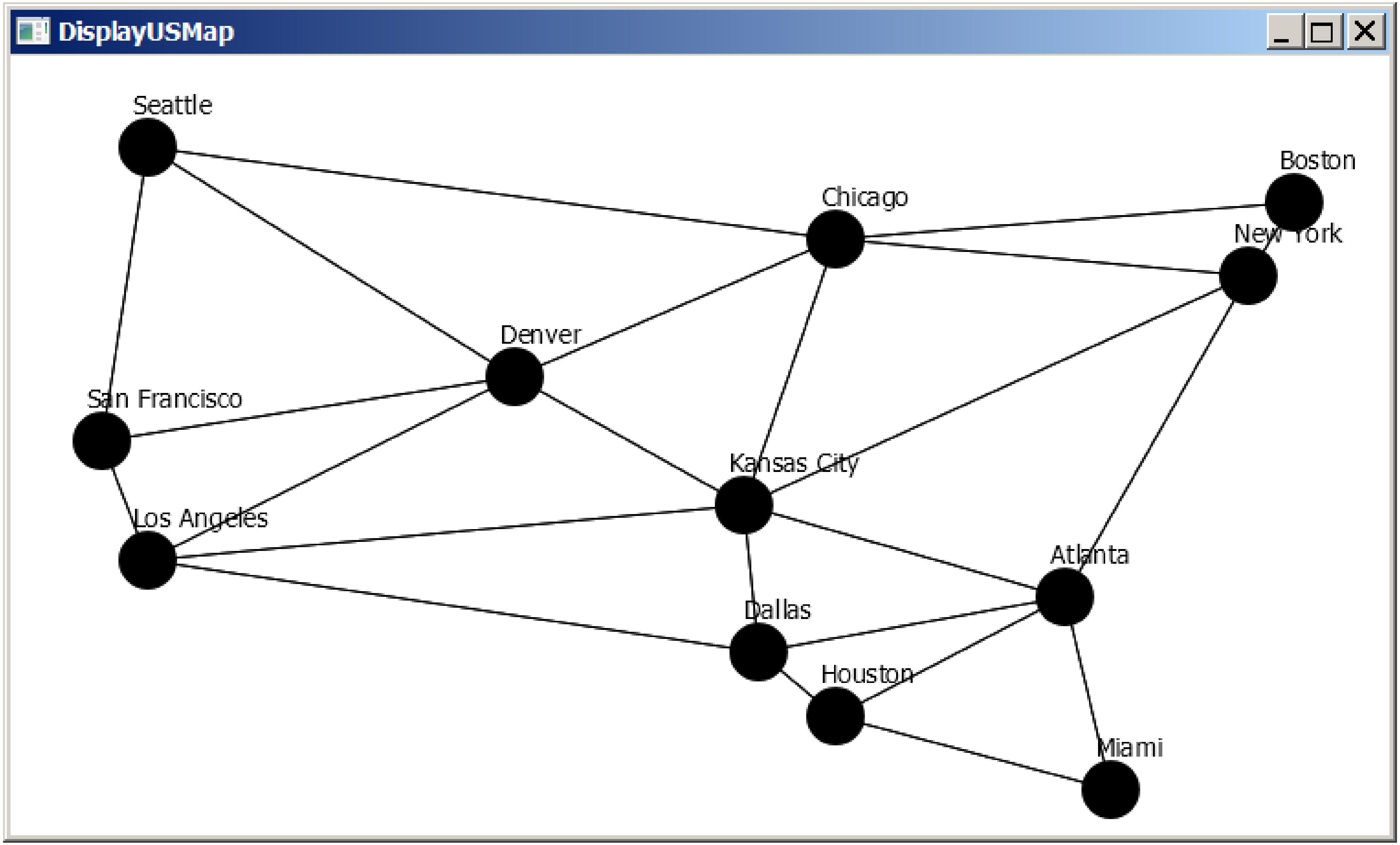
Graphs and Applications

CSE219, Computer Science III

Stony Brook University

http://www.cs.stonybrook.edu/~cse219


DisplayUSMap
Seattle
Boston
Chicago
New York
Denver
San Francisco
Kansas City
Los Angeles
Atlanta
Dallas
Houston
Miami

