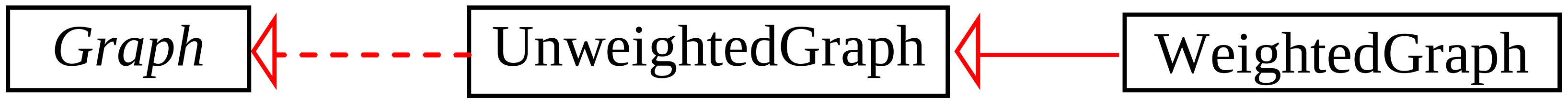
# Modeling Graphs



```java
public class Edge
{
 public int u;
 public int v;

 public Edge(int u, int v)
 {
   this.u = u;
   this.v = v;
 }

 public boolean equals(Object o)
 {
   if(o==null || !(o instanceof Edge)) return false;
   if(o==this) return true;

   Edge e = (Edge)o;
   return u == e.u && v == e.v;
 }
}
```

Edge

```java
public interface Graph<V>
{
 /** Return the number of vertices in the graph */
 public int getSize();

 /** Return the vertices in the graph */
 public java.util.List<V> getVertices();

 /** Return the object for the specified vertex index */
 public V getVertex(int index);

 /** Return the index for the specified vertex object */
 public int getIndex(V v);

 /** Return the neighbors of vertex with the specified index */
 public java.util.List<Integer> getNeighbors(int index);

 /** Return the degree for a specified vertex */
 public int getDegree(int v);

 /** Print the edges */
 public void printEdges();
 /** Clear the graph */
 public void clear();

 /** Add a vertex to the graph */
 public boolean addVertex(V vertex);

 /** Add an edge (u, v) to the graph */
 public boolean addEdge(int u, int v);

 /** Add an edge to the graph */
 public boolean addEdge(Edge e);

 /** Remove a vertex v from the graph, return true if successful */
 public boolean remove(V v);

 /** Remove an edge (u, v) from the graph */
 public boolean remove(int u, int v);

 /** Obtain a depth-first search tree */
 public UnweightedGraph<V>.SearchTree dfs(int v);

 /** Obtain a breadth-first search tree */
 public UnweightedGraph<V>.SearchTree bfs(int v);
}
```

Graph

```java
import java.util.*;
public class UnweightedGraph<V> implements Graph<V>
{
 protected List<V> vertices = new ArrayList<>(); // Store vertices
 protected List<List<Edge>> neighbors = new ArrayList<>(); // Adjacency lists

 /** Construct an empty graph */
 public UnweightedGraph() {  }

 /** Construct a graph from vertices and edges stored in arrays */
 public UnweightedGraph(V[] vertices, int[][] edges)
 {
   for (int i = 0; i < vertices.length; i++)
     addVertex(vertices[i]);

   createAdjacencyLists(edges, vertices.length);
 }

 /** Construct a graph from vertices and edges stored in List */
 public UnweightedGraph(List<V> vertices, List<Edge> edges)
 {
   for (int i = 0; i < vertices.size(); i++)
     addVertex(vertices.get(i));

   createAdjacencyLists(edges, vertices.size());
 }
```

```java
/** Construct a graph for integer vertices 0, 1, 2 and edge list */
@SuppressWarnings("unchecked")
public UnweightedGraph(List<Edge> edges, int numberOfVertices)
{
  for (int i = 0; i < numberOfVertices; i++)
    addVertex((V)(new Integer(i))); // vertices is {0, 1, ...}

  createAdjacencyLists(edges, numberOfVertices);
}

/** Construct a graph from integer vertices 0, 1, and edge array */
@SuppressWarnings("unchecked")
public UnweightedGraph(int[][] edges, int numberOfVertices)
{
  for (int i = 0; i < numberOfVertices; i++)
    addVertex((V)(new Integer(i))); // vertices is {0, 1, ...}

  createAdjacencyLists(edges, numberOfVertices);
}
```

```java
/** Create adjacency lists for each vertex */
private void createAdjacencyLists(int[][] edges, int numberOfVertices)
{
  for (int i = 0; i < edges.length; i++) {
    addEdge(edges[i][0], edges[i][1]);
  }
}

/** Create adjacency lists for each vertex */
private void createAdjacencyLists(List<Edge> edges, int numberOfVertices)
{
  for (Edge edge: edges) {
    addEdge(edge.u, edge.v);
  }
}

@Override /** Return the number of vertices in the graph */
public int getSize() { return vertices.size(); }

@Override /** Return the vertices in the graph */
public List<V> getVertices() { return vertices; }

@Override /** Return the object for the specified vertex */
public V getVertex(int index) { return vertices.get(index); }
```

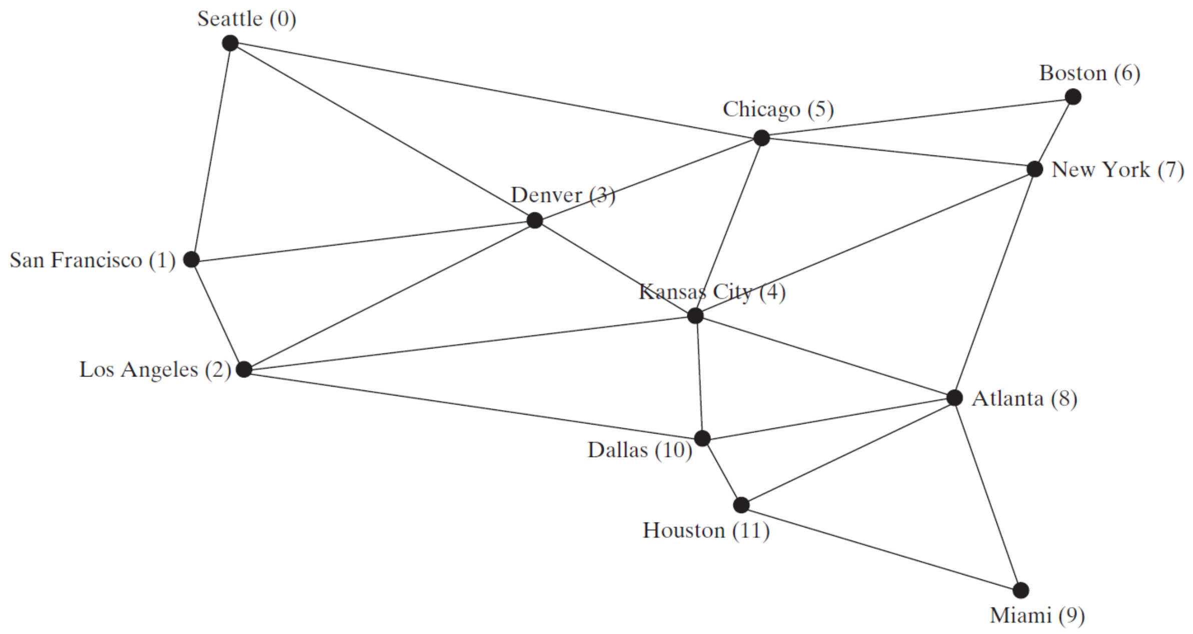
```java
@Override /** Return the index for the specified vertex object */
public int getIndex(V v)
{
  return vertices.indexOf(v);
}

@Override /** Return the neighbors of the specified vertex */
public List<Integer> getNeighbors(int index)
{
  List<Integer> result = new ArrayList<>();
  for (Edge e: neighbors.get(index))
    result.add(e.v);

  return result;
}

@Override /** Return the degree for a specified vertex */
public int getDegree(int v)
{
  return neighbors.get(v).size();
}
```

```java
@Override /** Print the edges */
public void printEdges()
{
  for (int u = 0; u < neighbors.size(); u++) {
    System.out.print(getVertex(u) + " (" + u + "): ");
    for (Edge e: neighbors.get(u)) {
      System.out.print("(" + getVertex(e.u) + ", " + getVertex(e.v) + ") ");
    }
    System.out.println();
  }
}

@Override /** Clear the graph */
public void clear() { vertices.clear(); neighbors.clear(); }

@Override /** Add a vertex to the graph */
public boolean addVertex(V vertex)
{
  if (!vertices.contains(vertex)) {
    vertices.add(vertex);
    neighbors.add(new ArrayList<Edge>());
    return true;
  } else { return false; }
}
```

```java
@Override /** Add an edge to the graph */
public boolean addEdge(Edge e)
{
  if (e.u < 0 || e.u > getSize() - 1)
    throw new IllegalArgumentException("No such index: " + e.u);

  if (e.v < 0 || e.v > getSize() - 1)
    throw new IllegalArgumentException("No such index: " + e.v);

  if (!neighbors.get(e.u).contains(e)) {
    neighbors.get(e.u).add(e);
    return true;
  }
  else {
    return false;
  }
}

@Override /** Add an edge to the graph */
public boolean addEdge(int u, int v)
{
  return addEdge(new Edge(u, v));
}
```

UnweightedGraph

```
public class TestGraph {
 public static void main(String[] args) {
   String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
     "Denver", "Kansas City", "Chicago", "Boston", "New York",
     "Atlanta", "Miami", "Dallas", "Houston"};

   int[][] edges = {
     {0, 1}, {0, 3}, {0, 5},
     {1, 0}, {1, 2}, {1, 3},
     {2, 1}, {2, 3}, {2, 4}, {2, 10},
     {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
     {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
     {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
     {6, 5}, {6, 7},
     {7, 4}, {7, 5}, {7, 6}, {7, 8},
     {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
     {9, 8}, {9, 11},
     {10, 2}, {10, 4}, {10, 8}, {10, 11},
     {11, 8}, {11, 9}, {11, 10}
   };
```



```java
Graph<String> graph1 = new UnweightedGraph<>(vertices, edges);
System.out.println("The number of vertices in graph1: "
  + graph1.getSize());
System.out.println("The vertex with index 1 is "
  + graph1.getVertex(1));
System.out.println("The index for Miami is " +
  graph1.getIndex("Miami"));
System.out.println("The edges for graph1:");
graph1.printEdges();
```

TestGraph

```
The number of vertices in graph1: 12
The vertex with index 1 is San Francisco
The index for Miami is 9
The edges for graph1:
Seattle (0): (Seattle, San Francisco) (Seattle, Denver) (Seattle, Chicago)
San Francisco (1): (San Francisco, Seattle) (San Francisco, Los Angeles) (
Los Angeles (2): (Los Angeles, San Francisco) (Los Angeles, Denver) (Los A
Denver (3): (Denver, Seattle) (Denver, San Francisco) (Denver, Los Angeles
Kansas City (4): (Kansas City, Los Angeles) (Kansas City, Denver) (Kansas
Chicago (5): (Chicago, Seattle) (Chicago, Denver) (Chicago, Kansas City) (
Boston (6): (Boston, Chicago) (Boston, New York)
```

Note: Partial output.

# Graph Traversals

```
UnweightedGraph<V>.SearchTree
```

| | |
|---|---|
| `-root: int` | The root of the tree. |
| `-parent: int[]` | The parents of the vertices. |
| `-searchOrder: List<Integer>` | The orders for traversing the vertices. |
| `+SearchTree(root: int, parent: int[], searchOrder: List<Integer>)` | Constructs a tree with the specified root, parent, and `searchOrder`. |
| `+getRoot(): int` | Returns the root of the tree. |
| `+getSearchOrder(): List<Integer>` | Returns the order of vertices searched. |
| `+getParent(index: int): int` | Returns the parent for the specified vertex index. |
| `+getNumberOfVerticesFound(): int` | Returns the number of vertices searched. |
| `+getPath(index: int): List<V>` | Returns a list of vertices from the specified vertex index to the root. |
| `+printPath(index: int): void` | Displays a path from the root to the specified vertex. |
| `+printTree(): void` | Displays tree with the root and all edges. |

