Weighted Graphs and Applications

Paul Fodor

CSE260, Computer Science B: Honors

Stony Brook University

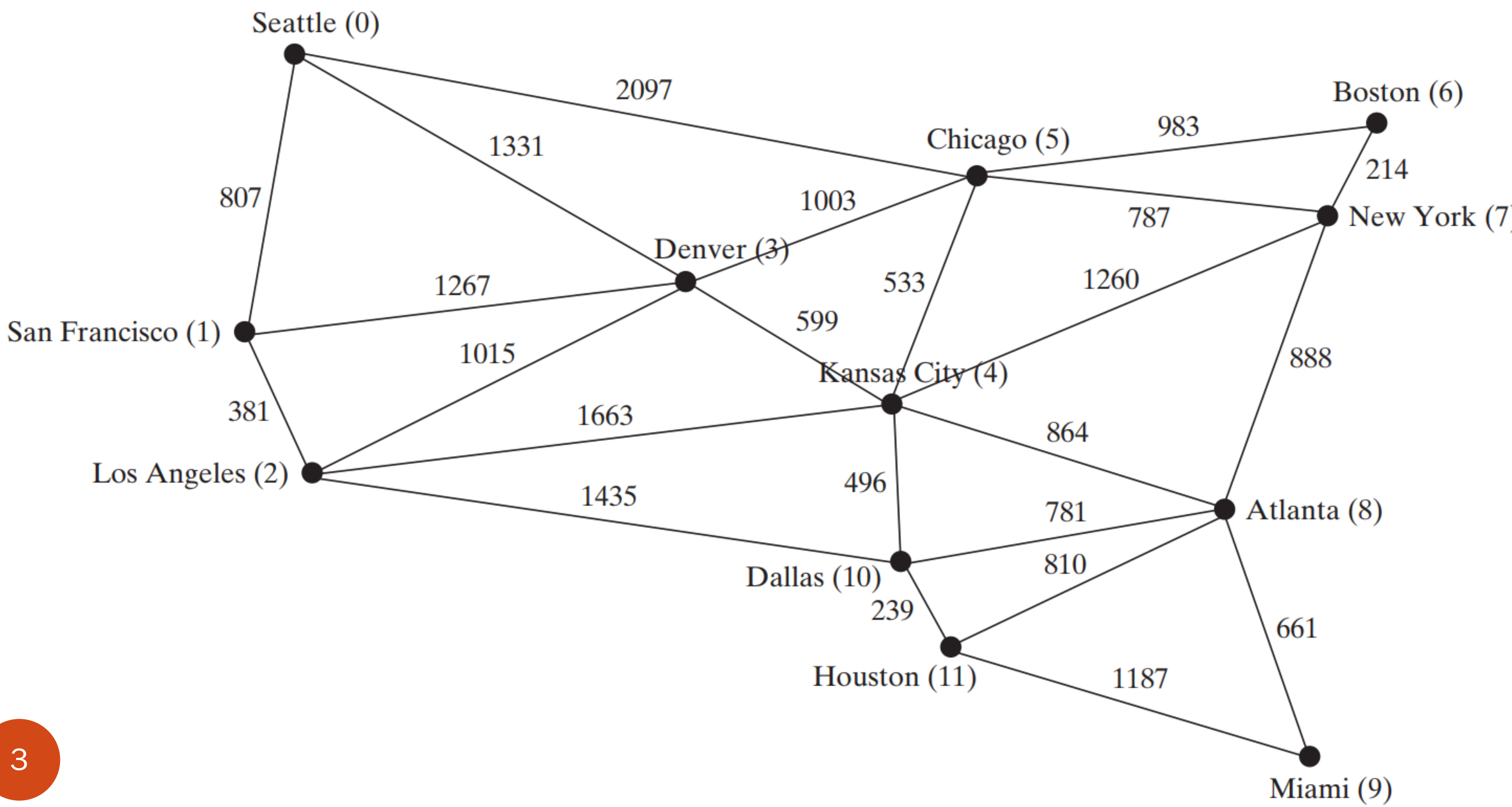
http://www.cs.stonybrook.edu/~cse260

Objectives

- To represent weighted edges using adjacency matrices and adjacency lists
- To model weighted graphs using the **`WeightedGraph`** class that extends the **`AbstractGraph`** class
- To design and implement the algorithm for finding a minimum spanning tree
  - To define the **`MST`** class that extends the **`Tree`** class
- To design and implement the algorithm for finding single-source shortest paths
  - To define the **`ShortestPathTree`** class that extends the **`Tree`** class

