Graphs and Applications

CPT204 Advanced Object-Oriented Programming

Objectives

- To model real-world problems using graphs
- To describe the graph terminologies: ***vertices (nodes)***, ***edges***, ***directed/ undirected, weighted/unweighted***, ***connected graphs, loops, parallel edges, simple graphs, cycles, subgraphs*** and ***spanning tree***
- To **represent vertices and edges** using ***edge arrays***, ***edge objects***, ***adjacency matrices***, ***adjacency vertices list*** and ***adjacency edge lists***
- To model graphs using the **`Graph`** interface and the **`UnweightedGraph`** class
- To design and implement ***depth-first search***
- To design and implement ***breadth-first search***

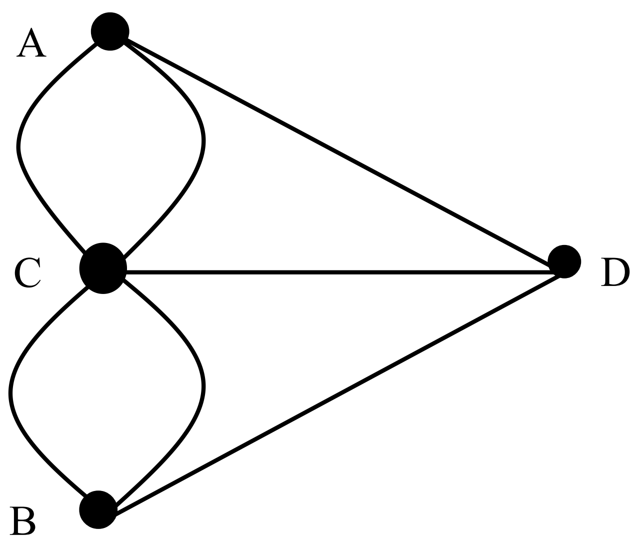
Modeling real-world problems using graphs

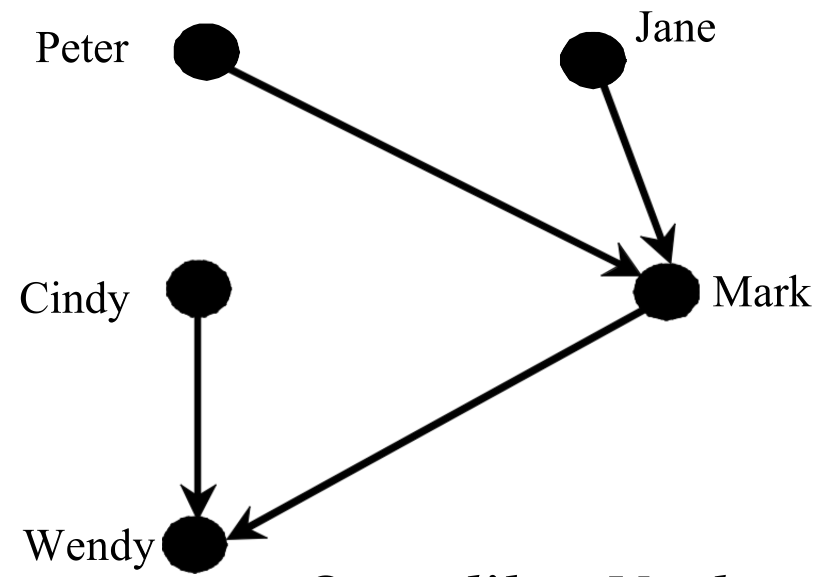
- Graphs are useful in modeling and solving real-world problems
  - For example, the problem to find the least number of flights between two cities is to find a shortest path between two vertices in a graph

- Other examples:
  - Social Media Analysis (e.g., modelling social network)
  - Computer chip design
  - Search Engine Algorithms

Basic Graph Terminology

- A ***graph*** **`G = (V, E)`**, where **V** represents a set of vertices (or nodes) and **E** represents a set of edges (or links).
- A graph may be ***undirected*** (i.e., if (x,y) is in E, then (y,x) is also in E) or ***directed***



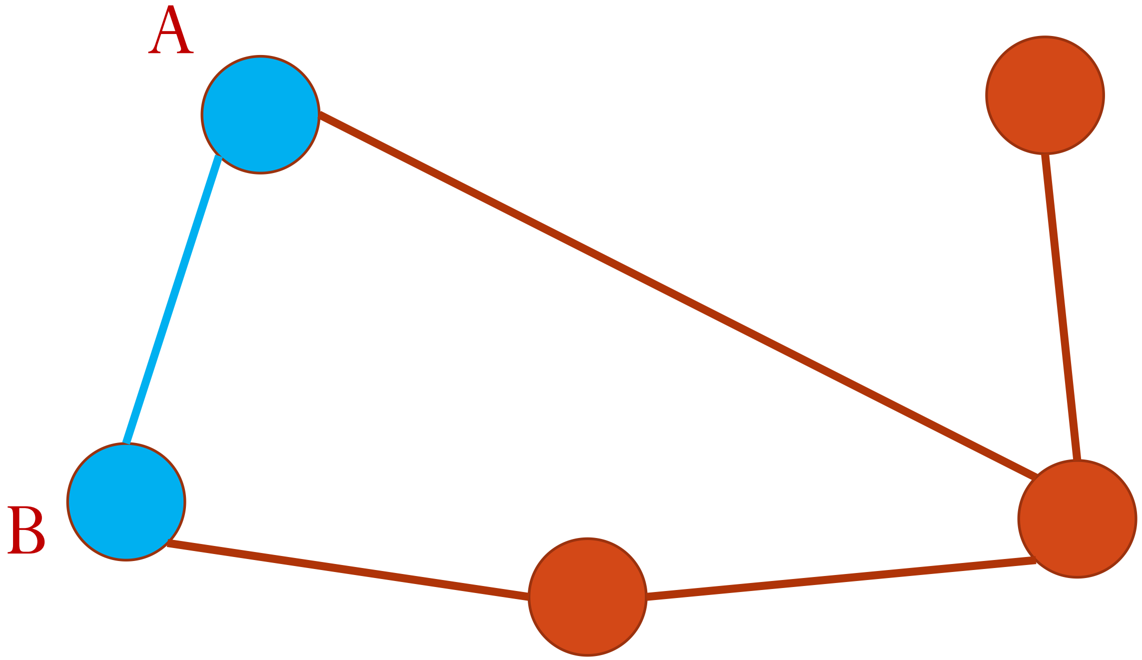


Peter likes Mark

Mark does not like Peter

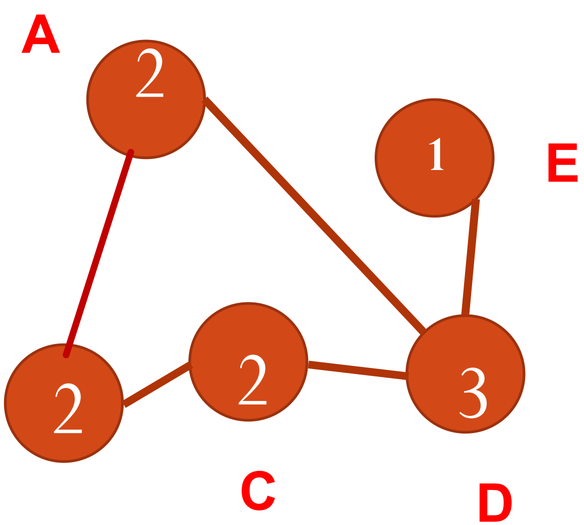
Adjacent Vertices

- Two vertices in a graph are said to be ***adjacent*** (or ***neighbors***) if they are connected by an edge
  - An edge in a graph that joins two vertices is said to be ***incident*** to both vertices
  - For example, A and B are adjacent



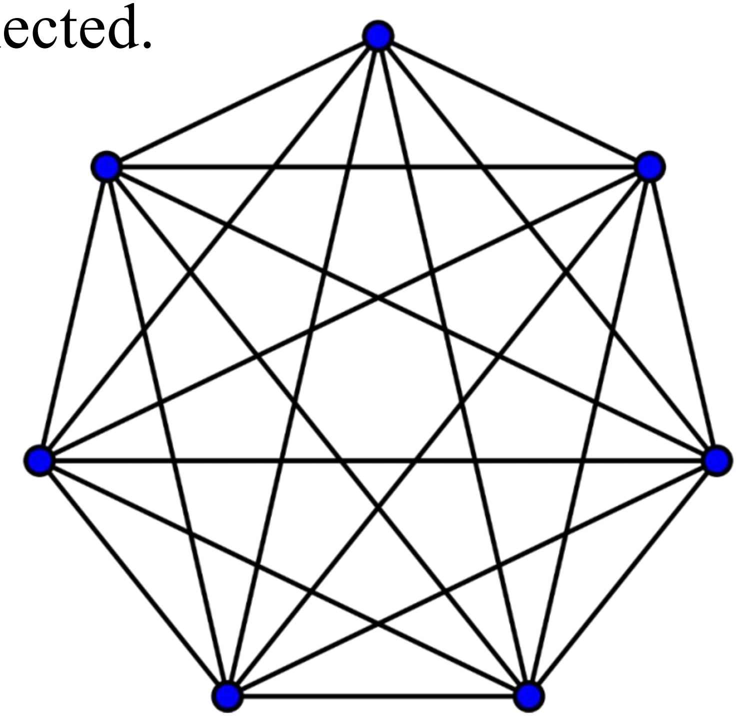
Degree

The ***degree*** of a vertex is the number of edges incident to it:



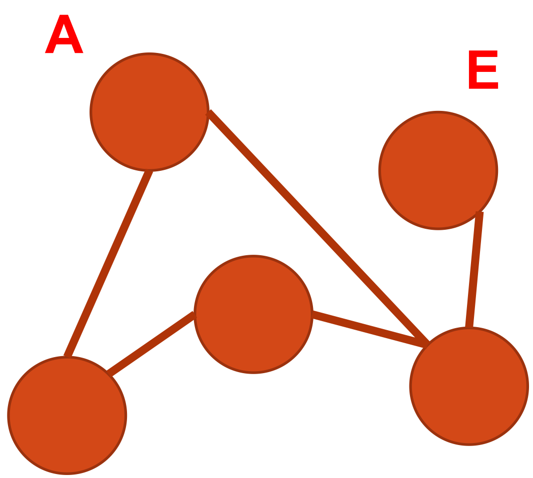
Complete graph

Every two pairs of vertices is directly connected.

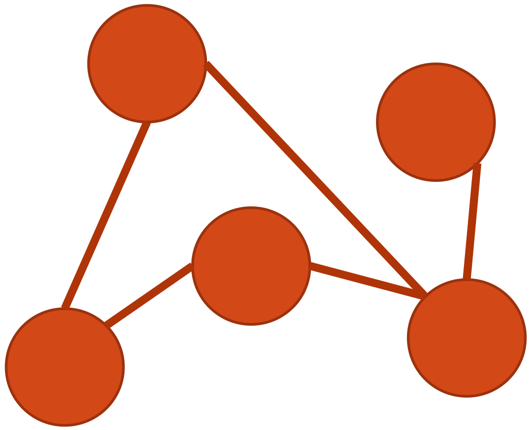


Incomplete graph

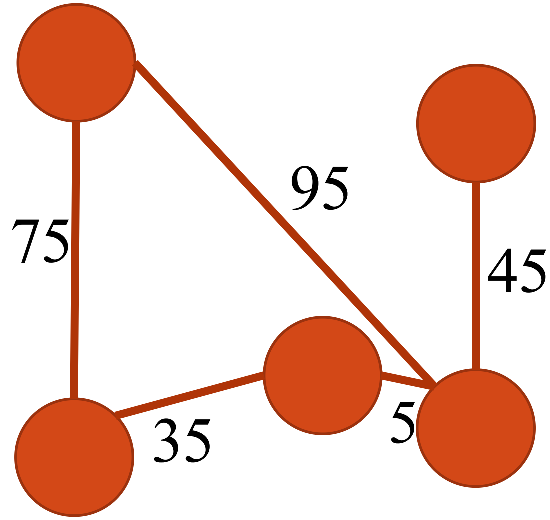
Vertex A and vertex E do not have a direct connection (no edge between them)



Unweighted

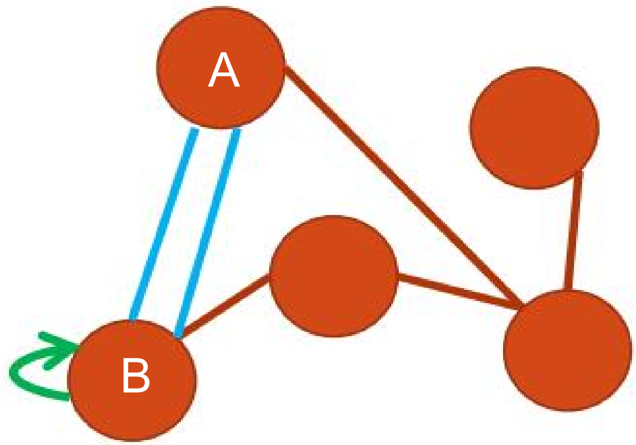


Weighted



Parallel Edges

If two vertices are connected by two or more edges, these edges are called ***parallel edges***