UnweightedGraph

# Depth-First Search (DFS) (or depth-first traversal)

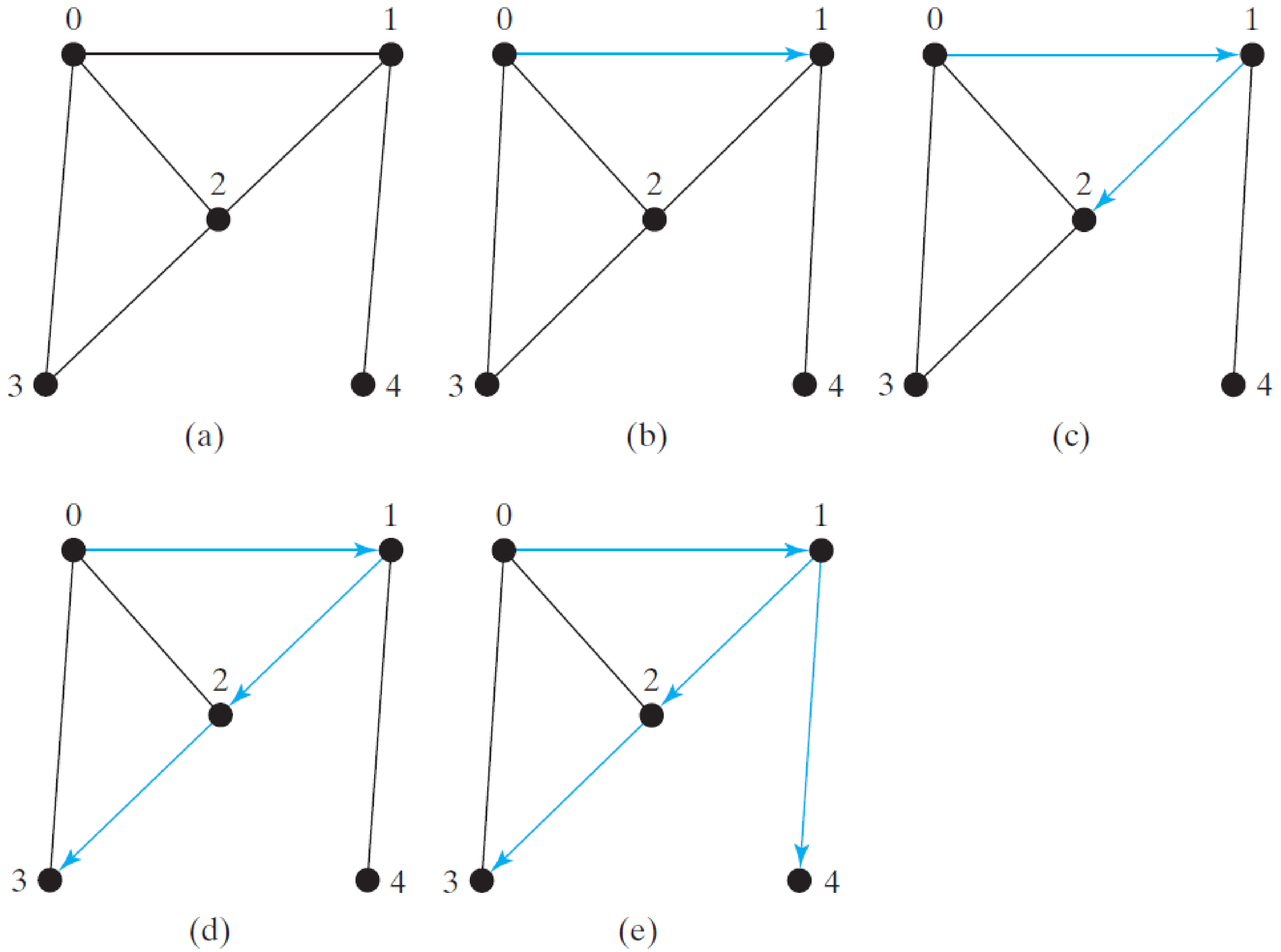
- The search is called depth-first because it searches “deeper” in the graph as much as possible.
- The search start from some vertex $v$.
  - After visiting $v$, it visits an unvisited neighbor of $v$.
  - If $v$ has no unvisited neighbors, the search backtracks to the vertex from which it reached $v$.
- We assume that the graph is connected and the search starting from any vertex can reach all the vertices.
  - It is easy to relax this condition, in which case you are finding connected components in a graph.

# Depth-First Search Algorithm

Input: G = (V, E) and a starting vertex v

Output: a DFS tree rooted at v

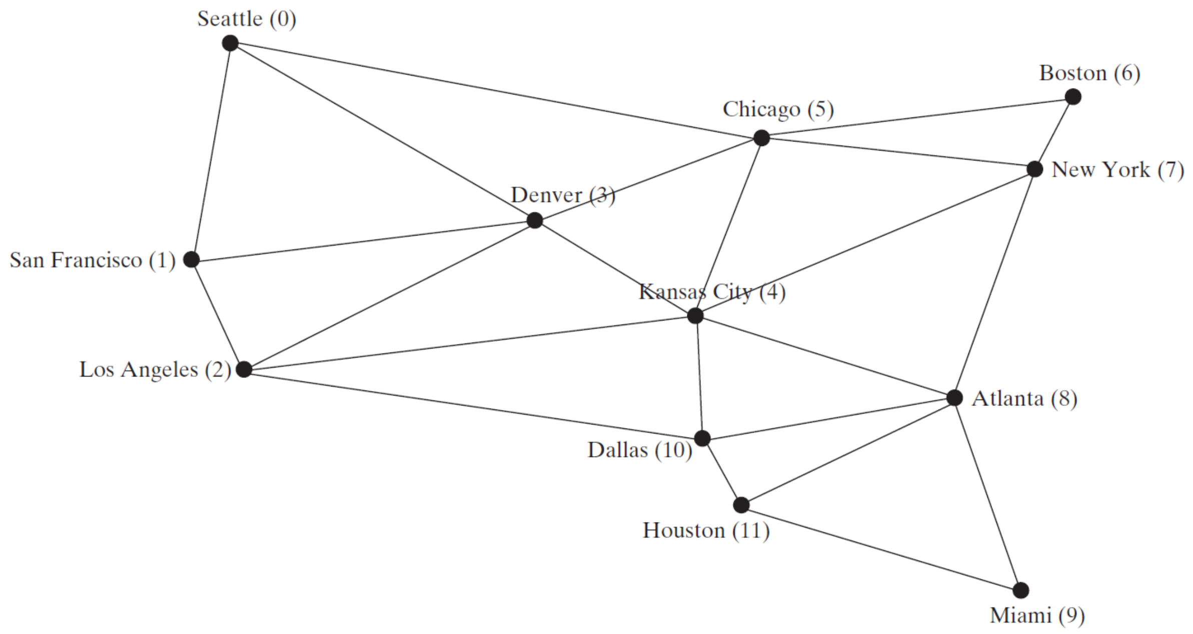
```
1.  SearchTree **dfs**(vertex v) {
2.    visit v;
3.    *for* each neighbor w of v
4.      *if* (w has not been visited) {
5.        set v as the parent for w in the tree;
6.        dfs(w);
7.      } // end if
8.  } // end dfs
```


0
1
2
3
4
(a)
(b)
(c)
(d)
(e)

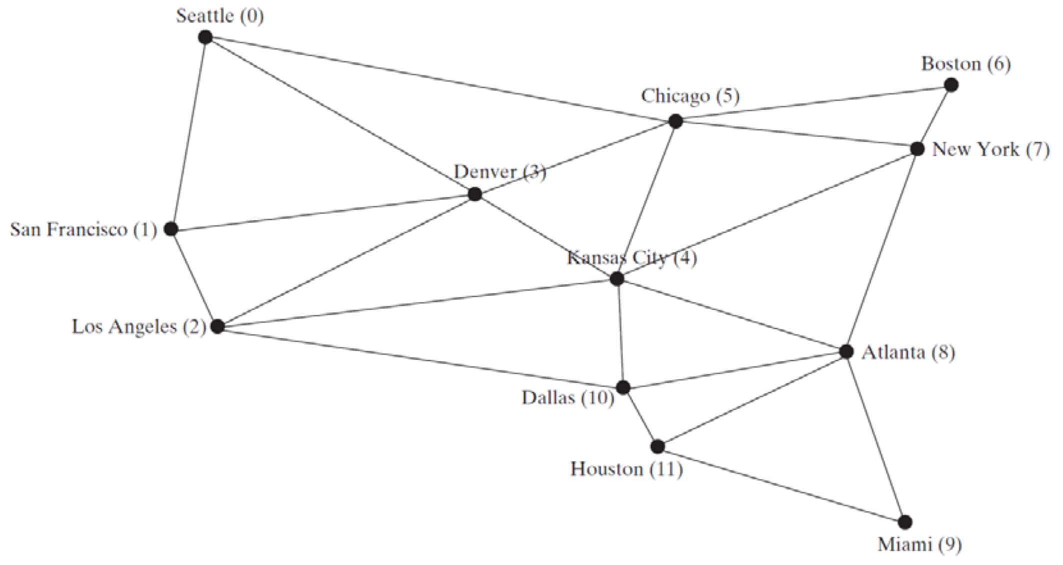
```java
@Override /** Obtain a DFS tree starting from vertex u */
public SearchTree dfs(int v)
{
  List<Integer> searchOrder = new ArrayList<>();
  int[] parent = new int[vertices.size()];
  for (int i = 0; i < parent.length; i++) parent[i] = -1;

  boolean[] isVisited = new boolean[vertices.size()];
  dfs(v, parent, searchOrder, isVisited);
  return new SearchTree(v, parent, searchOrder);
}

/** Recursive method for DFS search */
private void dfs(int v, int[] parent,
                 List<Integer> searchOrder, boolean[] isVisited) {
  searchOrder.add(v);
  isVisited[v] = true;
  for (Edge e : neighbors.get(v))
  {
    if (!isVisited[e.v])
    {
      parent[e.v] = v;
      dfs(e.v, parent, searchOrder, isVisited);
    }
  }
}
```

UnweightedGraph

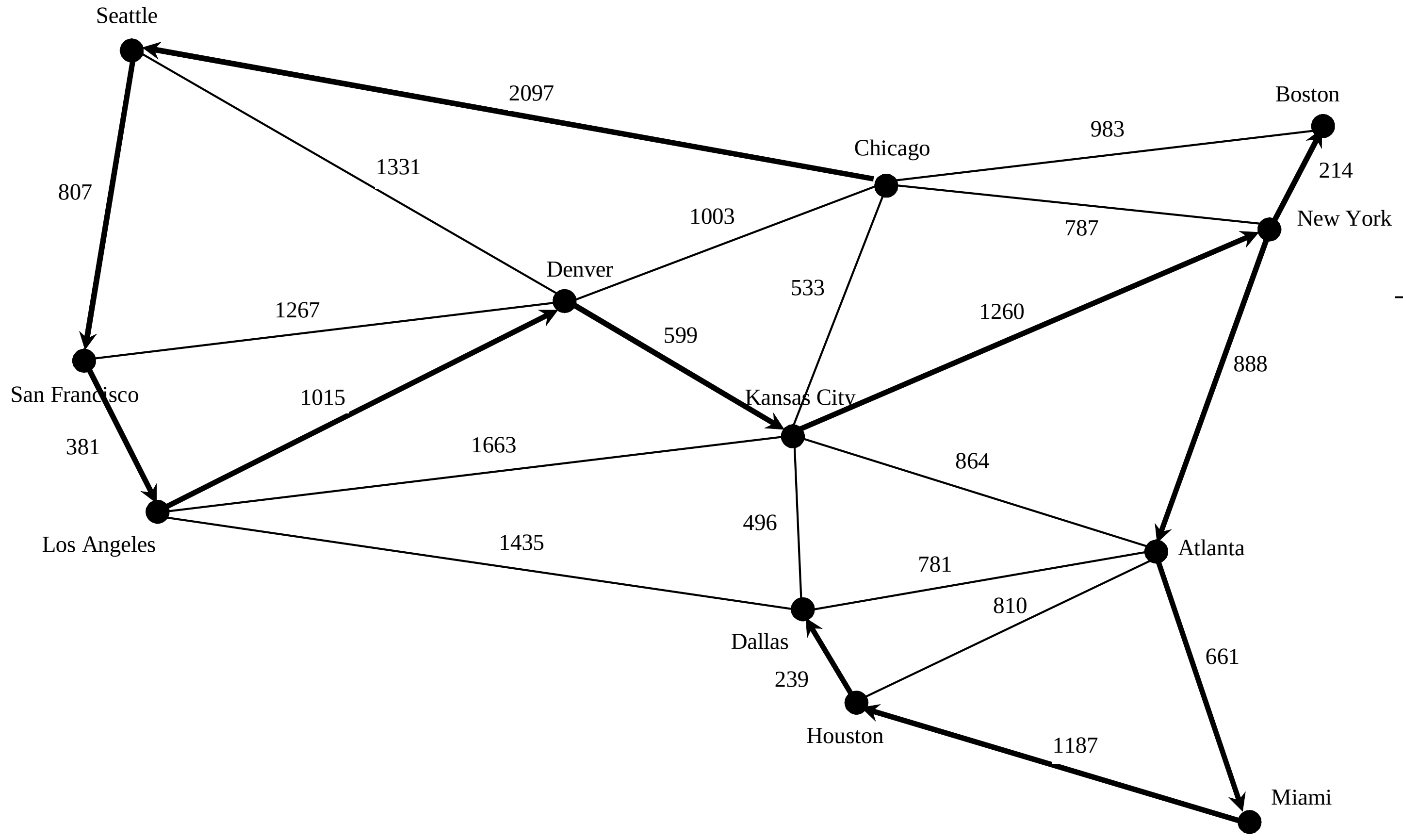
```java
public class TestDFS {
 public static void main(String[] args) {
   String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
     "Denver", "Kansas City", "Chicago", "Boston", "New York",
     "Atlanta", "Miami", "Dallas", "Houston"};

   int[][] edges = {
     {0, 1}, {0, 3}, {0, 5},
     {1, 0}, {1, 2}, {1, 3},
     {2, 1}, {2, 3}, {2, 4}, {2, 10},
     {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
     {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
     {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
     {6, 5}, {6, 7},
     {7, 4}, {7, 5}, {7, 6}, {7, 8},
     {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
     {9, 8}, {9, 11},
     {10, 2}, {10, 4}, {10, 8}, {10, 11},
     {11, 8}, {11, 9}, {11, 10}
   };
```



```
    Graph<String> graph = new UnweightedGraph<>(vertices, edges);
    UnweightedGraph<String>.SearchTree dfs =
      graph.dfs(graph.getIndex("Chicago"));

    java.util.List<Integer> searchOrders = dfs.getSearchOrder();
    System.out.println(dfs.getNumberOfVerticesFound() +
      " vertices are searched in this DFS order:");
    for (int i = 0; i < searchOrders.size(); i++)
      System.out.print(graph.getVertex(searchOrders.get(i)) + " ");
    System.out.println();

    for (int i = 0; i < searchOrders.size(); i++)
      if (dfs.getParent(i) != -1)
        System.out.println("parent of " + graph.getVertex(i) +
          " is " + graph.getVertex(dfs.getParent(i)));
  }
}
```



```
12 vertices are searched in this DFS order:
Chicago Seattle San Francisco Los Angeles Denver Kansas City New York Boston Atlanta Miami Houston Dallas
parent of Seattle is Chicago
parent of San Francisco is Seattle
parent of Los Angeles is San Francisco
parent of Denver is Los Angeles
parent of Kansas City is Denver
parent of Boston is New York
parent of New York is Kansas City
parent of Atlanta is New York
parent of Miami is Atlanta
parent of Dallas is Houston
parent of Houston is Miami
```



TestDFS

## Applications of the DFS

- Detecting whether a graph is connected.
  - Search the graph starting from any vertex. If the number of vertices searched is the same as the number of vertices in the graph, the graph is connected. Otherwise, the graph is not connected.
- Detecting whether there is a path between two vertices.
- Finding a path between two vertices.
- Finding all connected components.
  - A connected component is a maximal connected subgraph in which every pair of vertices are connected by a path.
- Detecting whether there is a cycle in the graph.
- Finding a cycle in the graph.
- Finding a Hamiltonian path/cycle.
  - A *Hamiltonian path* in a graph is a path that visits each vertex in the graph exactly once. A *Hamiltonian cycle* visits each vertex in the graph exactly once and returns to the starting vertex.