Weighted Graphs

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

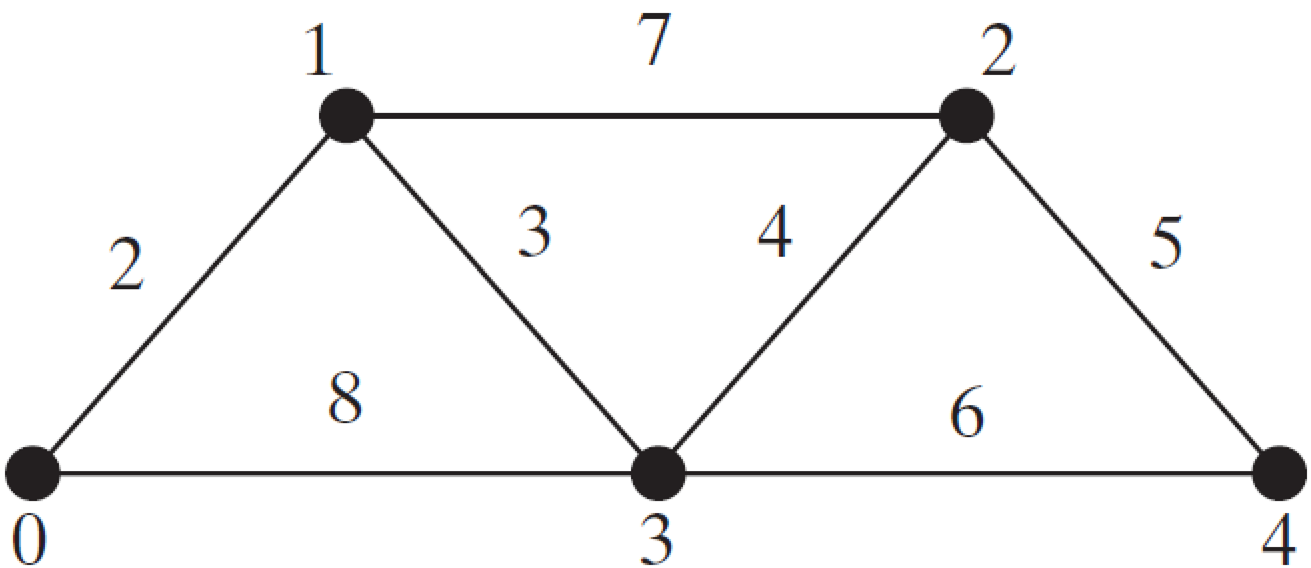


Representing Weighted Graphs

- Representing Weighted Edges
  - Edge Array
  - Weighted Adjacency Matrices
  - Adjacency Lists

Representing Weighted Edges: Edge Array

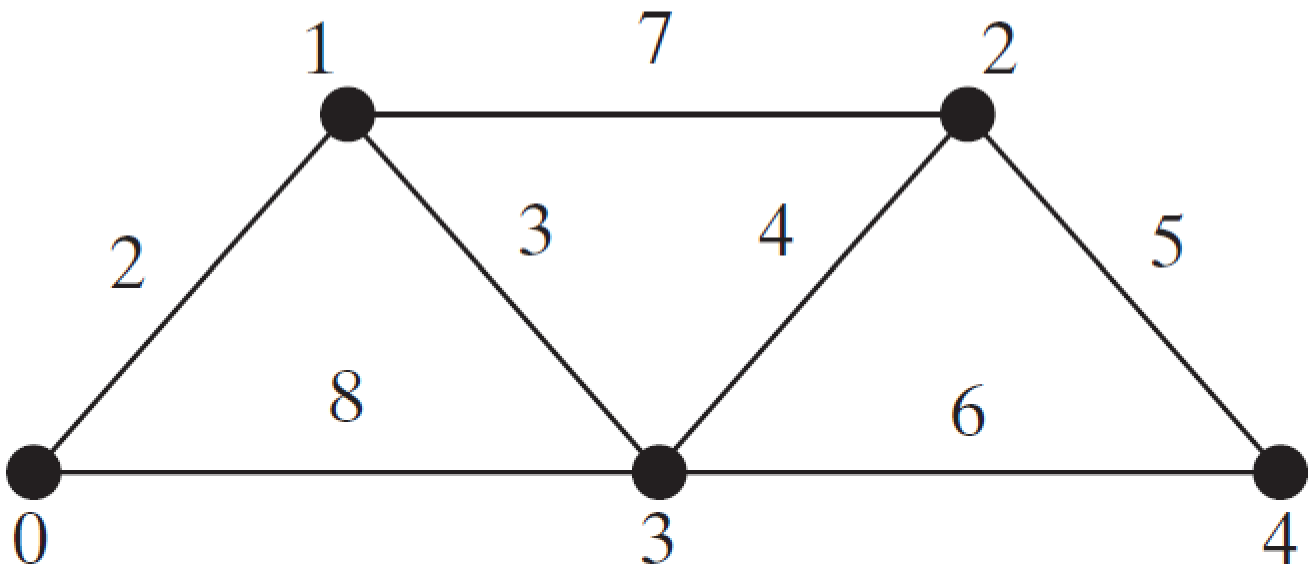
```
int[][] edges = {{0, 1, 2}, {0, 3, 8},
   {1, 0, 2}, {1, 2, 7}, {1, 3, 3},
   {2, 1, 7}, {2, 3, 4}, {2, 4, 5},
   {3, 0, 8}, {3, 1, 3}, {3, 2, 4}, {3, 4, 6},
   {4, 2, 5}, {4, 3, 6}
};
```



