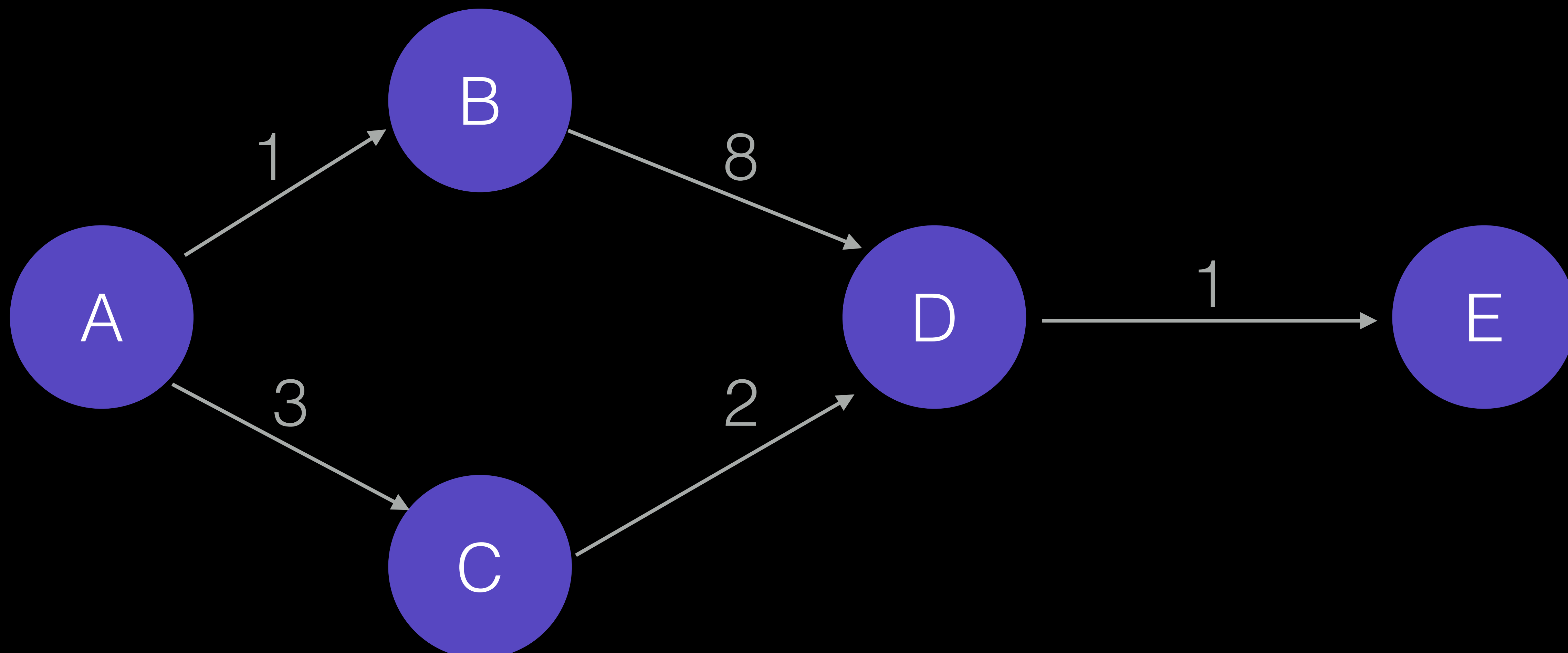
Graph algorithms in their many shapes and sizes

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

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*,

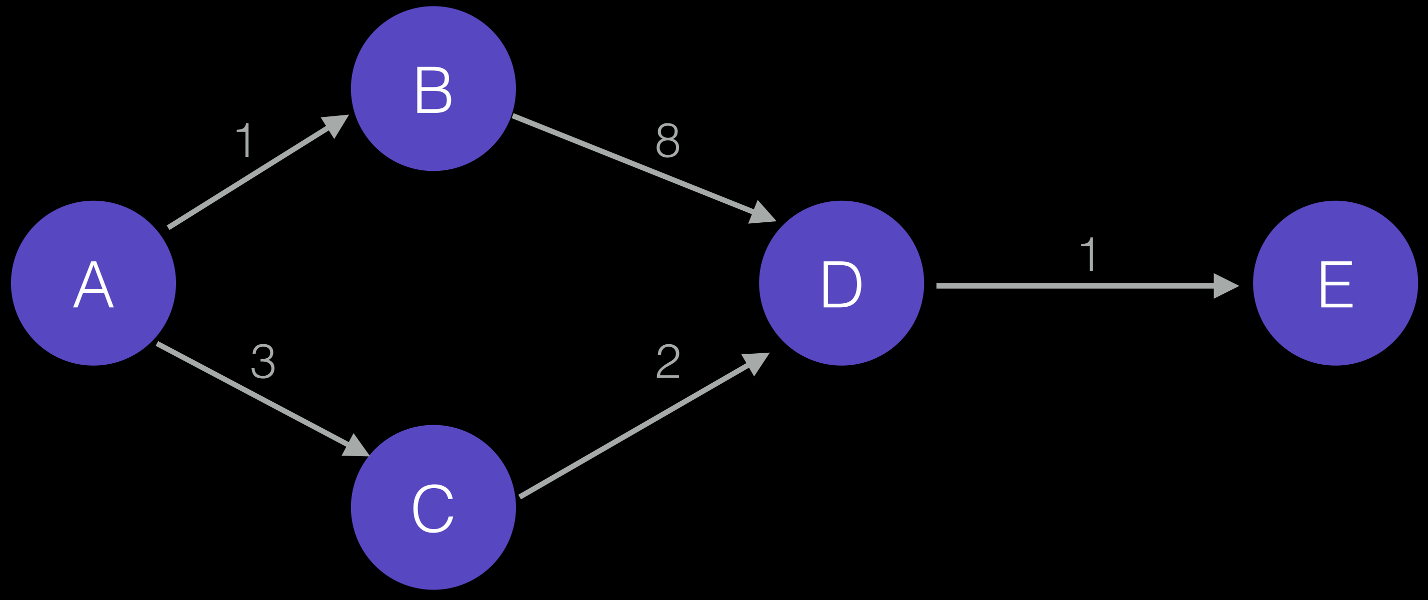
Nodes and Edges!

Graph Representation



Arrays

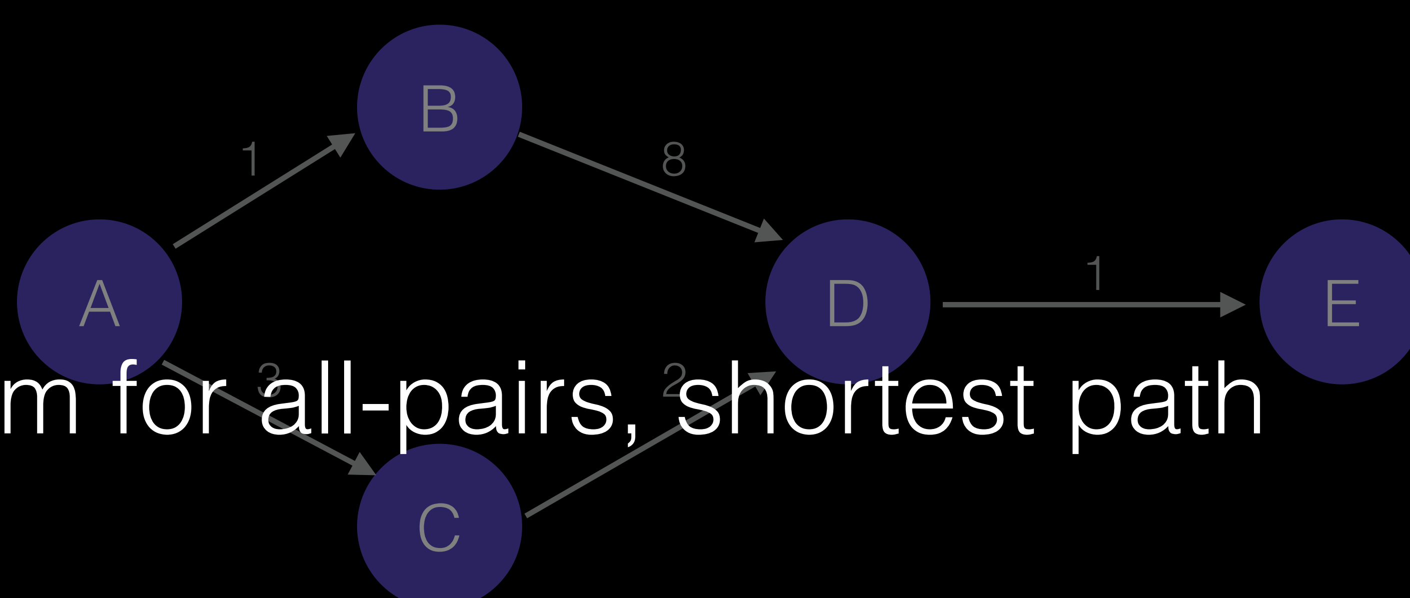
```
import java.io.*;
import java.util.*;

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

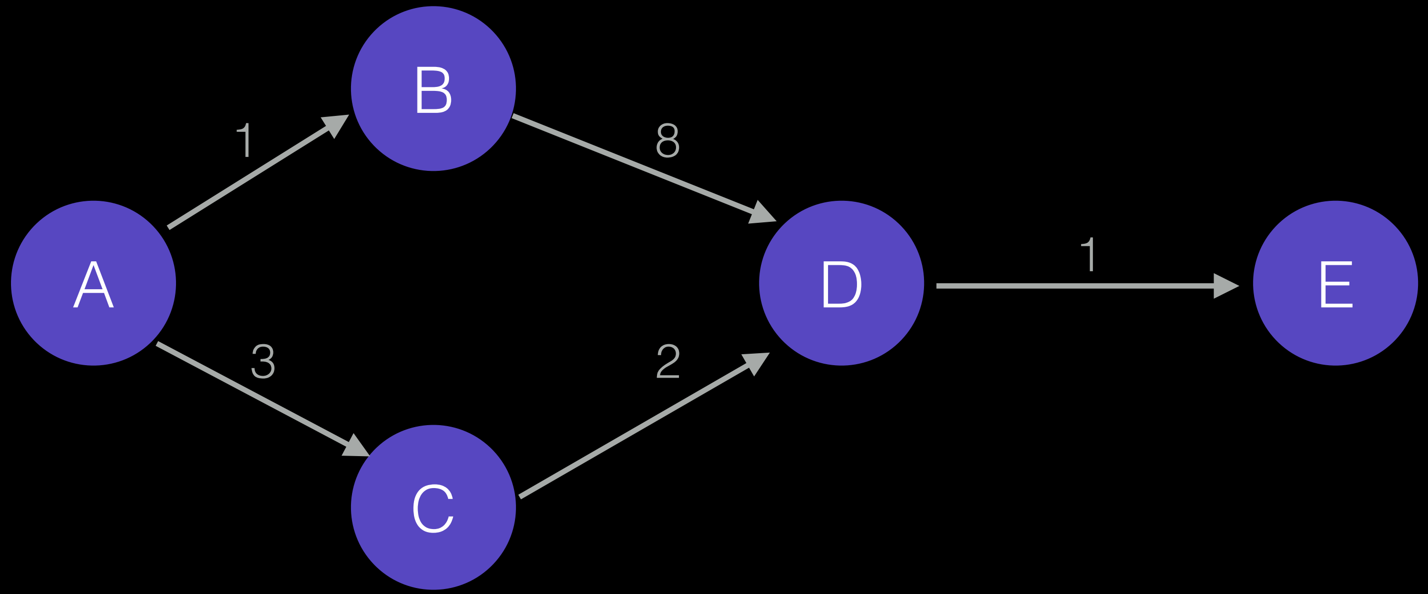
```
import java.io.*;
import java.util.*;

