ArrayList<Node>(Arrays.asList(new Node[]{e, d, c, b, a}));
    bfs(a, allNodes);
    System.out.printf("Distance to Node e is: %d\n", e.distance);
  }

  public static void bfs(Node root, ArrayList<Node> allNodes) {
    Queue<Node> q = new LinkedList<Node>();
    root.distance = 0;
    q.add(root);

    while(q.size() > 0) {
      Node u = q.poll();
      for(Node n : u.edges) {
        if(n.distance == Integer.MAX_VALUE) { // Has not been visited yet
          n.distance = u.distance + 1;
          q.add(n);
        }
      }
    }
  }
}

class Node implements Comparable<Node> {
  public ArrayList<Node> edges = new ArrayList<Node>();
  public int distance = Integer.MAX_VALUE;

  public int compareTo(Node o) {
    return (distance < o.distance) ? -1 : ((distance == o.distance) ? 0 : 1);
  }
}
```

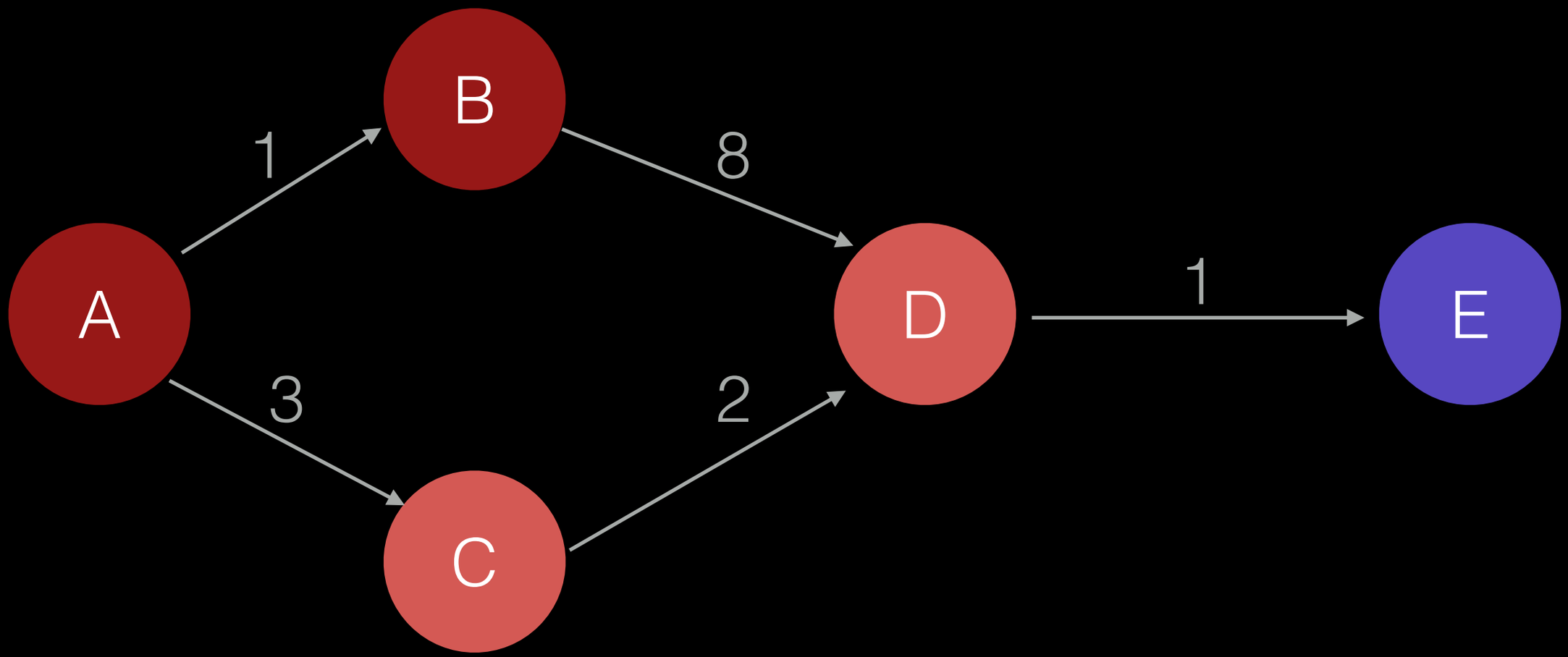
```java
public static void bfs(Node root, ArrayList<Node> allNodes) {
  Queue<Node> q = new LinkedList<Node>();
  root.distance = 0;
  q.add(root);