Weighted 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}
```

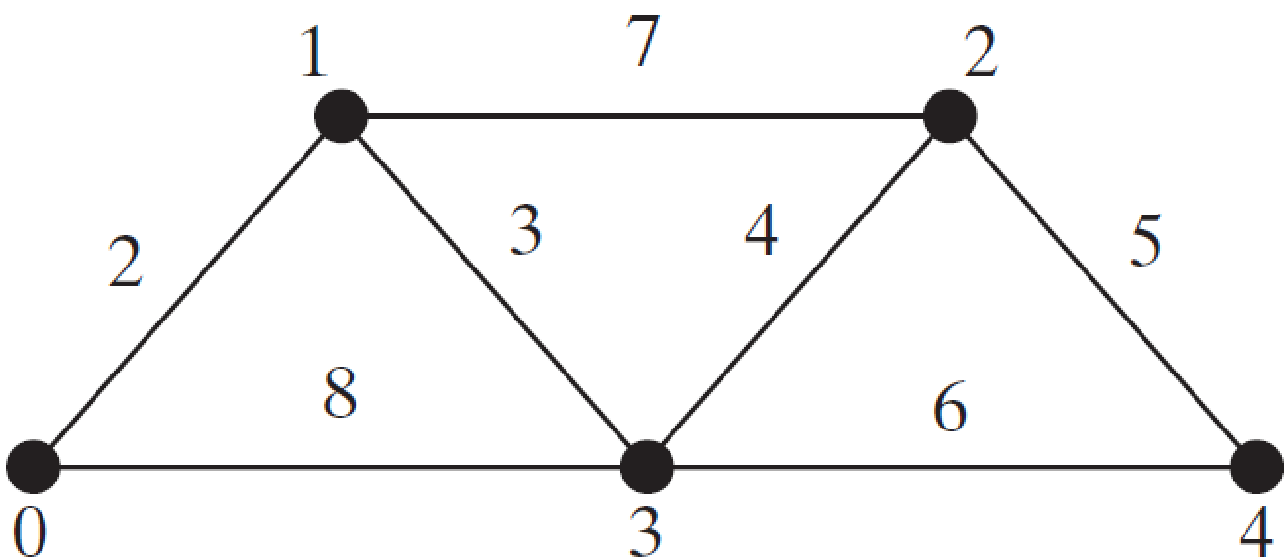
	0	1	2	3	4
0	null	2	null	8	null
1	2	null	7	3	null
2	null	7	null	4	5
3	8	3	4	null	6
4	null	null	5	6	null



Edge Adjacency Lists

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

list[0]	WeightedEdge(0, 1, 2)	WeightedEdge(0, 3, 8)		
list[1]	WeightedEdge(1, 0, 2)	WeightedEdge(1, 3, 3)	WeightedEdge(1, 2, 7)	
list[2]	WeightedEdge(2, 3, 4)	WeightedEdge(2, 4, 5)	WeightedEdge(2, 1, 7)	
list[3]	WeightedEdge(3, 1, 3)	WeightedEdge(3, 2, 4)	WeightedEdge(3, 4, 6)	WeightedEdge(3, 0, 8)
list[4]	WeightedEdge(4, 2, 5)	WeightedEdge(4, 3, 6)		

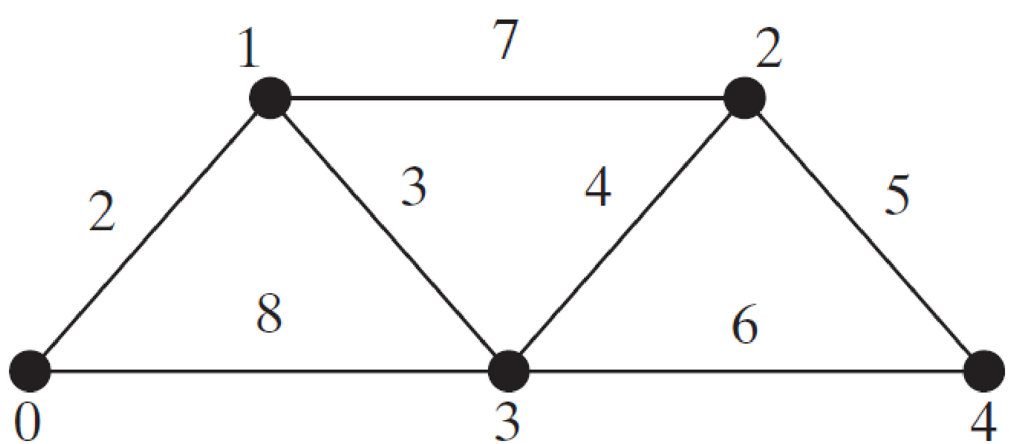


Edge Adjacency Lists

For flexibility, we will use an ArrayList rather than a fixed-sized array to represent the **list** as follows:

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

list[0]	WeightedEdge(0, 1, 2)	WeightedEdge(0, 3, 8)		
list[1]	WeightedEdge(1, 0, 2)	WeightedEdge(1, 3, 3)	WeightedEdge(1, 2, 7)	
list[2]	WeightedEdge(2, 3, 4)	WeightedEdge(2, 4, 5)	WeightedEdge(2, 1, 7)	
list[3]	WeightedEdge(3, 1, 3)	WeightedEdge(3, 2, 4)	WeightedEdge(3, 4, 6)	WeightedEdge(3, 0, 8)
list[4]	WeightedEdge(4, 2, 5)	WeightedEdge(4, 3, 6)		