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

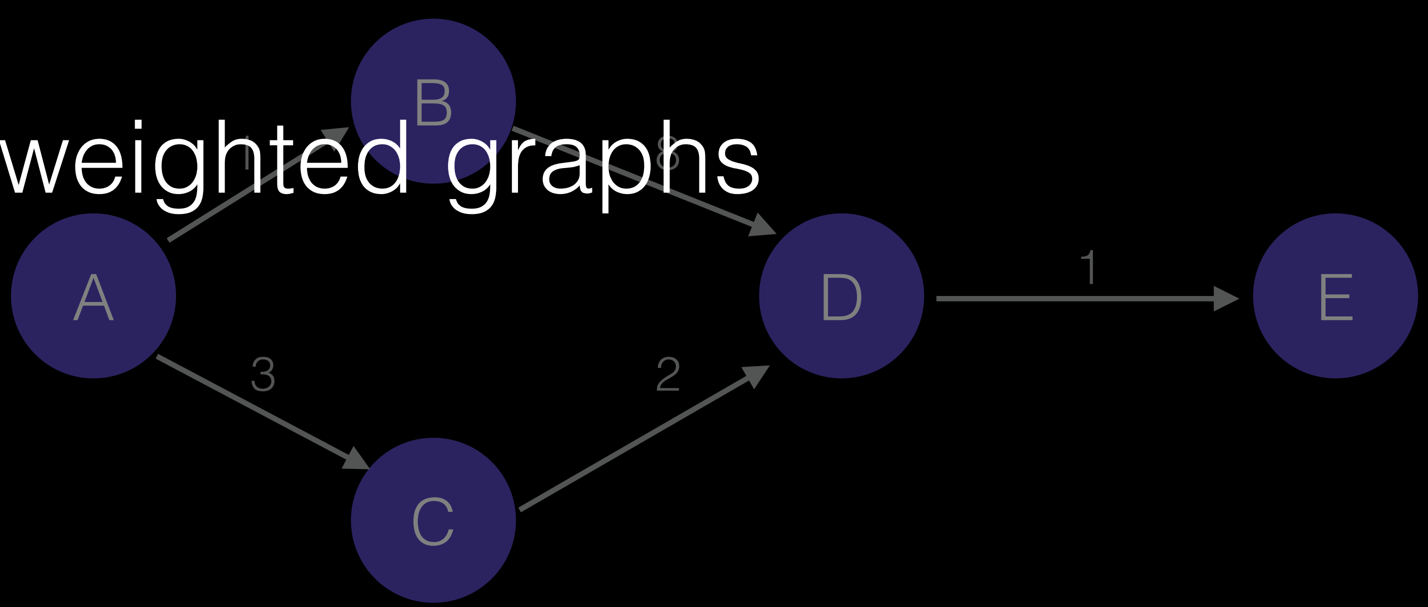
```
class Node {
    public ArrayList<Edge> neighbors = new ArrayList<Edge>();
}

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

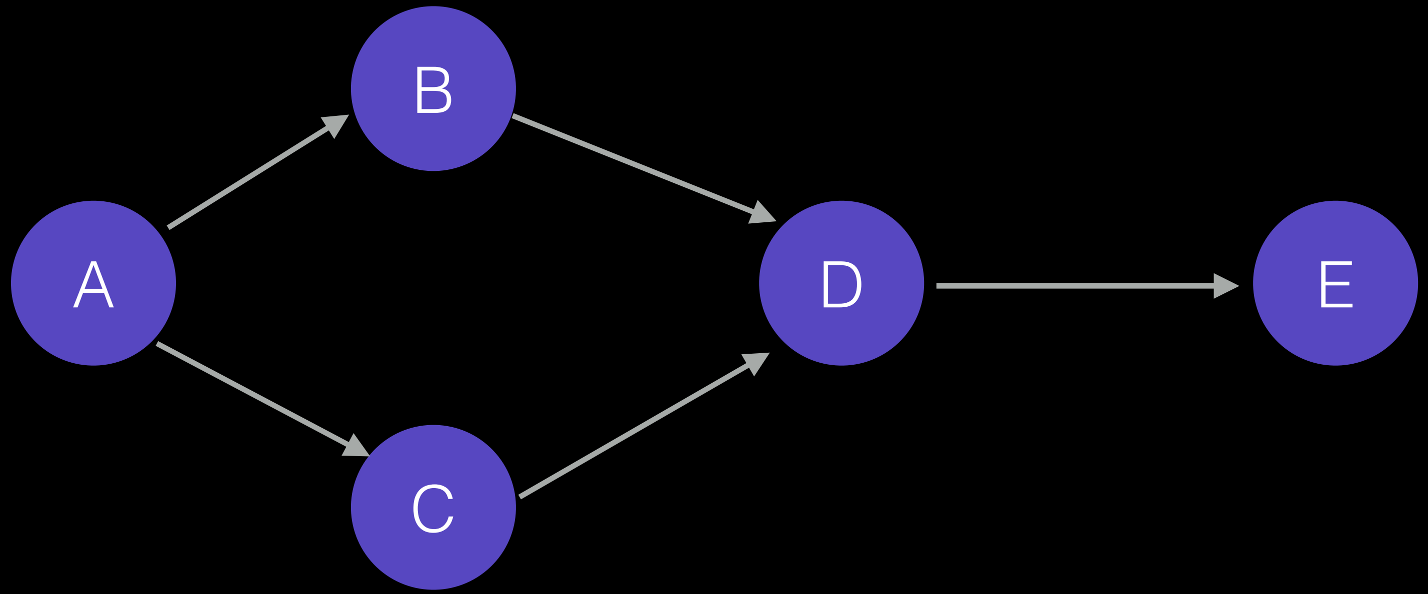
- Well-organized
- Adjacency Lists
    - Easy to get all edges out of a node
    - Difficult to find a particular edge
- This is what I use for weighted graphs