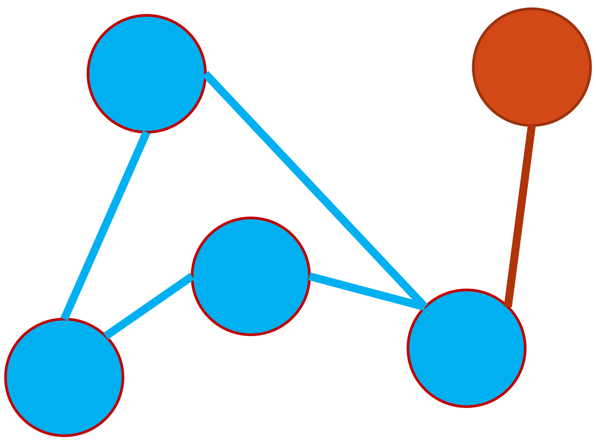
A ***loop*** is an edge that links a vertex to itself

A ***simple graph*** is one that has doesn't have any parallel edges or loops

Cycles

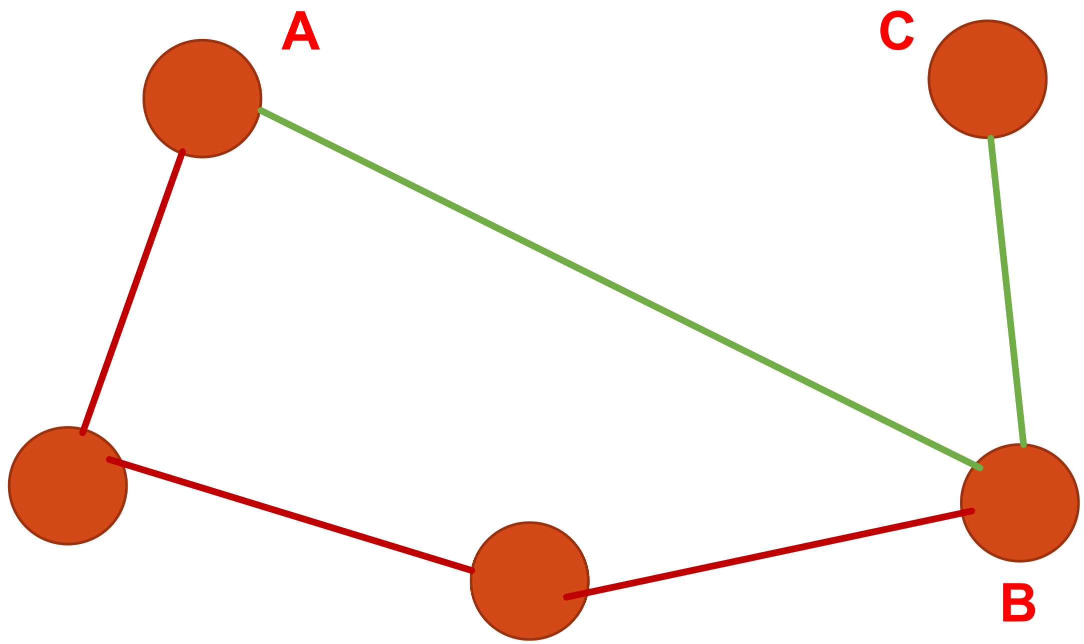
A ***closed path*** is a path where all vertices have 2 edges incident to them

A ***cycle*** is a closed path that starts from a vertex and ends at the same vertex



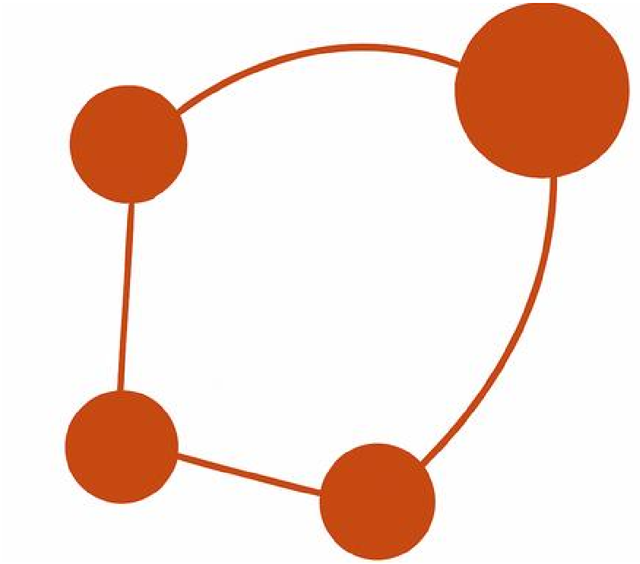
Connected graph

- A graph is ***connected*** if there exists a **path** between any two vertices in the graph

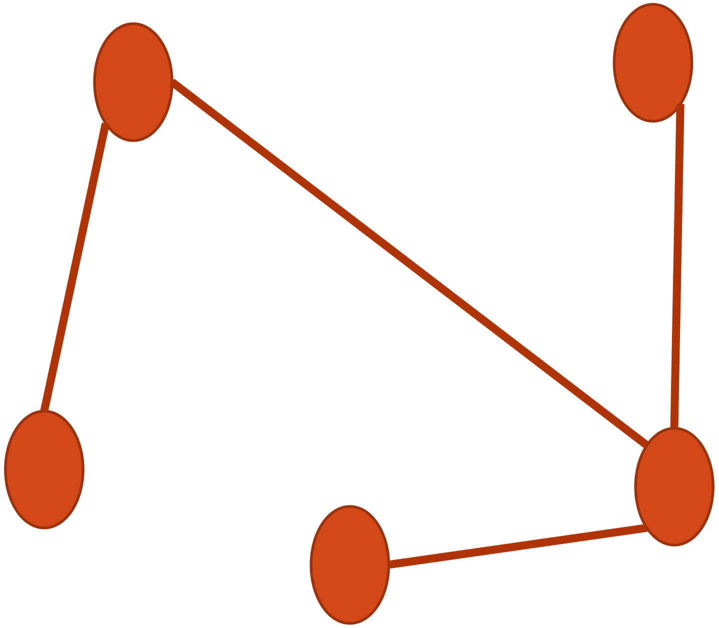


Tree

- A **connected** graph is a ***tree*** but **does not have cycles (there is no way to loop back to where we started)**



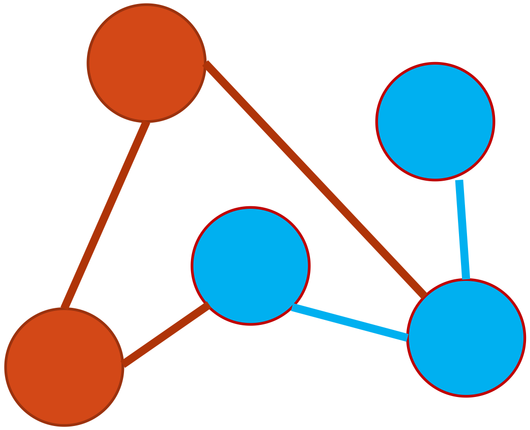
Not a Tree



A Tree

Subgraphs

A ***subgraph*** of a graph G is a graph whose vertex set is a subset of that of G and whose edge set is a subset of that of G



Representing Graphs

$G = (V, E)$

- Representing Vertices
- Representing Edges: Edge Array
- Representing Edges: Edge Objects
- Representing Edges: Adjacency Matrices
- Representing Edges: Adjacency Lists

Representing Vertices

Example 1:

```
String[] vertices = {"Seattle",
"San Francisco", "Los Angles",
"Denver", "Kansas City", …};
```

Example 2:

```
List<String> vertices;
vertices.add("Seattle");…
```

Representing Vertices

A more object-oriented approach

Example 3

```
public class City {
// define the attributes (e.g., private String cityName;)
// define the constructor (public City(String cityName,...){})
// getter and setter methods (public String getCityName(),...)
}

City city0 = new City("Seattle")
City city1 = new City("San Francisco");
City[] vertices = {city0, city1,...}
```

Representing Edges: Edge Array

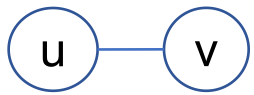
- The edges can be represented using a two-dimensional array of all the edges:

```
int[][] edges = {
  {0, 1}, {0, 3}, {0, 5}, // edges from vertex 0
  {1, 0}, {1, 2}, {1, 3}, // edges from vertex 1
  {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}, ... };
```

- **Each line indicates the edges from a particular vertex.**
- **Each pair, e.g., {0,1}, means there is an edge from vertex 0 to vertex 1**
- **In the case of unweighted graph, {0,1} and {1,0} are the same edge**

Representing Edges: Edge Objects

```
public class Edge {
   // Define u and v to the two endpoints of an edge
   int u, v;
```

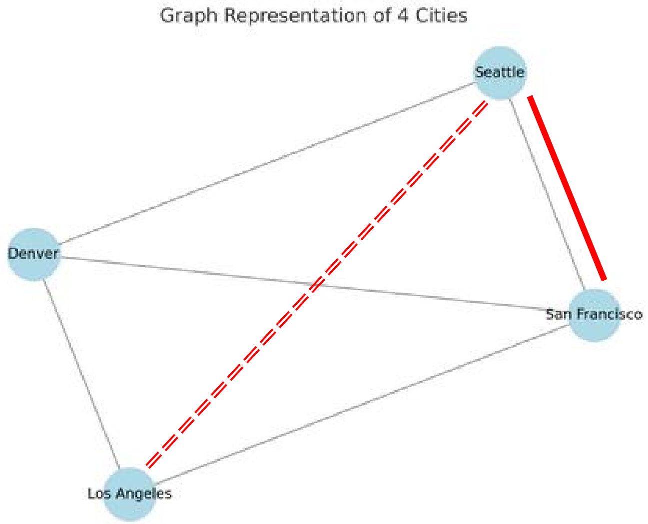



```
  // The rest codes
  // e.g., constructor, getter, setter and so on
  }

   List<Edge> list = new ArrayList();
   list.add(new Edge(0, 1));
   list.add(new Edge(0, 3));
```

- Storing **`Edge`** objects in an **`ArrayList`** is useful if you don't know the number of edges in advance

Representing Edges: Adjacency Matrix

- Knowing that the graph has **N** vertices and we can use a two-dimensional **N * N** matrix to represent the existence of edges, where we use 1 to indicate an edge, and 0 when there is no edge.



Adjacency Matrix (4 Cities)

	Seattle	San Francisco	Los Angeles	Denver
Seattle	0	1	0	1
San Francisco	1	0	1	1
Los Angeles	0	1	0	1
Denver	1	1	1	0

```
int[][] adjacencyMatrix = {
    {0, 1, 0, 1}, // Seattle
    {1, 0, 1, 1}, // San Francisco
    {0, 1, 0, 1}, // Los Angeles
    {1, 1, 1, 0}, // Denver
};
```

Representing Edges: Adjacency Vertex List

```
// Create an array of 12 integer
lists, each list representing a
city

List<Integer>[] neighbors = new
List[12];

// Add integer (i.e., the city
index) to the neighbor class,
representing neighboring cities
connected to the current city

neighbors[0].add(1); // San Francisco
neighbors[0].add(3); // Denver
neighbors[0].add(5); // Chicago

neighbors[1].add(0); // Seattle
neighbors[1].add(2); // Los Angeles
neighbors[1].add(3); // Denver
```

Seattle	neighbors[0]	1	3	5			
San Francisco	neighbors[1]	0	2	3			
Los Angeles	neighbors[2]	1	3	4	10		
Denver	neighbors[3]	0	1	2	4	5	
Kansas City	neighbors[4]	2	3	5	7	8	10
Chicago	neighbors[5]	0	3	4	6	7	
Boston	neighbors[6]	5	7				
New York	neighbors[7]	4	5	6	8		
Atlanta	neighbors[8]	4	7	9	10	11	
Miami	neighbors[9]	8	11				
Dallas	neighbors[10]	2	4	8	11		
Houston	neighbors[11]	8	9	10			

Representing Edges: Adjacency Edge List

```
List<Edge>[] neighbors = new List[12];
```

Seattle	neighbors[0]	Edge(0, 1)	Edge(0, 3)	Edge(0, 5)			
San Francisco	neighbors[1]	Edge(1, 0)	Edge(1, 2)	Edge(1, 3)			
Los Angeles	neighbors[2]	Edge(2, 1)	Edge(2, 3)	Edge(2, 4)	Edge(2, 10)		
Denver	neighbors[3]	Edge(3, 0)	Edge(3, 1)	Edge(3, 2)	Edge(3, 4)	Edge(3, 5)	
Kansas City	neighbors[4]	Edge(4, 2)	Edge(4, 3)	Edge(4, 5)	Edge(4, 7)	Edge(4, 8)	Edge(4, 10)
Chicago	neighbors[5]	Edge(5, 0)	Edge(5, 3)	Edge(5, 4)	Edge(5, 6)	Edge(5, 7)	
Boston	neighbors[6]	Edge(6, 5)	Edge(6, 7)				
New York	neighbors[7]	Edge(7, 4)	Edge(7, 5)	Edge(7, 6)	Edge(7, 8)		
Atlanta	neighbors[8]	Edge(8, 4)	Edge(8, 7)	Edge(8, 9)	Edge(8, 10)	Edge(8, 11)	
Miami	neighbors[9]	Edge(9, 8)	Edge(9, 11)				
Dallas	neighbors[10]	Edge(10, 2)	Edge(10, 4)	Edge(10, 8)	Edge(10, 11)		
Houston	neighbors[11]	Edge(11, 8)	Edge(11, 9)	Edge(11, 10)			

```
public class Edge {
    int u; // from vertex
    int v; // to vertex
    public Edge(int u, int v) {
        this.u = u;
        this.v = v;
    }
}

......

List<Edge>[] neighbors = new List[12];

neighbors[0].add(new Edge(0, 1));  // Seattle → San Francisco
neighbors[0].add(new Edge(0, 3));  // Seattle → Denver
neighbors[0].add(new Edge(0, 5));  // Seattle → Chicago
```

Modeling Graphs

- We will define an interface named **Graph** that contains all the common operations of graphs.
- The concrete graphs classes (e.g., **UnweightedGraph and WeightedGraph**) implement the graph interface.