```java
public class WeightedEdge extends AbstractGraph.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;
  }

  /** 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;
    }
  }
}
```

«interface»
Graph<V>

AbstractGraph<V>

WeightedGraph<V>

+WeightedGraph()	Constructs an empty graph.
+WeightedGraph(vertices: V[], edges: int[][])	Constructs a weighted graph with the specified edges and the number of vertices in arrays.
+WeightedGraph(vertices: List<V>, edges: List<WeightedEdge>)	Constructs a weighted graph with the specified edges and the number of vertices.
+WeightedGraph(edges: int[][], numberOfVertices: int)	Constructs a weighted graph with the specified edges in an array and the number of vertices.
+WeightedGraph(edges: List<WeightedEdge>, numberOfVertices: int)	Constructs a weighted graph with the specified edges in a list and the number of vertices.
+printWeightedEdges(): void	Displays all edges and weights.
+getWeight(int u, int v): double	Returns the weight on the edge from u to v. Throw an exception if the edge does not exist.
+addEdges(u: int, v: int, weight: double): void	Adds a weighted edge to the graph and throws an IllegalArgumentException if u, v, or w is invalid. If (u, v) is already in the graph, the new weight is set.
+getMinimumSpanningTree(): MST	Returns a minimum spanning tree starting from vertex 0.
+getMinimumSpanningTree(index: int): MST	Returns a minimum spanning tree starting from vertex v.
+getShortestPath(index: int): ShortestPathTree	Returns all single-source shortest paths.

```java
import java.util.*;
public class WeightedGraph<V> extends AbstractGraph<V> {
  /** Construct an empty */
  public WeightedGraph() {
  }

  /** Construct a WeightedGraph from vertices and edges in list */
  public WeightedGraph(int[][] edges, int numberOfVertices) {
    List<V> vertices = new ArrayList<>();
    for (int i = 0; i < numberOfVertices; i++)
      vertices.add((V)(new Integer(i)));
    createWeightedGraph(vertices, edges);
  }

  /** Construct a WeightedGraph from vertices and edged in arrays */
  public WeightedGraph(V[] vertices, int[][] edges) {
    createWeightedGraph(java.util.Arrays.asList(vertices), edges);
  }


  /** Construct a WeightedGraph for vertices 0, 1, 2 and edge list */
  public WeightedGraph(List<V> vertices, List<WeightedEdge> edges) {
    createWeightedGraph(vertices, edges);
  }
```

```
/** Create adjacency lists from edge arrays */
private void createWeightedGraph(List<V> vertices, int[][] edges) {
   this.vertices = vertices;
   for (int i = 0; i < vertices.size(); i++)
     // Create an edge list for each vertex
     neighbors.add(new ArrayList<Edge>());
   for (int i = 0; i < edges.length; i++)
     neighbors.get(edges[i][0]).add(
       new WeightedEdge(edges[i][0], edges[i][1], edges[i][2]));
}

/** Create adjacency lists from edge lists */
private void createWeightedGraph(List<V> vertices, List<WeightedEdge> edges) {
  this.vertices = vertices;
  for (int i = 0; i < vertices.size(); i++)
    neighbors.add(new ArrayList<Edge>()); // Create a list for vertices
  for (WeightedEdge edge: edges)
    neighbors.get(edge.u).add(edge); // Add an edge into the list
}

/** Construct a WeightedGraph from vertices 0, 1, and edge array */
public WeightedGraph(List<WeightedEdge> edges, int numberOfVertices) {
  List<V> vertices = new ArrayList<>();
  for (int i = 0; i < numberOfVertices; i++)
    vertices.add((V)(new Integer(i)));
  createWeightedGraph(vertices, edges);
}
```