```
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);
    }
}
```



Unweighted Graphs

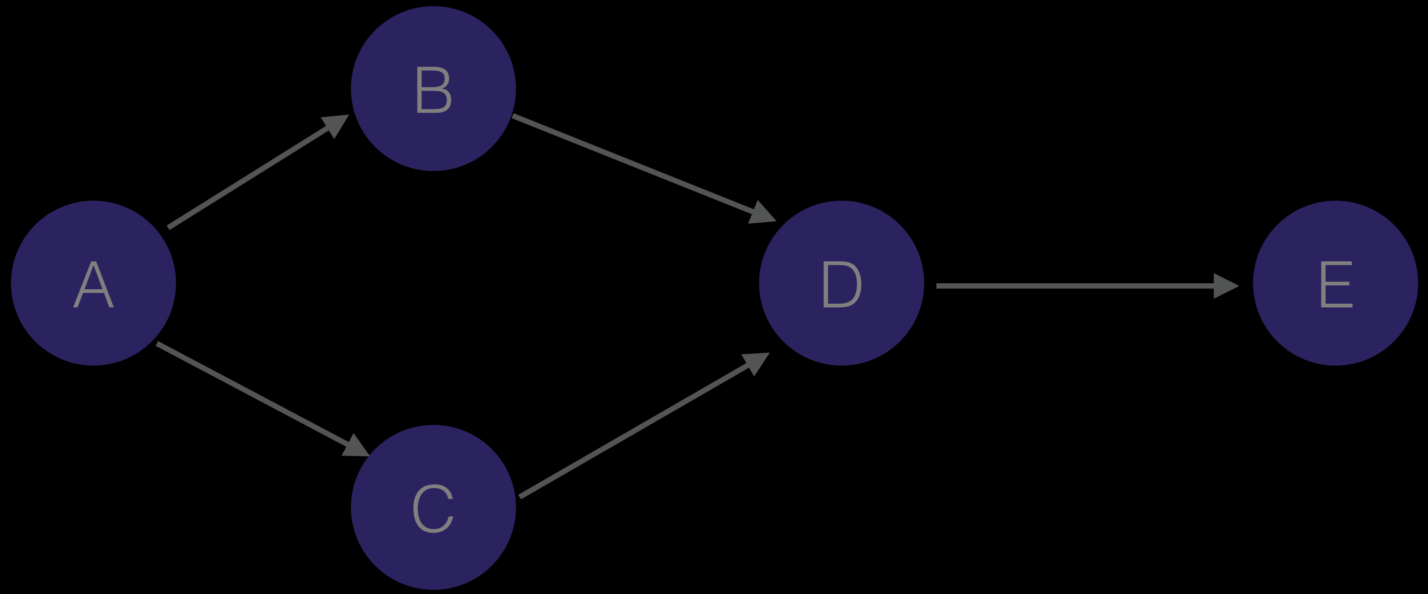
```
class Node {
    public ArrayList<Node> neighbors = new ArrayList<Node>();
}
```



Unweighted Graphs

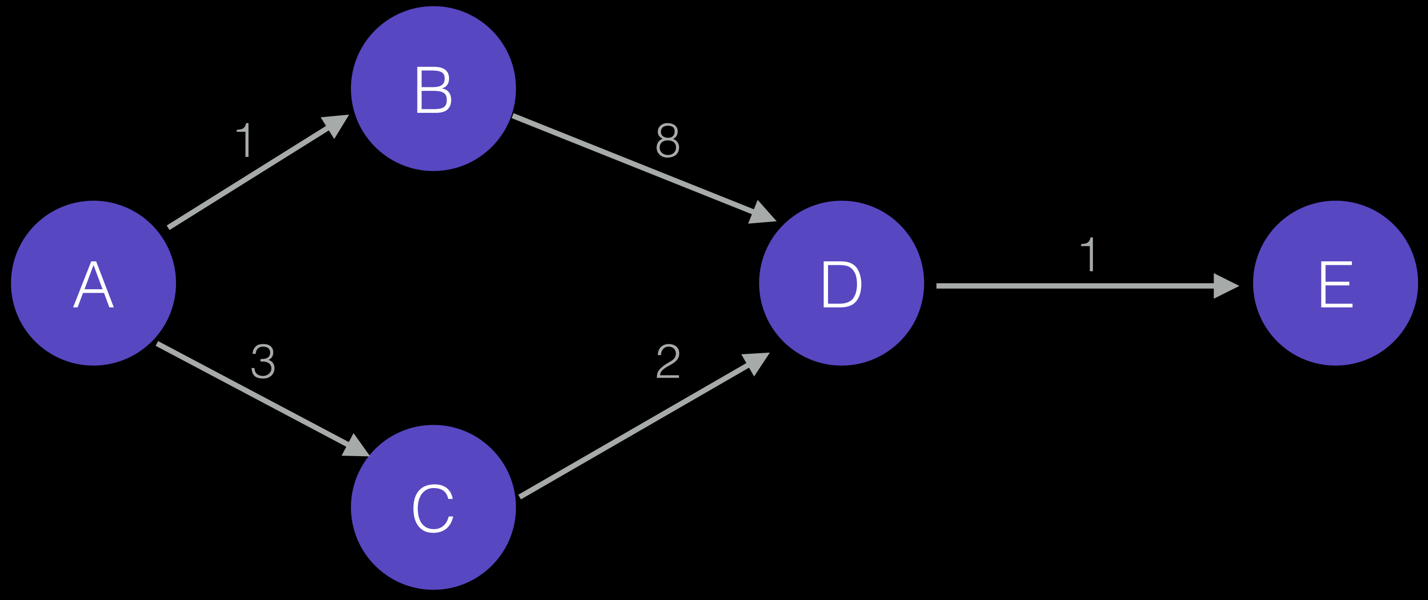
- Easy. You don’t need an edge class!

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



HashMaps

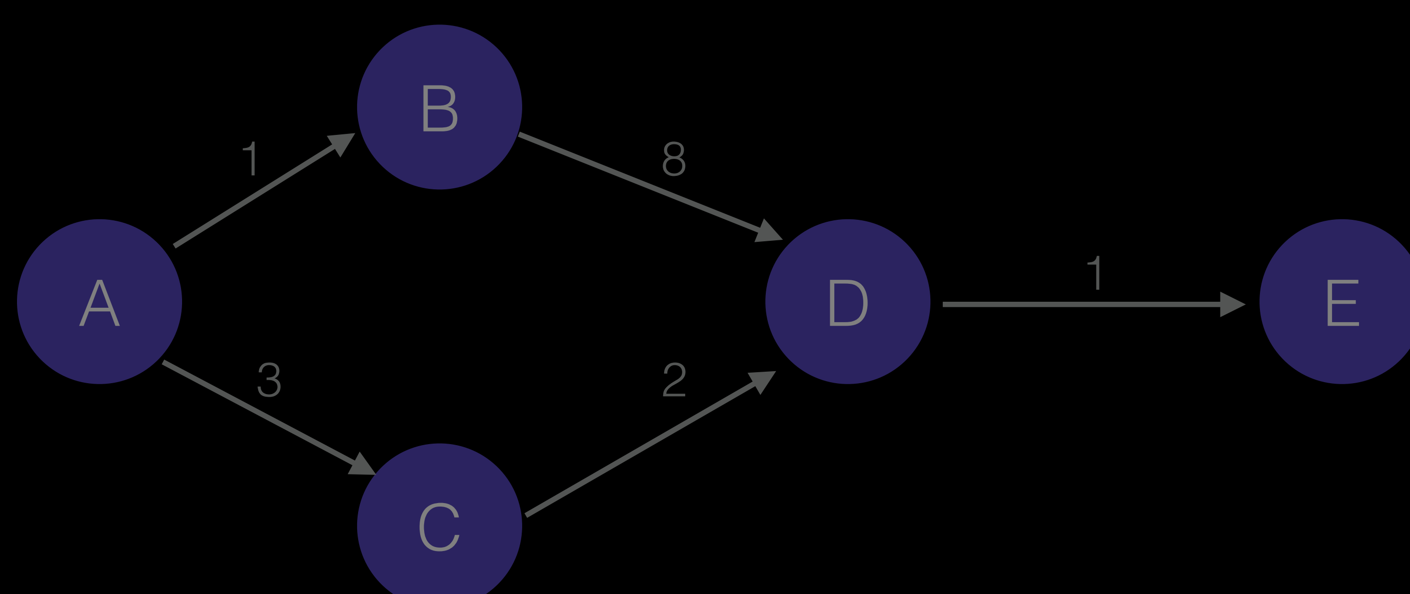
```java
//For a weighted graph
class Node {
   public HashMap<Node, ArrayList<Node, Integer>> neighbors = new HashMap<Node,
ArrayList<Node, Integer>>();
}

//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

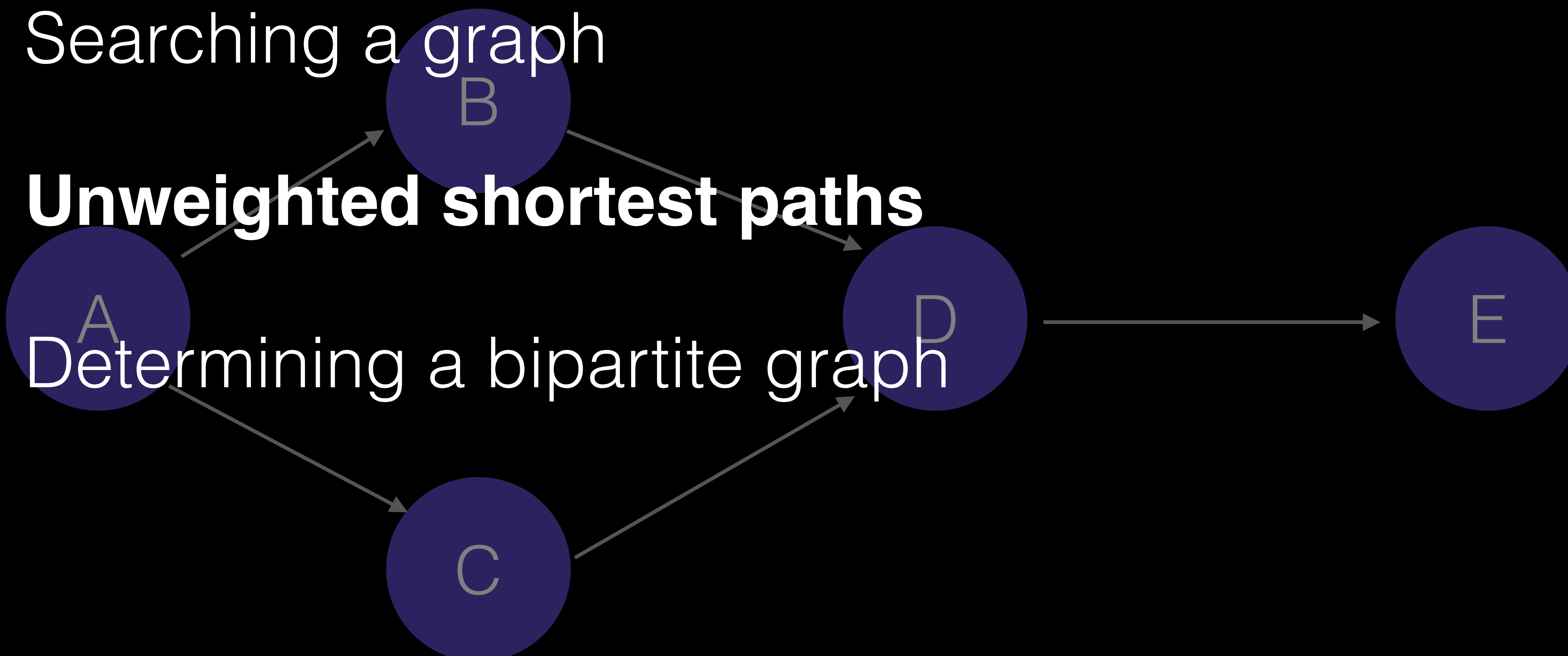
- Searches (Breadth-First, Depth-First)
- Shortest Path (Dijkstra's)
- Minimum Spanning Tree (Prim's, Kruskal's)
- Topological Sort
- Negative-Edge Shortest Path (Bellman-Ford)
- All-Pairs, Shorest Path (Floyd-Warshall)
- Max Flow (Ford-Fulkerson, Edmonds-Karp, Preflow Push)
- Minimum-Cost Flow