  - They define **internal data structures** to store graph information, such as the list or adjacent list of vertices.
  - They provide **multiple constructors** to allow users to initialize graphs from various types of input, such as arrays, lists of vertices, or edge sets
  - They **implement the abstract methods** declared in the Graph interface by providing concrete logic for operations

Graph Interface

```java
package org.example;

public interface Graph<V> {  1 usage  2 implementations
    /**
     * Return the number of vertices in the graph
     */
    public int getSize();  11 usages  1 implementation

    /**
     * Return the vertices in the graph
     */
    public java.util.List<V> getVertices();  no usages  1 implementation

    /**
     * Return the object for the specified vertex index
     */
    public V getVertex(int index);  4 usages  1 implementation

    /**
     * Return the index for the specified vertex object
     */
    public int getIndex(V v);  2 usages  1 implementation

    /**
     * Return the neighbors of vertex with the specified index
     */
    public java.util.List<Integer> getNeighbors(int index);  no usages  1 implementation

    /**
     * Return the degree for a specified vertex
     */
    public int getDegree(int v);  no usages  1 implementation

    /**
     * Print the edges
     */
    public void printEdges();  no usages  1 implementation

    /**
     * Clear the graph
     */
    public void clear();  no usages  1 implementation

    /**
     * Add a vertex to the graph
     */
    public boolean addVertex(V vertex);  4 usages  1 implementation

    /**
     * Add an edge (u, v) to the graph
     */
    public boolean addEdge(int u, int v);  2 usages  1 implementation

    /**
     * Add an edge to the graph
     */
    public boolean addEdge(Edge e);  2 usages  1 implementation

    /**
     * Remove a vertex v from the graph, return true if successful
     */
    public boolean remove(V v);  no usages  1 implementation

    /**
     * Remove an edge (u, v) from the graph
     */
    public boolean remove(int u, int v);  no usages  1 implementation

    /**
     * Obtain a depth-first search tree
     */
    public UnweightedGraph<V>.SearchTree dfs(int v);  no usages  1 implementation

    /**
     * Obtain a breadth-first search tree
     */
    public UnweightedGraph<V>.SearchTree bfs(int v);  no usages  1 implementation
}
```

Defines the abstract methods like getSize(), with **no** specific implementation details inside.

UnweightedGraph.java

```java
public class UnweightedGraph<V> implements Graph<V> {  3 usages  1 inheritor
    protected List<V> vertices = new ArrayList<>(); // Store vertices  20 usages
    protected List<List<Edge>> neighbors  18 usages
            = new ArrayList<>(); // Adjacency lists
```

Declare and initialize the essential **data structures** used to store vertices and edges in the graph.

UnweightedGraph.java (cont.)

```java
/**
 * Construct an empty graph
 */
public UnweightedGraph() {  5 usages
}

/**
 * Construct a graph from vertices and edges stored in arrays
 */
public UnweightedGraph(V[] vertices, int[][] edges) {  no usages
    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) {  no usages
    for (int i = 0; i < vertices.size(); i++)
        addVertex(vertices.get(i));

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

Provide **multiple constructors** to allow users to initialize graphs from various types of input

UnweightedGraph.java (cont.)

```java
@Override  11 usages
/** Return the number of vertices in the graph */
public int getSize() { return vertices.size(); }

@Override  no usages
/** Return the vertices in the graph */
public List<V> getVertices() { return vertices; }

@Override  4 usages
/** Return the object for the specified vertex */
public V getVertex(int index) { return vertices.get(index); }

@Override  2 usages
/** Return the index for the specified vertex object */
public int getIndex(V v) { return vertices.indexOf(v); }

@Override  no usages
/** 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;
}
```

Implement the abstract methods declared in the Graph interface by providing concrete logic for operations.

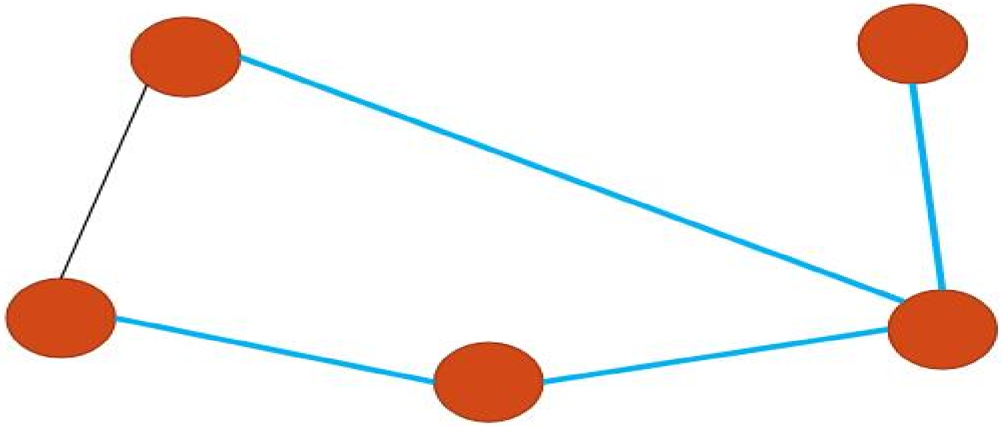
- *In the interface:* **public int getSize();**
- *In the concrete class:*

  @Override

  public int getSize() {return vertices.size();}

Graph Traversals

- **Graph traversal** is the process of visiting **each vertex** in the graph exactly **once.**
- There are two popular ways to traverse a graph: **depth-first search (DFS) and breadth-first search (BFS)**
- Both traversals result in **a spanning tree,** which is a subgraph of the whole graph, containing **ALL** vertices

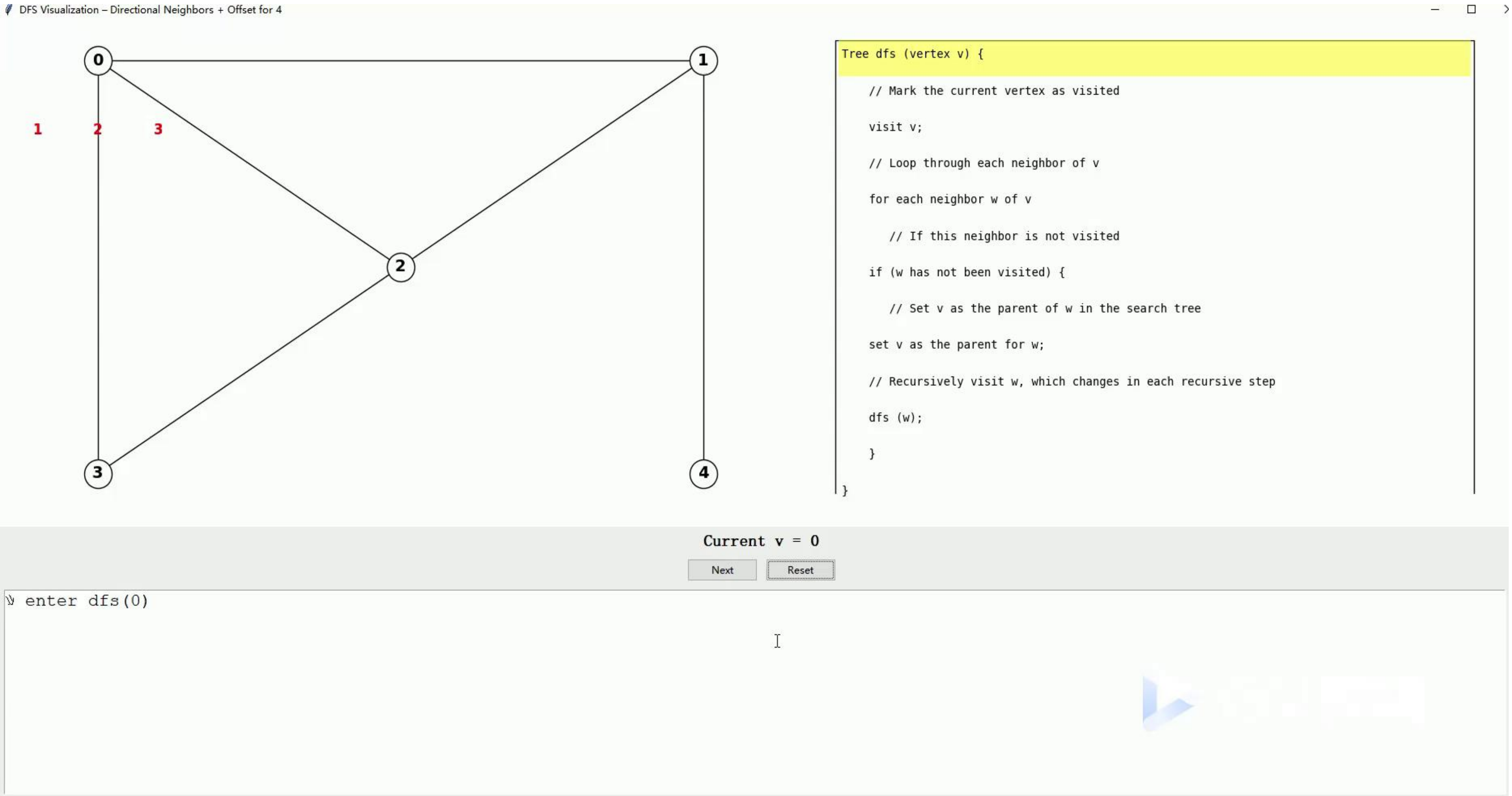


Depth-First Search

- The ***depth-first search*** of a graph starts from a vertex in the graph and visits all vertices in the graph as far as possible before backtracking

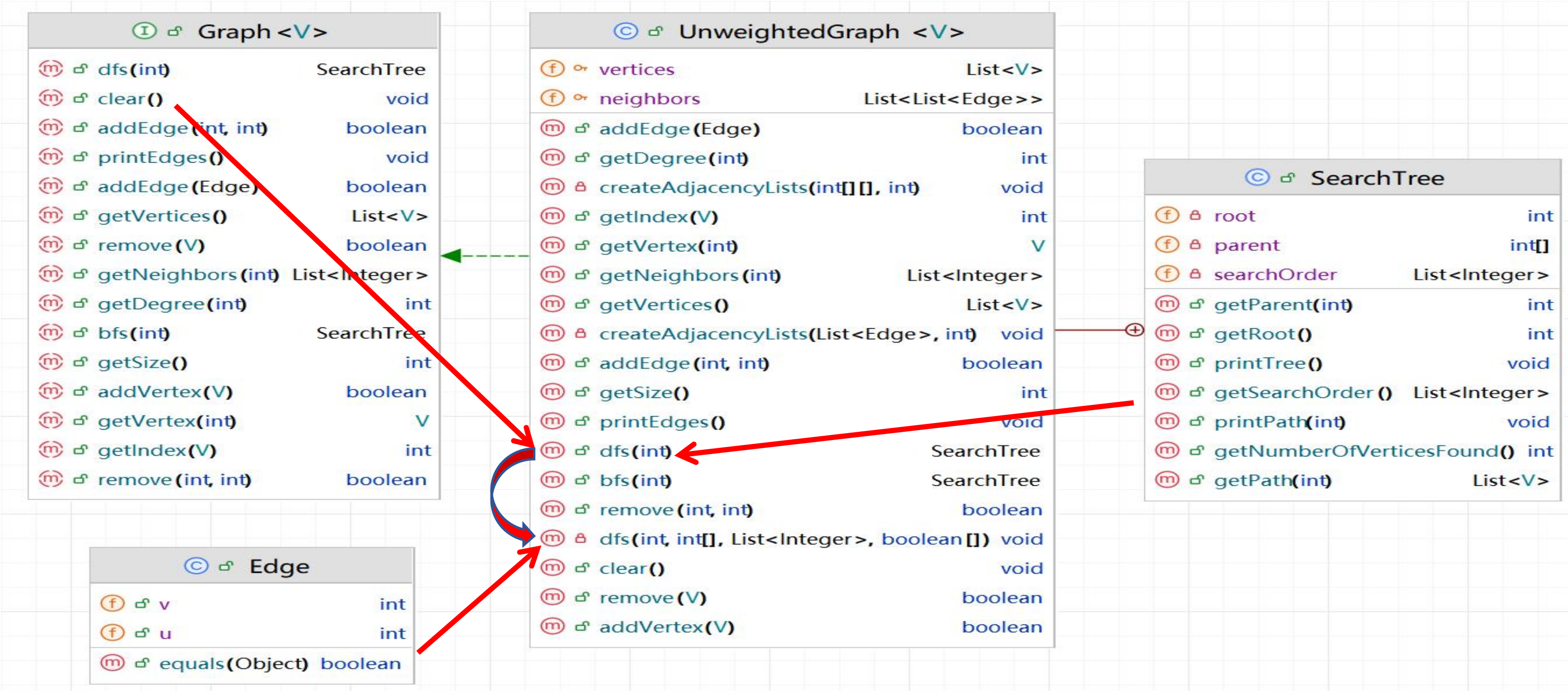
```
Input: G =  (V, E) and a starting vertex v
Output: a DFS tree rooted at v
Logic:
Tree dfs(vertex v)  {
    //Mark the current vertex as visited
    visit v;
    //Loop through each neighbor of v
    for each neighbor w of v
        //If this neighbor is not visited
        if  (w has not been visited)  {
            //Set v as the parent of w in the search tree
            set v as the parent for w;
            // Recursively visit w, which changes in each recusive step
            dfs(w);
        }
}
```

DFS Visualization



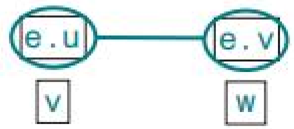
Since each edge and each vertex is visited only once, the time complexity of the dfs method is **O(|E| + |V|)**, where |E| denotes the number of edges and |V| the number of vertices.

How DFS is Implemented



1. The dfs(int) method in the interface is implemented by the **public dfs(int) method** in the UnweightedGraph
2. Inside the public dfs(int) method, a **private recursive dfs()** method is called to perform the actual dfs operation
3. The **Edge class** supports the DFS operation by providing the edge object represented by u (starting vertex) and v (ending vertex).
4. **SearchTree (inner class)** serves as a container for storing and printing the results of a DFS traversal.