  while(q.size() > 0) {
    Node u = q.poll();
    for(Node n : u.edges) {
      if(n.distance == Integer.MAX_VALUE) { // Has not been visited yet
        n.distance = u.distance + 1;
        q.add(n);
      }
    }
  }
}
```



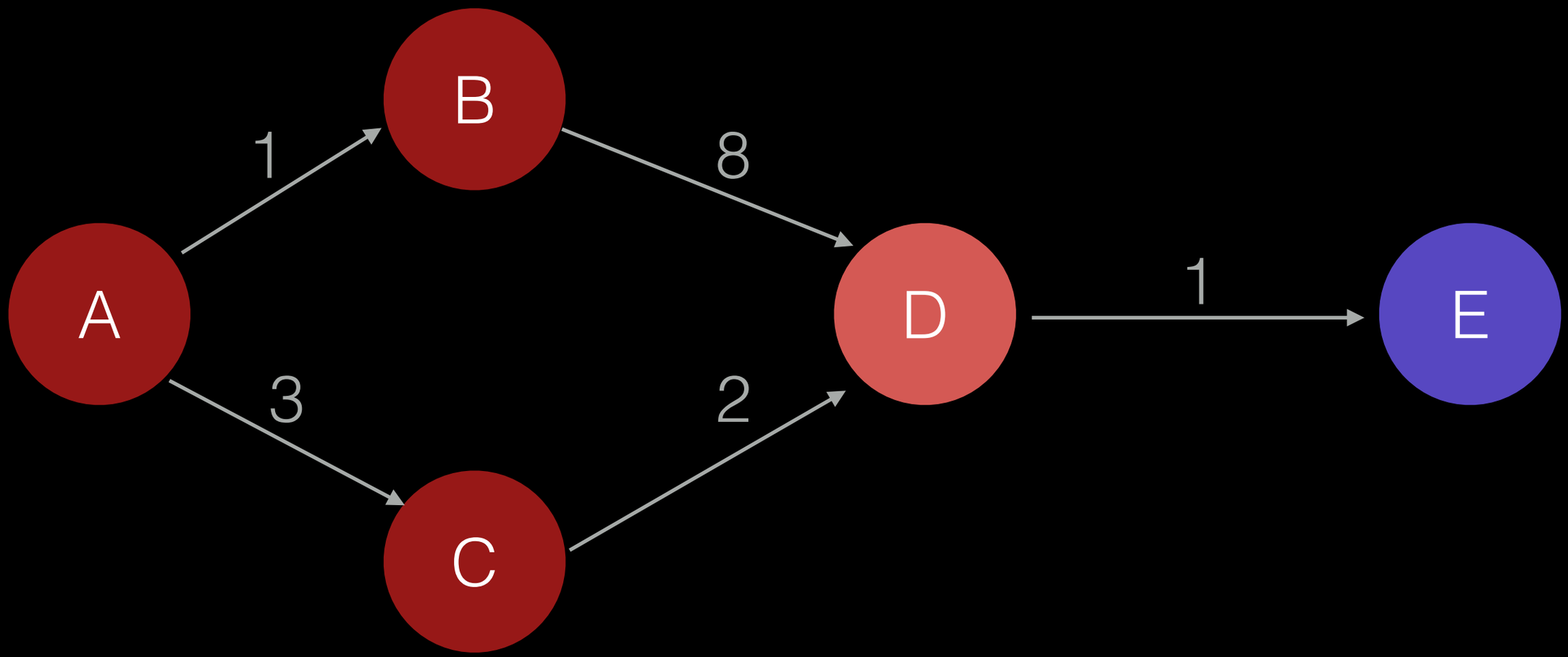
A 0



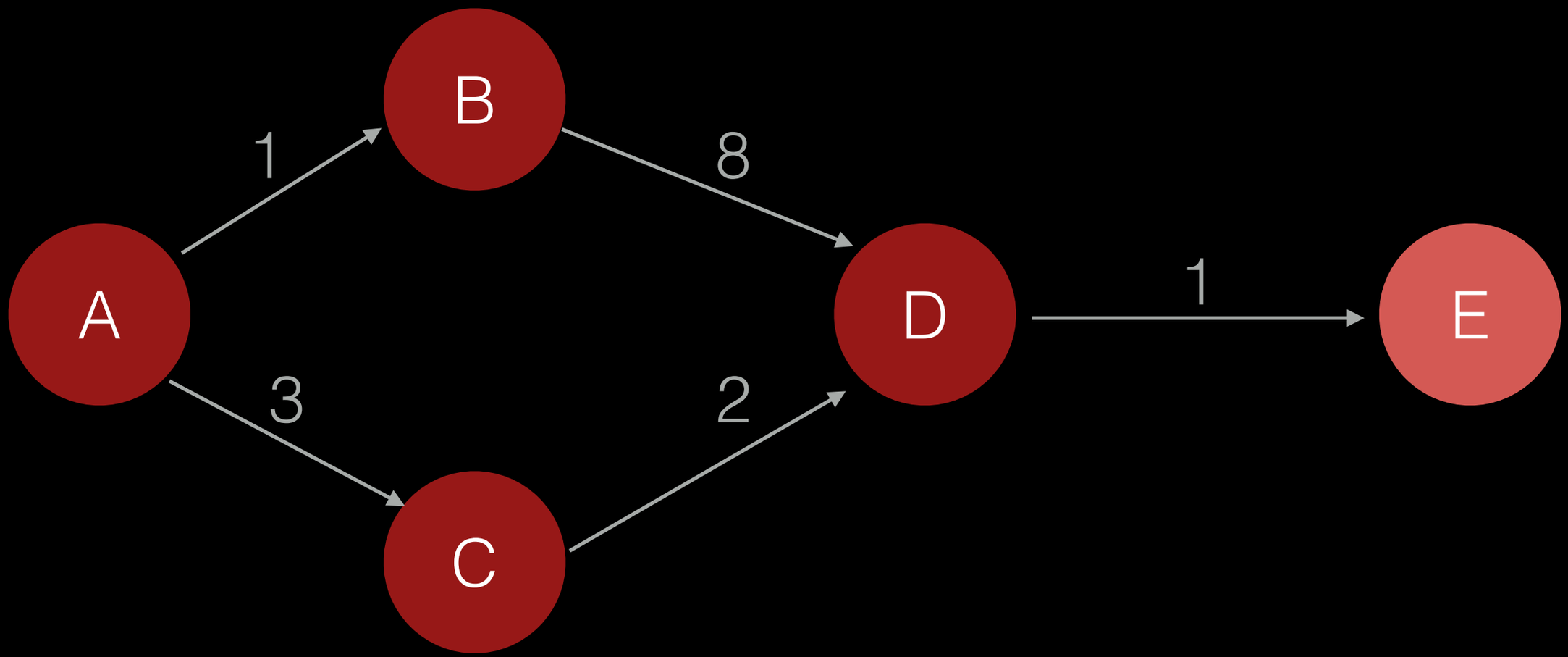
A	B	C
0	1	3



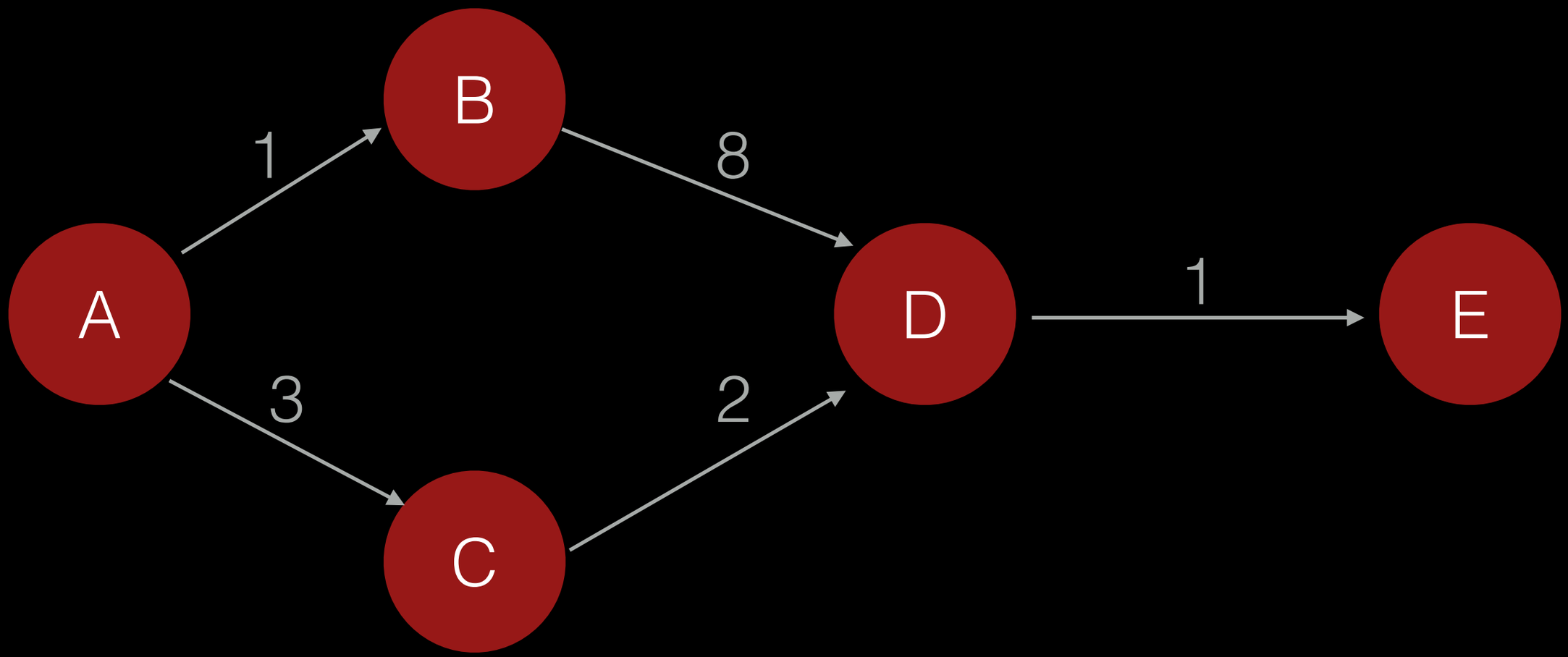
A	B	C	D
0	1	3	9



A	B	C	D
0	1	3	5



A	B	C	D	E
0	1	3	5	6



A	B	C	D	E
0	1	3	5	6

```java
import java.io.*;
import java.util.*;

public class dijkstra {

  public static void main(String[] args) {
    Node a = new Node();
    Node b = new Node();
    Node c = new Node();
    Node d = new Node();
    Node e = new Node();
    a.edges.put(b, 1);
    a.edges.put(c, 3);
    b.edges.put(d, 8);
    c.edges.put(d, 2);
    d.edges.put(e, 1);
    ArrayList<Node> allNodes = new ArrayList<Node>(Arrays.asList(new Node[]{e, d, c, b, a}));
    dijkstra(a, allNodes);
    System.out.printf("Distance to Node e is: %d\n", e.distance);
  }

  public static void dijkstra(Node root, ArrayList<Node> allNodes) {
    PriorityQueue<Node> q = new PriorityQueue<Node>();
    root.distance = 0;
    q.add(root);

    while(q.size() > 0) {
      Node u = q.poll();
      for(Node n : u.edges.keySet()) {
        if(n.distance == Integer.MAX_VALUE) { // Update the distance to node n
          q.remove(n);
        }
        n.distance = Math.min(n.distance, u.distance + u.edges.get(n));
        q.add(n);
      }
    }
  }
}

class Node implements Comparable<Node> {
  public HashMap<Node, Integer> edges = new HashMap<Node, Integer>();
  public int distance = Integer.MAX_VALUE;

  public int compareTo(Node o) {
    return (distance < o.distance) ? -1 : ((distance == o.distance) ? 0 : 1);
  }
}
```

```java
public static void dijkstra(Node root, ArrayList<Node> allNodes) {
  PriorityQueue<Node> q = new PriorityQueue<Node>();
  root.distance = 0;
  q.add(root);

  while(q.size() > 0) {
    Node u = q.poll();
    for(Node n : u.edges.keySet()) {
      if(n.distance == Integer.MAX_VALUE) { // Update the distance to node n
        q.remove(n);
      }
      n.distance = Math.min(n.distance, u.distance + u.edges.get(n));
      q.add(n);
    }
  }
}
```

```java
public static void bfs(Node root, ArrayList<Node> allNodes) {
  Queue<Node> q = new LinkedList<Node>();
  root.distance = 0;
  q.add(root);

  while(q.size() > 0) {
    Node u = q.poll();
    for(Node n : u.edges) {
      if(n.distance == Integer.MAX_VALUE) { // Has not been visited yet
        n.distance = u.distance + 1;
        q.add(n);
      }
    }
  }
}
```

```java
public static void dijkstra(Node root, ArrayList<Node> allNodes) {
  PriorityQueue<Node> q = new PriorityQueue<Node>();
  root.distance = 0;
  q.add(root);

  while(q.size() > 0) {
    Node u = q.poll();
    for(Node n : u.edges.keySet()) {
      if(n.distance == Integer.MAX_VALUE) { // Update the distance to node n
        q.remove(n);
      }
      n.distance = Math.min(n.distance, u.distance + u.edges.get(n));
      q.add(n);
    }
  }
}
```

```java
public static void bfs(Node root, ArrayList<Node> allNodes) {
  Queue<Node> q = new LinkedList<Node>();
  root.distance = 0;
  q.add(root);

  while(q.size() > 0) {
    Node u = q.poll();
    for(Node n : u.edges) {
      if(n.distance == Integer.MAX_VALUE) { // Has not been visited yet
        n.distance = u.distance + 1;
        q.add(n);
      }
    }
  }
}
```

```java
public static void dijkstra(Node root, ArrayList<Node> allNodes) {
  PriorityQueue<Node> q = new PriorityQueue<Node>();
  root.distance = 0;
  q.add(root);

  while(q.size() > 0) {
    Node u = q.poll();
    for(Node n : u.edges.keySet()) {
      if(n.distance == Integer.MAX_VALUE) { // Update the distance to node n
        q.remove(n);
      }
      n.distance = Math.min(n.distance, u.distance + u.edges.get(n));
      q.add(n);
    }
  }
}
```

```java
public static void bfs(Node root, ArrayList<Node> allNodes) {
  Queue<Node> q = new LinkedList<Node>();
  root.distance = 0;
  q.add(root);

  while(q.size() > 0) {
    Node u = q.poll();
    for(Node n : u.edges) {
      if(n.distance == Integer.MAX_VALUE) { // Has not been visited yet
        n.distance = u.distance + 1;
        q.add(n);
      }
    }
  }
}
```