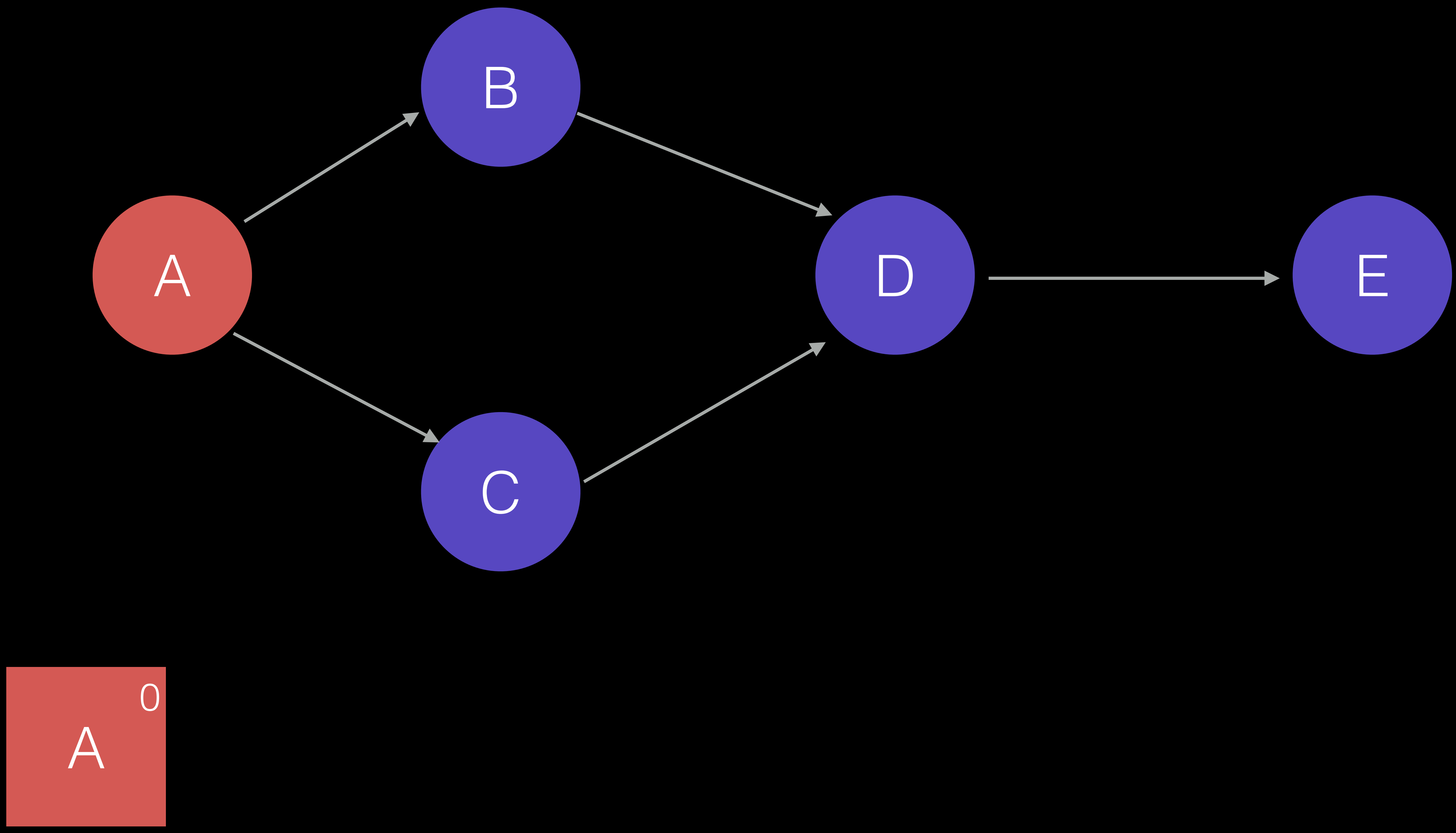
Graph Algorithms

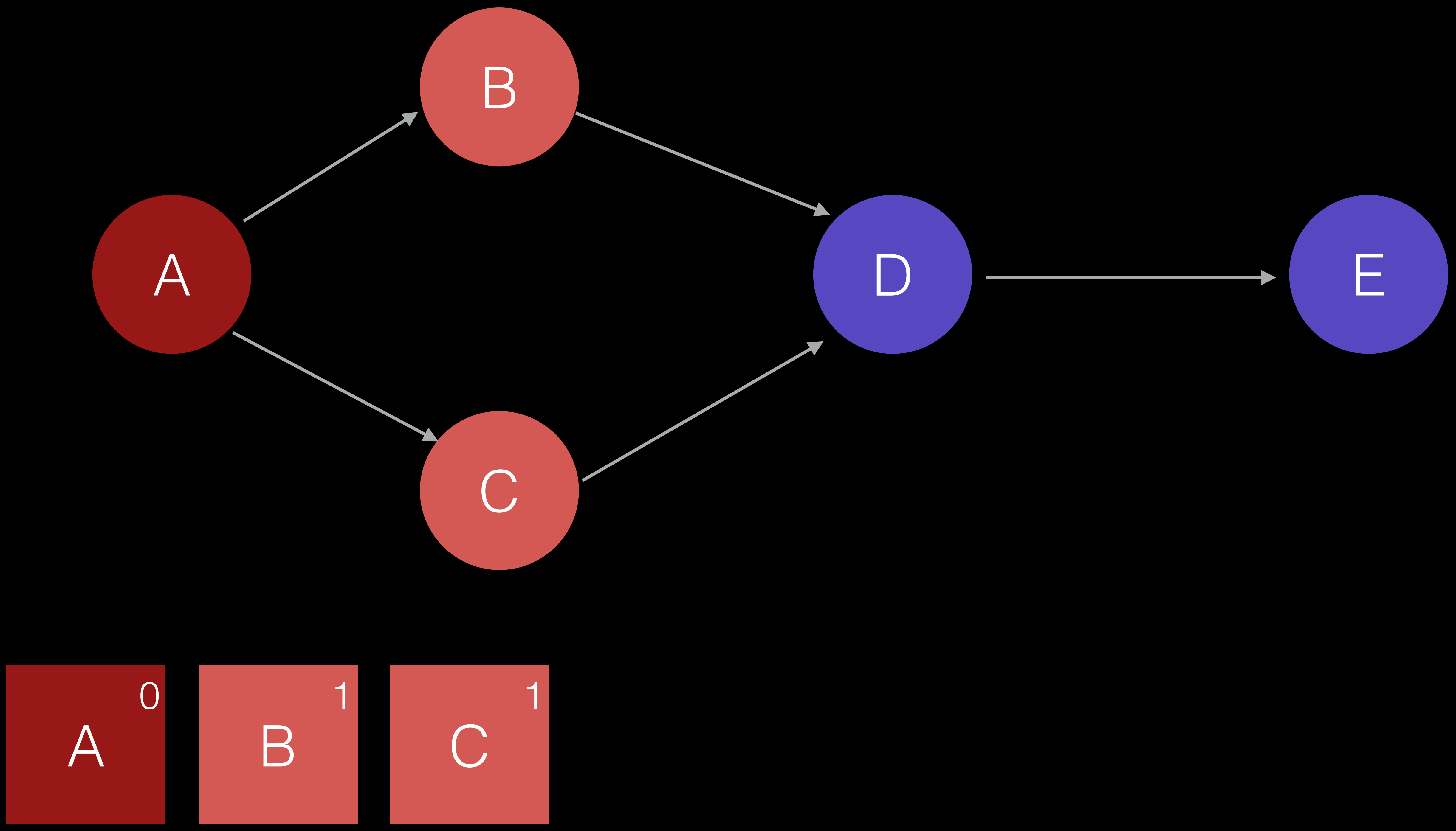
- Searches (Breadth-First, Depth-First)
- Shortest Path (Dijkstra's)
- Minimum Spanning Tree (Prim's, Kruskal's)
- Topological Sort
- Negative-Edge Shortest Path (Bellman-Ford)
- All-Pairs, Shorest Path (Floyd-Warshall)
- Max Flow (Ford-Fulkerson, Edmonds-Karp, Preflow Push)
- Minimum-Cost Flow

Breadth-First Search

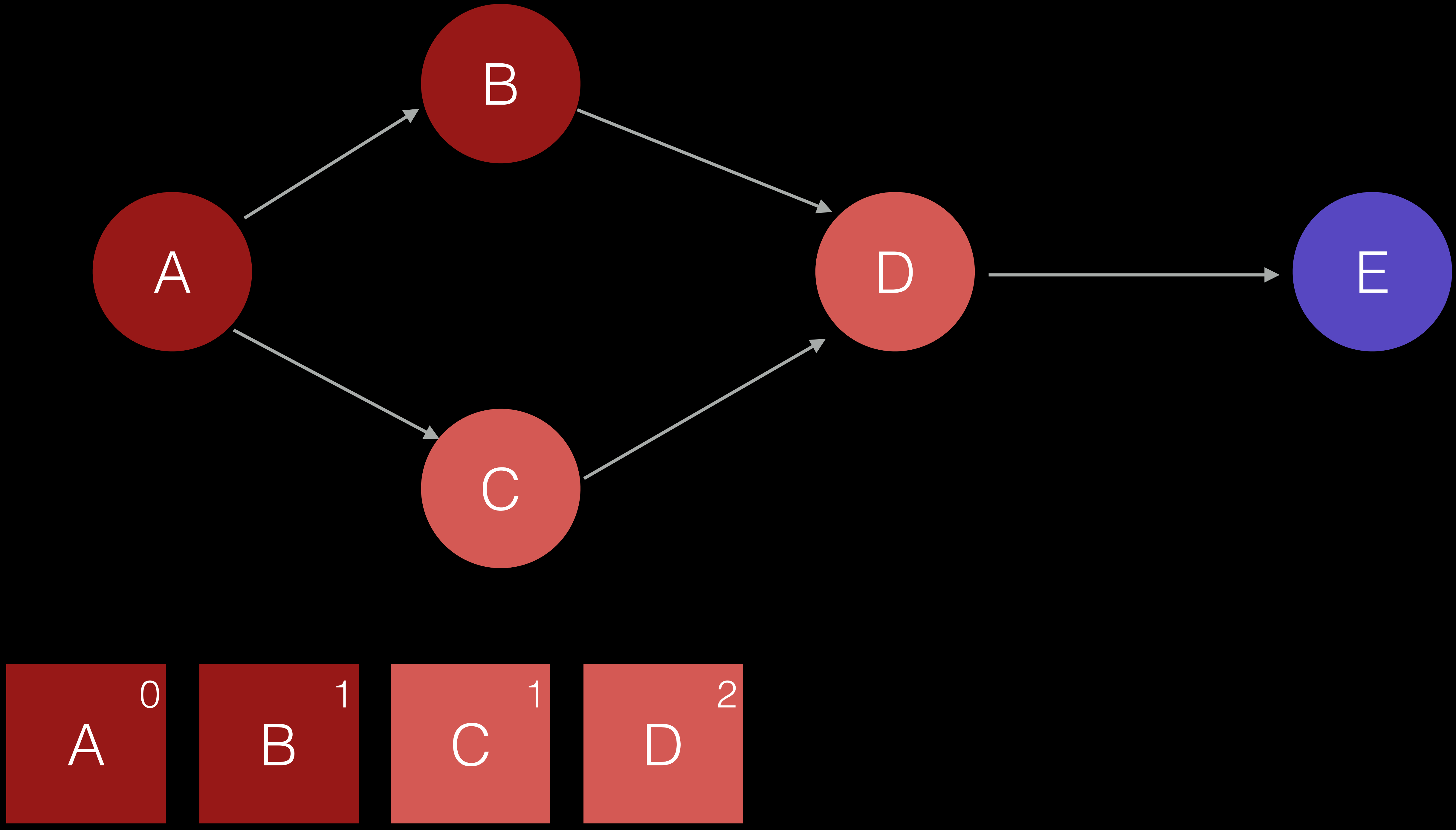
- Searching a graph
- **Unweighted shortest paths**
- Determining a bipartite graph