## Breadth-First Search (BFS) (or breadth-first traversal)

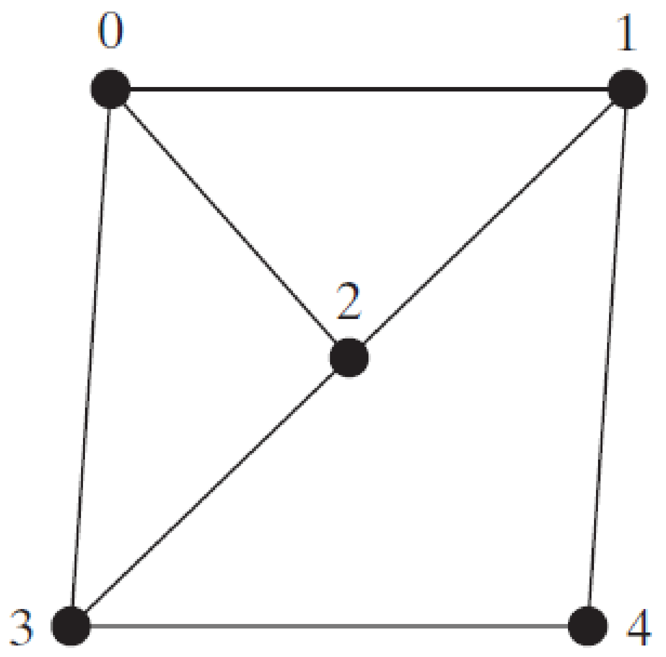
- The breadth-first search of a graph visits the vertices level by level.
  - The first level consists of the starting vertex.
  - Each next level consists of the vertices adjacent to the vertices in the preceding level.
- To ensure each vertex is visited only once, it skips a vertex if it has already been visited.

# Breadth-First Search Algorithm

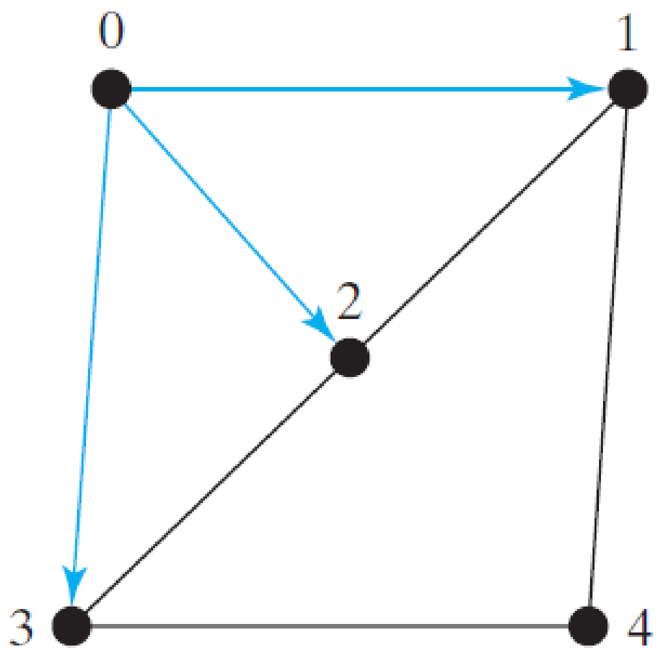
Input: G = (V, E) and a starting vertex v

Output: a BFS tree rooted at v

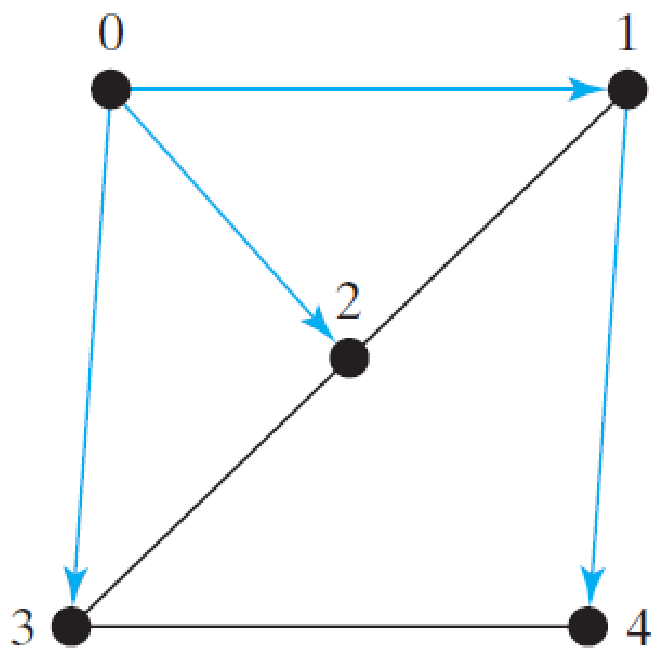
```
1.   SearchTree bfs(vertex v) {
2.     create an empty queue for storing vertices to be visited;
3.     add v into the queue;
4.     mark v visited;
5.
6.     while (the queue is not empty) {
7.       dequeue a vertex, say u, from the queue;
8.       add u into a list of traversed vertices;
9.       for each neighbor w of u
10.        if (w has not been visited) {
11.          add w into the queue;
12.          set u as the parent for w in the tree;
13.          mark w visited;
14.        } // end if
15.    } // end while
16.  } // end bfs
```



(a)



(b)



(c)

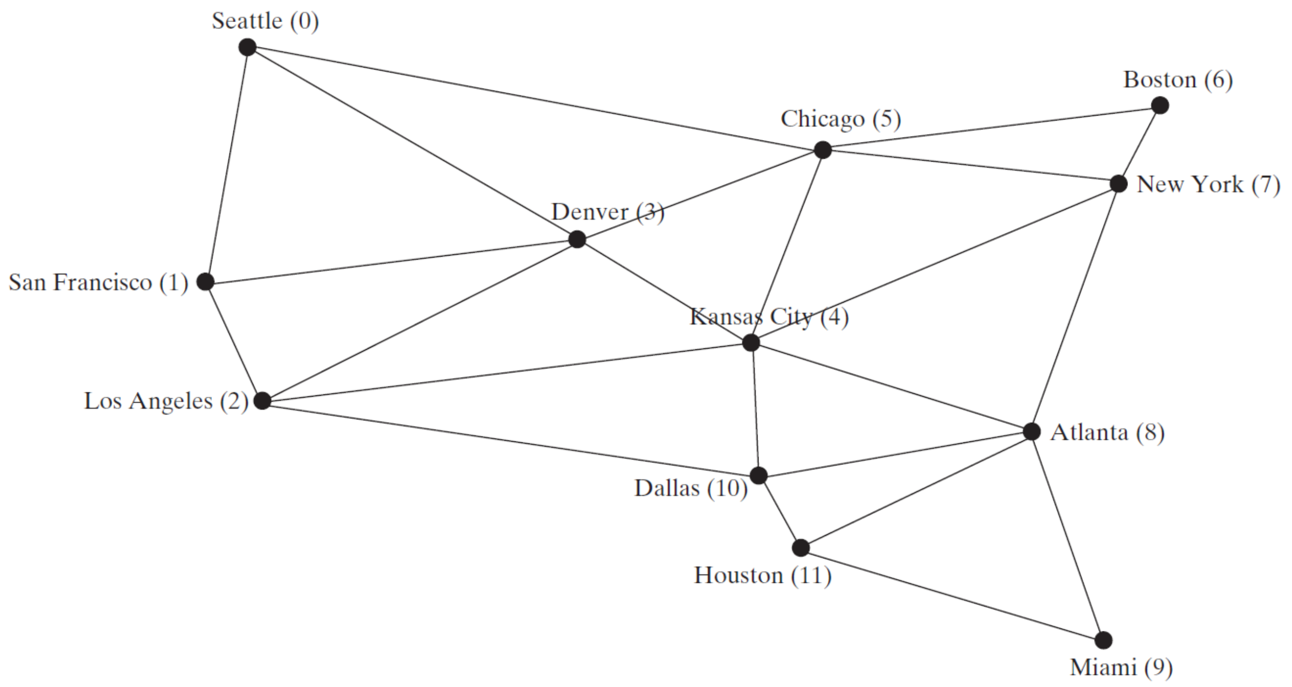
| | |
|---|---|
| Queue: 0 | isVisited[0] = true |
| Queue: 1 2 3 | isVisited[1] = true, isVisited[2] = true, isVisited[3] = true |
| Queue: 2 3 4 | isVisited[4] = true |

```
@Override /** Starting bfs search from vertex v */
public SearchTree bfs(int v)
{
  List<Integer> searchOrder = new ArrayList<>();
  int[] parent = new int[vertices.size()];
  for (int i = 0; i < parent.length; i++) parent[i] = -1;

  java.util.LinkedList<Integer> queue = new java.util.LinkedList<>();
  boolean[] isVisited = new boolean[vertices.size()];
  queue.offer(v);
  isVisited[v] = true;

  while (!queue.isEmpty()) {
    int u = queue.poll();
    searchOrder.add(u);
    for (Edge e: neighbors.get(u)) {
      if (!isVisited[e.v]) {
        queue.offer(e.v);
        parent[e.v] = u;
        isVisited[e.v] = true;
      }
    }
  }
  return new SearchTree(v, parent, searchOrder);
}
```