Representing Graphs

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

Representing Vertices

- String[] vertices = {“Seattle“, “San Francisco“, “Los Angles”, “Denver”, “Kansas City”, “Chicago”, … };

OR

- public class City {
- }
- …
- City[] vertices = {city0, city1, … };
- OR
- List<String> vertices;

Representing Edges: Edge Array

- int[][] edges = {{0, 1}, {0, 3} {0, 5}, {1, 0}, {1, 2}, … };

Representing Edges: Edge Objects

```
public class Edge {
  int u, v;
  public Edge(int u, int v) {
    this.u = u;
    this.v = v;
  }
…
List<Edge> list = new ArrayList<>();
list.add(new Edge(0, 1)); list.add(new Edge(0, 3)); …
```

Representing Edges: Adjacency Vertex List

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

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			

`List<List<Integer>> neighbors = new ArrayList<>();`

Representing Edges: Adjacency Edge List

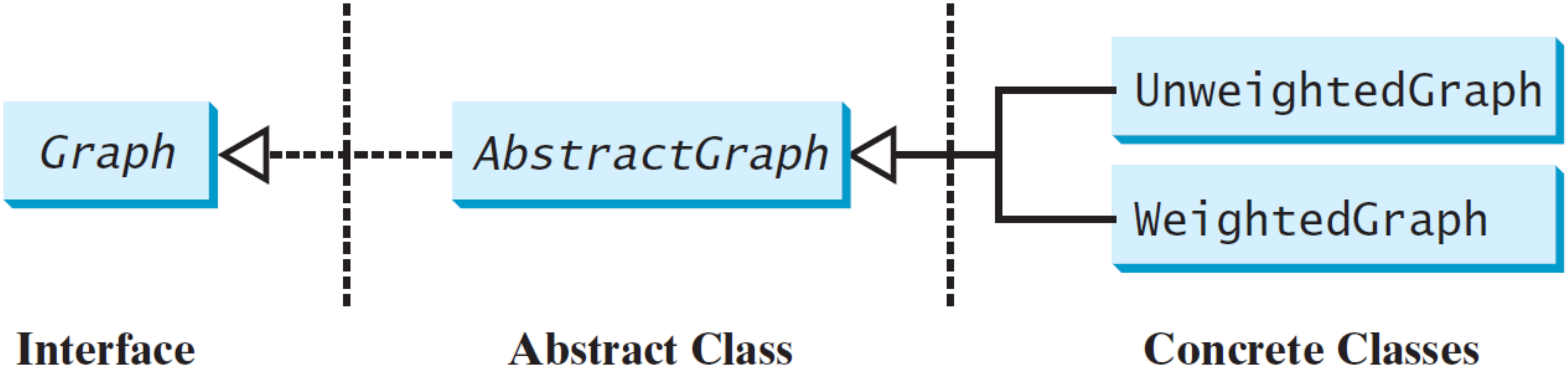
`List<Edge>[] neighbors = new List[12];`

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)			

Representing Adjacency Edge List Using ArrayList

```
List<ArrayList<Edge>> neighbors = new ArrayList<>();
neighbors.add(new ArrayList<Edge>());
neighbors.get(0).add(new Edge(0, 1));
neighbors.get(0).add(new Edge(0, 3));
neighbors.get(0).add(new Edge(0, 5));
neighbors.add(new ArrayList<Edge>());
neighbors.get(1).add(new Edge(1, 0));
neighbors.get(1).add(new Edge(1, 2));
neighbors.get(1).add(new Edge(1, 3));
...
...
neighbors.get(11).add(new Edge(11, 8));
neighbors.get(11).add(new Edge(11, 9));
neighbors.get(11).add(new Edge(11, 10));
```