```java
/** Return the weight on the edge (u, v) */
public double getWeight(int u, int v) throws Exception {
  for (Edge edge : neighbors.get(u))
    if (edge.v == v)
      return ((WeightedEdge)edge).weight;
  throw new Exception("Edge does not exit");
}

/** Add edges to the weighted graph */
public boolean addEdge(int u, int v, double weight) {
  return addEdge(new WeightedEdge(u, v, weight));
}

/** Display edges with weights */
public void printWeightedEdges() {
  for (int i = 0; i < getSize(); i++) {
    System.out.print(getVertex(i) + " (" + i + "): ");
    for (Edge edge : neighbors.get(i))
      System.out.print("(" + edge.u +
        ", " + edge.v + ", " + ((WeightedEdge)edge).weight + ") ");
    System.out.println();
  }
}
// to continue later with minimum spanning tree and
//  single-source shortest paths
```

```java
public class TestWeightedGraph {
  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);

    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.printWeightedEdges();
```

```
    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);
    System.out.println("\nThe edges for graph2:");
    graph2.printWeightedEdges();
  }
}
```

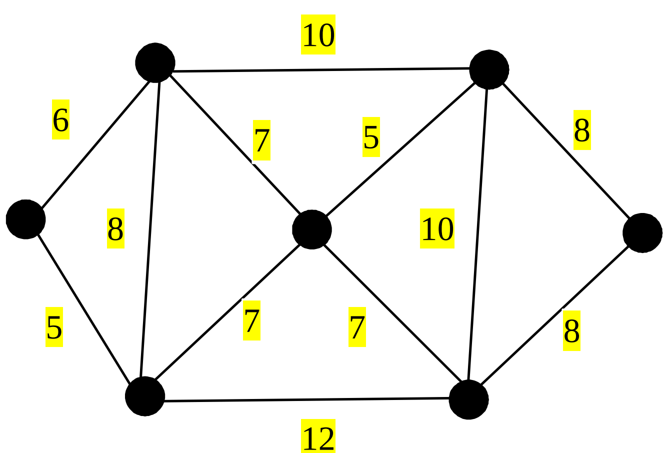
```
The number of vertices in graph1: 12
The vertex with index 1 is San Francisco
The index for Miami is 9
The edges for graph1:
Vertex 0: (0, 1, 807) (0, 3, 1331) (0, 5, 2097)
Vertex 1: (1, 2, 381) (1, 0, 807) (1, 3, 1267)
Vertex 2: (2, 1, 381) (2, 3, 1015) (2, 4, 1663) (2, 10, 1435)
Vertex 3: (3, 4, 599) (3, 5, 1003) (3, 1, 1267)
 (3, 0, 1331) (3, 2, 1015)
Vertex 4: (4, 10, 496) (4, 8, 864) (4, 5, 533) (4, 2, 1663)
 (4, 7, 1260) (4, 3, 599)
Vertex 5: (5, 4, 533) (5, 7, 787) (5, 3, 1003)
 (5, 0, 2097) (5, 6, 983)
Vertex 6: (6, 7, 214) (6, 5, 983)
Vertex 7: (7, 6, 214) (7, 8, 888) (7, 5, 787) (7, 4, 1260)
Vertex 8: (8, 9, 661) (8, 10, 781) (8, 4, 864)
 (8, 7, 888) (8, 11, 810)
Vertex 9: (9, 8, 661) (9, 11, 1187)
Vertex 10: (10, 11, 239) (10, 4, 496) (10, 8, 781) (10, 2, 1435)
Vertex 11: (11, 10, 239) (11, 9, 1187) (11, 8, 810)