UnweightedGraph

# TestBFS

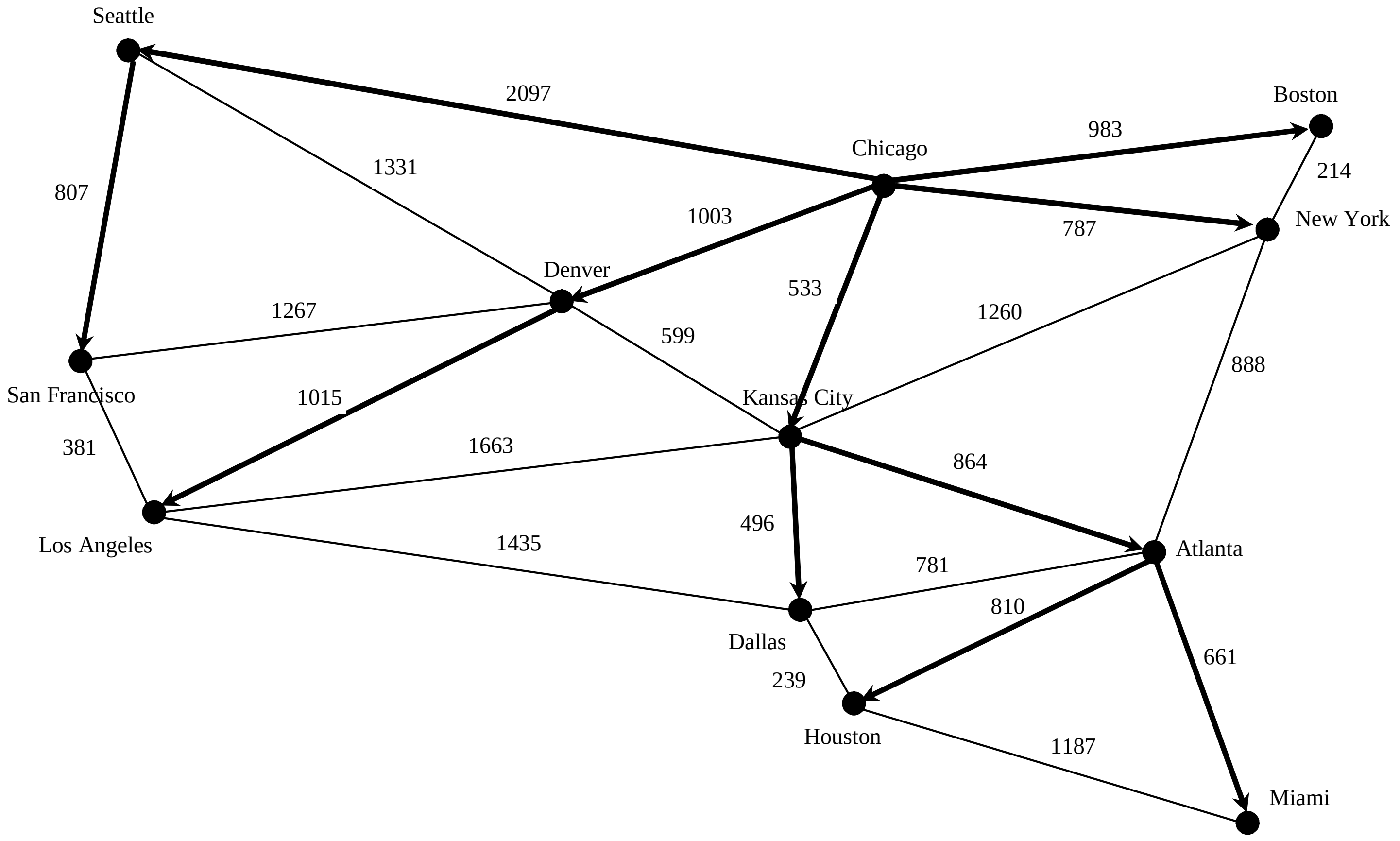
```java
public class TestBFS {
 public static void main(String[] args) {
   String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
     "Denver", "Kansas City", "Chicago", "Boston", "New York",
     "Atlanta", "Miami", "Dallas", "Houston"};

   int[][] edges = {
     {0, 1}, {0, 3}, {0, 5},
     {1, 0}, {1, 2}, {1, 3},
     {2, 1}, {2, 3}, {2, 4}, {2, 10},
     {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
     {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
     {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
     {6, 5}, {6, 7},
     {7, 4}, {7, 5}, {7, 6}, {7, 8},
     {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
     {9, 8}, {9, 11},
     {10, 2}, {10, 4}, {10, 8}, {10, 11},
     {11, 8}, {11, 9}, {11, 10}
   };
```



```java
    Graph<String> graph = new UnweightedGraph<>(vertices, edges);
    UnweightedGraph<String>.SearchTree bfs =
      graph.bfs(graph.getIndex("Chicago"));

    java.util.List<Integer> searchOrders = bfs.getSearchOrder();
    System.out.println(bfs.getNumberOfVerticesFound() +
      " vertices are searched in this order:");
    for (int i = 0; i < searchOrders.size(); i++)
      System.out.println(graph.getVertex(searchOrders.get(i)));

    for (int i = 0; i < searchOrders.size(); i++)
      if (bfs.getParent(i) != -1)
        System.out.println("parent of " + graph.getVertex(i) +
          " is " + graph.getVertex(bfs.getParent(i)));
  }
 }
```

TestBFS

```
12 vertices are searched in this order:
Chicago
Seattle
Denver
Kansas City
Boston
New York
San Francisco
Los Angeles
Atlanta
Dallas
Miami
Houston
parent of Seattle is Chicago
parent of San Francisco is Seattle
parent of Los Angeles is Denver
parent of Denver is Chicago
parent of Kansas City is Chicago
parent of Boston is Chicago
parent of New York is Chicago
parent of Atlanta is Kansas City
parent of Miami is Atlanta
parent of Dallas is Kansas City
parent of Houston is Atlanta
```



TestBFS

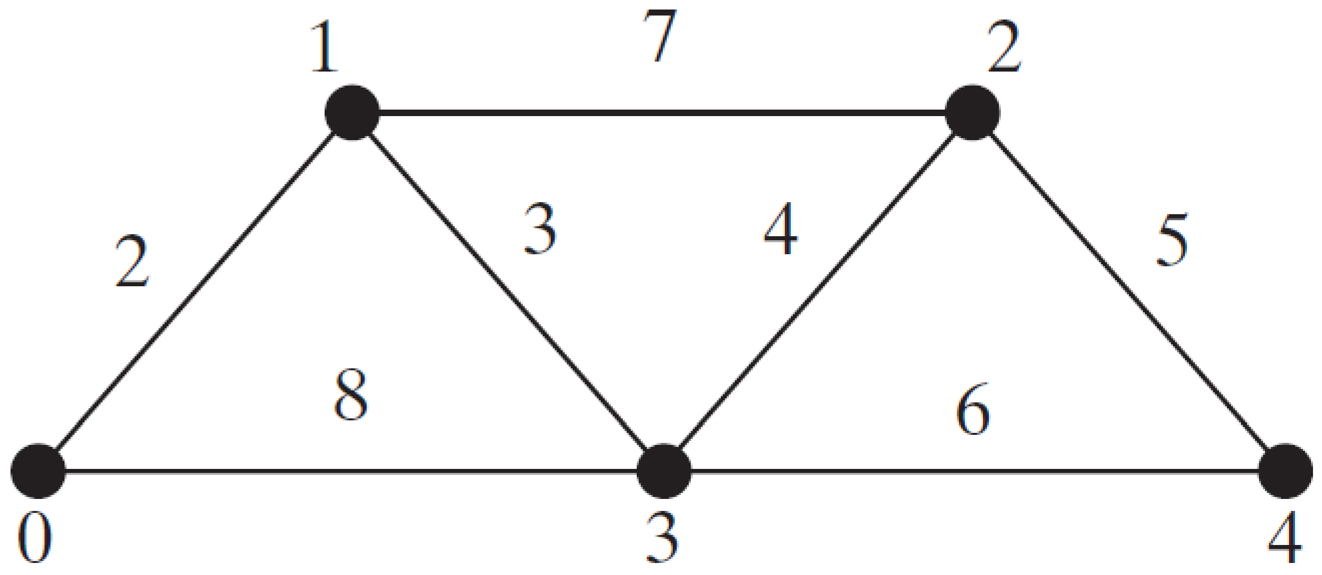
## Applications of the BFS

- Detecting whether a graph is connected.
  - A graph is connected if there is a path between any two vertices in the graph.
- Detecting whether there is a path between two vertices.
- Finding the shortest path between two vertices.
  - You can prove that the path between the root and any node in the BFS tree is the shortest path between the root and the node (in case of unweighted graphs).
- Finding all connected components.
- Detecting whether there is a cycle in the graph.
- Finding a cycle in the graph.
- Testing whether a graph is bipartite.
  - A graph is bipartite if the vertices of the graph can be divided into two disjoint sets such that no edges exist between vertices in the same set.

## Weighted Graphs

# Representing Weighted Edges: Edge Array

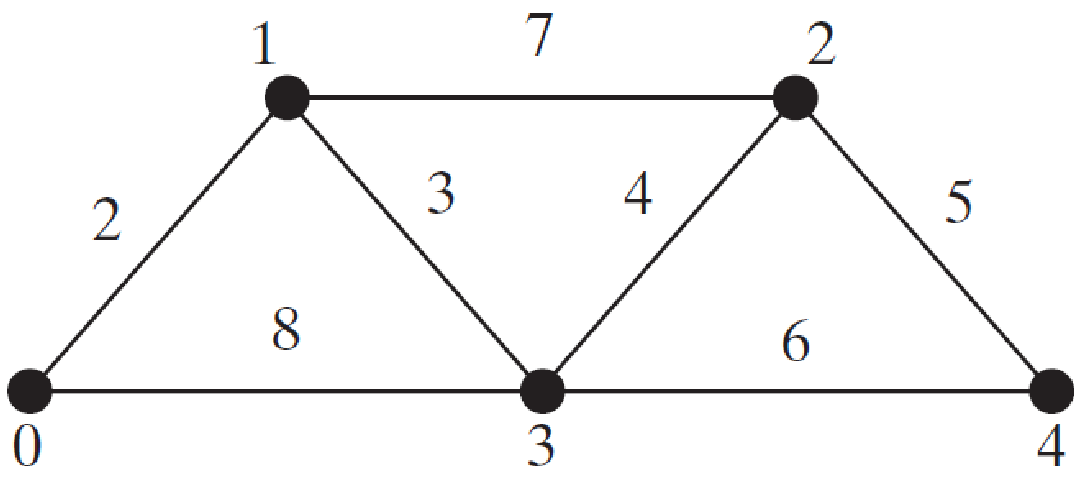
```
int[][] edges = {
  {0, 1, 2}, {0, 3, 8},
  {1, 0, 2}, {1, 2, 7}, {1, 3, 3},
  {2, 1, 7}, {2, 3, 4}, {2, 4, 5},
  {3, 0, 8}, {3, 1, 3}, {3, 2, 4}, {3, 4, 6},
  {4, 2, 5}, {4, 3, 6}
};
```



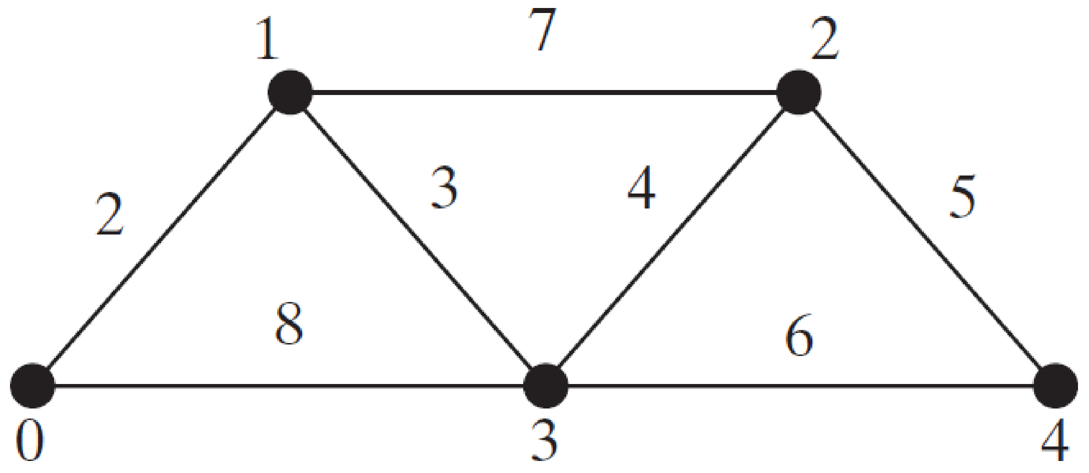
# Representing Weighted Edges: Edge Array (cont.)

```
Integer[][] adjacencyMatrix = {
  {null, 2,    null, 8,    null},
  {2,    null, 7,    3,    null},
  {null, 7,    null, 4,    5},
  {8,    3,    4,    null, 6},
  {null, null, 5,    6,    null}
 };
```

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | null | 2 | null | 8 | null |
| 1 | 2 | null | 7 | 3 | null |
| 2 | null | 7 | null | 4 | 5 |
| 3 | 8 | 3 | 4 | null | 6 |
| 4 | null | null | 5 | 6 | null |



# Representing Weighted Edges: Edge Adjacency Lists



```
java.util.List<WeightedEdge>[] neighbors = new java.util.List[5];
```

| | | | | |
|---|---|---|---|---|
| neighbors[0] | WeightedEdge(0, 1, 2) | WeightedEdge(0, 3, 8) | | |
| neighbors[1] | WeightedEdge(1, 0, 2) | WeightedEdge(1, 3, 3) | WeightedEdge(1, 2, 7) | |
| neighbors[2] | WeightedEdge(2, 3, 4) | WeightedEdge(2, 4, 5) | WeightedEdge(2, 1, 7) | |
| neighbors[3] | WeightedEdge(3, 1, 3) | WeightedEdge(3, 2, 4) | WeightedEdge(3, 4, 6) | WeightedEdge(3, 0, 8) |
| neighbors[4] | WeightedEdge(4, 2, 5) | WeightedEdge(4, 3, 6) | | |

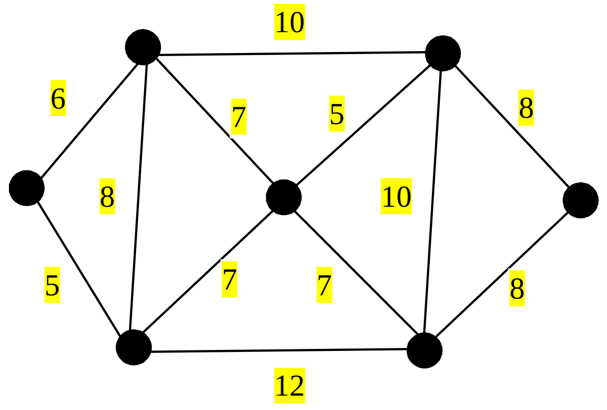
## WeightedEdge

```java
public class WeightedEdge extends Edge
       implements Comparable<WeightedEdge>
{
 public double weight; // The weight on edge (u, v)

 /** Create a weighted edge on (u, v) */
 public WeightedEdge(int u, int v, double weight)
 {
  super(u, v);
  this.weight = weight;
 }

 @Override /** Compare two edges on weights */
 public int compareTo(WeightedEdge edge)
 {
  if (weight > edge.weight) { return 1; }
  else if (weight == edge.weight) { return 0; }
  else { return -1; }
 }
}
```

```java
List<List<WeightedEdge>> list = new java.util.ArrayList<>();
```

# Minimum Spanning Tree

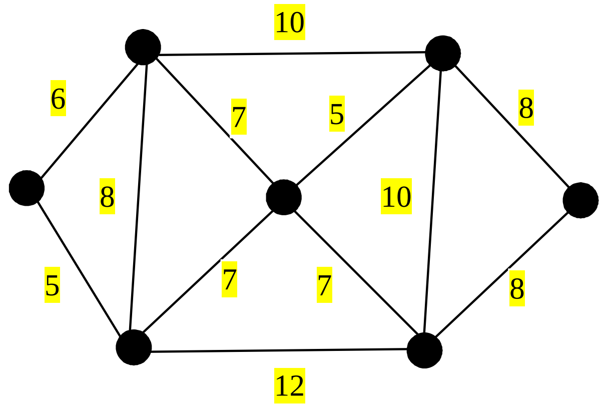
- A graph may have many spanning trees.
- A *minimum spanning tree* of a graph is a spanning tree with the minimal total weights.