Modeling Graphs



Graph

AbstractGraph

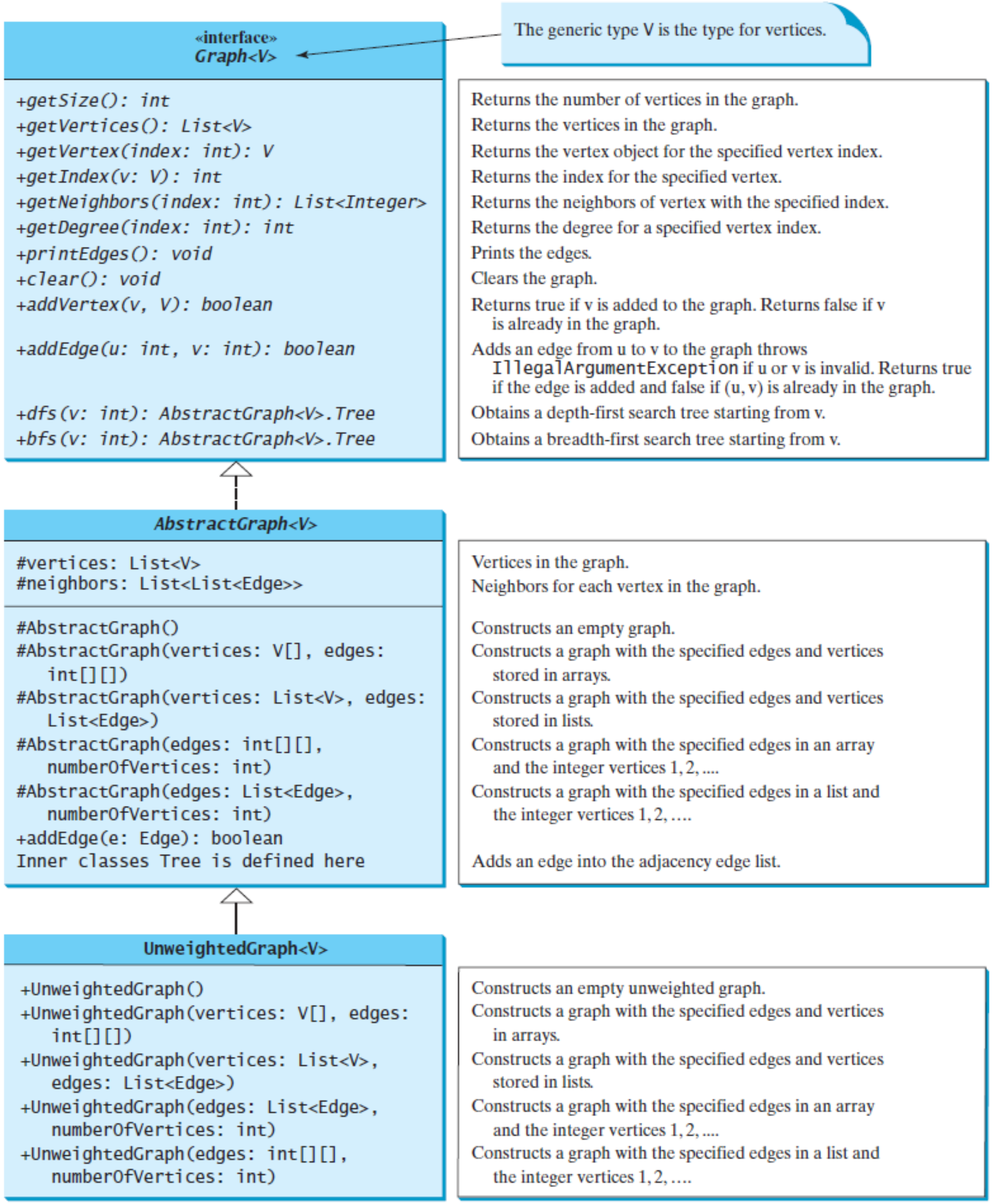
UnweightedGraph

WeightedGraph

Interface

Abstract Class

Concrete Classes



«interface»
Graph<V>

The generic type V is the type for vertices.