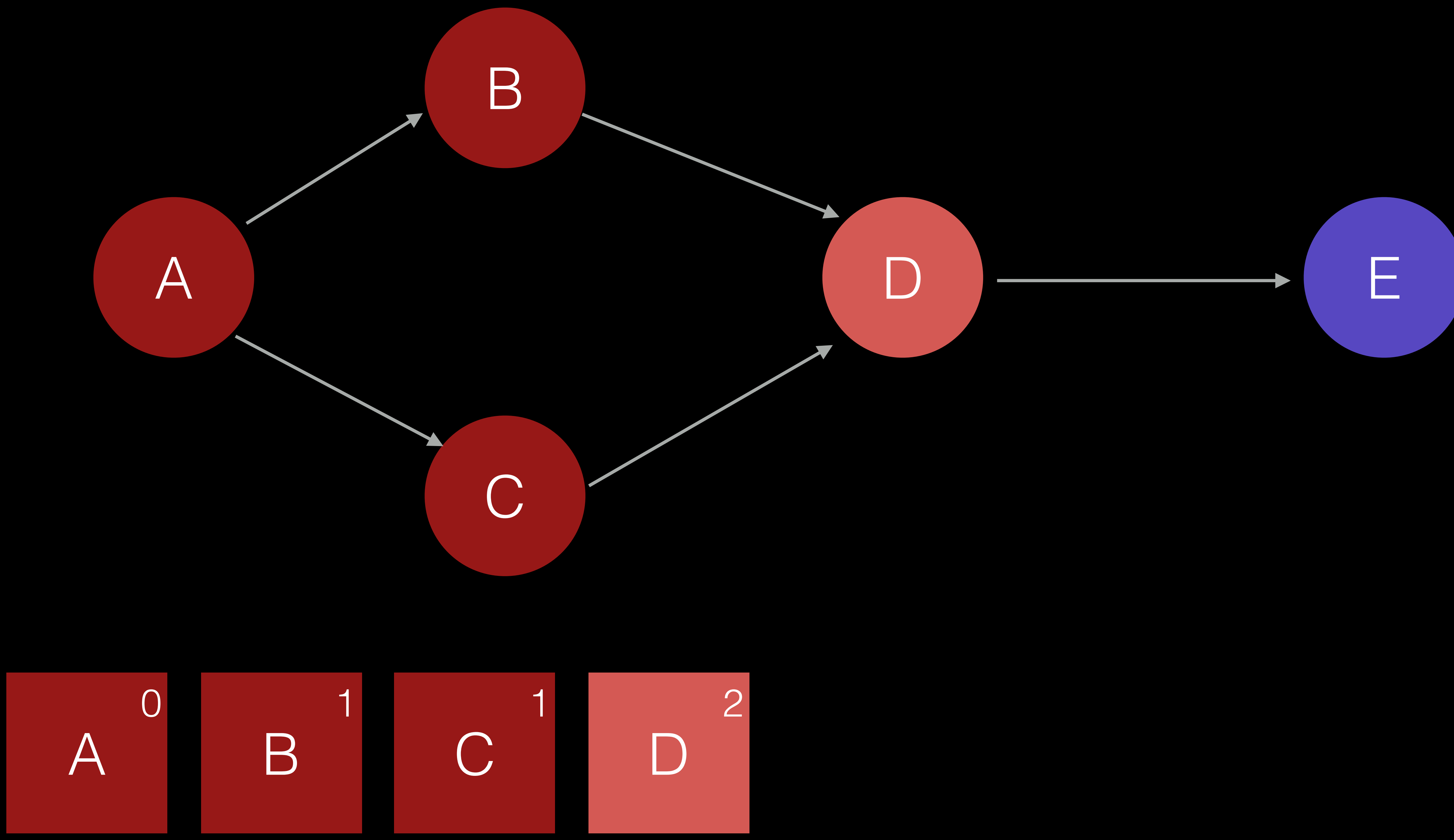


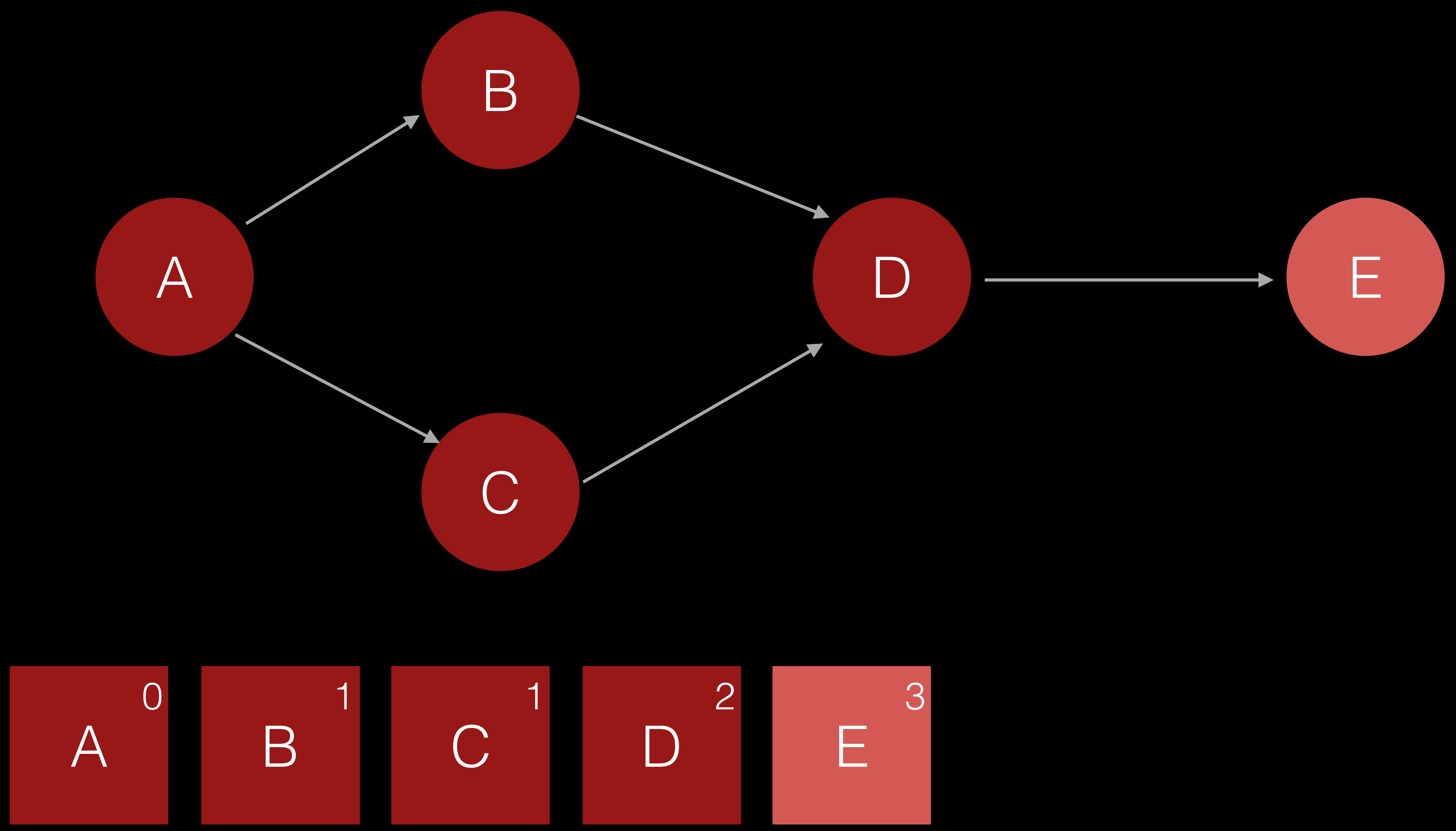

A
B
C
D
E
A
0



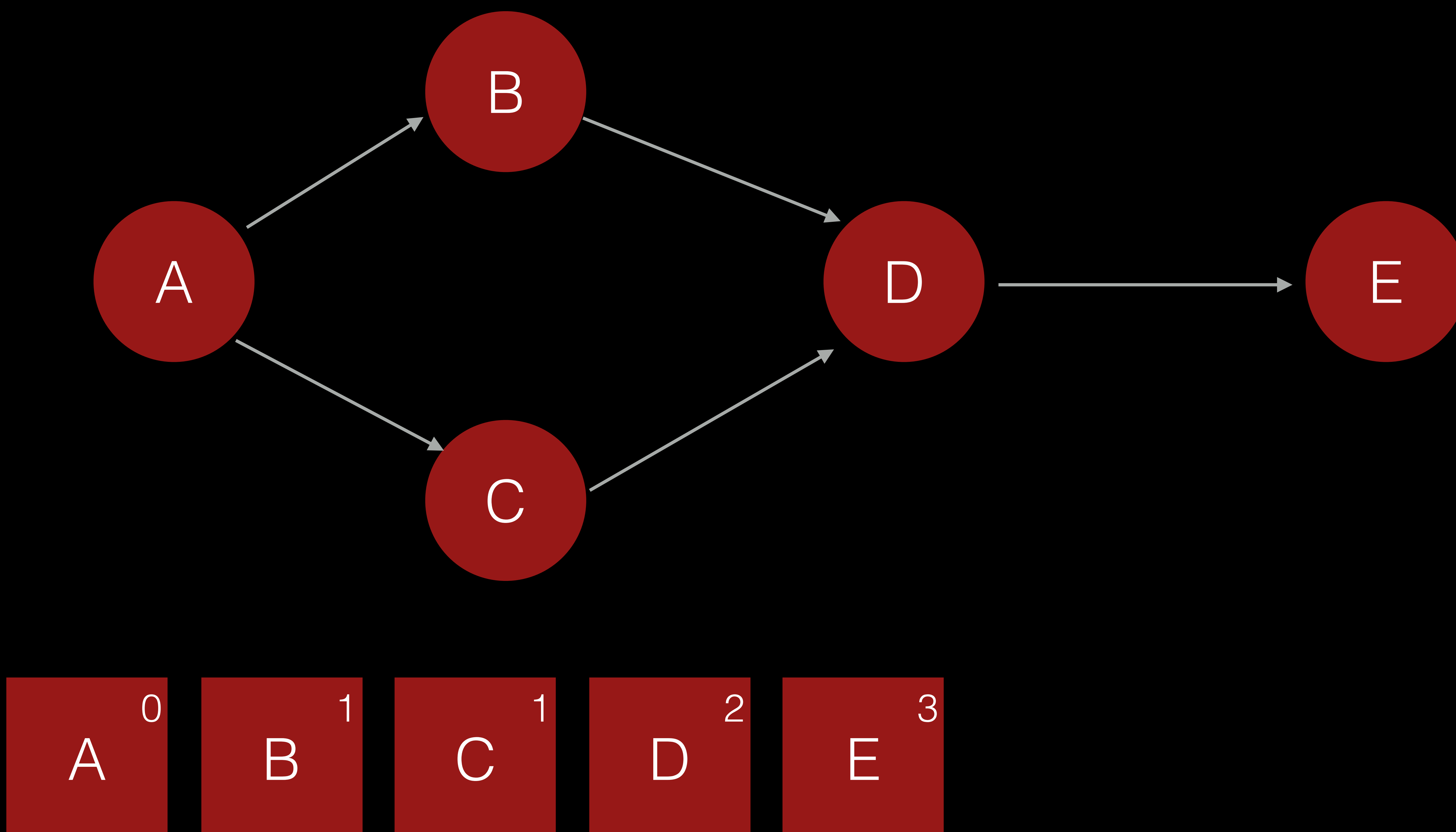
B
A
D
E
C
A 0
B 1
C 1


A
B
C
D
E
A 0
B 1
C 1
D 2


A
B
C
D
E
A 0
B 1
C 1
D 2



A
B
C
D
E
A 0
B 1