```java
public static void dijkstra(Node root, ArrayList<Node> allNodes) {
  PriorityQueue<Node> q = new PriorityQueue<Node>();
  root.distance = 0;
  q.add(root);

  while(q.size() > 0) {
    Node u = q.poll();
    for(Node n : u.edges.keySet()) {
      if(n.distance == Integer.MAX_VALUE) { // Update the distance to node n
        q.remove(n);
      }
      n.distance = Math.min(n.distance, u.distance + u.edges.get(n));
      q.add(n);
    }
  }
}
```

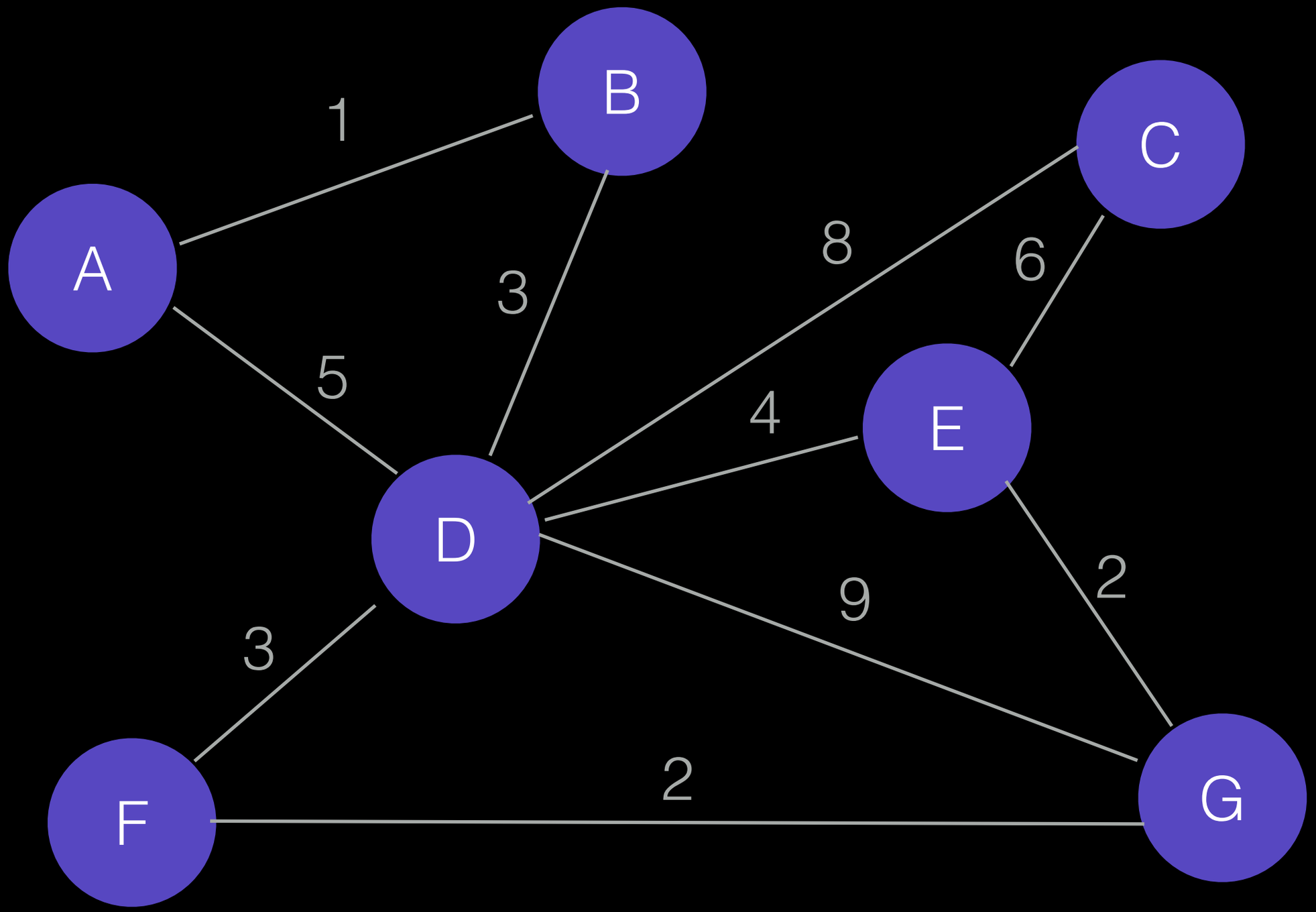
```java
public static void bfs(Node root, ArrayList<Node> allNodes) {
  Queue<Node> q = new LinkedList<Node>();
  root.distance = 0;
  q.add(root);

  while(q.size() > 0) {
    Node u = q.poll();
    for(Node n : u.edges) {
      if(n.distance == Integer.MAX_VALUE) { // Has not been visited yet
        n.distance = u.distance + 1;
        q.add(n);
      }
    }
  }
}
```

```java
public static void dijkstra(Node root, ArrayList<Node> allNodes) {
  PriorityQueue<Node> q = new PriorityQueue<Node>();
  root.distance = 0;
  q.add(root);

  while(q.size() > 0) {
    Node u = q.poll();
    for(Node n : u.edges.keySet()) {
      if(n.distance == Integer.MAX_VALUE) { // Update the distance to node n
        q.remove(n);
      }
      n.distance = Math.min(n.distance, u.distance + u.edges.get(n));
      q.add(n);
    }
  }
}
```

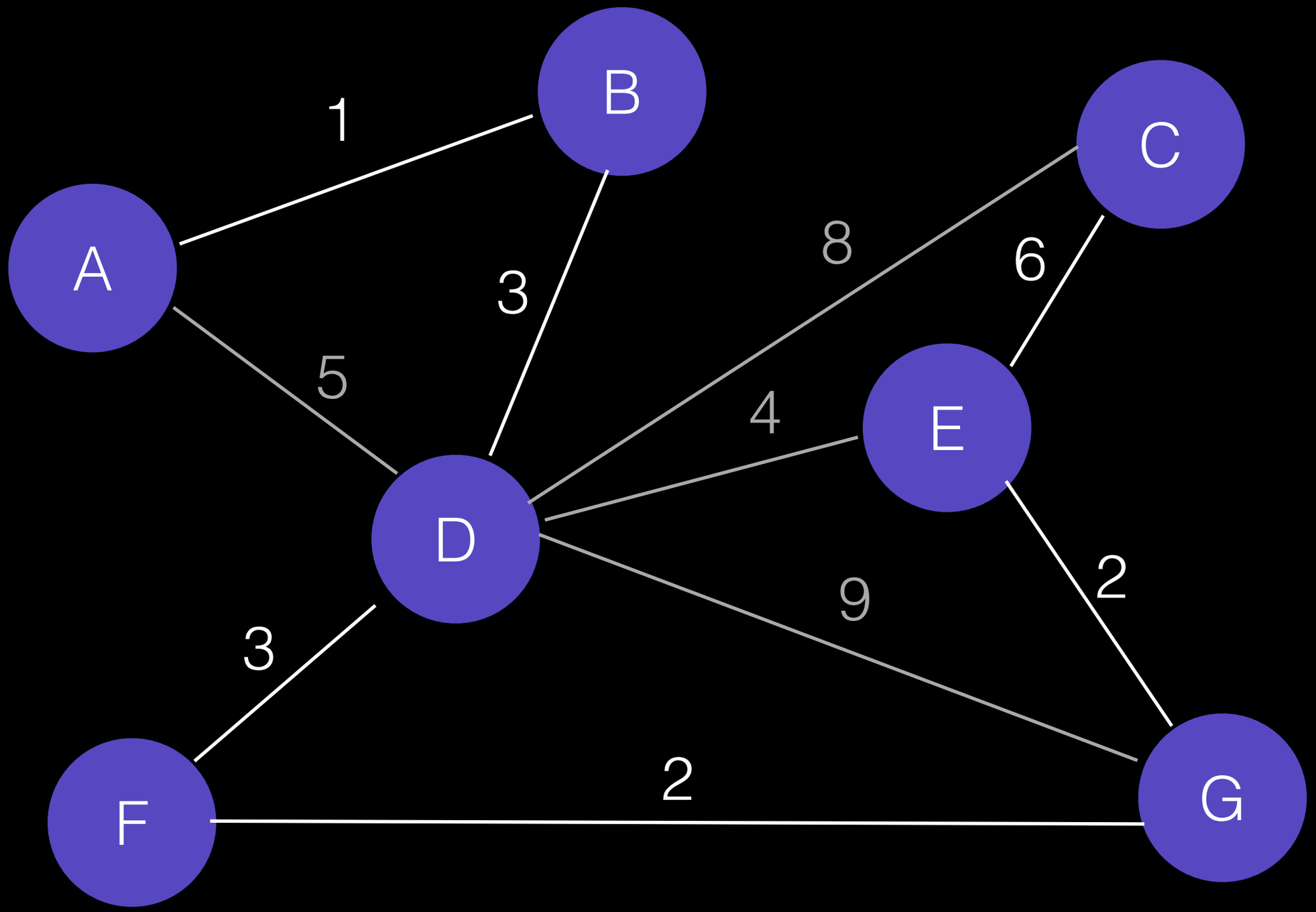
```java
public static void bfs(Node root, ArrayList<Node> allNodes) {
  Queue<Node> q = new LinkedList<Node>();
  root.distance = 0;
  q.add(root);

  while(q.size() > 0) {
    Node u = q.poll();
    for(Node n : u.edges) {
      if(n.distance == Integer.MAX_VALUE) { // Has not been visited yet
        n.distance = u.distance + 1;
        q.add(n);
      }
    }
  }
}
```