Graph<V>	
+getSize(): int	Returns the number of vertices in the graph.
+getVertices(): List<V>	Returns the vertices in the graph.
+getVertex(index: int): V	Returns the vertex object for the specified vertex index.
+getIndex(v: V): int	Returns the index for the specified vertex.
+getNeighbors(index: int): List<Integer>	Returns the neighbors of vertex with the specified index.
+getDegree(index: int): int	Returns the degree for a specified vertex index.
+printEdges(): void	Prints the edges.
+clear(): void	Clears the graph.
+addVertex(v, V): boolean	Returns true if v is added to the graph. Returns false if v is already in the graph.
+addEdge(u: int, v: int): boolean	Adds an edge from u to v to the graph throws `IllegalArgumentException` if u or v is invalid. Returns true if the edge is added and false if (u, v) is already in the graph.
+dfs(v: int): AbstractGraph<V>.Tree	Obtains a depth-first search tree starting from v.
+bfs(v: int): AbstractGraph<V>.Tree	Obtains a breadth-first search tree starting from v.

AbstractGraph<V>	
#vertices: List<V>	Vertices in the graph.
#neighbors: List<List<Edge>>	Neighbors for each vertex in the graph.
#AbstractGraph()	Constructs an empty graph.
#AbstractGraph(vertices: V[], edges: int[][])	Constructs a graph with the specified edges and vertices stored in arrays.
#AbstractGraph(vertices: List<V>, edges: List<Edge>)	Constructs a graph with the specified edges and vertices stored in lists.
#AbstractGraph(edges: int[][], numberOfVertices: int)	Constructs a graph with the specified edges in an array and the integer vertices 1, 2, ...
#AbstractGraph(edges: List<Edge>, numberOfVertices: int)	Constructs a graph with the specified edges in a list and the integer vertices 1, 2, ….
+addEdge(e: Edge): boolean	Adds an edge into the adjacency edge list.
Inner classes Tree is defined here	

UnweightedGraph<V>	
+UnweightedGraph()	Constructs an empty unweighted graph.
+UnweightedGraph(vertices: V[], edges: int[][])	Constructs a graph with the specified edges and vertices in arrays.
+UnweightedGraph(vertices: List<V>, edges: List<Edge>)	Constructs a graph with the specified edges and vertices stored in lists.
+UnweightedGraph(edges: List<Edge>, numberOfVertices: int)	Constructs a graph with the specified edges in an array and the integer vertices 1, 2, ...
+UnweightedGraph(edges: int[][], numberOfVertices: int)	Constructs a graph with the specified edges in a list and the integer vertices 1, 2, ….

```
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 graph */
  public void clear();
  /** Add a vertex to the graph */
  public boolean addVertex(V vertex);
  /** Add an edge to the graph */
  public boolean addEdge(int u, int v);
  /** Obtain a depth-first search tree */
  public AbstractGraph<V>.Tree dfs(int v);
  /** Obtain a breadth-first search tree */
  public AbstractGraph<V>.Tree bfs(int v);
}
```

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

    createAdjacencyLists(edges, vertices.size());
  }
  /** Construct a graph for integer vertices 0, 1, 2 and edge list */
  protected AbstractGraph(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 */
protected AbstractGraph(int[][] edges, int numberOfVertices) {
  for (int i = 0; i < numberOfVertices; i++)
    addVertex((V)(new Integer(i))); // vertices is {0, 1, ...}

  createAdjacencyLists(edges, numberOfVertices);
}
/** 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);
}
@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();
}
@Override /** Clear the graph */
public void clear() {
  vertices.clear();
  neighbors.clear();
}
```

```
@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 /** 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;
  }
}
/** Add an edge to the graph */
protected 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));
}
/** Edge inner class inside the AbstractGraph class */
public static class Edge {
  public int u; // Starting vertex of the edge
  public int v; // Ending vertex of the edge

  /** Construct an edge for (u, v) */
  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;
  }
```

```
@Override /** Obtain a DFS tree starting from vertex v */
public Tree 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; // Initialize parent[i] to -1
  // Mark visited vertices
  boolean[] isVisited = new boolean[vertices.size()];
  // Recursively search
  dfs(v, parent, searchOrder, isVisited);
  // Return a search tree
  return new Tree(v, parent, searchOrder);
}
/** Recursive method for DFS search */
private void dfs(int u, int[] parent, List<Integer> searchOrder,
    boolean[] isVisited) {
  // Store the visited vertex
  searchOrder.add(u);
  isVisited[u] = true; // Vertex v visited
  for (Edge e : neighbors.get(u)) {
    if (!isVisited[e.v]) {
      parent[e.v] = u; // The parent of vertex e.v is u
      dfs(e.v, parent, searchOrder, isVisited); // Recursive search
    }
  }
}
```