```java
public class Edge {
  public int u; // starting point index
  public int v; // endpoint index

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

  public boolean equals(Object o) {
    return u == ((Edge)o).u && v == ((Edge)o).v;
  }
}
```



This class stores the index of the starting point u and the index of the endpoint v, allowing the graph to be represented as a list of such edges.

e.u corresponds to the current vertex v (the one we're expanding), and e.v represents a neighbor of that vertex.

```java
public class SearchTree {
    private int root; // The root of the tree
    private int[] parent; // Store the parent of each vertex
    private List<Integer> searchOrder; // Store the search order

    /** Construct a tree with root, parent, and searchOrder */
    public SearchTree(int root, int[] parent,
        List<Integer> searchOrder) {
      this.root = root;
      this.parent = parent;
      this.searchOrder = searchOrder;
    }
    /**
     * Return the root of the tree
     */
        public int getRoot() {
            return root;
        }
     (and so on...)
```

(In unweightedGraph.java)

The SeachTree class serves as a container for storing and printing the results of a DFS traversal. It define the 1. root (the starting vertex), 2. parent (an array storing the parent of each vertex), and 3. searchOrder (a list showing the order in which vertices were visited)

It also construct a tree based on these 3 variables.

```java
@Override
  public SearchTree dfs(int v) {
    // A list to record the order in which vertices are visited
    List<Integer> searchOrder = new ArrayList<>();
    // An array to store the parent of each vertex in the DFS tree
    int[] parent = new int[vertices.size()];
    for (int i = 0; i < parent.length; i++)
      parent[i] = -1; // Initialize parent[i] to -1
    // A boolean array to mark whether a vertex has been visited
    boolean[] isVisited = new boolean[vertices.size()];
    // Perform recursive DFS traversal (the private void dfs() below)
    dfs(v, parent, searchOrder, isVisited);
    // Return a search tree
    return new SearchTree(v, parent, searchOrder);
  }
```

(In unweightedGraph.java)

A public method named dfs, returning a SearchTree object.
It does:
1. Initializes variables,
2. calls the recursive DFS method,
3. returns a SearchTree.

```java
@Override
  private void dfs(int v, int[] parent, List<Integer> searchOrder,
      boolean[] isVisited) {
    // Store the visited vertex
    searchOrder.add(v);
    isVisited[v] = true; // Vertex v visited

    for (Edge e : neighbors.get(v)) { // Note that e.u is v
      if (!isVisited[e.v]) { // If w is not visited yet
        parent[e.v] = v; // The parent of w is v
        // Recursive search
        dfs(e.v, parent, searchOrder, isVisited);
      }
    }
  }
```

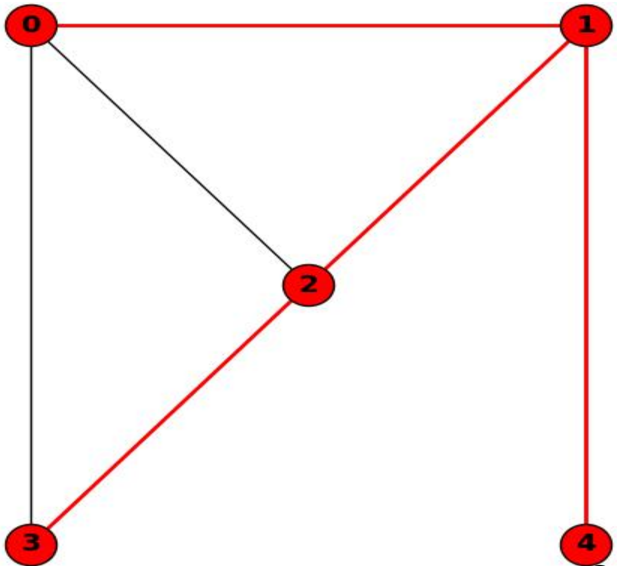
(In unweightedGraph.java)

This method performs the actual recursive depth-first search, marking each visited vertex, recording its parent, and maintaining the visit order

The Main() is in the TestDFS class

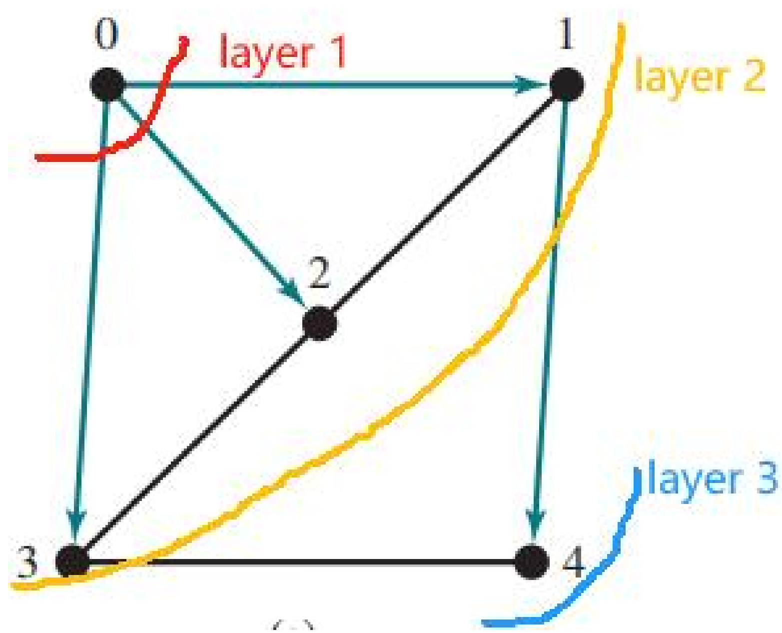
Applications of the DFS

- Detecting whether a graph is **<u>connected</u>**
  - 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.
- Finding all connected components
- Detecting whether there is a path between two vertices AND find it (not the shortest)
  - Case: From 0 to 2
  - Shortest should be: 0-2
  - In DFS Result: 0-1-2



Breadth-First Search (BFS)

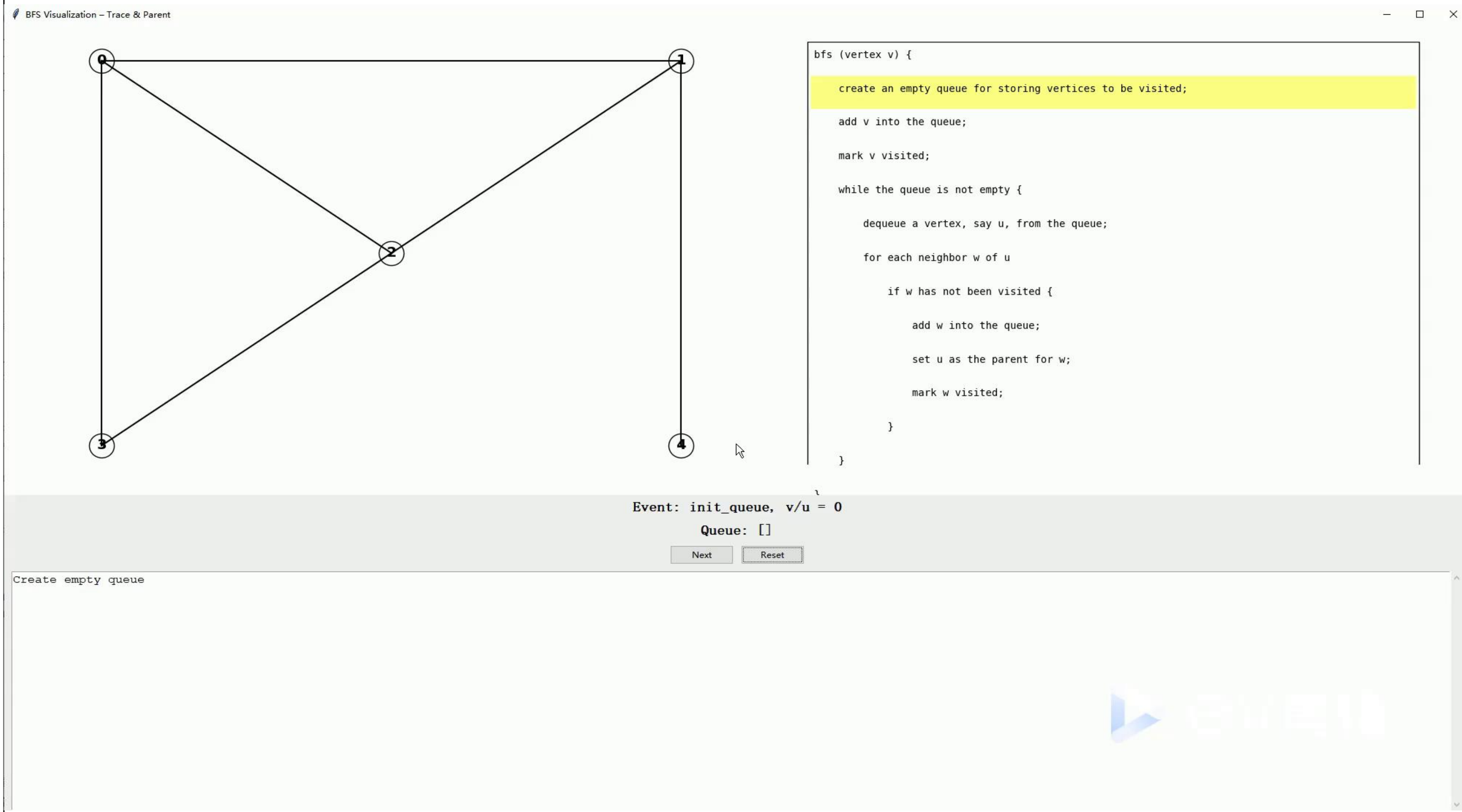
- Breadth-first search (BFS) visits the graph layer by layer.
- It starts from the root, then visits all its neighbors (children), then the neighbors' neighbors (grandchildren), and so on — moving outward layer by layer.



Breadth-First Search Algorithm

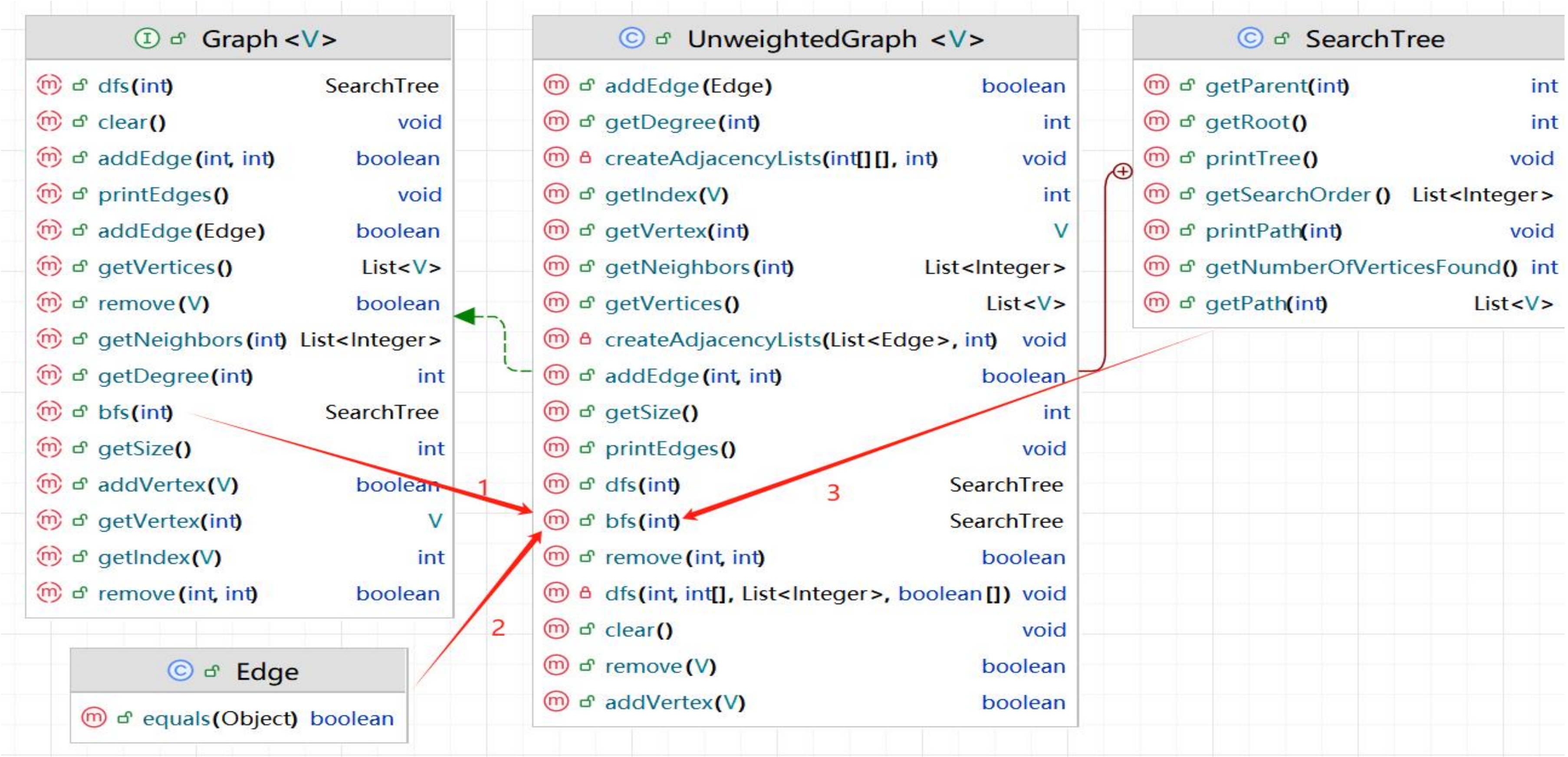
```
Input: G =  (V, E) and a starting vertex v
Output:  a BFS tree rooted at v
bfs(vertex v)  {
   // Starts from a given vertex, adds it to a queue, and marks it visited
   create an empty queue for storing vertices to be visited;
   add v into the queue;
   mark v visited;
   // While the queue is not empty, it dequeues (extracts) a vertext,
   visit all its unvisited neighbors, marks these neighbors visited,
   records their parent, and add them to the queue
   while the queue is not empty {
      dequeue a vertex, say u, from the queue
      for each neighbor w of u
         if w has not been visited {
            add w into the queue;
            set u as the parent for w;
            mark w visited;
         }
   }
}
```

BFS Visualization



Since each edge and each vertex is visited only once, the time complexity of the dfs method is **O(|E| + |V|)**, where |E| denotes the number of edges and |V| the number of vertices.

How BFS is Implemented



1. The bfs method in UnweightedGraph implements the bfs method in the interface
2. Edge class provides the edge objects to BFS for operation
3. SearchTree (inner class) serves as the container and printer of the BFS results

Main method is in the TestBFS.java

Applications of the BFS

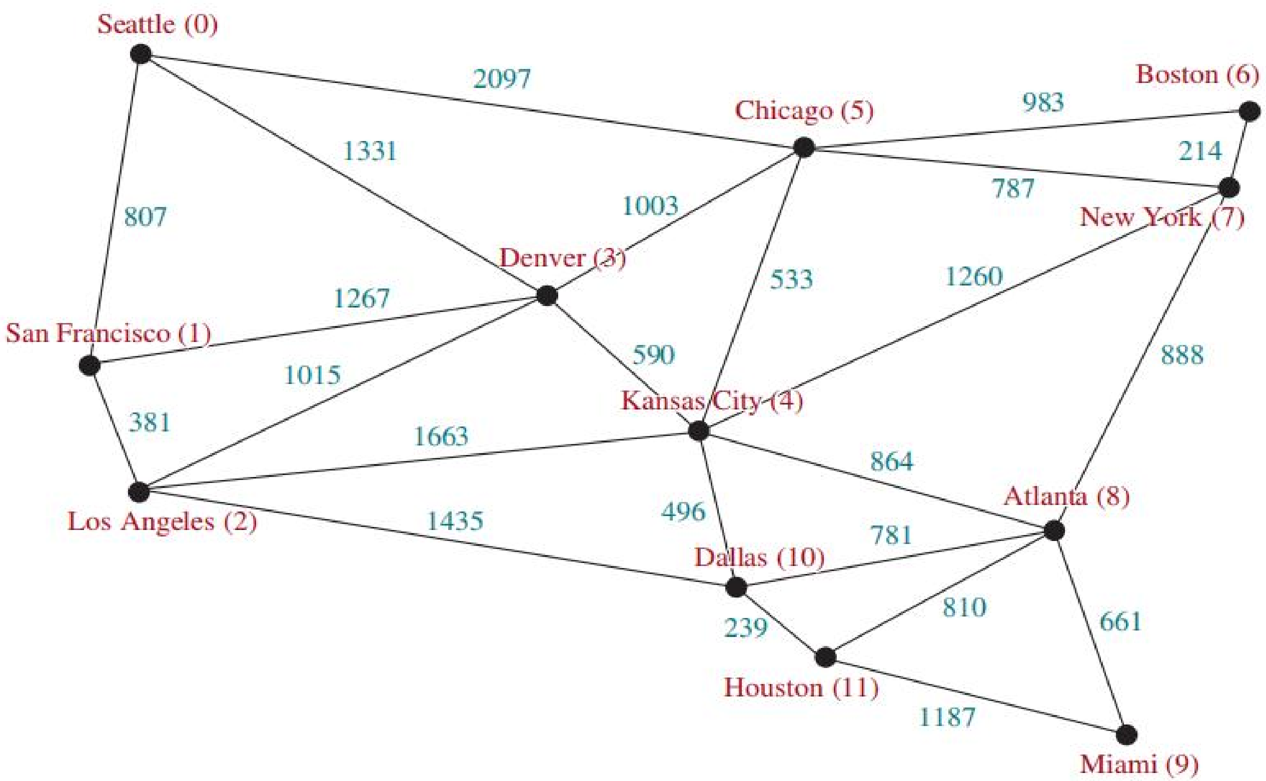
- Detecting whether a graph is connected (i.e., if there is a path between any two vertices in the graph)
  - check is the size of the spanning tree is the same with the number of vertices
- Detecting whether there is a path between two vertices
  - Compute the BFS from the first vertex and check if the second vertex is reached
- Finding all connected components
- Finding a ***shortest path*** between two vertices (in unweighted graph case)
  - Because BFS explores all vertices with 1 edge away, then 2 edges away, etc., it guarantees that the first time we reach a vertex, it is through the shortest possible edge count

Weighted Graphs

CPT204 Advanced Object-Oriented Programming

Weighted Graphs

- A graph is a ***weighted graph*** if each edge is assigned a weight (value).
  - For example: assume that the edges represent the driving distances among the cities



Representing Weighted Edges

1. Using Edge Array

vertex weight