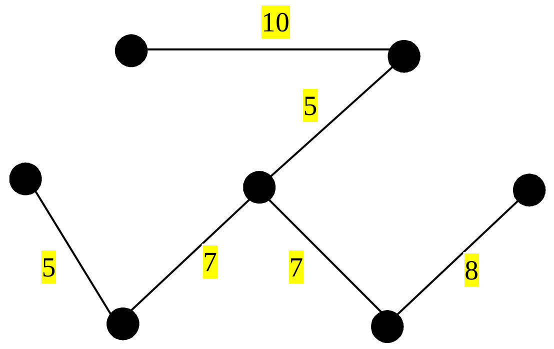
(a)

# Minimum Spanning Tree

- A graph may have many spanning trees.
- A ***minimum spanning tree*** of a graph is a spanning tree with the minimal total weights.



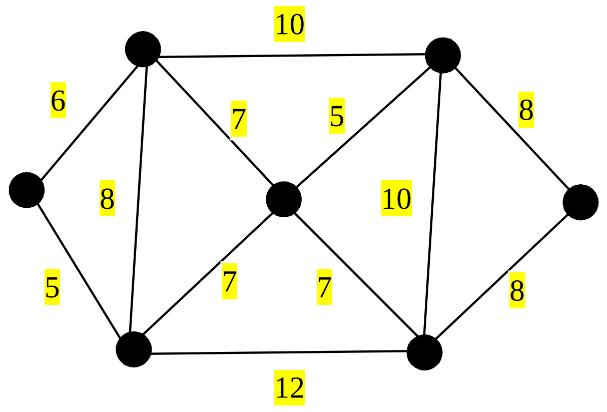
(a)



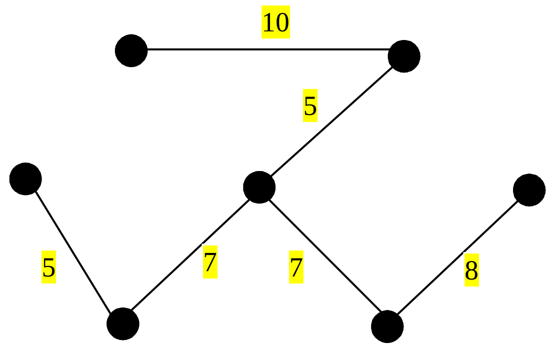
(b) Total weight is 42

# Minimum Spanning Tree

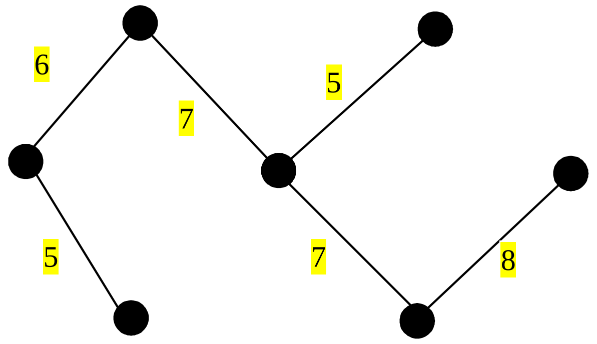
- A graph may have many spanning trees.
- A ***minimum spanning tree*** of a graph is a spanning tree with the minimal total weights.



(a)



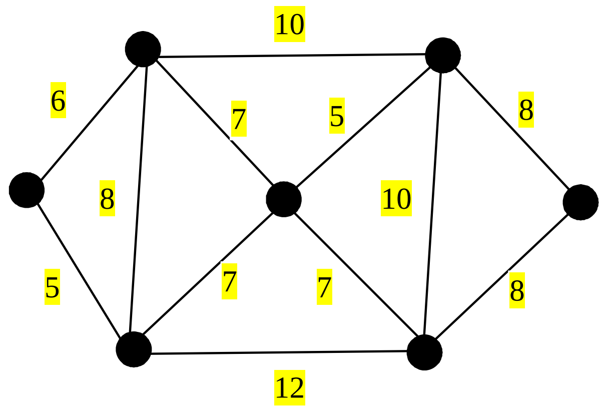
(b) Total weight is 42



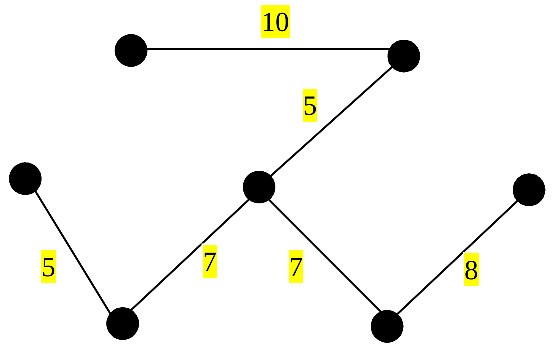
(c) Total weight is 38

# Minimum Spanning Tree

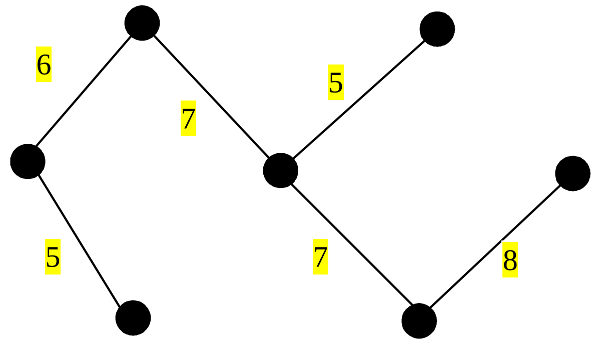
- A graph may have many spanning trees.
- A *minimum spanning tree* of a graph is a spanning tree with the minimal total weights.



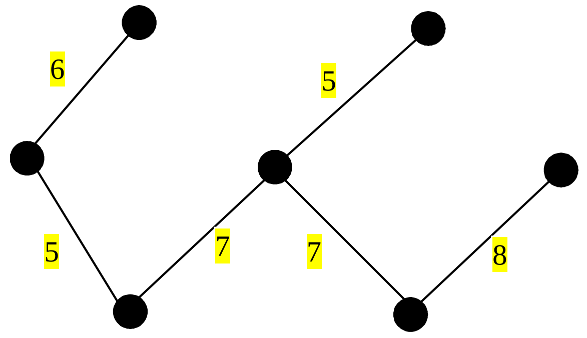
(a)



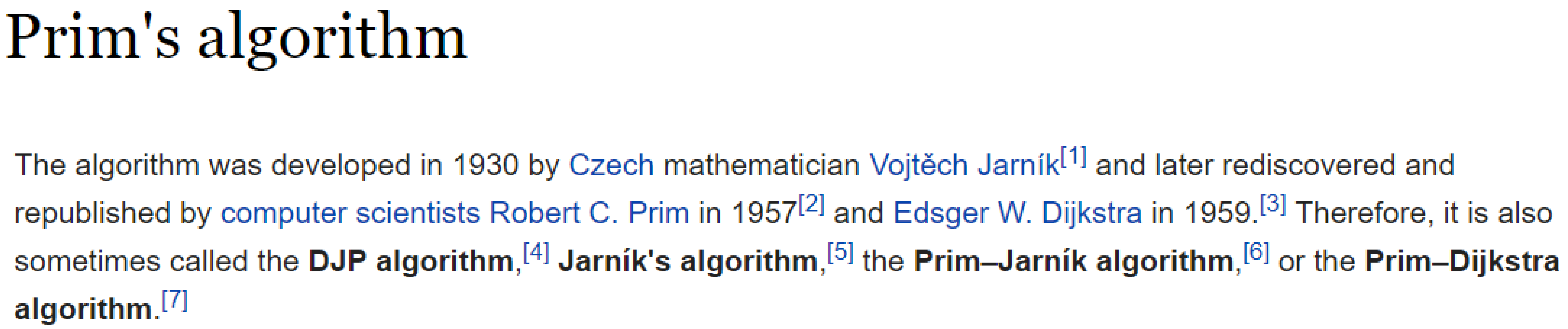
(b) Total weight is 42



(c) Total weight is 38



(d) Total weight is 38



# Prim's algorithm

The algorithm was developed in 1930 by Czech mathematician Vojtěch Jarník[1] and later rediscovered and republished by computer scientists Robert C. Prim in 1957[2] and Edsger W. Dijkstra in 1959.[3] Therefore, it is also sometimes called the **DJP algorithm**,[4] **Jarník's algorithm**,[5] the **Prim–Jarník algorithm**,[6] or the **Prim–Dijkstra algorithm**.[7]

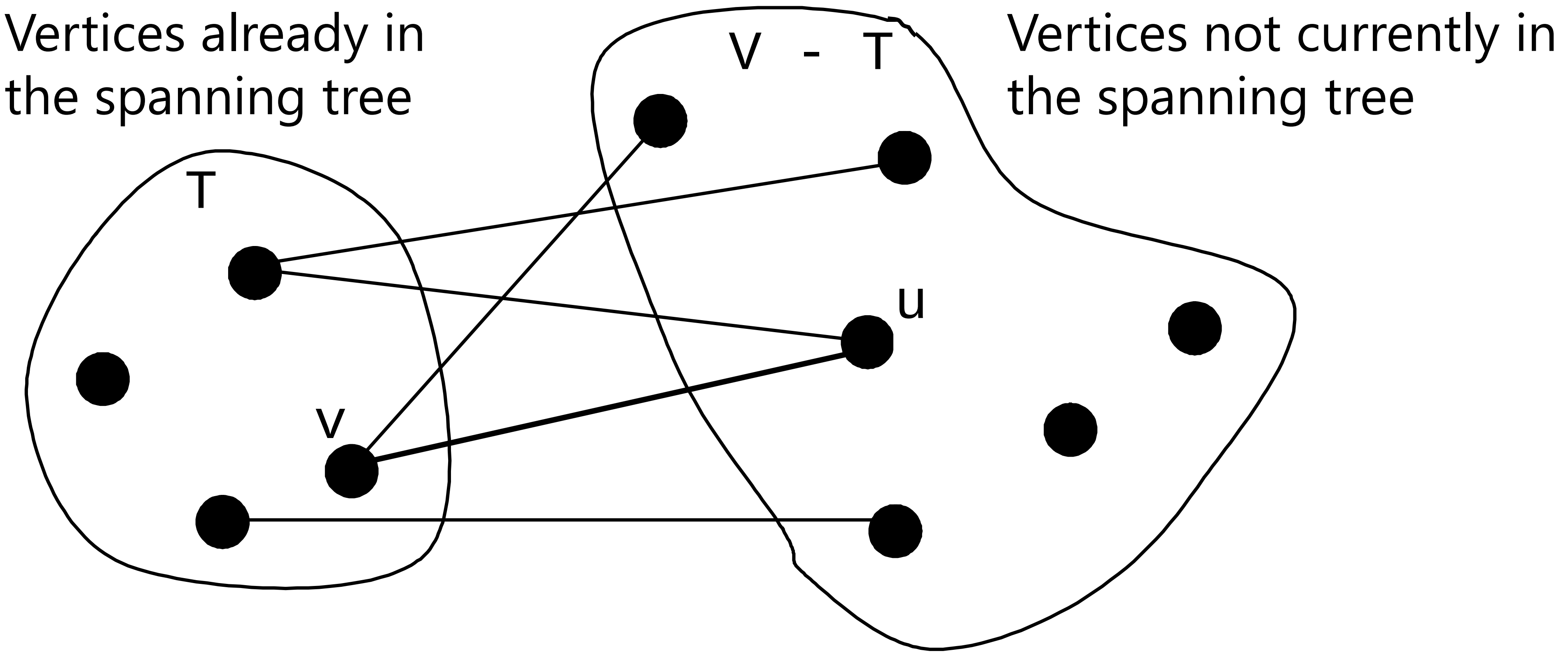
Source: https://en.wikipedia.org/wiki/Prim%27s_algorithm