The edges for graph2:
Vertex 0: (0, 1, 2) (0, 3, 8)
Vertex 1: (1, 0, 2) (1, 2, 7) (1, 3, 3)
Vertex 2: (2, 3, 4) (2, 1, 7) (2, 4, 5)
Vertex 3: (3, 1, 3) (3, 4, 6) (3, 2, 4) (3, 0, 8)
Vertex 4: (4, 2, 5) (4, 3, 6)
```

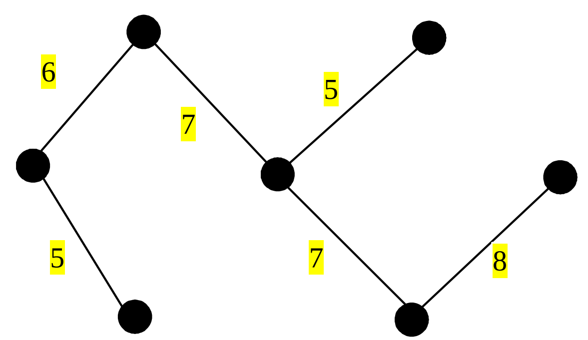
Minimum Spanning Trees

- A graph may have many spanning trees
  - A ***minimum spanning tree*** is a spanning tree (i.e., a subset of the edges of a connected undirected graph that connects all the vertices together without any cycles) with the minimum total weight

Minimum Spanning Trees

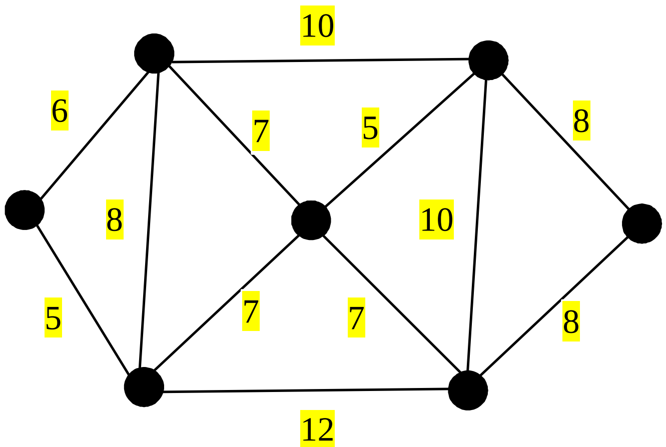
- Application example: a company wants to create wired Internet lines to connect all the customers together
  - There are many ways (i.e., streets) to connect all customers together
  - Different lines have different cost (e.g., length)