```
@Override /** Starting bfs search from vertex v */
public Tree 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; // Initialize parent[i] to -1
  java.util.LinkedList<Integer> queue =
    new java.util.LinkedList<>(); // list used as a queue
  boolean[] isVisited = new boolean[vertices.size()];
  queue.offer(v); // Enqueue v
  isVisited[v] = true; // Mark it visited
  while (!queue.isEmpty()) {
    int u = queue.poll(); // Dequeue to u
    searchOrder.add(u); // u searched
    for (Edge e: neighbors.get(u)) {
      if (!isVisited[e.v]) {
        queue.offer(e.v); // Enqueue w
        parent[e.v] = u; // The parent of w is u
        isVisited[e.v] = true; // Mark it visited
      }
    }
  }
  return new Tree(v, parent, searchOrder);
}
```

```
public class Tree {
  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 Tree(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;
  }
  /** Return the parent of vertex v */
  public int getParent(int v) {
    return parent[v];
  }
  /** Return an array representing search order */
  public List<Integer> getSearchOrder() {
    return searchOrder;
  }
  /** Return number of vertices found */
  public int getNumberOfVerticesFound() {
    return searchOrder.size();
  }
```

```
/** Return the path of vertices from a vertex to the root */
public List<V> getPath(int index) {
  ArrayList<V> path = new ArrayList<>();
  do { path.add(vertices.get(index));
    index = parent[index];
  } while (index != -1);
  return path;
}
/** Print a path from the root to vertex v */
public void printPath(int index) {
  List<V> path = getPath(index);
  System.out.print("A path from " + vertices.get(root) + " to " +
    vertices.get(index) + ": ");
  for (int i = path.size() - 1; i >= 0; i--)
    System.out.print(path.get(i) + " ");
}
/** Print the whole tree */
public void printTree() {
  System.out.println("Root is: " + vertices.get(root));
  System.out.print("Edges: ");
  for (int i = 0; i < parent.length; i++) {
    if (parent[i] != -1) {
      // Display an edge
      System.out.print("(" + vertices.get(parent[i]) + ", " +
        vertices.get(i) + ") ");
    } }
  System.out.println(); }}}
```

```
import java.util.*;
public class UnweightedGraph<V> extends AbstractGraph<V> {
  /** Construct an empty graph */
  public UnweightedGraph() {
  }
  /** Construct a graph from vertices and edges stored in arrays */
  public UnweightedGraph(V[] vertices, int[][] edges) {
    super(vertices, edges);
  }
  /** Construct a graph from vertices and edges stored in List */
  public UnweightedGraph(List<V> vertices, List<Edge> edges) {
    super(vertices, edges);
  }
  /** Construct a graph for integer vertices 0, 1, 2 and edge list */
  public UnweightedGraph(List<Edge> edges, int numberOfVertices) {
    super(edges, numberOfVertices);
  }
  /** Construct a graph from integer vertices 0, 1, and edge array */
  public UnweightedGraph(int[][] edges, int numberOfVertices) {
    super(edges, numberOfVertices);
  }
}
```

```
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}
    };
    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();