```
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}
};
```

2. Using Adjacency Matrices

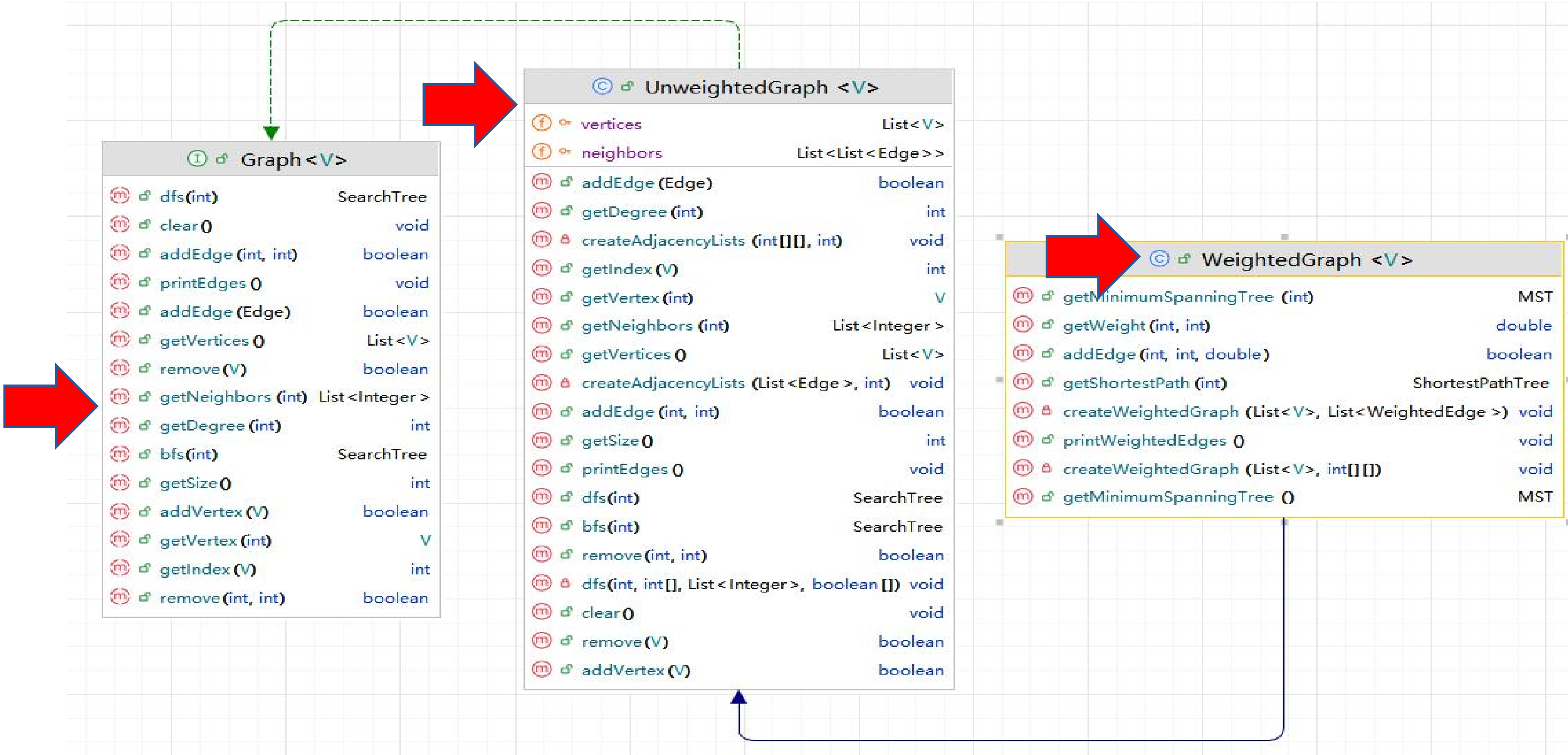
```
Integer[][] adjacencyMatrix = {
  {null, 2, null, 8, null},
  {2, null, 7, 3, null},
  {null, 7, null, 4, 5},
  {8, 3, 4, null, 6},
  {null, null, 5, 6, null}
};
```

	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

3. Using Adjacency List

```
// An array of 5 lists, each intends to store
WeightedEdge objects
java.util.List<WeightedEdge>[] list = new
java.util.List[5];

// The WeightedEdge class
public class WeightedEdge extends Edge implements
Comparable<WeightedEdge> {
 // Define weight
 public double weight;

 // Constructs a weighted edge with the weights
 public WeightedEdge(int u, int v, double weight) {
        super(u, v);
        this.weight = weight;
    }