  - The cheapest way is to find a spanning tree with the minimum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

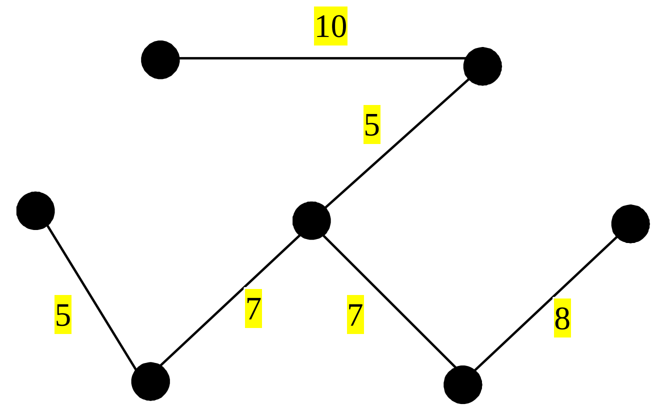




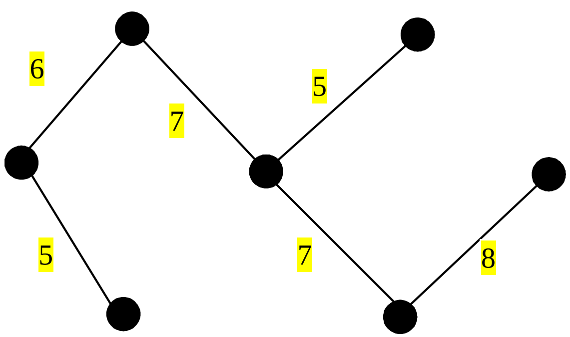
Total w: 38

Minimum Spanning Trees

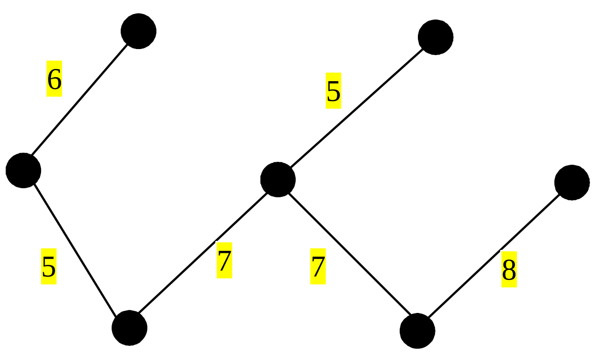




Total w: 42
Not a minimum
spanning tree



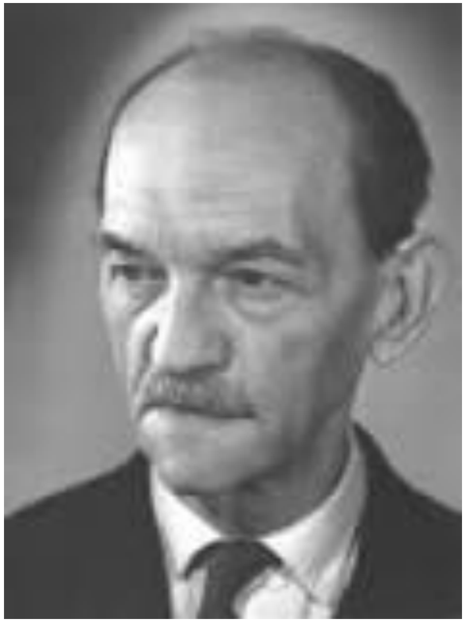
Total w: 38



Total w: 38

Minimum Spanning Trees

- An algorithm was developed in 1930 by Czech mathematician Vojtěch Jarník and later rediscovered and republished by computer scientists Robert C. Prim in 1957 and Edsger W. Dijkstra in 1959



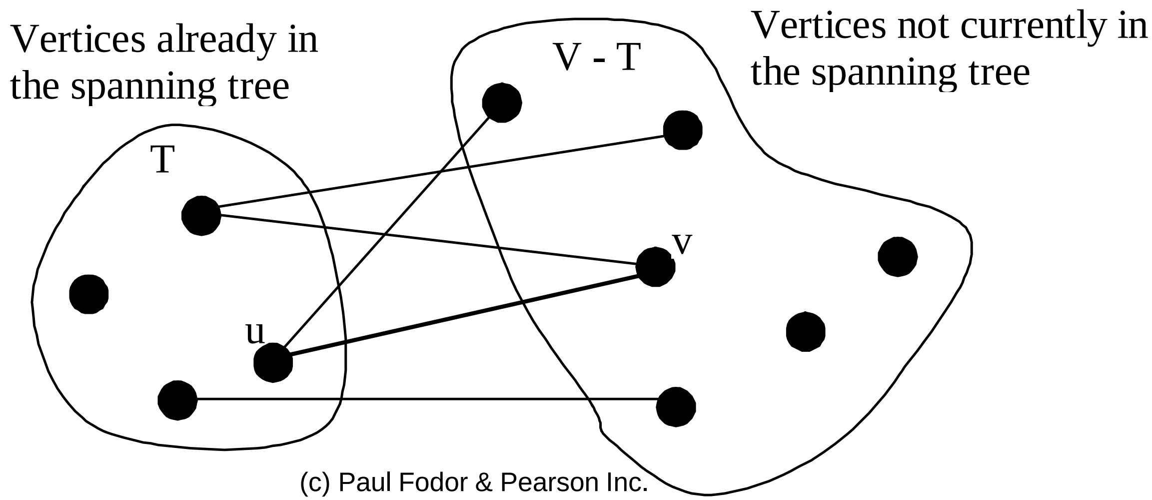
Vojtěch Jarník

Robert C. Prim

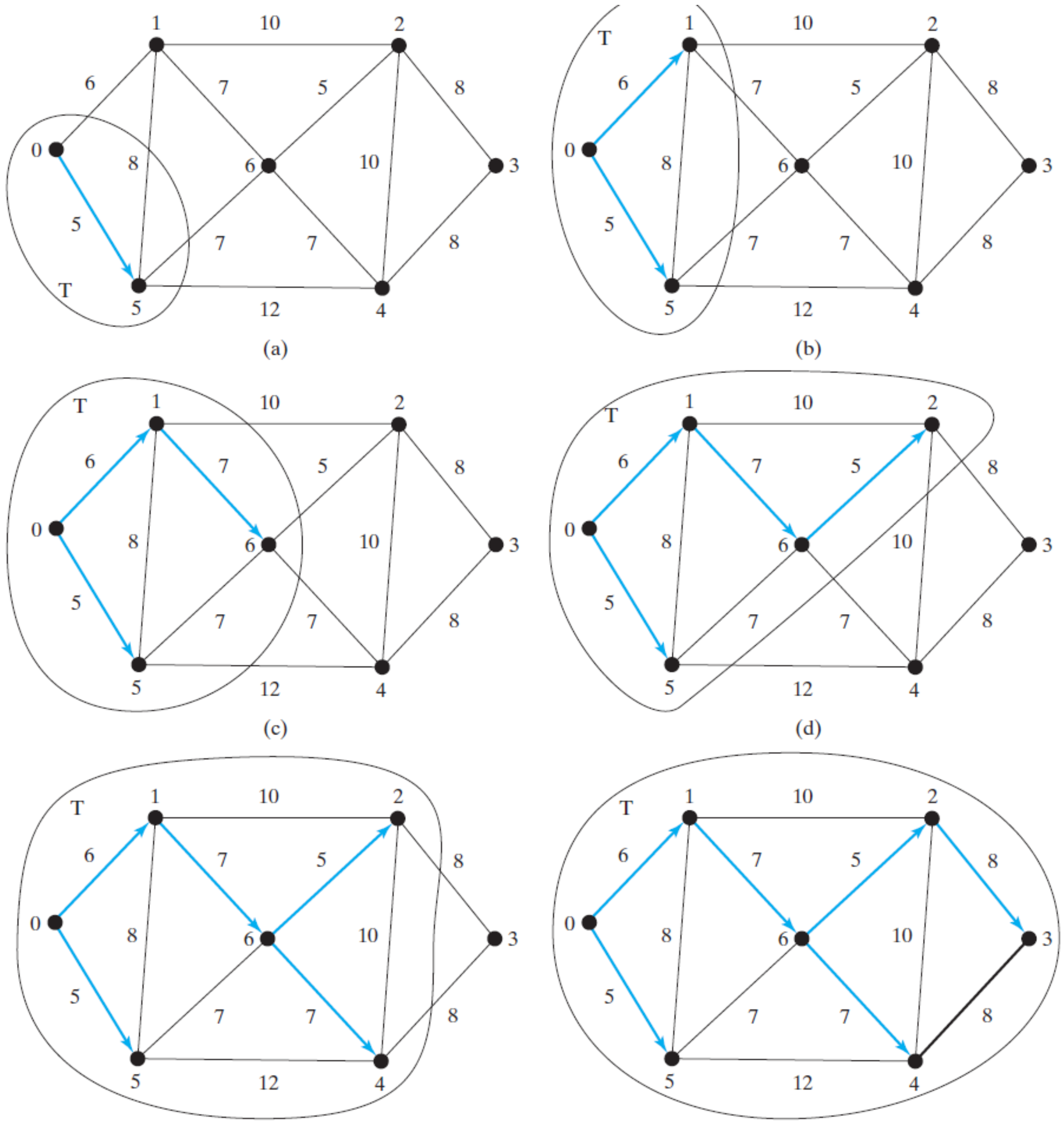
Edsger W. Dijkstra

Prim's Minimum Spanning Tree Algorithm