```java
public static void dijkstra(Node root, ArrayList<Node> allNodes) {
  PriorityQueue<Node> q = new PriorityQueue<Node>();
  root.distance = 0;
  q.add(root);

  while(q.size() > 0) {
    Node u = q.poll();
    for(Node n : u.edges.keySet()) {
      if(n.distance == Integer.MAX_VALUE) { // Update the distance to node n
        q.remove(n);
      }
      n.distance = Math.min(n.distance, u.distance + u.edges.get(n));
      q.add(n);
    }
  }
}
```

Minimum Spanning Tree

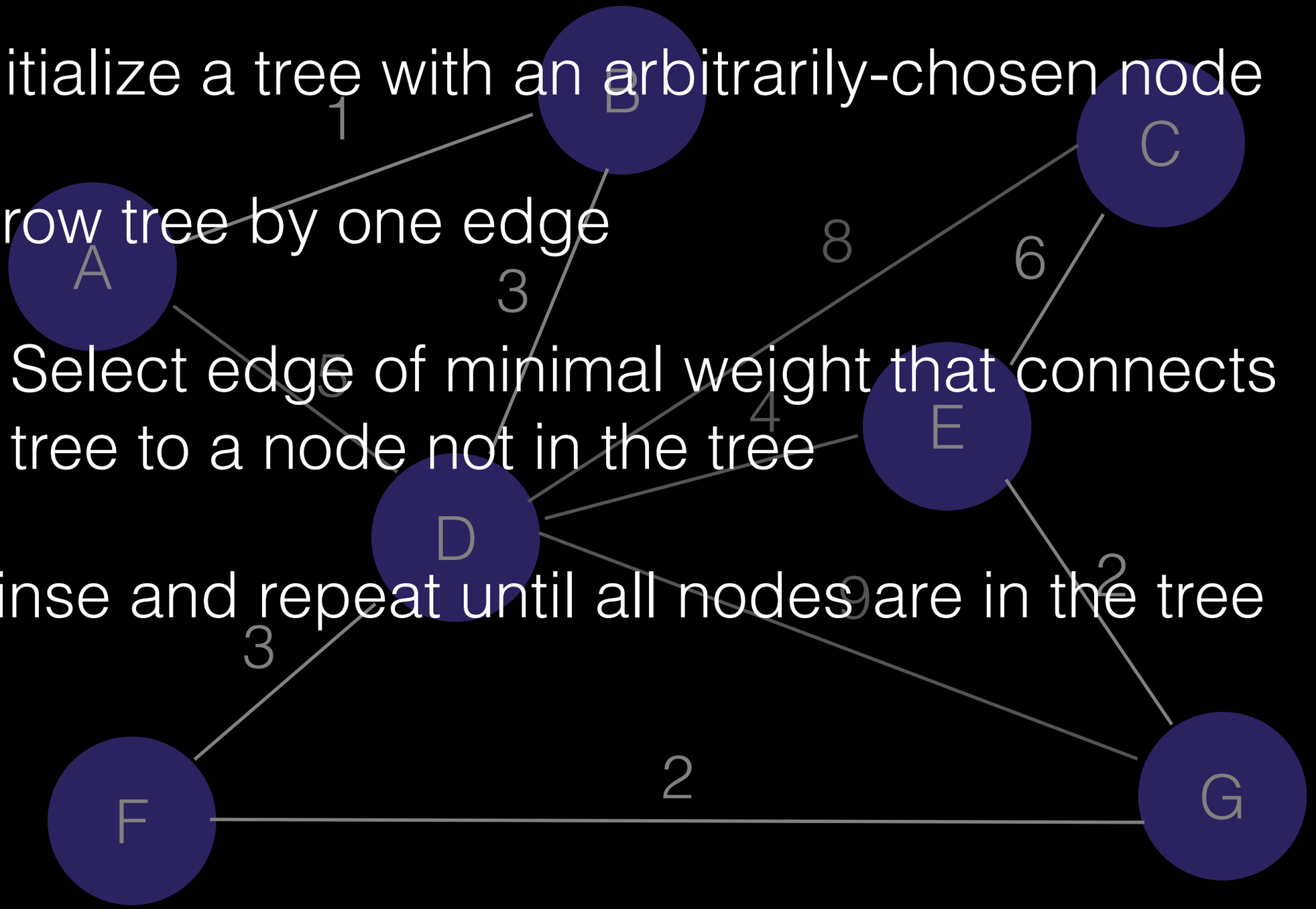


Minimum Spanning Tree

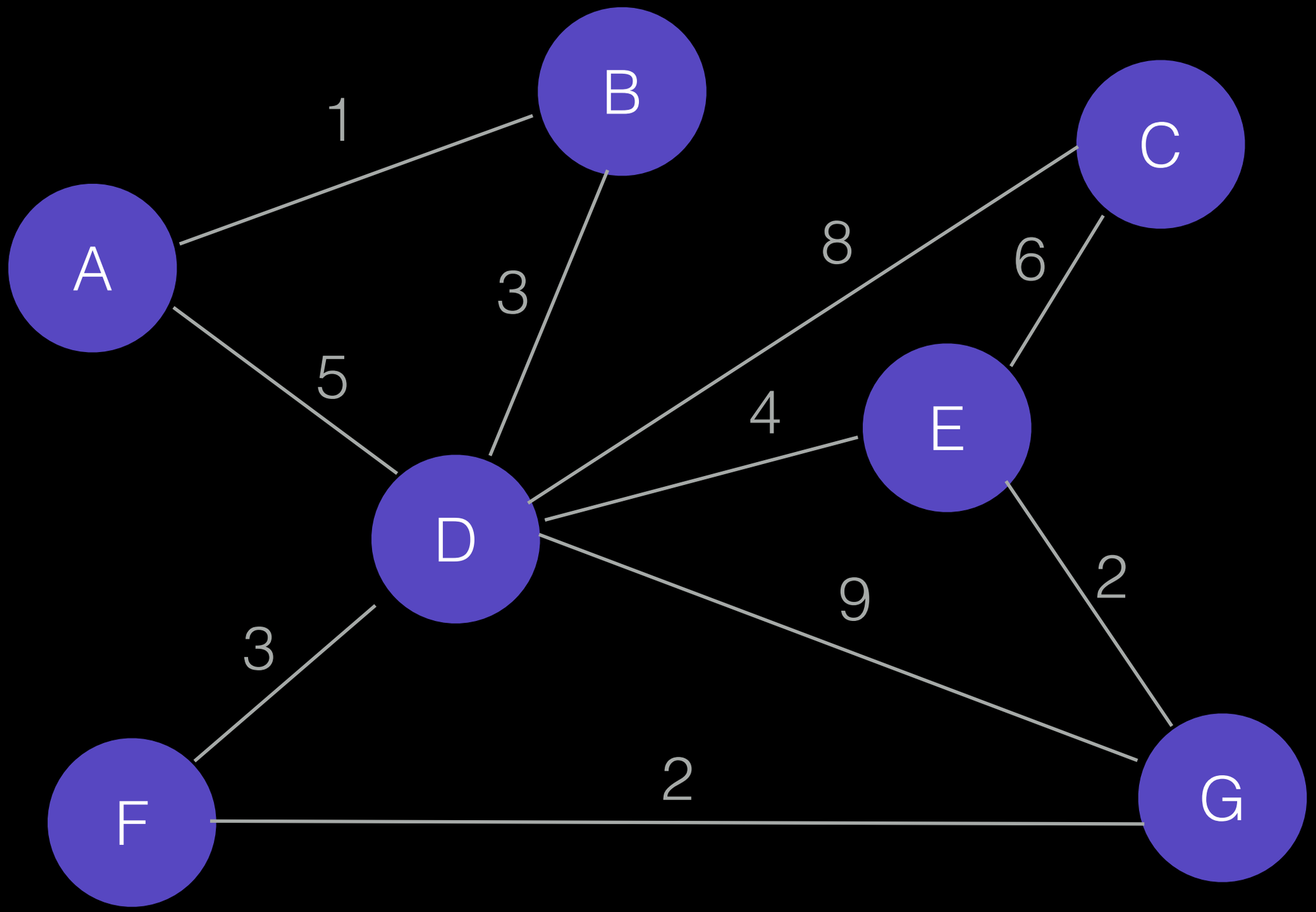


Prim's Algorithm

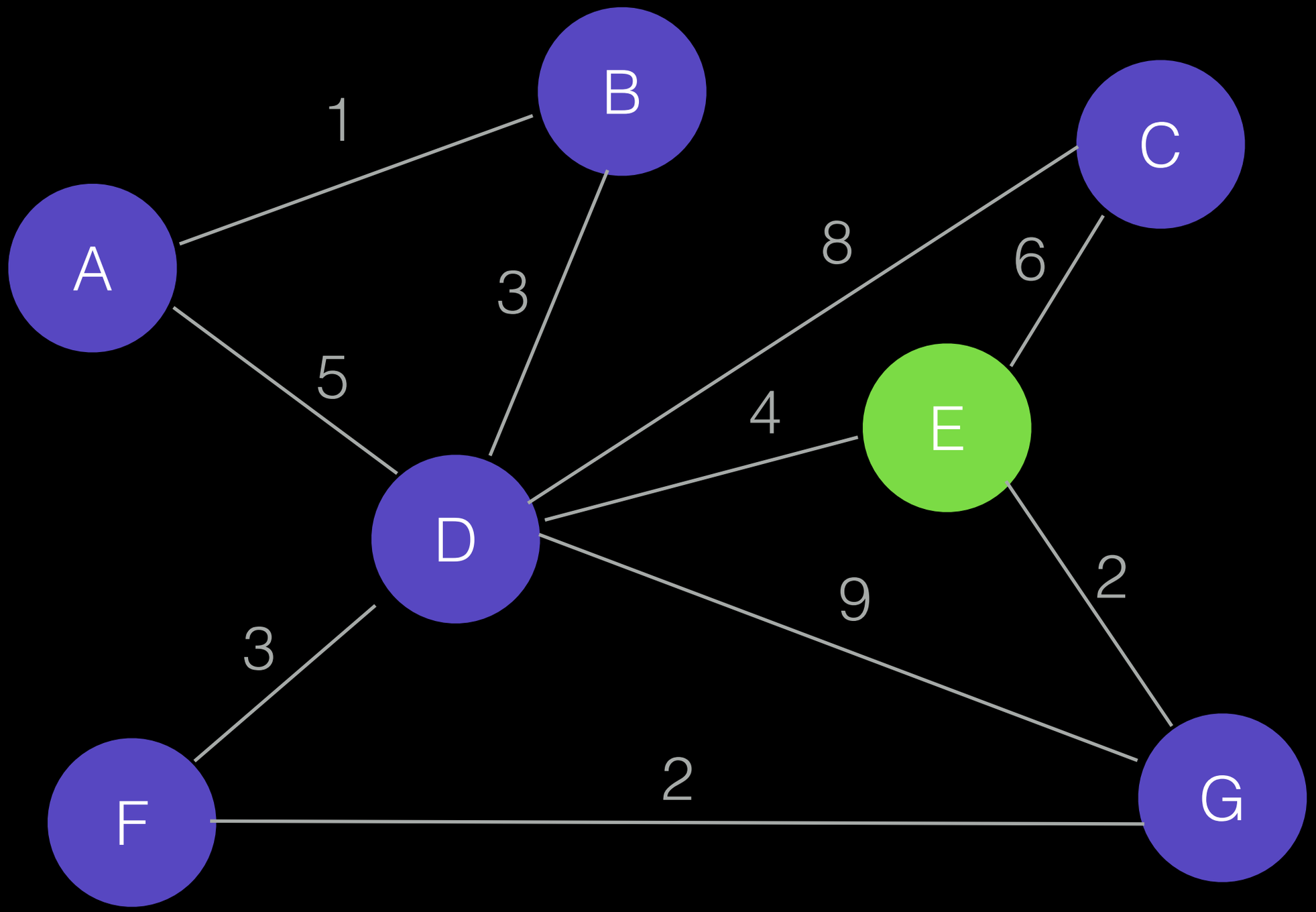
- Initialize a tree with an arbitrarily-chosen node
- Grow tree by one edge
  - Select edge of minimal weight that connects tree to a node not in the tree
- Rinse and repeat until all nodes are in the tree



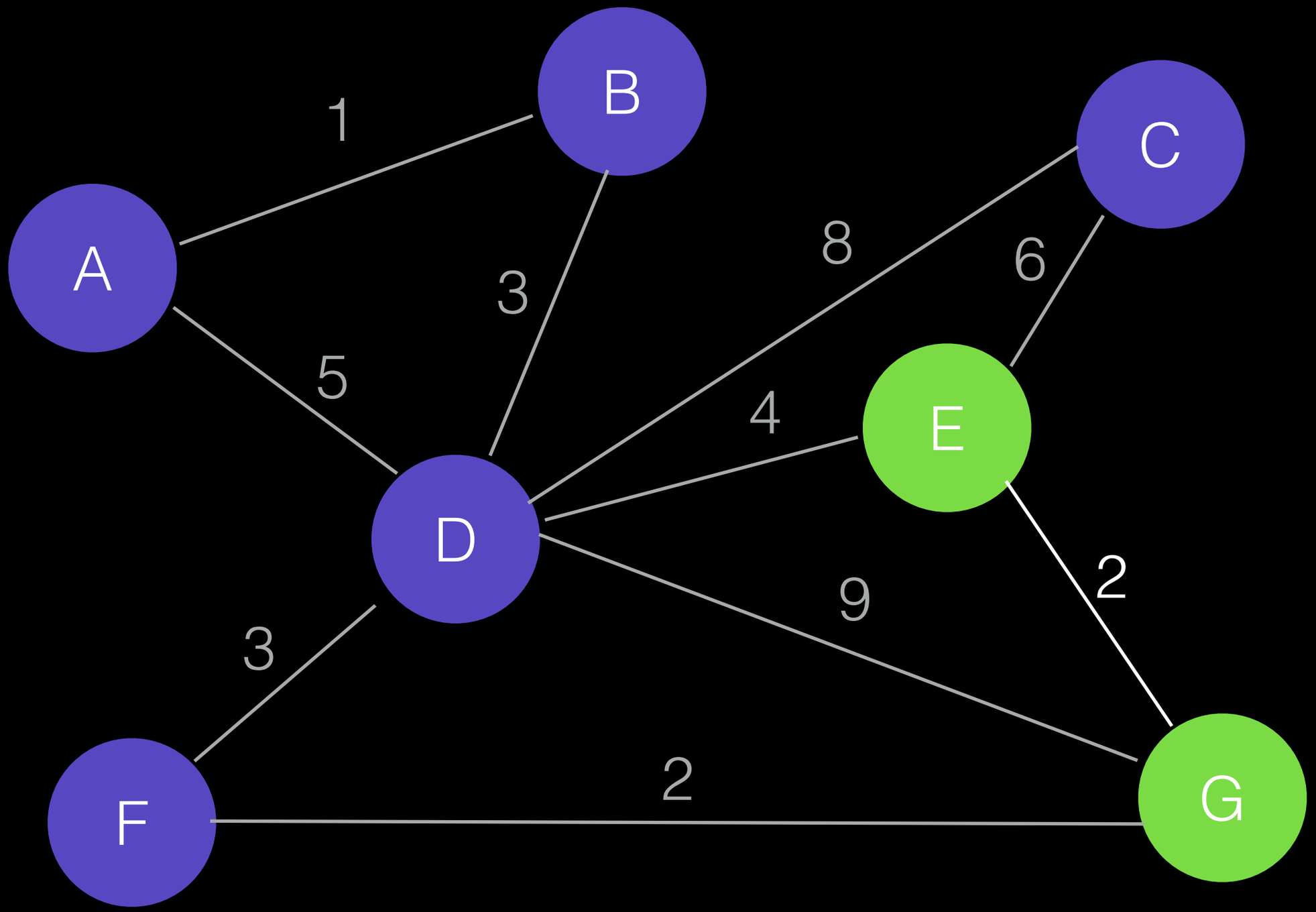
Prim’s Algorithm



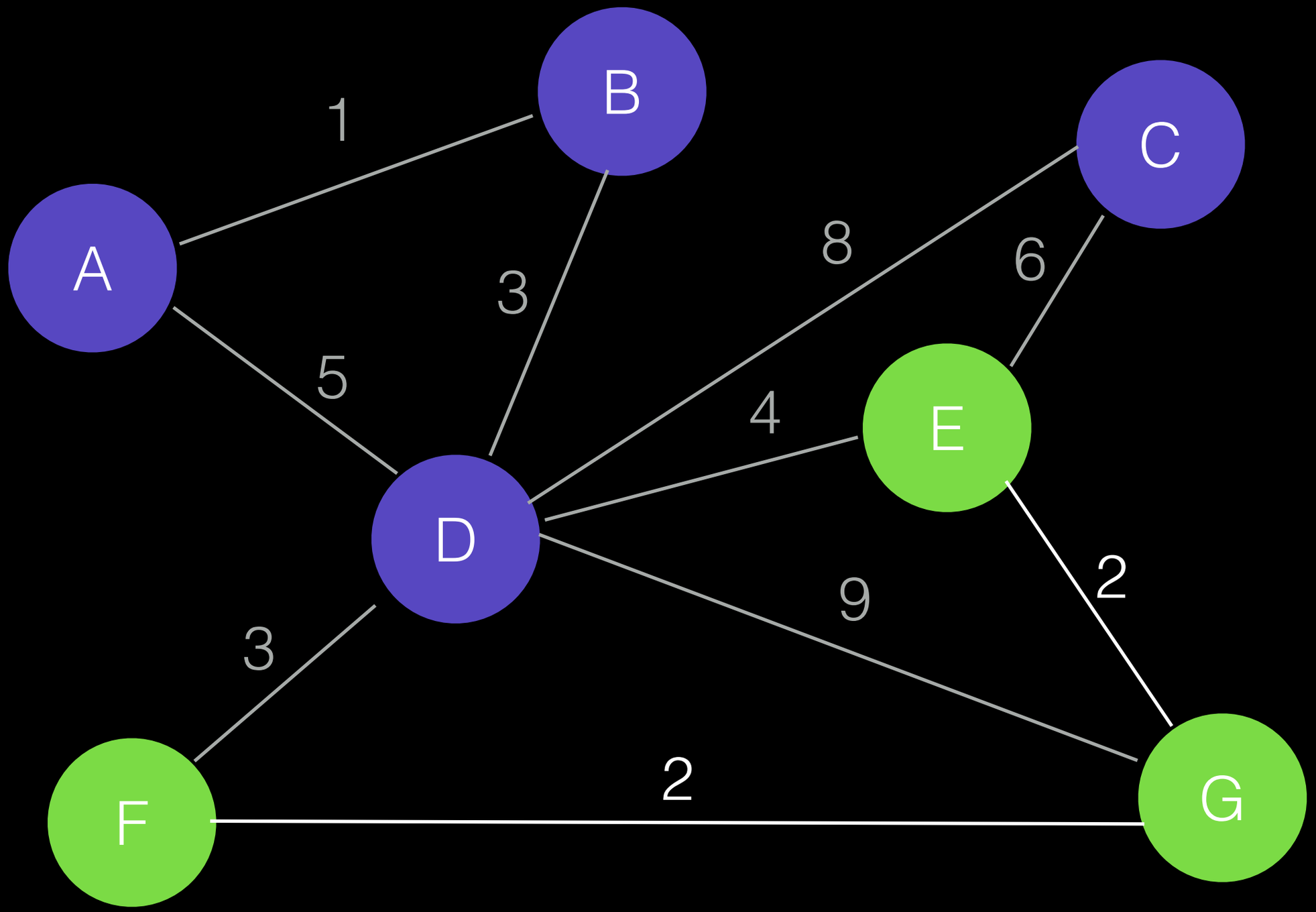
Prim’s Algorithm



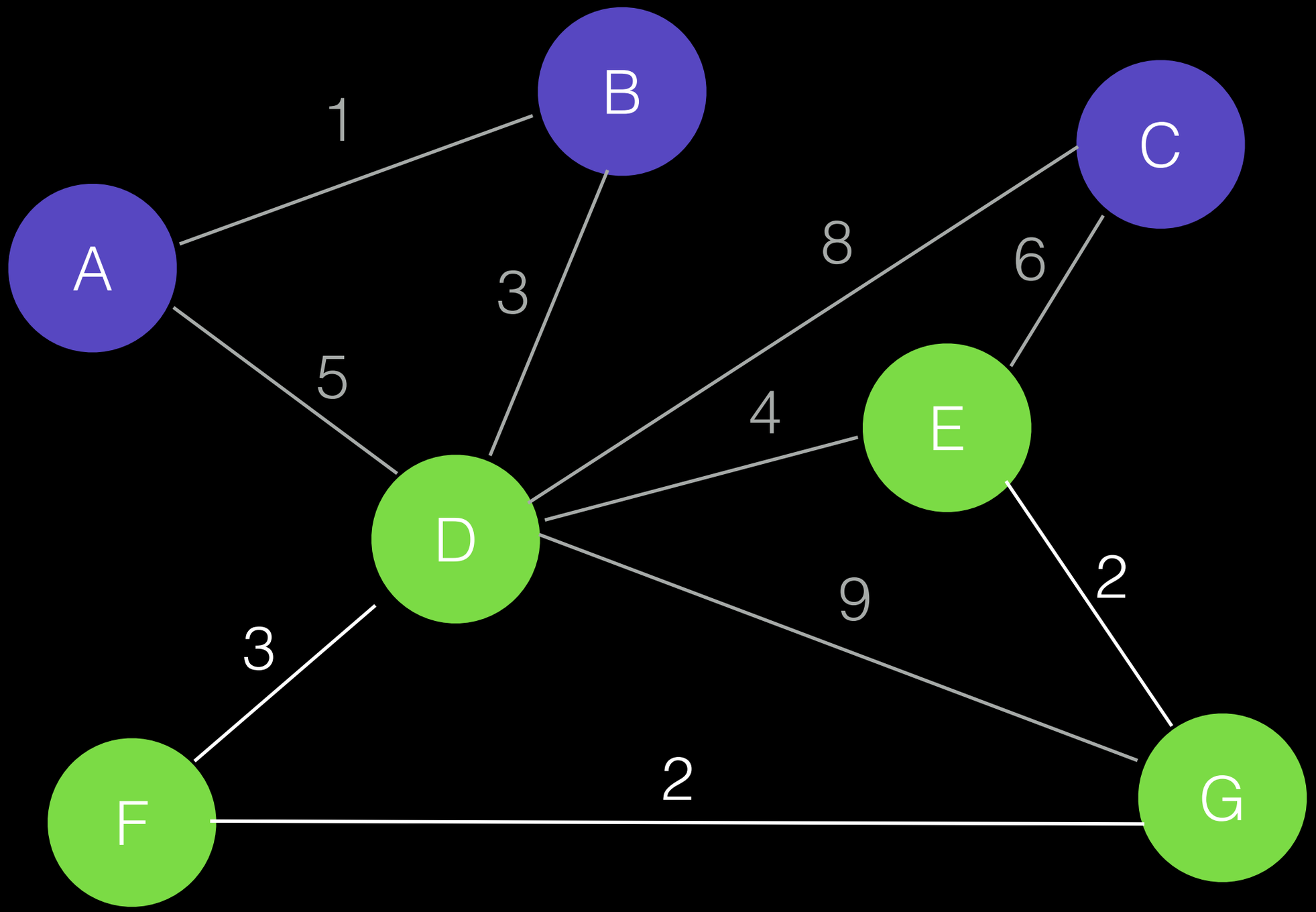