  // Compare two edges based on weights
  public int compareTo(WeightedEdge edge) {
     // Return 1 if this.weight > other,
     // 0 if equal,
     // -1 if less
     }
}
```

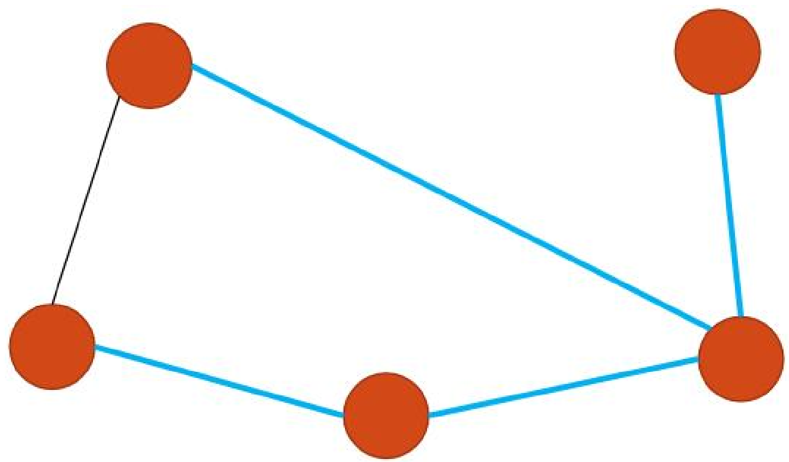
Modeling Weightd Graphs



- Graph<V> is an interface that defines the common behavior of a graph.
- UnweightedGraph<V> implements the Graph<V> interface, focusing on dealing with unweighted graphs.
- WeightedGraph<V> extends UnweightedGraph<V> to reuse foundational methods and data structures, while introducing additional capabilities for handling weighted graphs

Minimum Spanning Trees (MST)

- A ***spanning tree*** of a graph G is a connected subgraph of G and the subgraph **is a tree that contains ALL vertices in G**

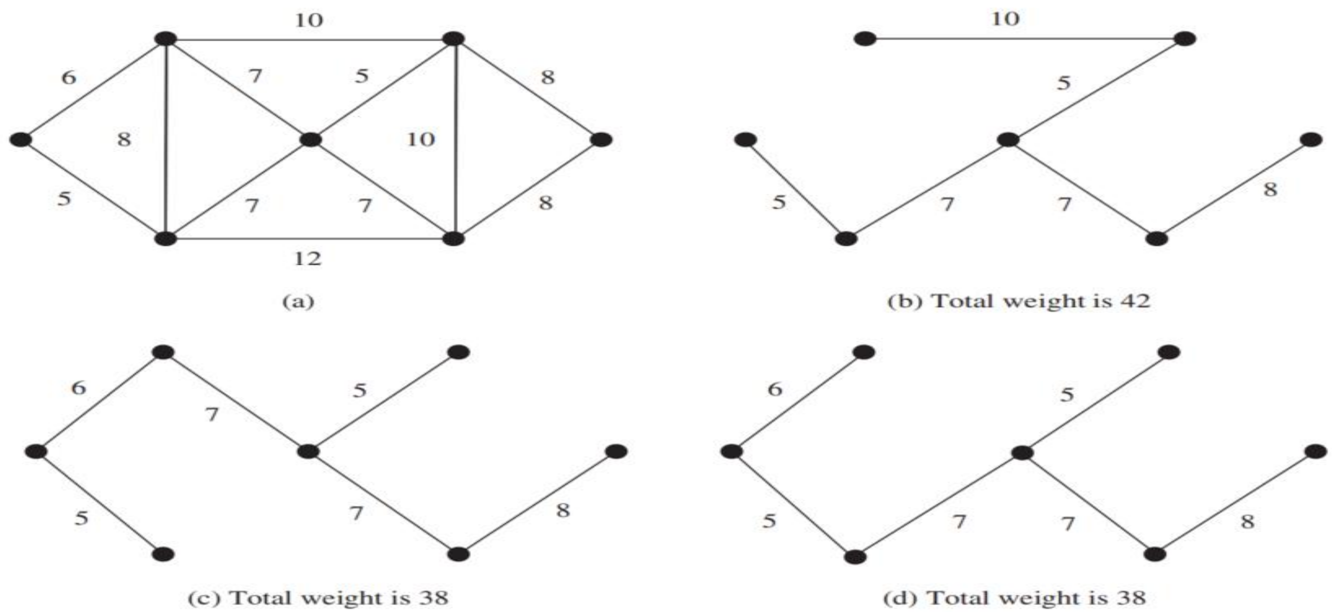


- A ***minimum spanning tree*** is a spanning tree **with the minimum total weight**

Minimum Spanning Trees (MST)

Application example: a company wants to create the Internet lines to connect all the customers together

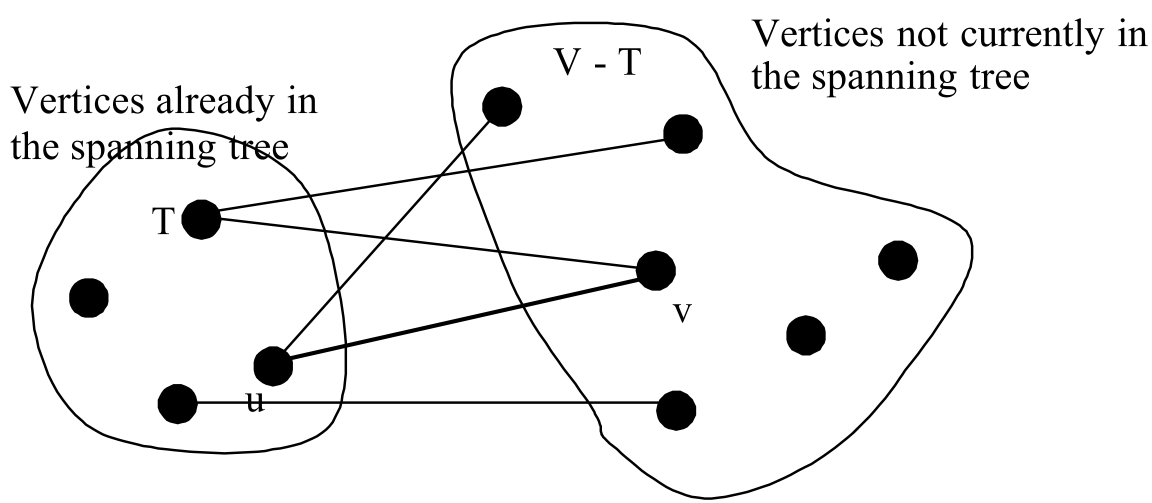
- There are many ways (i.e., streets) to connect all customers together
- Different lines have different cost (e.g., length)
- The cheapest way is to find a spanning tree with the minimum total cost



A graph may have many minimum spanning trees

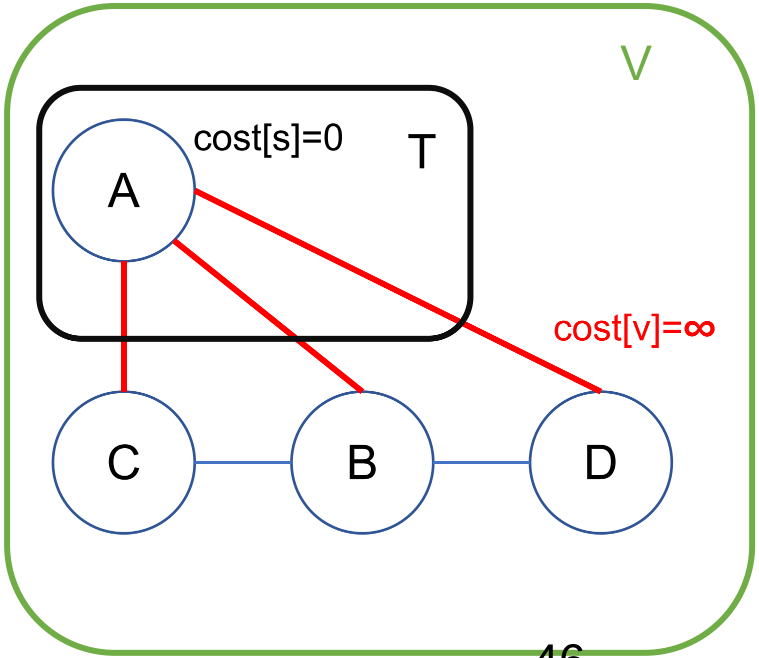
Prim's Minimum Spanning Tree Algorithm

```
Input: G = (V, E)
Output: a MST
MST getMinimumSpanningTree(s) {
  Let V denote the set of vertices in the graph;
  Let T be a set for the vertices in the spanning tree;
  Initially, add the starting vertex to T;

  while (size of T < n) {
    find u in T and v in V – T with the smallest weight
      on the edge (u, v), as shown in the figure;
    add v to T;
  }
}
```



Refined Version of Prim’s MST Algorithm

```
MST getMinimumSpanngingTree(s) {
  // Initialize T = [vertices in spanning tree]; V=[all vertices in the graph]
  // cost[s]=0; cost[v](the cost of adding a vertex v to the spanning tree T)= ∞
  Let T be a set that contains the vertices in the spanning tree; Initially T is
  empty;
   Set cost[s] = 0; and cost[v] = infinity for all other vertices in V;

   // While vertices outside the T set, find one vertex with the min cost, add
   it to the T set, and update the cost of its nerighbors (for next round)
   while  (size of T < n)  {
      Find u not in T with the smallest cost[u];
      Add u  to T;
      for(each  (u, v)  in E)
         if  ( v not in T && cost[v] > w(u, v))  {
            cost[v] = w(u, v);
            parent[v] = u;
         }
   }
 }
```



Refined MST Visualization

Prim Visualization

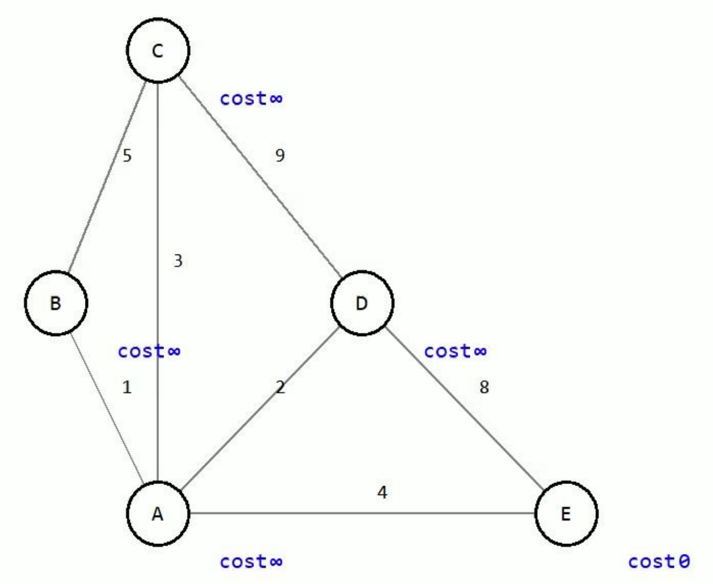
Next Step

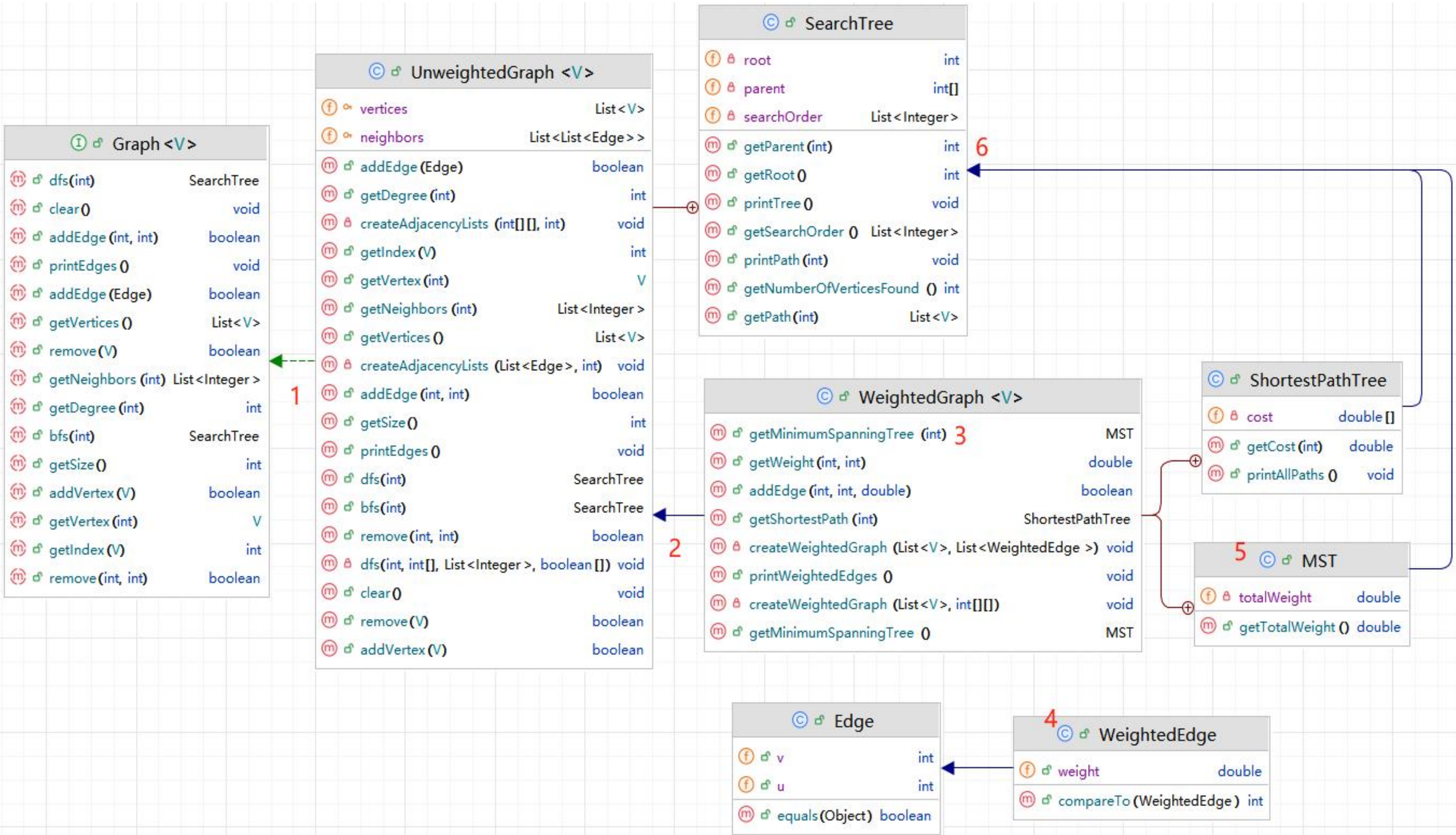
Reset

u = -, v = -