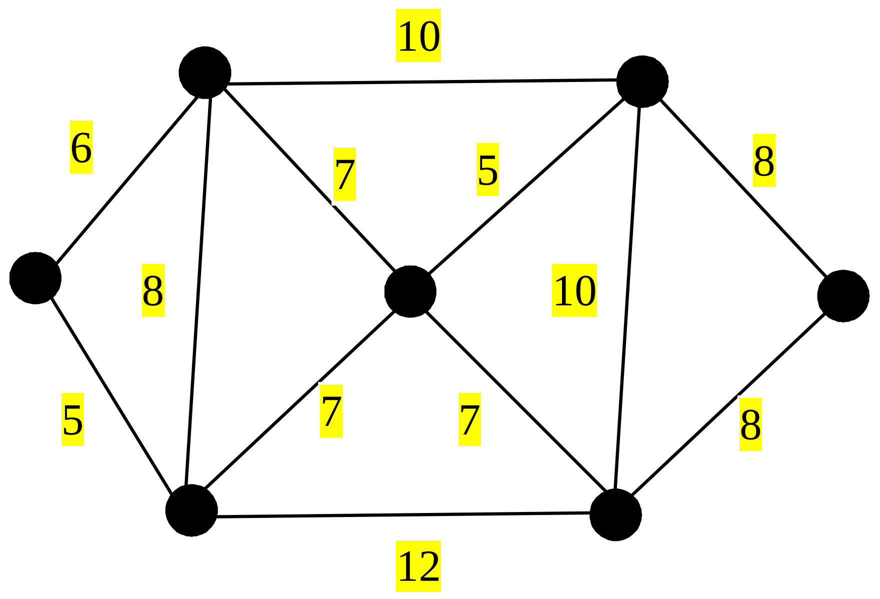
# Prim's Minimum Spanning Tree Algorithm

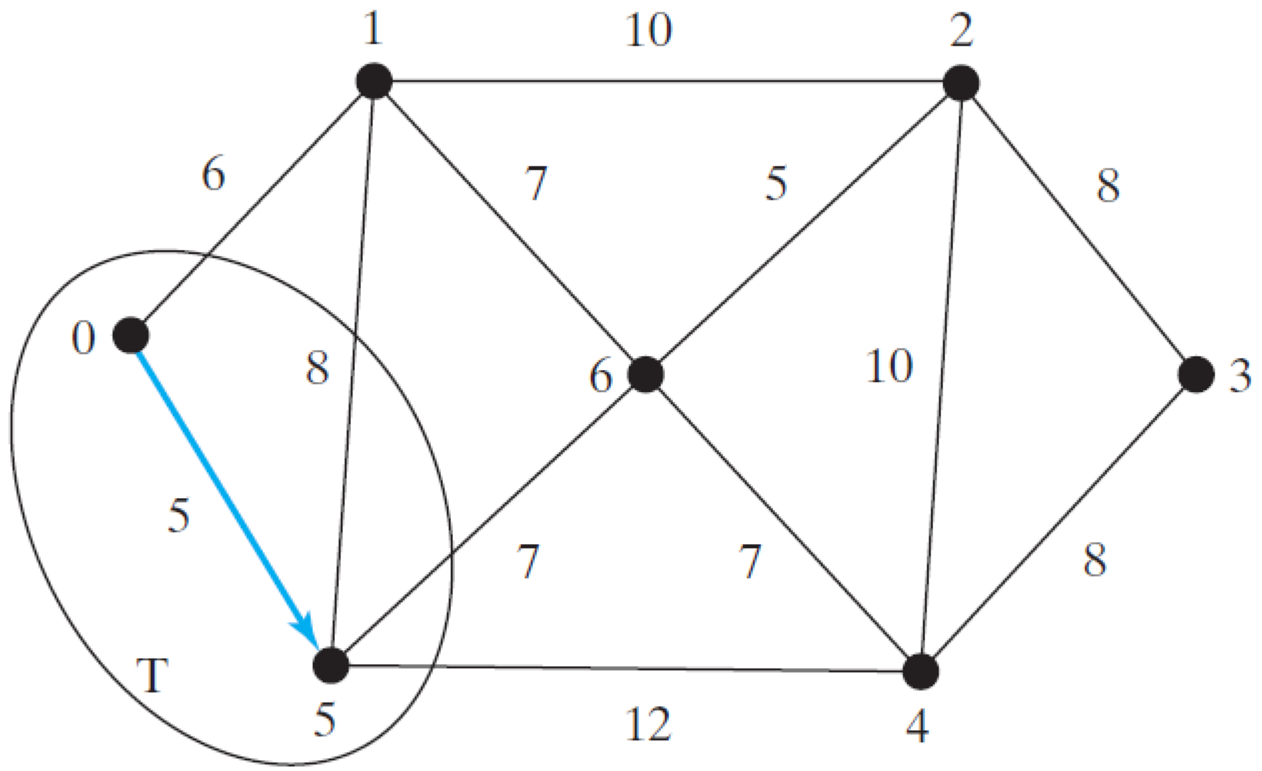
Input: A connected undirected weighted G = (V, E) with nonnegative weights
Output: MST (a minimum spanning tree)

```
1.  MST **minimumSpanningTree**() {
2.    Let T be a set for the vertices in the spanning tree;
3.    Initially, add the starting vertex, s, to T;
4.
5.    *while* (size of T < n) {
6.      Find x in T and y in V – T with the smallest weight on the edge (x, y);
7.      Add y to T and set parent[y] = x;
8.    } // end while
9.  } // minimumSpanningTree
```

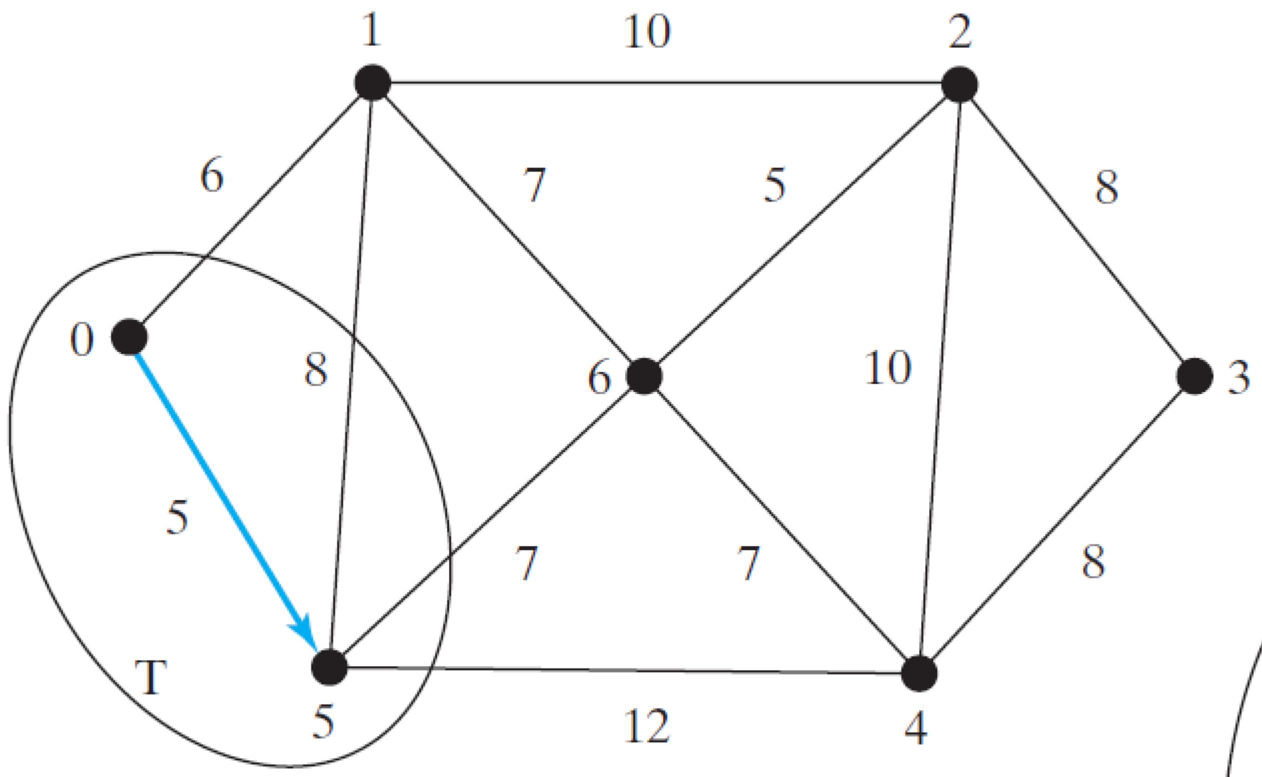
## 6. Find x in T and y in V – T with the smallest weight on the edge (x, y);


Vertices already in the spanning tree
T
v
V - T
u
Vertices not currently in the spanning tree

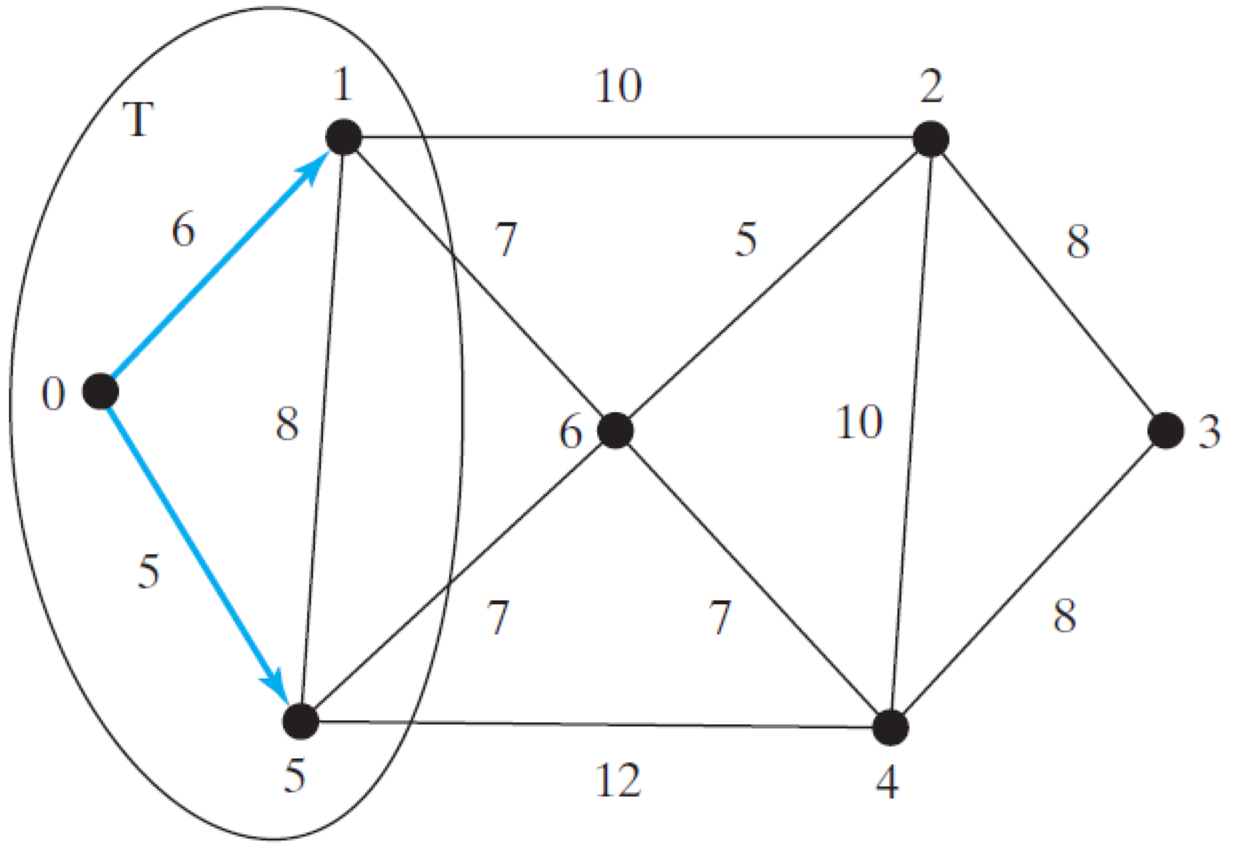




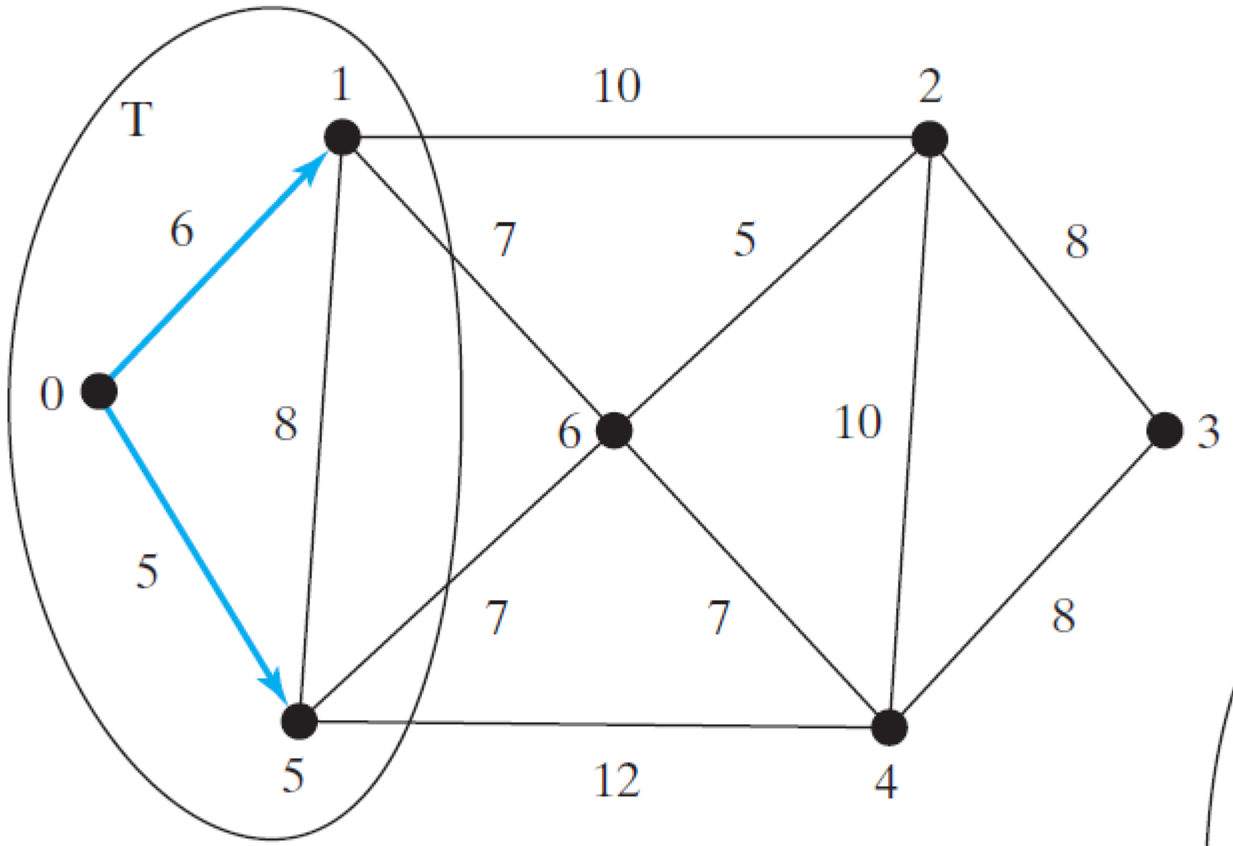
(a)