    String[] names = {"Peter", "Jane", "Mark", "Cindy", "Wendy"};
    java.util.ArrayList<AbstractGraph.Edge> edgeList
      = new java.util.ArrayList<>();
    edgeList.add(new AbstractGraph.Edge(0, 2));
    edgeList.add(new AbstractGraph.Edge(1, 2));
    edgeList.add(new AbstractGraph.Edge(2, 4));
    edgeList.add(new AbstractGraph.Edge(3, 4));
    // Create a graph with 5 vertices
    Graph<String> graph2 = new UnweightedGraph<>
      (java.util.Arrays.asList(names), edgeList);
    System.out.println("\nThe number of vertices in graph2: "
      + graph2.getSize());
    System.out.println("The edges for graph2:");
    graph2.printEdges();
  }
}
```

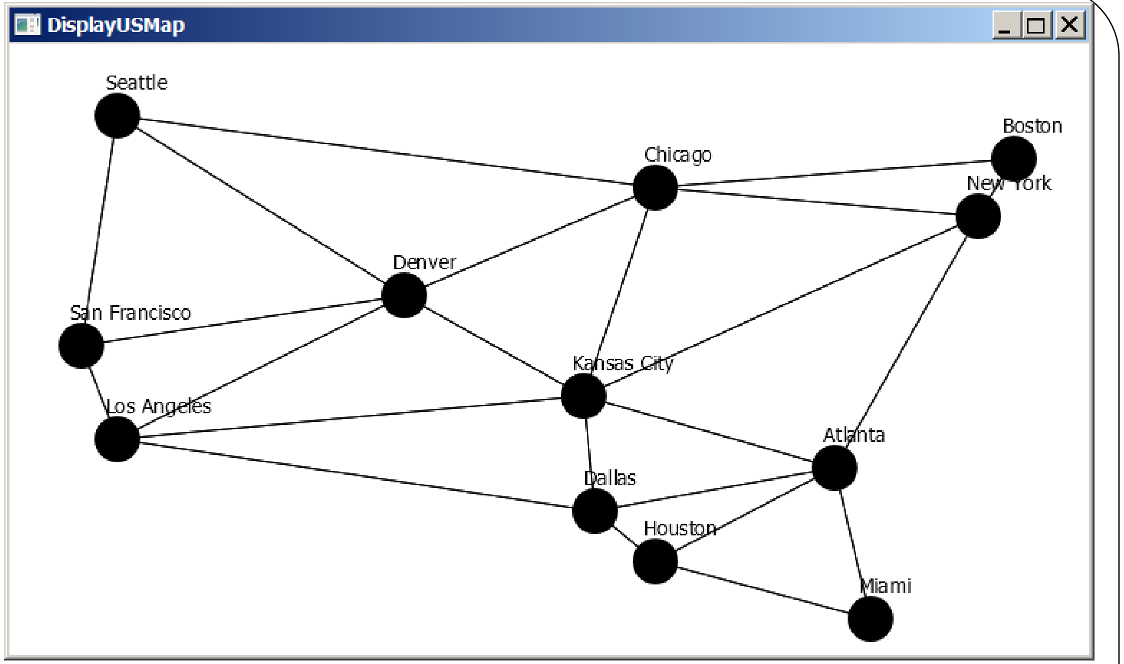
Graph Visualization

```
public interface Displayable {
  public int getX(); // Get x-coordinate of the vertex
  public int getY(); // Get x-coordinate of the vertex
  public String getName(); // Get display name of the vertex
}
import javafx.scene.layout.Pane;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Line;
import javafx.scene.text.Text;
public class GraphView extends Pane {
  private Graph<? extends Displayable> graph;
  public GraphView(Graph<? extends Displayable> graph) {
    this.graph = graph;
    // Draw vertices
    java.util.List<? extends Displayable> vertices
      = graph.getVertices();
    for (int i = 0; i < graph.getSize(); i++) {
      int x = vertices.get(i).getX();
      int y = vertices.get(i).getY();
      String name = vertices.get(i).getName();
      getChildren().add(new Circle(x, y, 16)); // Display a vertex
      getChildren().add(new Text(x - 8, y - 18, name));
    }
```

```
  // Draw edges for pair of vertices
  for (int i = 0; i < graph.getSize(); i++) {
   java.util.List<Integer> neighbors = graph.getNeighbors(i);
   int x1 = graph.getVertex(i).getX();
   int y1 = graph.getVertex(i).getY();
   for (int v: neighbors) {
    int x2 = graph.getVertex(v).getX();
    int y2 = graph.getVertex(v).getY();

    // Draw an edge for (i, v)
    getChildren().add(new Line(x1, y1, x2, y2));
   }
  }
 }
}
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
public class DisplayUSMap extends Application {
 @Override
 public void start(Stage primaryStage) {
  City[] vertices = {new City("Seattle", 75, 50),
   new City("San Francisco", 50, 210),
   new City("Los Angeles", 75, 275), new City("Denver", 275, 175),
   new City("Kansas City", 400, 245),
   new City("Chicago", 450, 100), new City("Boston", 700, 80),
   new City("New York", 675, 120), new City("Atlanta", 575, 295),
```