C 1
D 2
E 3

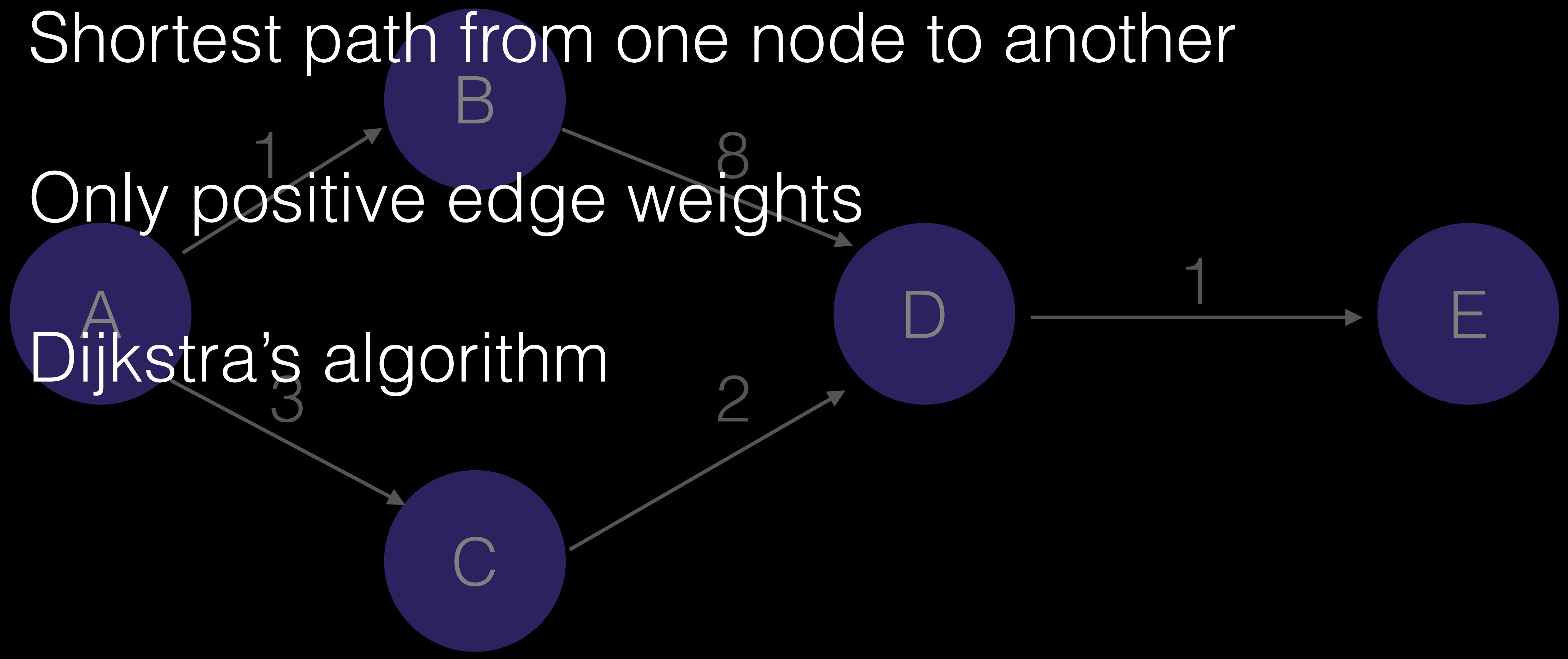

A
B
C
D
E
A 0
B 1
C 1
D 2
E 3

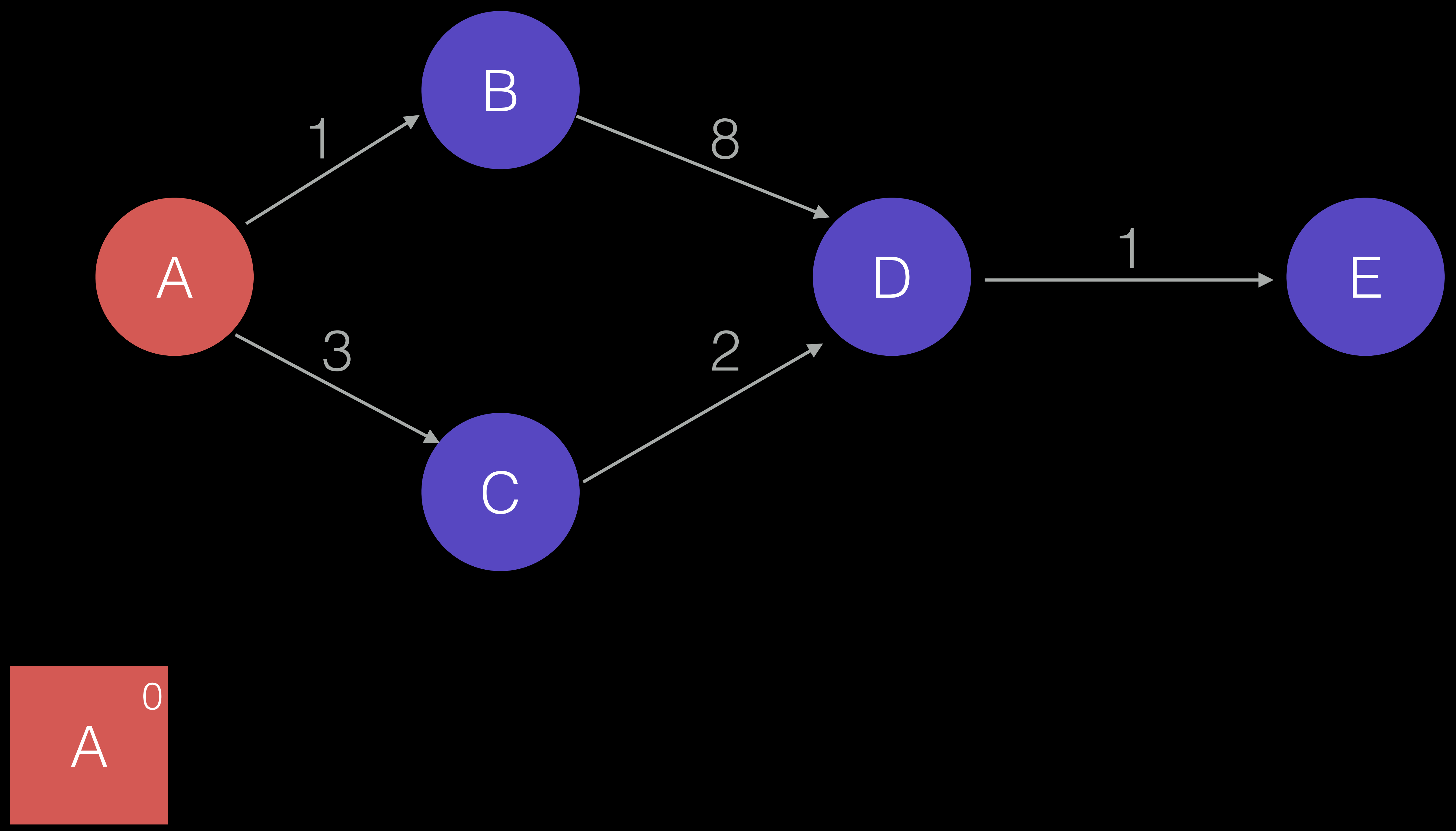
```java
import java.io.*;
import java.util.*;

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);
  }
}
```