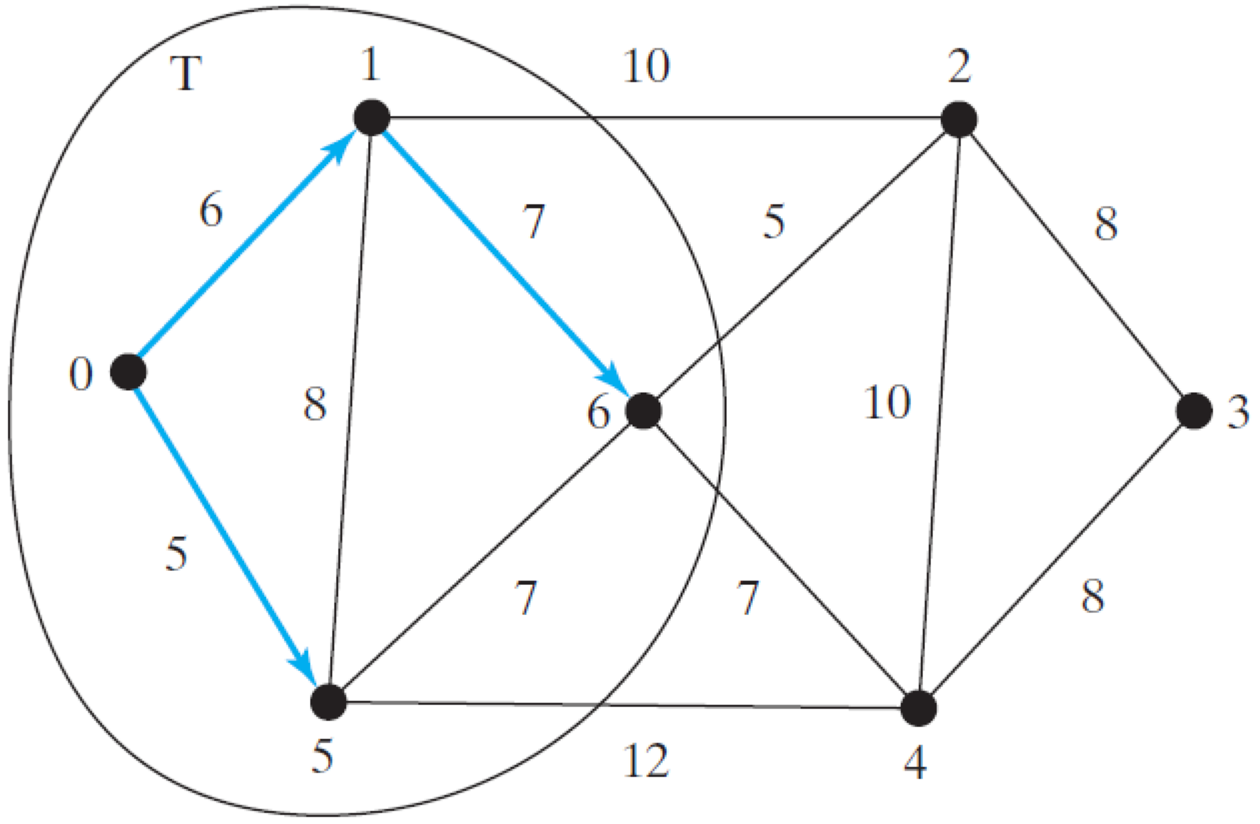
(a)



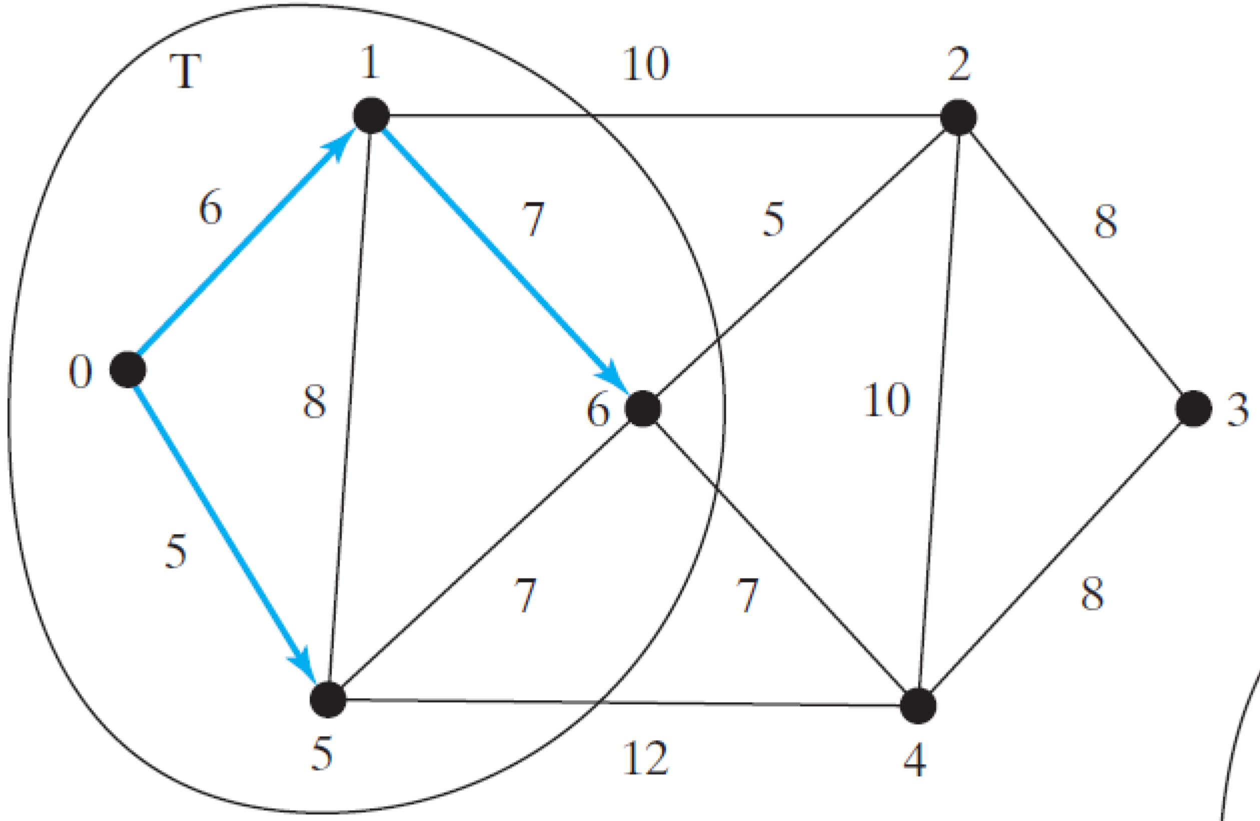
(b)



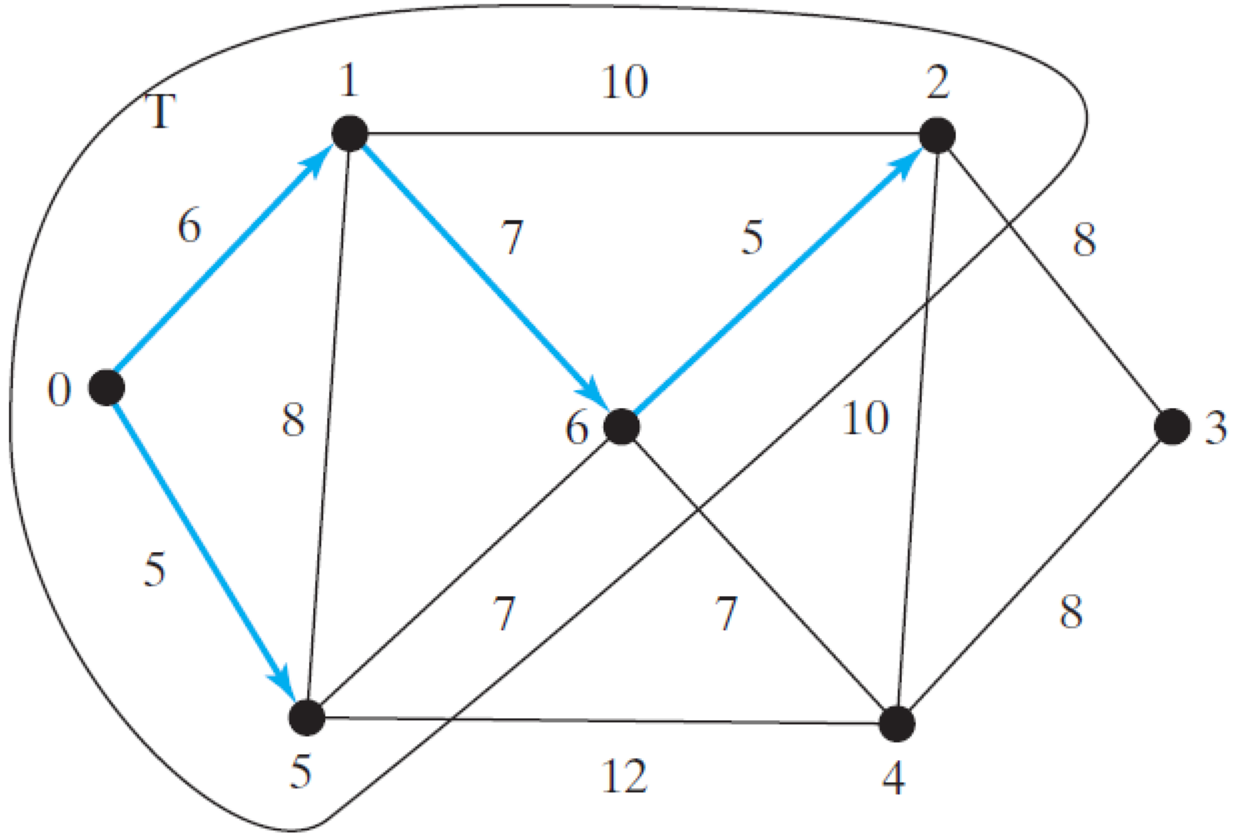
(b)