```
MST getMinimumSpanningTree(s) {
Let T be a set that contains the vertices in the spanning tree; Initially T is empty;
Set cost[s] = 0 and cost[v] = infinity for all other vertices in V
while (size of T < n) {
    Find u not in T with the smallest cost[u]
    Add u to T
    for (each (u, v) in E)
        if (v not in T and cost[v] > w(u, v)) {
            cost[v] = w(u, v)
            parent[v] = u
        }
    }
}
```

cost	∞	∞	∞	∞	0
	A	B	C	D	E
parent	-	-	-	-	-
	A	B	C	D	E





1. UnweightedGraph implements Interface
2. WeightedGraph extends the UnweightedGraph (for methods/data structures reuse)
3. WeightedGraph defines the getMinimumSpanningTree method
4. WeightedEdge extends Edge (for reuse) to provide weighted edge objects to MST operation.
5. MST method serves as the container and printer for the MST results
6. MST extends SearchTree (for reuse)

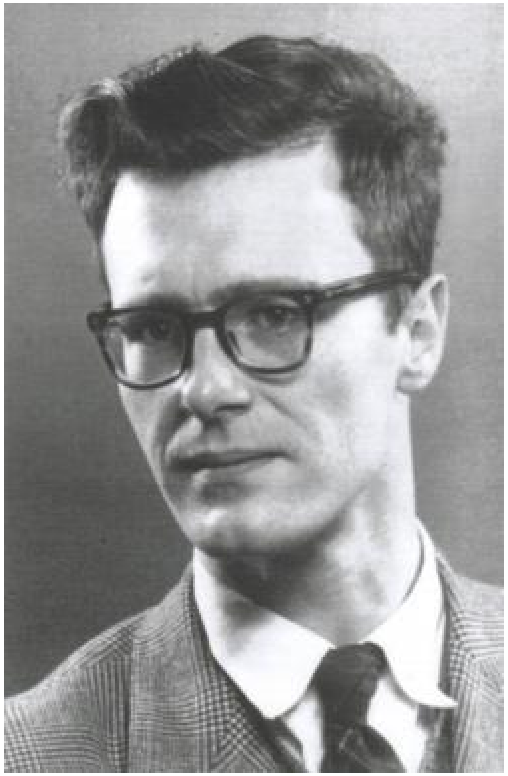
```java
/** Get a minimum spanning tree rooted at vertex 0 */
public MST getMinimumSpanningTree() {
  return getMinimumSpanningTree(0);
}
/** Get a minimum spanning tree rooted at a specified vertex */
public MST getMinimumSpanningTree(int startingVertex) {
  // cost[v] stores the cost by adding v to the tree
  double[] cost = new double[getSize()];
  for (int i = 0; i < cost.length; i++)
    cost[i] = Double.POSITIVE_INFINITY; // Initial cost
  cost[startingVertex] = 0; // Cost of source is 0
  int[] parent = new int[getSize()]; // Parent of a vertex
  parent[startingVertex] = -1; // startingVertex is the root
  double totalWeight = 0; // Total weight of the tree thus far
  List<Integer> T = new ArrayList<>();
   // Expand T
   while (T.size() < getSize()) {
     // Find smallest cost v in V - T
     int u = -1; // Vertex to be determined
     double currentMinCost = Double.POSITIVE_INFINITY;
     for (int i = 0; i < getSize(); i++)
       if (!T.contains(i) && cost[i] < currentMinCost) {
         currentMinCost = cost[i];
         u = i;
       }
     T.add(u); // Add a new vertex to T
     totalWeight += cost[u]; // Add cost[u] to the tree
     // Adjust cost[v] for v that is adjacent to u and v in V - T
     for (Edge e : neighbors.get(u))
       if (!T.contains(e.v) && cost[e.v] > ((WeightedEdge)e).weight) {
         cost[e.v] = ((WeightedEdge)e).weight;
         parent[e.v] = u;
       }
   } // End of while
   return new MST(startingVertex, parent, T, totalWeight);
 }
```

```java
public class TestMinimumSpanningTree {
  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, 807}, {0, 3, 1331}, {0, 5, 2097},
      {1, 0, 807}, {1, 2, 381}, {1, 3, 1267},
      {2, 1, 381}, {2, 3, 1015}, {2, 4, 1663}, {2, 10, 1435},
      {3, 0, 1331}, {3, 1, 1267}, {3, 2, 1015}, {3, 4, 599}, {3, 5, 1003},
      {4, 2, 1663}, {4, 3, 599},{4, 5, 533},{4, 7, 1260},{4, 8, 864}, {4, 10, 496},
      {5, 0, 2097}, {5, 3, 1003}, {5, 4, 533}, {5, 6, 983}, {5, 7, 787},
      {6, 5, 983}, {6, 7, 214},
      {7, 4, 1260}, {7, 5, 787}, {7, 6, 214}, {7, 8, 888},
      {8, 4, 864}, {8, 7, 888}, {8, 9, 661}, {8, 10, 781}, {8, 11, 810},
      {9, 8, 661}, {9, 11, 1187},
      {10, 2, 1435}, {10, 4, 496}, {10, 8, 781}, {10, 11, 239},
      {11, 8, 810}, {11, 9, 1187}, {11, 10, 239}
    };

    WeightedGraph<String> graph1 = new WeightedGraph<>(vertices, edges);

    WeightedGraph<String>.MST tree1 = graph1.getMinimumSpanningTree();

    System.out.println("Total weight is " + tree1.getTotalWeight());
    tree1.printTree();
```

```java
    edges = new int[][]{
      {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}
    };

    WeightedGraph<Integer> graph2 = new WeightedGraph<>(edges, 5);

    WeightedGraph<Integer>.MST tree2 = graph2 .getMinimumSpanningTree(1);

    System.out.println("\nTotal weight is " + tree2 .getTotalWeight());
    tree2 .printTree();

  }
}
```

Shortest Path

- Find a shortest path between two vertices in the graph
  - The *shortest path* between two vertices is a path with the minimum total weight
- A well-known algorithm for finding a shortest path between two vertices was discovered by Edsger Dijkstra,a Dutch computer scientist
- In order to find a shortest path from vertex **`s`** to vertex **`v`**, Dijkstra's algorithm finds the shortest path from **`s`** to all vertices



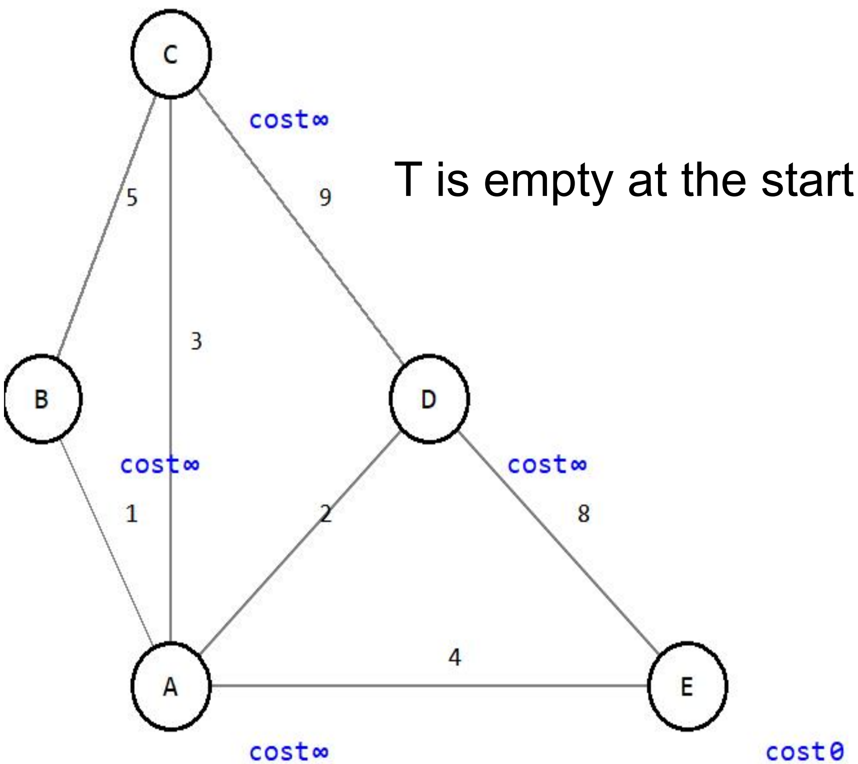
Edsger W. Dijkstra

Single Source Shortest Path Algorithm

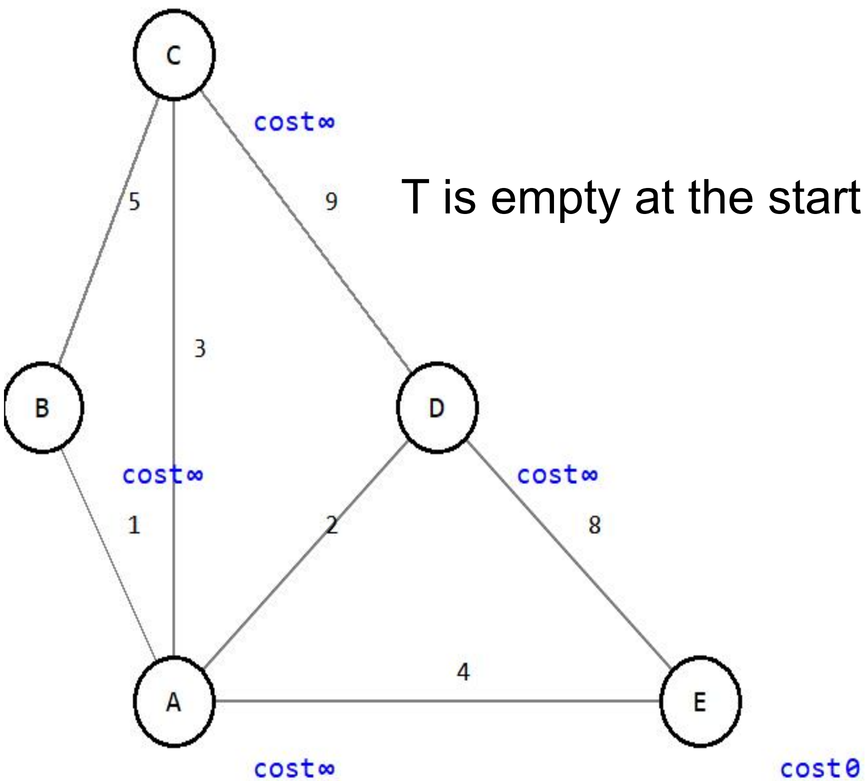
```
Input: a graph G = (V, E) with non-negative weights
Output: a shortest path tree with the source vertex s as the root

ShortestPathTree getShortestPath(s) {
  // Initialize T = [vertices whose path are known]; V=[all vertices in the graph]
  // cost[s]=0; cost[v](from starting point to a vertex)= ∞
  Let T be a set that contains the vertices whose paths to s are known; Initially
  T is empty;
  Set cost[s] = 0; and cost[v] = infinity for all other vertices in V;

  // While vertices outside the T set, find one vertex with the min cost, add it
  to the T set, and update the cost of its nerighbors (for next round)
  while (size of T < n) {
     Find u not in T with the smallest cost[u];
     Add u to T;
     for (each (u, v) in E)
       if (v not in T and cost[v] > cost[u] + w(u, v)) {
         cost[v] = w(u, v) + cost[u];
         parent[v] = u;
       }
  }
}
```

Prim v.s. Dijkstra

Both of them initialize an **empty T** set to keep track of the processed vertices and **cost[s]=0**, **cost[v]=∞**



Prim's



Dijkstra's

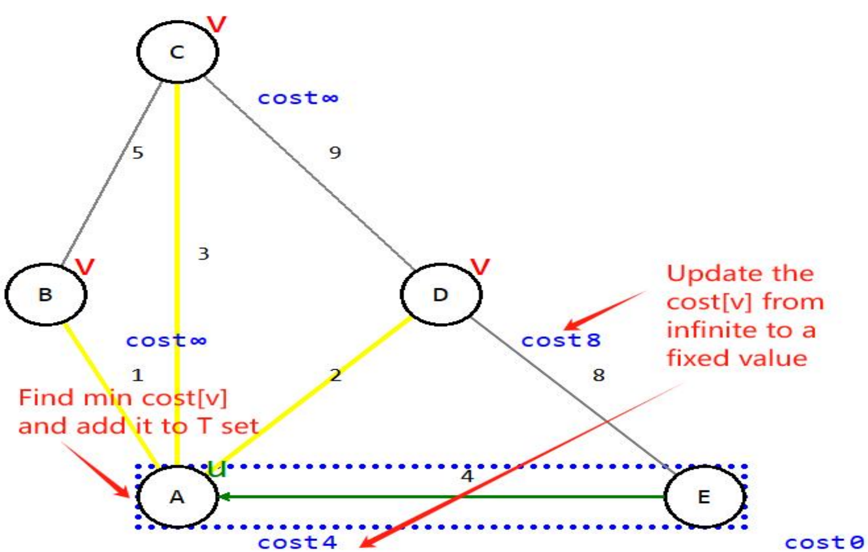
Prim v.s. Dijkstra

Both of them iteratively use the **while-for loop** structure to

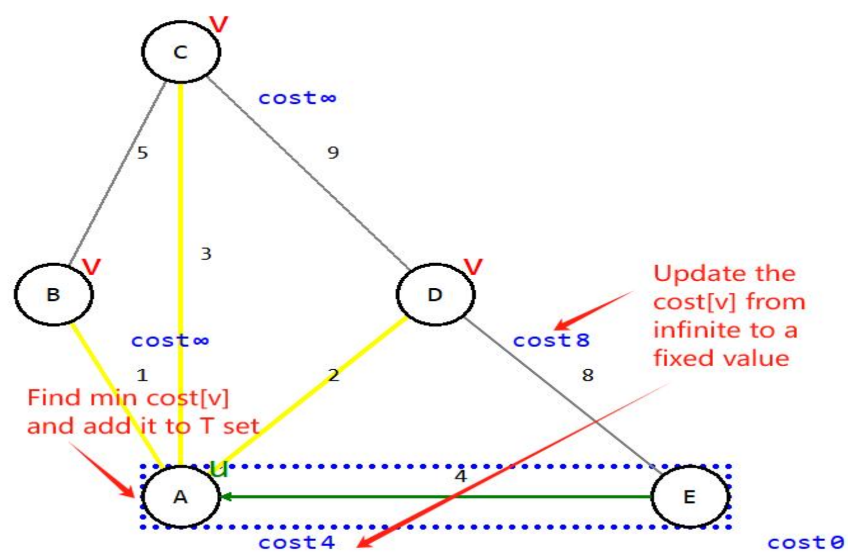
- Select the **minimum cost[v] not in T**, and add it to T set
- **Update** cost[v] and parent[v]