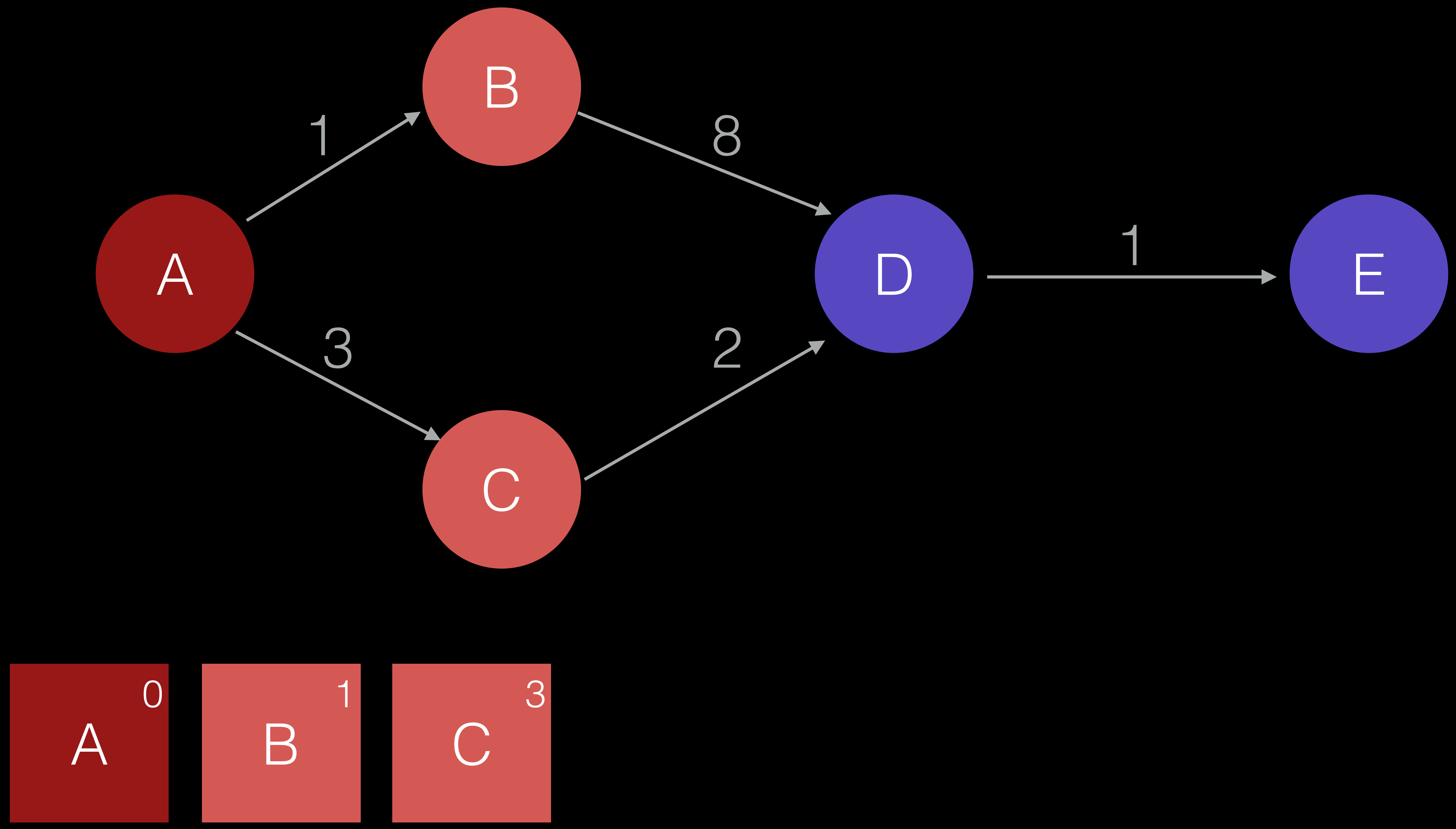
```java
public static void bfs(Node root, ArrayList<Node> allNodes) {
  Queue<Node> q = new LinkedList<Node>();
  root.distance = 0;
  q.add(root);

  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);
      }
    }
  }
}
```

Weighted Shortest Path

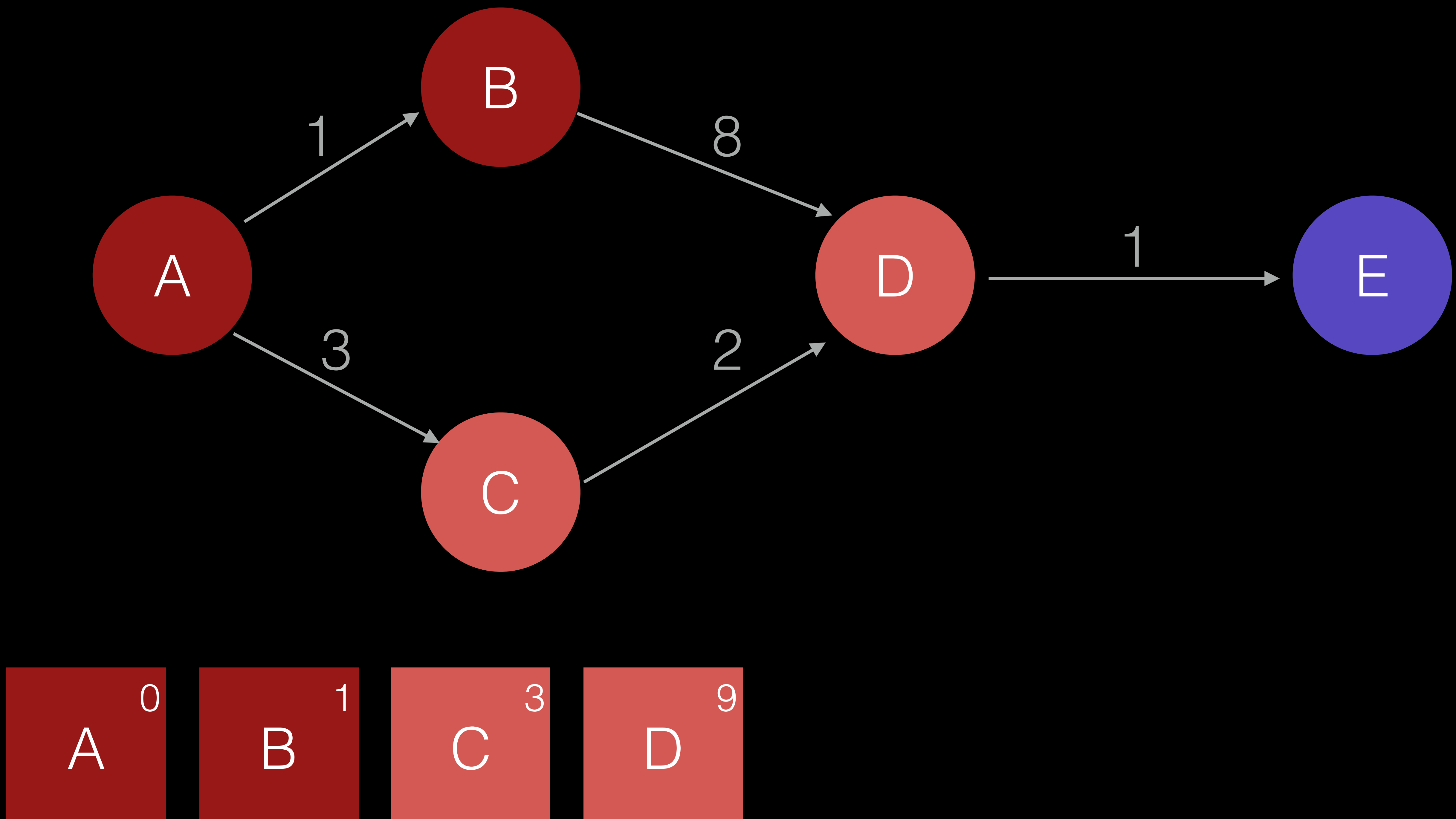
- Shortest path from one node to another
- Only positive edge weights
- Dijkstra’s algorithm