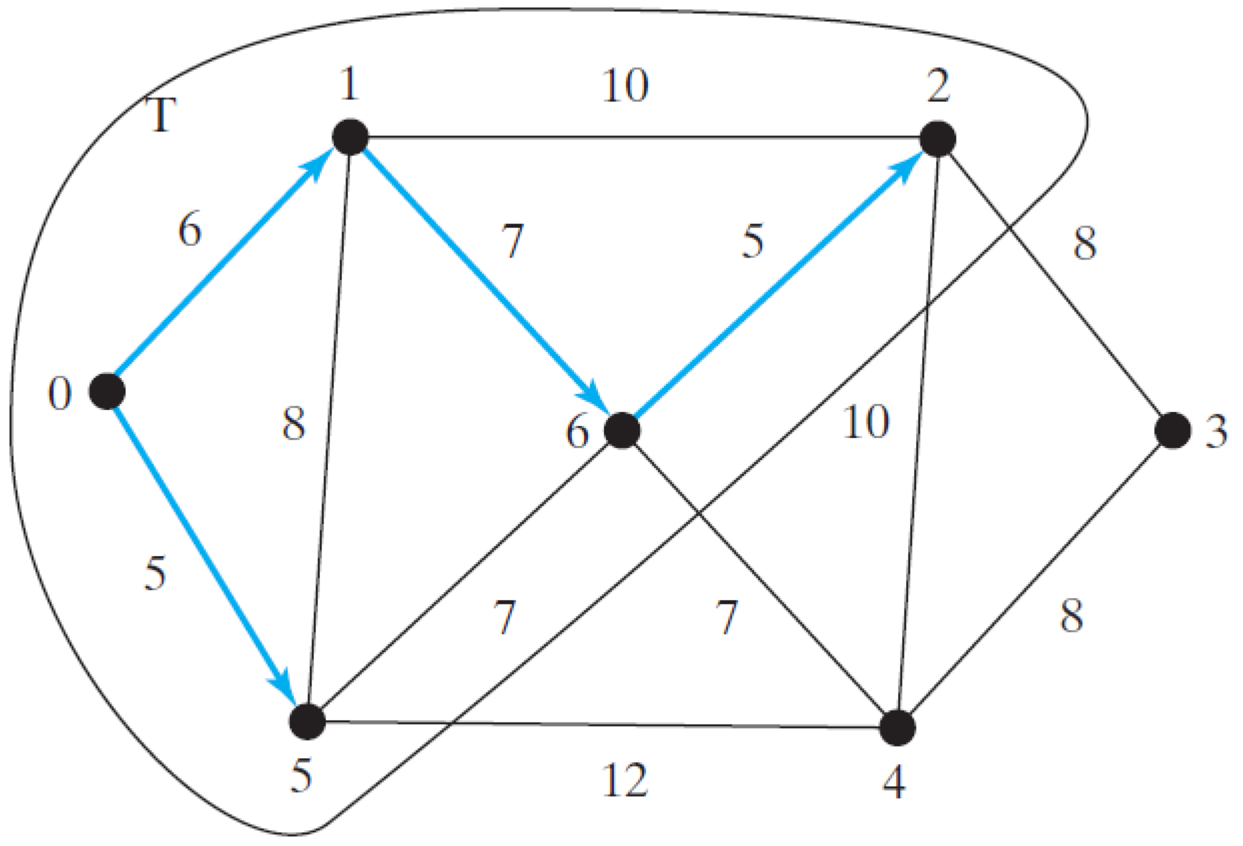
(c)



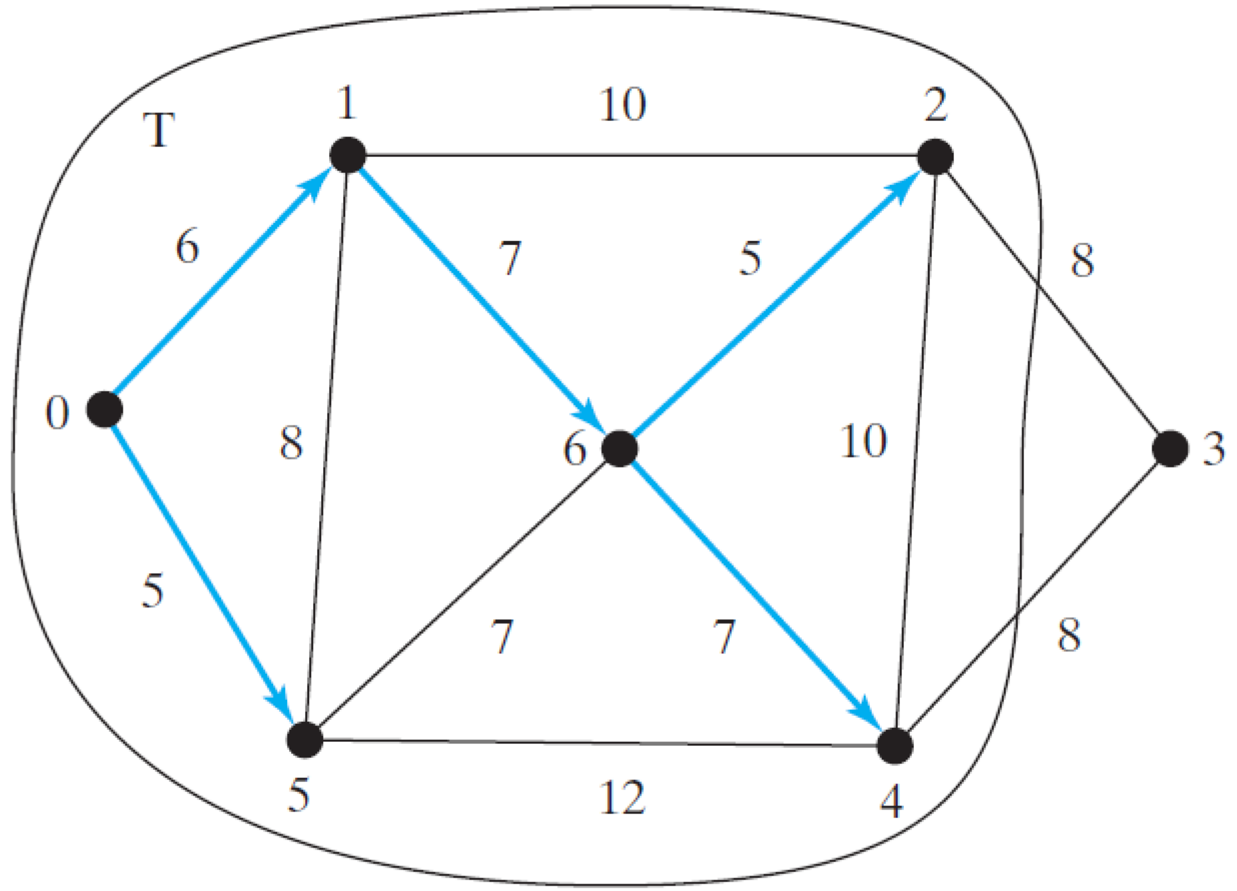
(c)



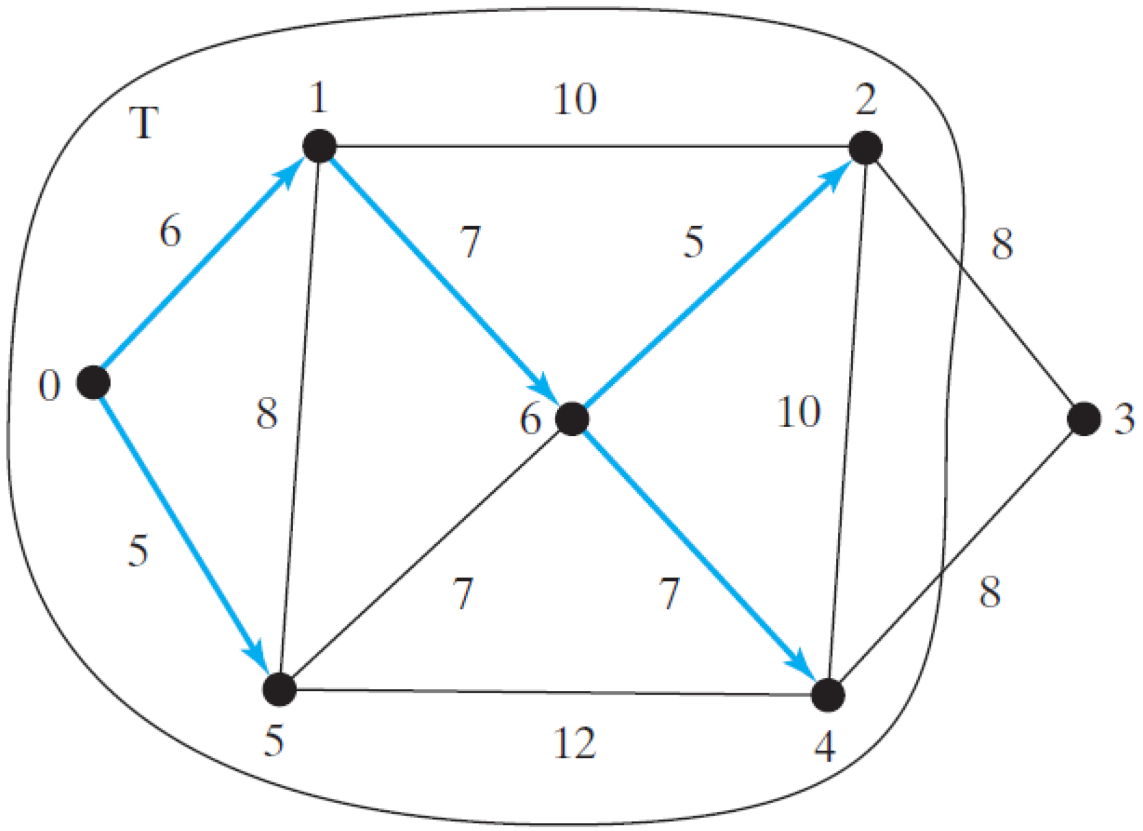
(d)



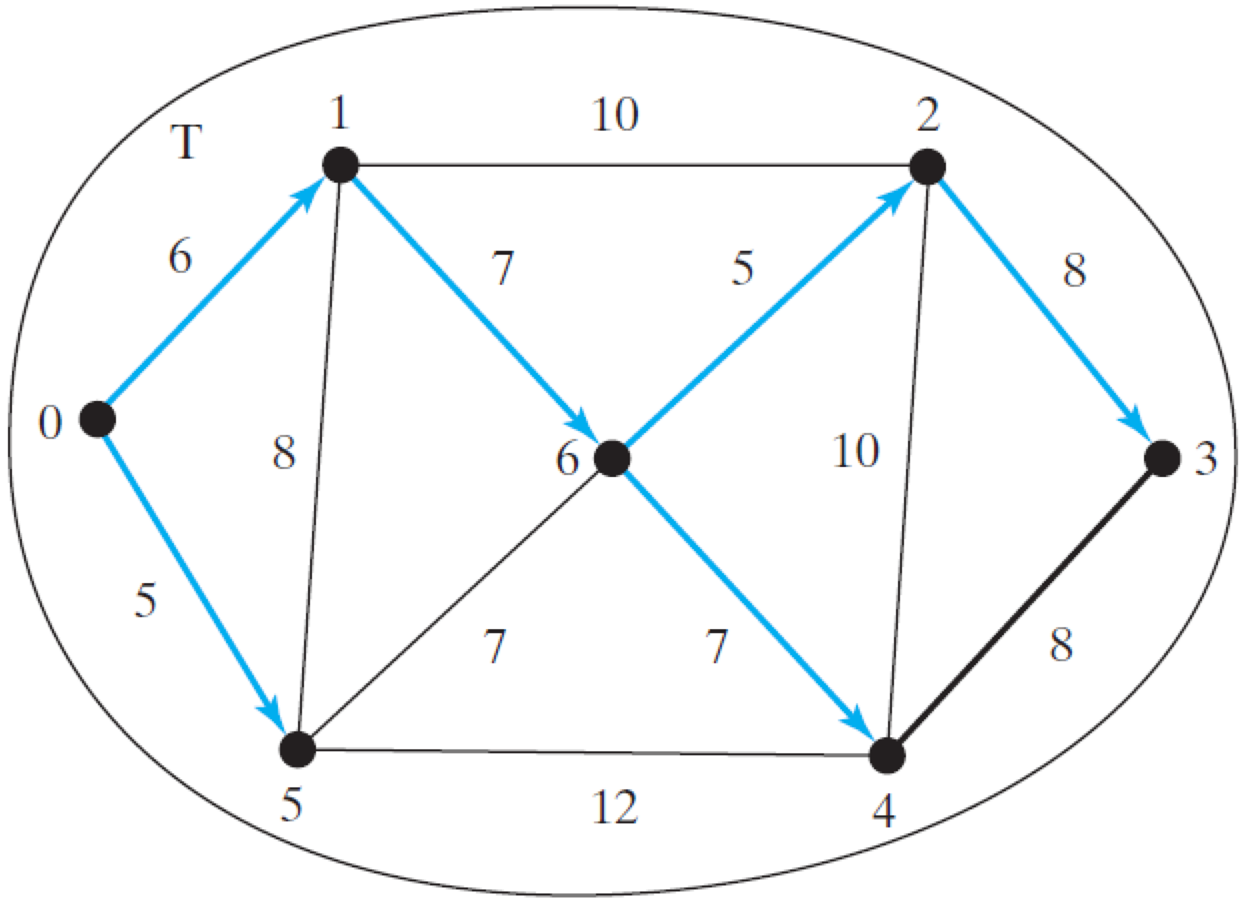
(d)



(e)



(e)



(f)

## Refined version of Prim's Minimum Spanning Tree Algorithm

Input: a graph G = (V, E) with non-negative weights
Output: a minimum spanning tree with the starting vertex s as the root

```
1.  MST **getMinimumSpanningTree**(s) {
2.    Let T be a set that contains the vertices in the spanning tree;
3.    Initially T is empty;
4.    Set cost[s] = 0; and cost[v] = infinity for all other vertices in V;
5.
6.    while (size of T < n) {
7.      Find u not in T with the smallest cost[u];
8.      Add u to T;
9.      for each v not in T and (u, v) in E
10.       if (cost[v] > w(u, v)) { cost[v] = w(u, v); parent[v] = u; } // end if
11.   } // end while
12. } // end getMinimumSpanningTree
```

## References

- Y. D. Liang, "Introduction to Java Programming and Data Structures," Comprehensive version, 11th ed. Pearson Education, Inc., 2018.