```
While (vertices outside T){
// find min cost[v] and add to T
   for (each (u,v) in E)
      if (conditions){
        //update the cost[v], parent[v]
      }
}
```

.



Prim's



Dijkstra's

Prim v.s. Dijkstra

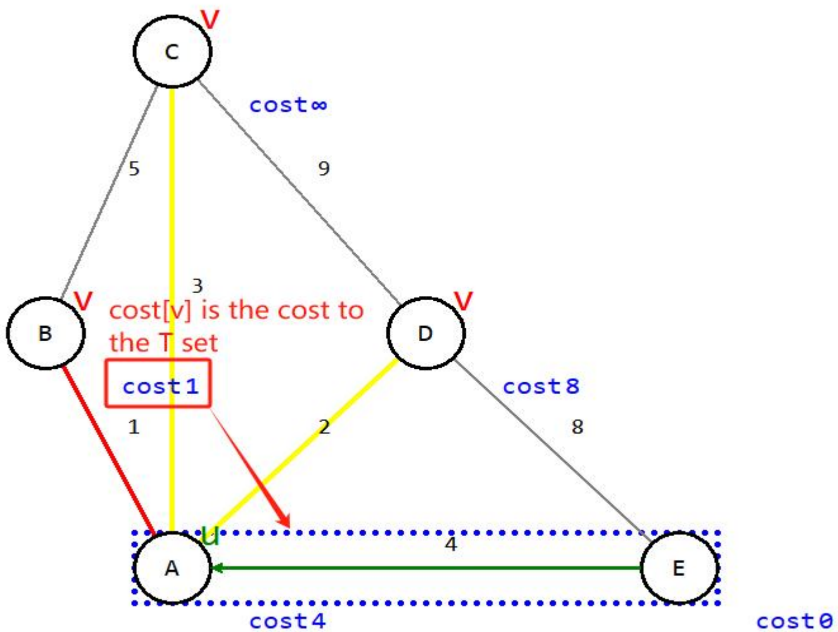
BUT(!)

The cost[v] in **Prim's algorithm** is the cost to the T set

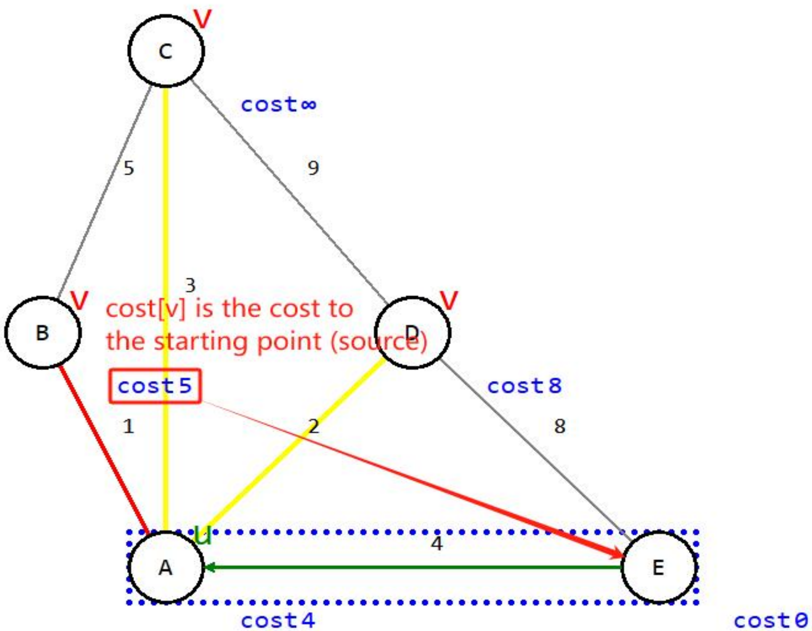
- cost[B] = 1, because **`cost[v] = w(u, v)`**.
- Because w(A,B)=1, cost[B]=1.

The cost[v] in **Dijkstra's algorithm** is the cost to the source (the very first starting vertex)

- cost[B] = 5, because **`cost[v] = w(u, v) + cost[u]`**.
- Because w(A,B)=1, cost[u]=4, cost[B]=1+4=5.



Prim's



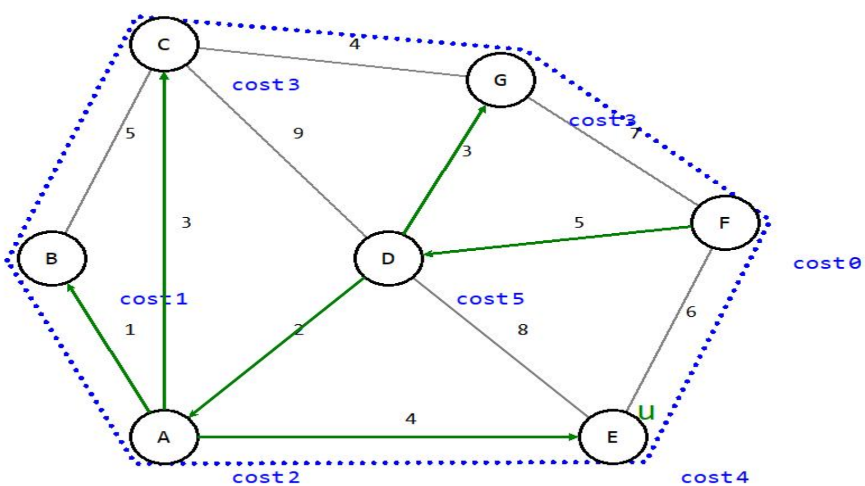
Dijkstra's

Prim v.s. Dijkstra

Overall

The goal of Prim's algorithm is to **connect all vertices with the minimum total edge weight**

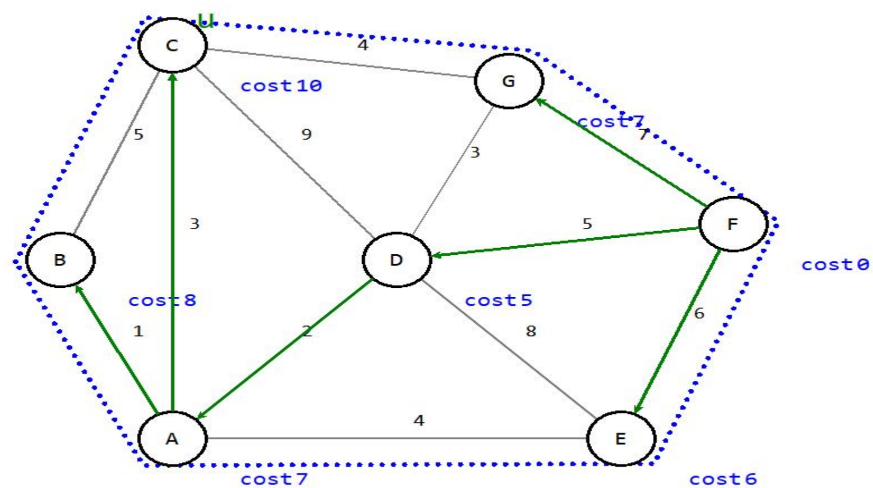
The goal of Dijkstra's algorithm is to **find the shortest path from a source s to every other vertex**



Prim's (starting F)

Total weight: 18

From F to E: F-D-A-E = 11



Dijkstra's (starting F)

Total weight: 24

From F to E: F-E = 6

```java
/** Find single source shortest paths */
public ShortestPathTree getShortestPath(int sourceVertex) {
   // cost[v] stores the cost of the path from v to the source
   double[] cost = new double[getSize()];
   for (int i = 0; i < cost.length; i++)
      cost[i] = Double.POSITIVE_INFINITY;
   // parent[v] stores the previous vertex of v in the path
   int[] parent = new int[getSize()];
   parent[sourceVertex] = -1; // The parent of source is set to -1
   // T stores the vertices whose path found so far
   List<Integer> T = new ArrayList<>();
   // Expand T
   while (T.size() < getSize()) {
      // Find smallest cost v in V - T
      int u = -1; // Vertex to be determined
      double currentMinCost = Double.POSITIVE_INFINITY;
      for (int i = 0; i < getSize(); i++)
         if (!T.contains(i) && cost[i] < currentMinCost) {
            currentMinCost = cost[i];
            u = i;
         }
      T.add(u); // Add a new vertex to T
      // Adjust cost[v] for v that is adjacent to u and v in V - T
      for (Edge e : neighbors.get(u)) {
         if (!T.contains(e.v)
               && cost[e.v] > cost[u] + ((WeightedEdge)e).weight) {
            cost[e.v] = cost[u] + ((WeightedEdge)e).weight;
            parent[e.v] = u;
         }
      }
   } // End of while
   // Create a ShortestPathTree
   return new ShortestPathTree(sourceVertex, parent, T, cost);
}
```

```
// Executes n times, where n = number of vertices
while (T.size() < getSize()) {
   // n times for loop to find the min cost[v]
   // !T.contains(i) is O(n) per call, (since T is an ArrayList)
   // so n*n = O(n^2)
   for (int i = 0; i < getSize(); i++) {
      if (!T.contains(i) && cost[i] < min) {
         ...
      }
   }

   // n times to update neighbor values
   // !T.contains(i) is O(n) per call, (since T is an ArrayList)
   // so n*n = O(n^2)
   for (each neighbor v) {
      if (!T.contains(v) && ...) {
         ...
      }
   }
}
```

Time complexity: $O(n) \times O(n^2) + O(n) \times O(n^2) = O(n^3)$

Similar in the Prim's MST algorithm, but is this efficient?

```java
public class TestShortestPath {
  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, 807}, {0, 3, 1331}, {0, 5, 2097},
      {1, 0, 807}, {1, 2, 381}, {1, 3, 1267},
      {2, 1, 381}, {2, 3, 1015}, {2, 4, 1663}, {2, 10, 1435},
      {3, 0, 1331}, {3, 1, 1267}, {3, 2, 1015}, {3, 4, 599}, {3, 5, 1003},
      {4, 2, 1663}, {4, 3, 599},{4, 5, 533},{4, 7, 1260},{4, 8, 864}, {4, 10, 496},
      {5, 0, 2097}, {5, 3, 1003}, {5, 4, 533}, {5, 6, 983}, {5, 7, 787},
      {6, 5, 983}, {6, 7, 214},
      {7, 4, 1260}, {7, 5, 787}, {7, 6, 214}, {7, 8, 888},
      {8, 4, 864}, {8, 7, 888}, {8, 9, 661}, {8, 10, 781}, {8, 11, 810},
      {9, 8, 661}, {9, 11, 1187},
      {10, 2, 1435}, {10, 4, 496}, {10, 8, 781}, {10, 11, 239},
      {11, 8, 810}, {11, 9, 1187}, {11, 10, 239}
    };

    WeightedGraph<String> graph1 = new WeightedGraph<>(vertices, edges);

    WeightedGraph<String>.ShortestPathTree tree1 =
      graph1.getShortestPath(graph1.getIndex("Chicago"));

    tree1 .printAllPaths();
```

```java
    // Display shortest paths from Houston to Chicago
    System.out.print("Shortest path from Houston to Chicago: ");
    java.util.List<String> path = tree1.getPath(graph1.getIndex("Houston"));
    for (String s: path) {
      System.out.print(s + " ");
    }

    edges = new int[][] {
      {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}
    };
    WeightedGraph<Integer> graph2 = new WeightedGraph<>(edges, 5);

    WeightedGraph<Integer>.ShortestPathTree tree2 = graph2.getShortestPath(3);

    System.out.println("\n");
    tree2.printAllPaths();
  }

}
```

```
All shortest paths from Chicago are:
A path from Chicago to Seattle: Chicago Seattle (cost: 2097.0)
A path from Chicago to San Francisco:
 Chicago Denver San Francisco (cost: 2270.0)
A path from Chicago to Los Angeles:
 Chicago Denver Los Angeles (cost: 2018.0)
A path from Chicago to Denver: Chicago Denver (cost: 1003.0)
A path from Chicago to Kansas City: Chicago Kansas City (cost: 533.0)
A path from Chicago to Chicago: Chicago (cost: 0.0)
A path from Chicago to Boston: Chicago Boston (cost: 983.0)
A path from Chicago to New York: Chicago New York (cost: 787.0)
A path from Chicago to Atlanta:
 Chicago Kansas City Atlanta (cost: 1397.0)
A path from Chicago to Miami:
 Chicago Kansas City Atlanta Miami (cost: 2058.0)
A path from Chicago to Dallas: Chicago Kansas City Dallas (cost: 1029.0)
A path from Chicago to Houston:
 Chicago Kansas City Dallas Houston (cost: 1268.0)
Shortest path from Houston to Chicago:
 Houston Dallas Kansas City Chicago

All shortest paths from 3 are:
A path from 3 to 0: 3 1 0 (cost: 5.0)
A path from 3 to 1: 3 1 (cost: 3.0)
A path from 3 to 2: 3 2 (cost: 4.0)
A path from 3 to 3: 3 (cost: 0.0)
A path from 3 to 4: 3 4 (cost: 6.0)
```