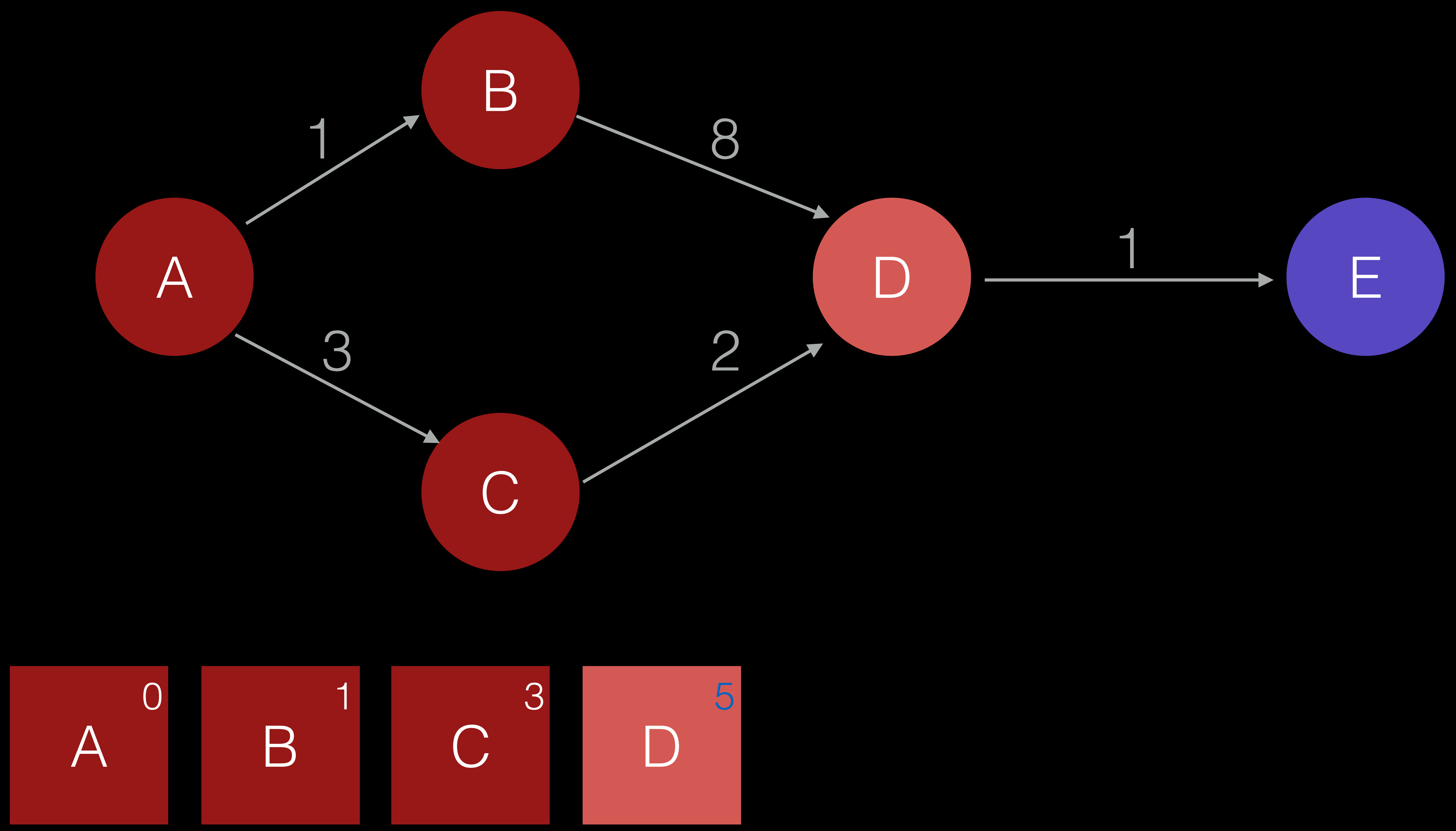


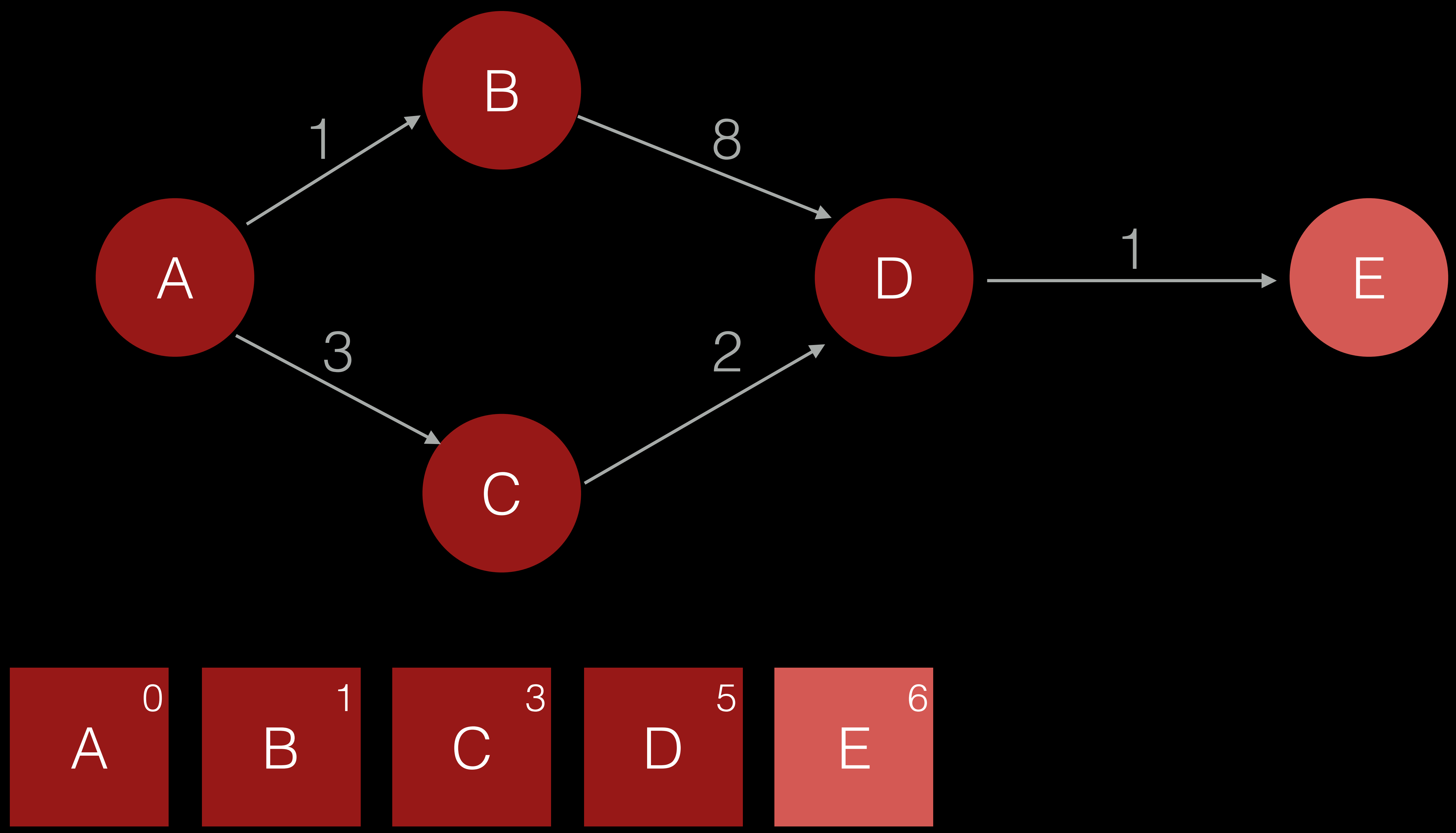

1
8
A
B
D
1
E
3
2
C
A
0

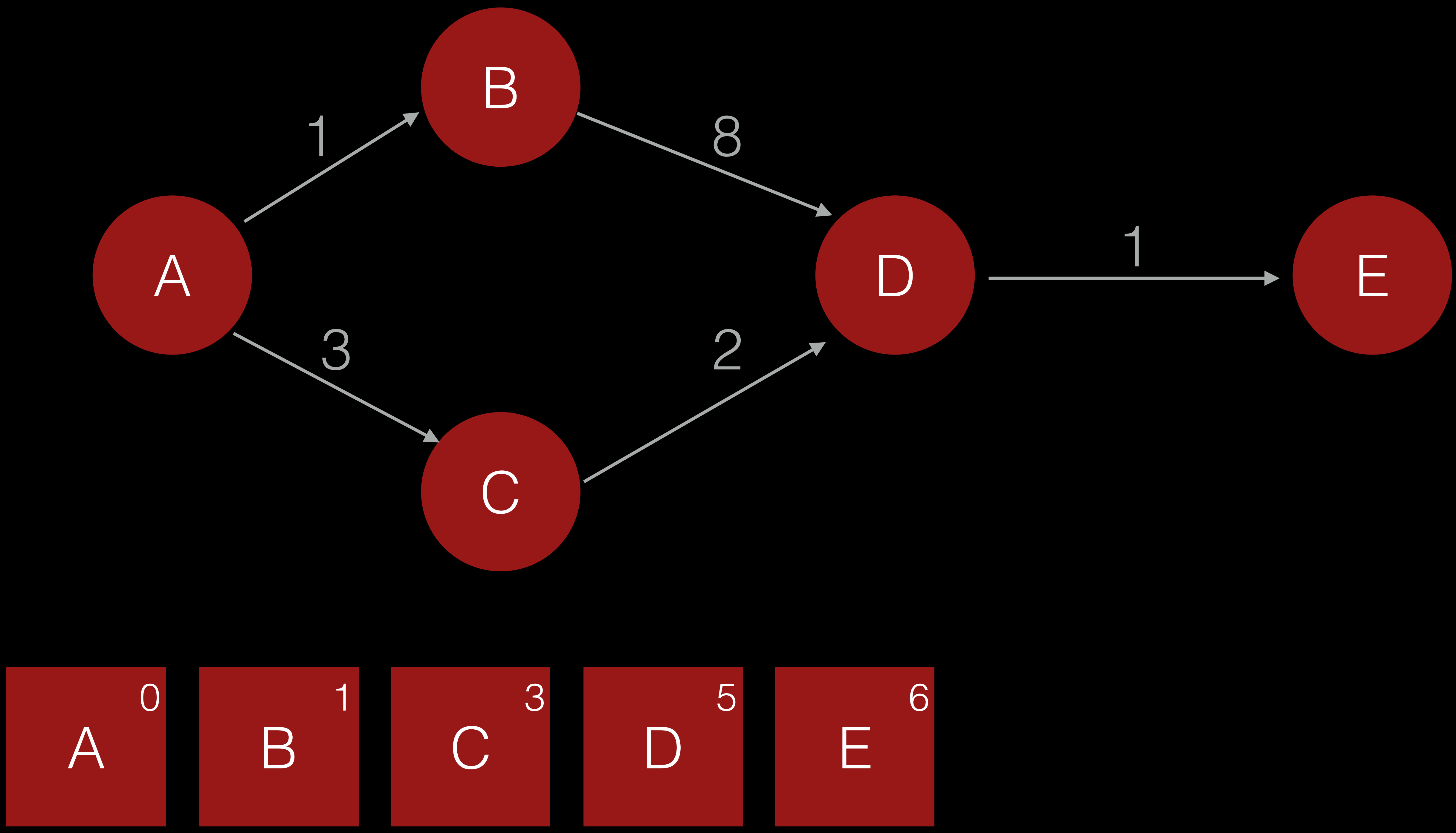


A
B
C
D
E
1
8
3
2
1
A 0
B 1
C 3


B
1
8
A
D
1
E
3
2
C
A 0
B 1
C 3
D 9


A
B
C
D
E
1
8
3
2
1
A 0
B 1
C 3
D 5


A
B
C
D
E
1
3
8
2
1
A 0
B 1
C 3
D 5
E 6


A
B
C
D
E
1
3
8
2
1
A 0
B 1
C 3
D 5
E 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);
    }
  }
}
```

Graph algorithms in their many shapes and sizes

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