Prim’s Algorithm



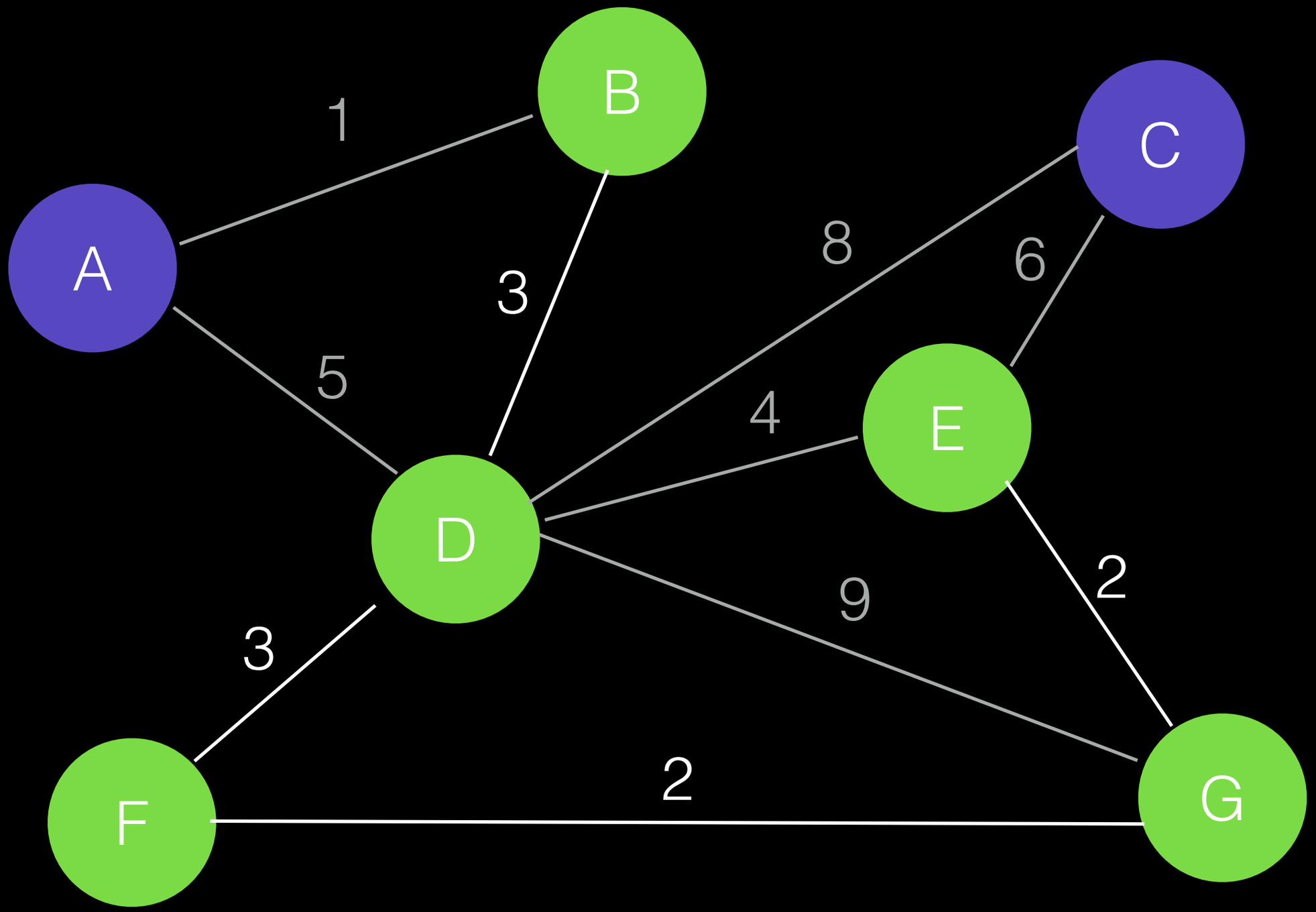
Prim’s Algorithm



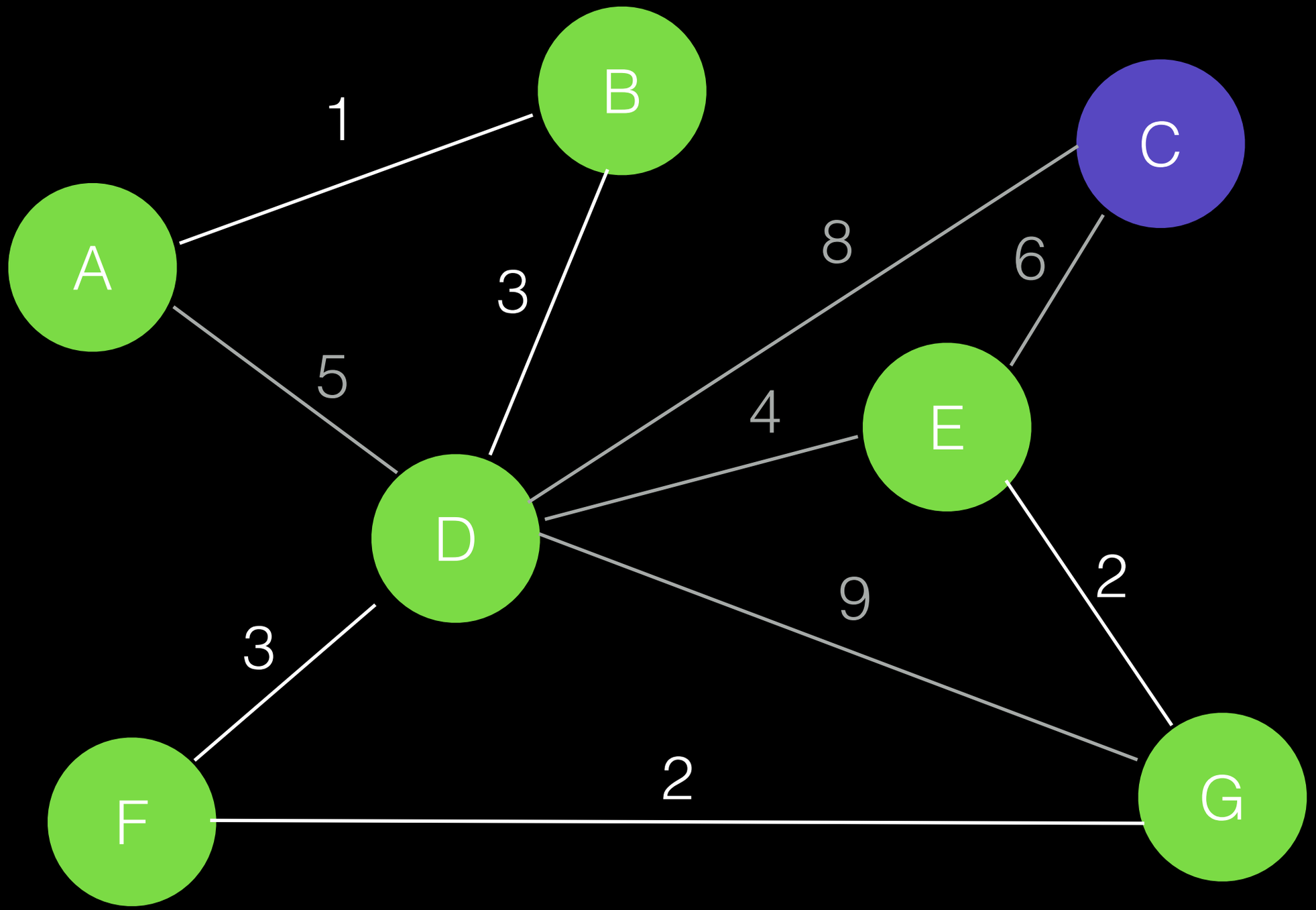
Prim’s Algorithm



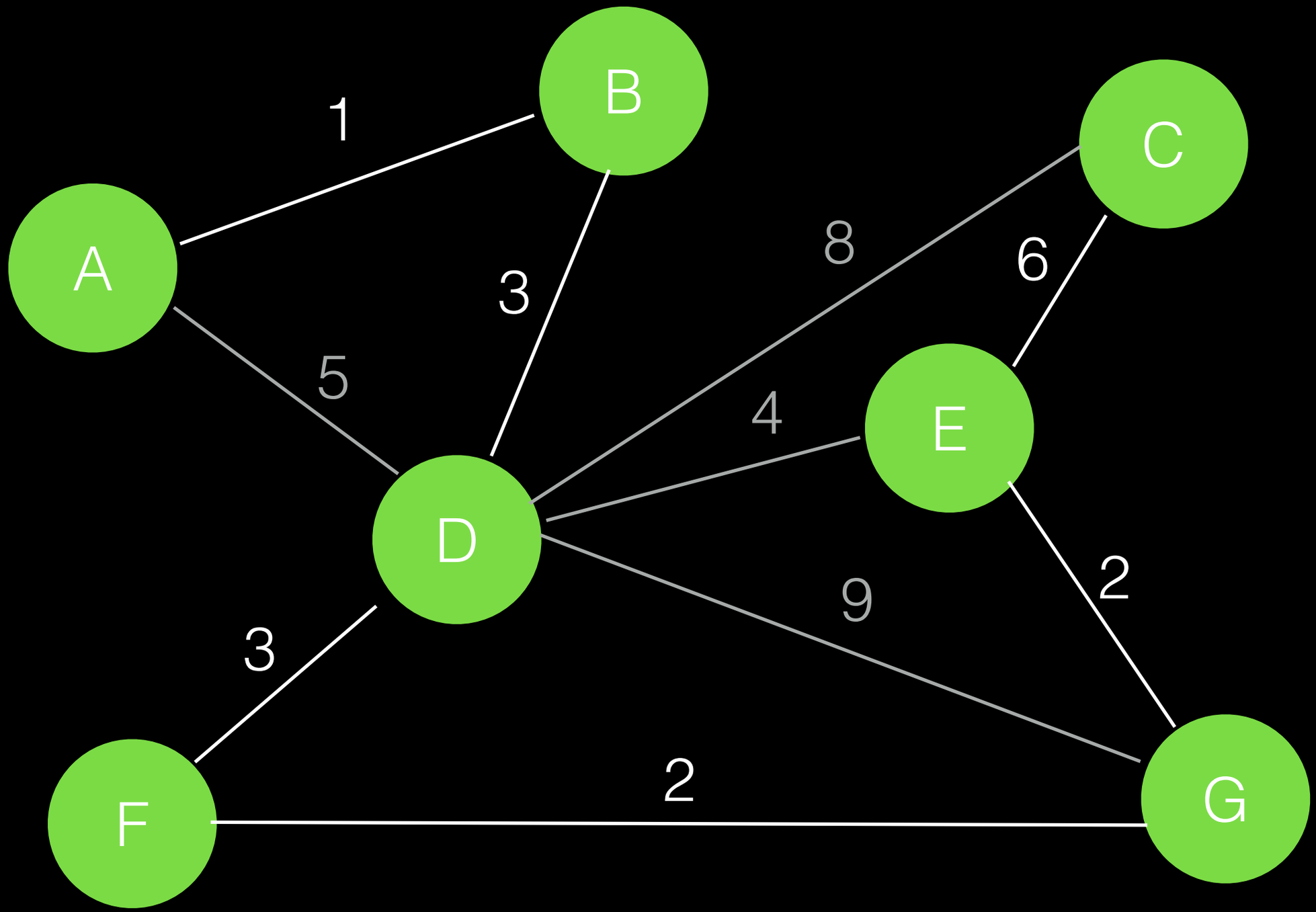
Prim’s Algorithm



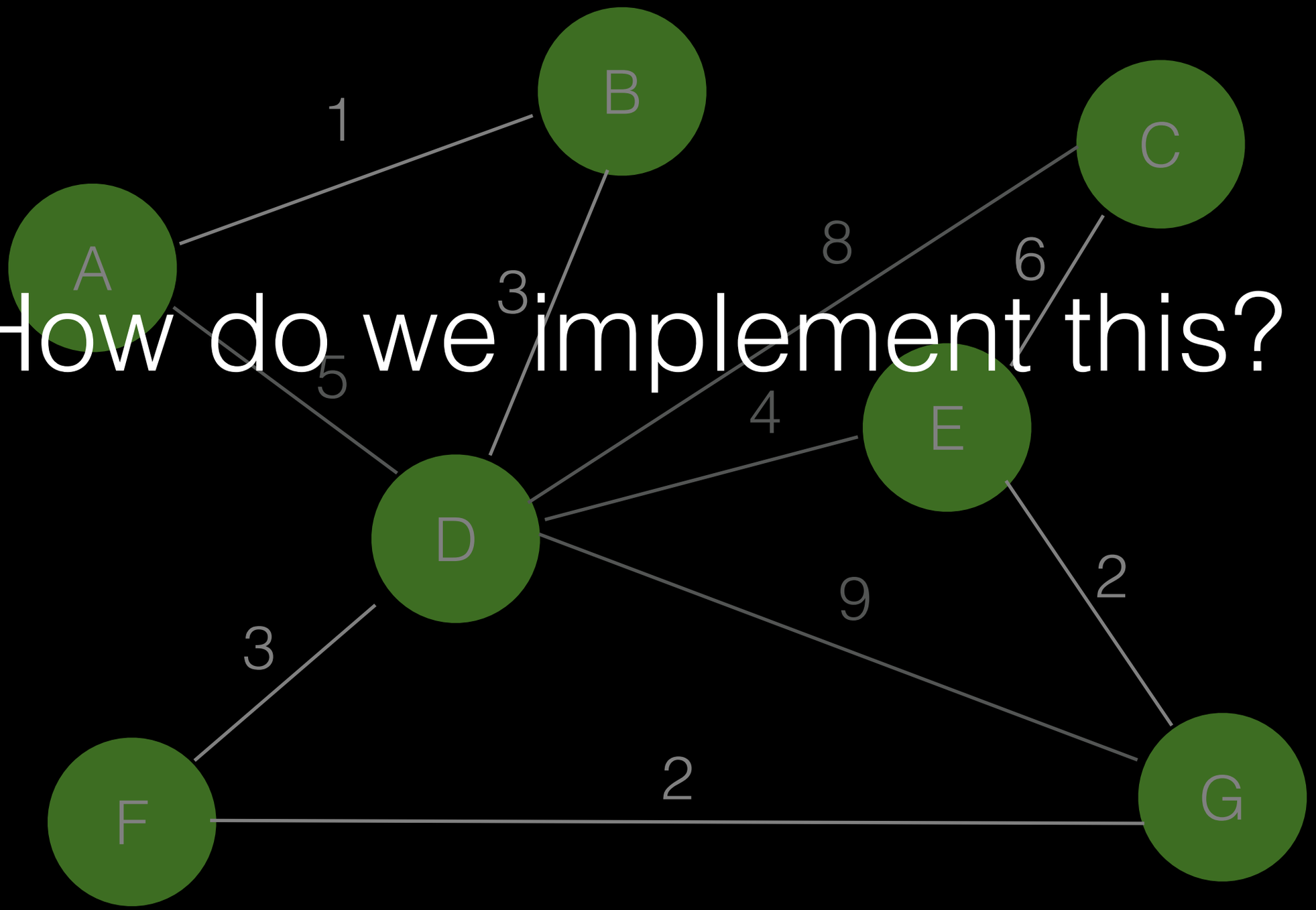
Prim's Algorithm



Prim's Algorithm



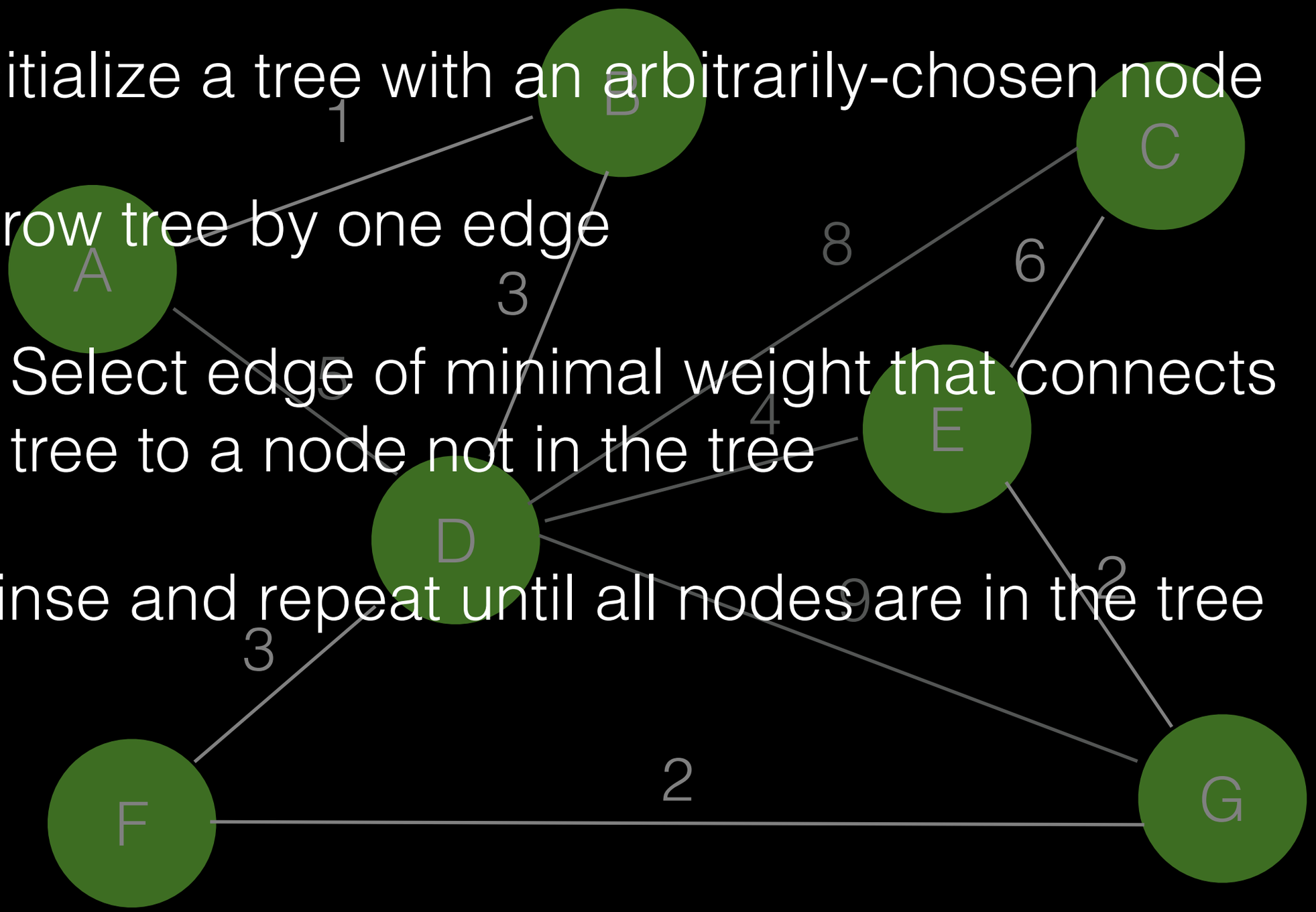
Prim's Algorithm



How do we implement this?

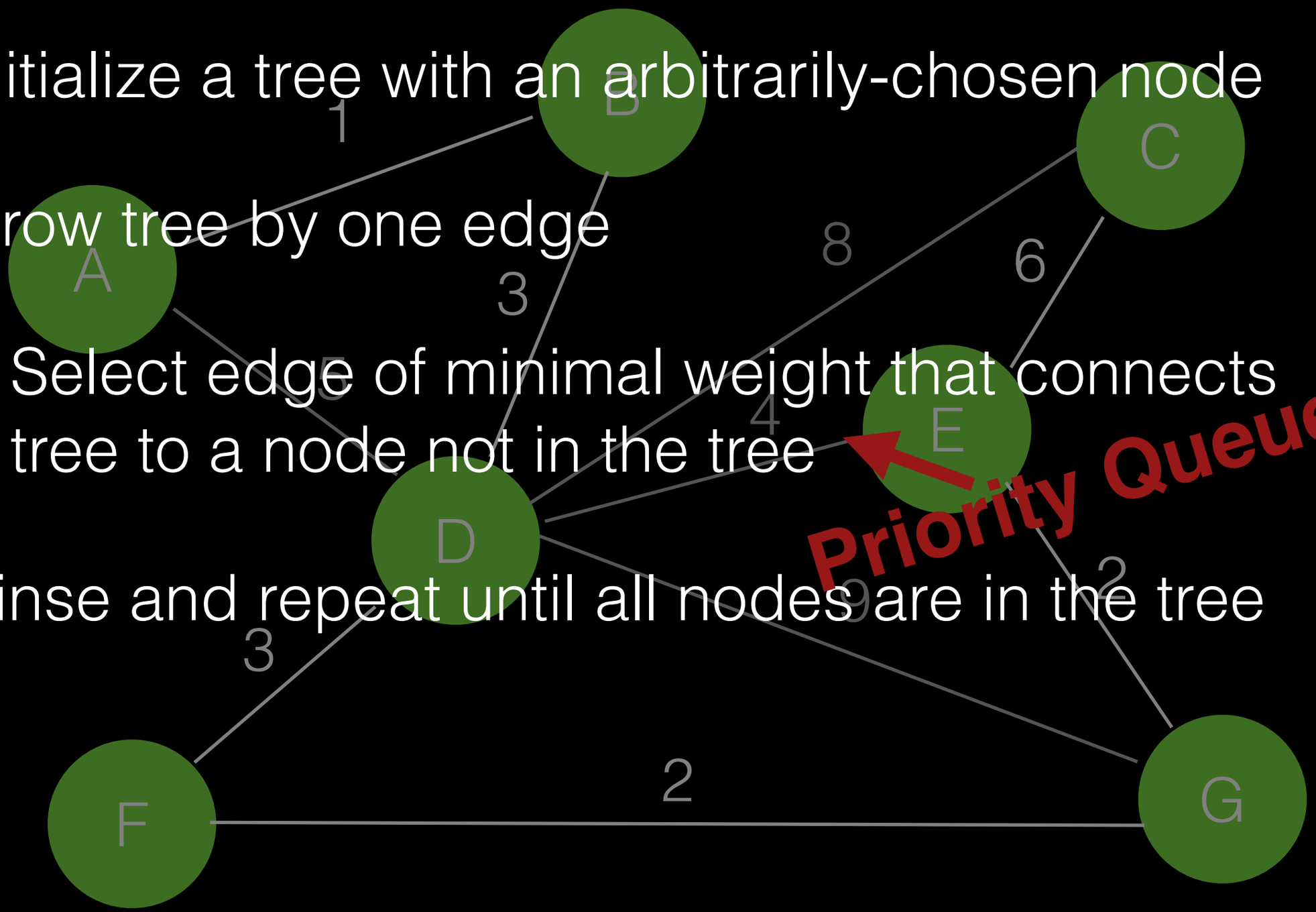
Prim's Algorithm

- Initialize a tree with an arbitrarily-chosen node
- Grow tree by one edge
  - Select edge of minimal weight that connects tree to a node not in the tree
- Rinse and repeat until all nodes are in the tree



Prim's Algorithm

- Initialize a tree with an arbitrarily-chosen node
- Grow tree by one edge
  - Select edge of minimal weight that connects tree to a node not in the tree
- Rinse and repeat until all nodes are in the tree



Prim's Algorithm

```
public static void prims(ArrayList<Node> allNodes) {
  PriorityQueue<Node> q = new PriorityQueue<Node>();
  Node root = allNodes.get(0);
  root.distance = 0;
  q.add(root);

  while(!q.isEmpty()) {
    Node cur = q.poll();
    cur.visited = true;
    for(Node n : cur.edges.keySet()) {
      if(!n.visited && n.distance > cur.edges.get(n)) {
        n.distance = cur.edges.get(n);
        q.remove(n);
        q.add(n);
      }
    }
  }
}
```

Graph algorithms in their many shapes and sizes

Daniel Epstein, 5/8/14
depstein@cs.washington.edu
https://github.com/depstein/programming-competitions