```
  new City("Miami", 600, 400), new City("Dallas", 408, 325),
  new City("Houston", 450, 360) };
 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<City> graph = new UnweightedGraph<>(vertices, edges);
 // Create a scene and place it in the stage
 Scene scene = new Scene(new GraphView(graph), 750, 450);
 primaryStage.setTitle("DisplayUSMap");
 primaryStage.setScene(scene);
 primaryStage.show();
}
static class City implements Displayable {
 private int x, y;
 private String name;
 City(String name, int x, int y) {
  this.name = name;
  this.x = x;
  this.y = y;
 }
```



```
    @Override
    public int getX() {
      return x;
    }
    @Override
    public int getY() {
      return y;
    }
    @Override
    public String getName() {
      return name;
    }
  }
  public static void main(String[] args) {
    launch(args);
  }
}
```

Dijkstra Shortest Path

```
import java.util.PriorityQueue;
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;
class Vertex implements Comparable<Vertex> {
       public final String name;
       public Edge[] adjacencies;
       public double minDistance = Double.POSITIVE_INFINITY;
       public Vertex previous;
       public Vertex(String argName) {
              name = argName;
       }
       public String toString() {
              return name;
       }
       public int compareTo(Vertex other) {
              return Double.compare(minDistance, other.minDistance);
       }
}
class Edge {
       public final Vertex target;
       public final double weight;
       public Edge(Vertex argTarget, double argWeight) {
              target = argTarget;
              weight = argWeight;
       }
}
```

Dijkstra Shortest Path

```
public class Dijkstra {
  public static void computePaths(Vertex source) {
    source.minDistance = 0.;
    PriorityQueue<Vertex> vertexQueue = new PriorityQueue<Vertex>();
    vertexQueue.add(source);
    while (!vertexQueue.isEmpty()) {
      Vertex u = vertexQueue.poll();
      // Visit each edge exiting u
      for (Edge e : u.adjacencies) {
        Vertex v = e.target;
        double weight = e.weight;
        double distanceThroughU = u.minDistance + weight;
        if (distanceThroughU < v.minDistance) {
          vertexQueue.remove(v);
          v.minDistance = distanceThroughU;
          v.previous = u;
          vertexQueue.add(v);
        }
      }
    }
  }
  public static List<Vertex> getShortestPathTo(Vertex target) {
    List<Vertex> path = new ArrayList<Vertex>();
    for (Vertex vertex = target; vertex != null; vertex = vertex.previous)
      path.add(vertex);
    Collections.reverse(path);
    return path;
  }
```

Dijkstra Shortest Path

```
public static void main(String[] args) {
    Vertex v0 = new Vertex("London");
    Vertex v1 = new Vertex("Dover");
    Vertex v2 = new Vertex("Calais");
    Vertex v3 = new Vertex("Paris");
    Vertex v4 = new Vertex("Rotterdam");
    Vertex v5 = new Vertex("Brussels");
    Vertex v6 = new Vertex("Lille");
    v0.adjacencies = new Edge[] { new Edge(v1, 1), new Edge(v4, 6),
        new Edge(v3, 2) };
    v1.adjacencies = new Edge[] { new Edge(v0, 1), new Edge(v2, 1) };
    v2.adjacencies = new Edge[] { new Edge(v1, 1), new Edge(v3, 1),
        new Edge(v6, 1) };
    v3.adjacencies = new Edge[] { new Edge(v0, 2), new Edge(v2, 1),
        new Edge(v6, 1) };
    v4.adjacencies = new Edge[] { new Edge(v0, 6), new Edge(v5, 1) };
    v5.adjacencies = new Edge[] { new Edge(v4, 1), new Edge(v6, 1) };
    v6.adjacencies = new Edge[] { new Edge(v3, 1), new Edge(v5, 1) };
    Vertex[] vertices = { v0, v1, v2, v3, v4, v5, v6 };
    // Paths from London
    computePaths(v0);
    for (Vertex v : vertices) {
      System.out.println("Distance from London to " + v + ": "
          + v.minDistance);
      List<Vertex> path = getShortestPathTo(v);
      System.out.println("Path: " + path);
    }
  }}
```