```
Input: G = (V, E)
Output: a MST
Tree minimumSpanningTree() {
  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;
  }
}
```



Minimum Spanning Tree Algorithm Example



(a) (b) (c) (d) (e) (f)

Refined Version of Prim's Minimum Spanning Tree Algorithm

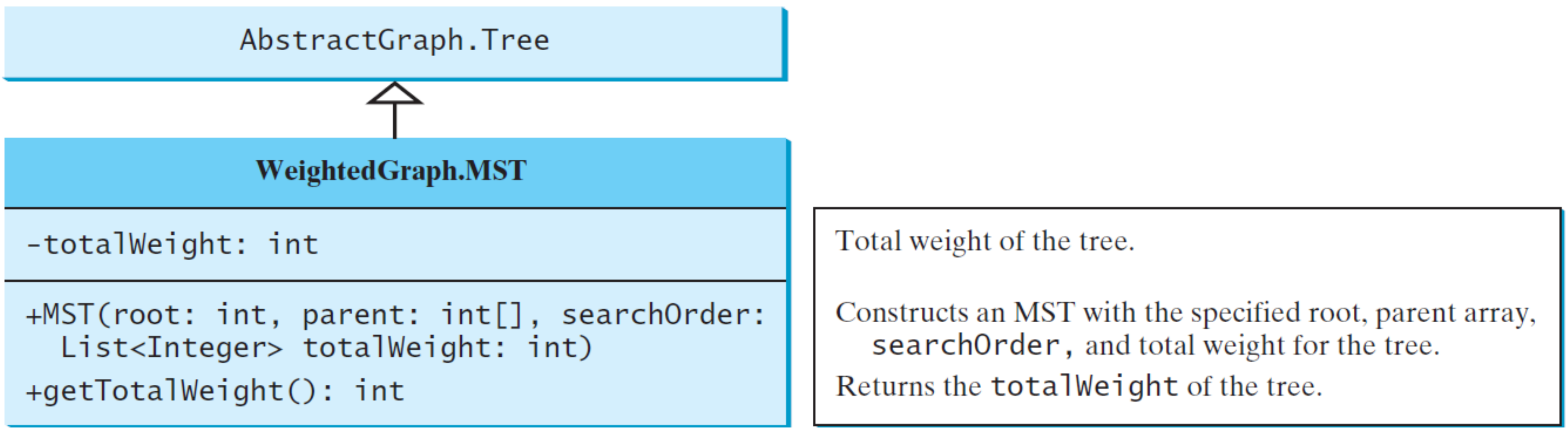
- To make it easy to identify the next vertex to add into the tree, we use `cost[v]` to store the cost of adding a vertex `v` to the spanning tree `T`
- Initially `cost[s]` is `0` for a starting vertex and assign infinity to `cost[v]` for all other vertices
- The algorithm repeatedly finds a vertex `u` in `V - T` with the smallest `cost[u]` and moves `u` to `T`
  - For all edges `(u,v)` where `v` in `V - T` update the `cost[v]` with min between the old cost and the weight of `(u,v)`

Refined Version of Prim's Minimum Spanning Tree Algorithm

```
Input: a graph G = (V, E)
Output: a minimum spanning tree with the starting vertex s as the root
  MST getMinimumSpanngingTree(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 &&
         cost[v] > w(u, v)) {
          cost[v] = w(u, v);
          parent[v] = u;
       }
   }
  }
```

- Testing whether a vertex **v** is in **T** by invoking **T.contains(v)** takes **O(|V|)** time since **T** is a list
  - Therefore, the overall time complexity for this implementation is **O(|V|²|E|)**, but it can be reduced to with binary heaps and hashing.

Implementing MST Algorithm



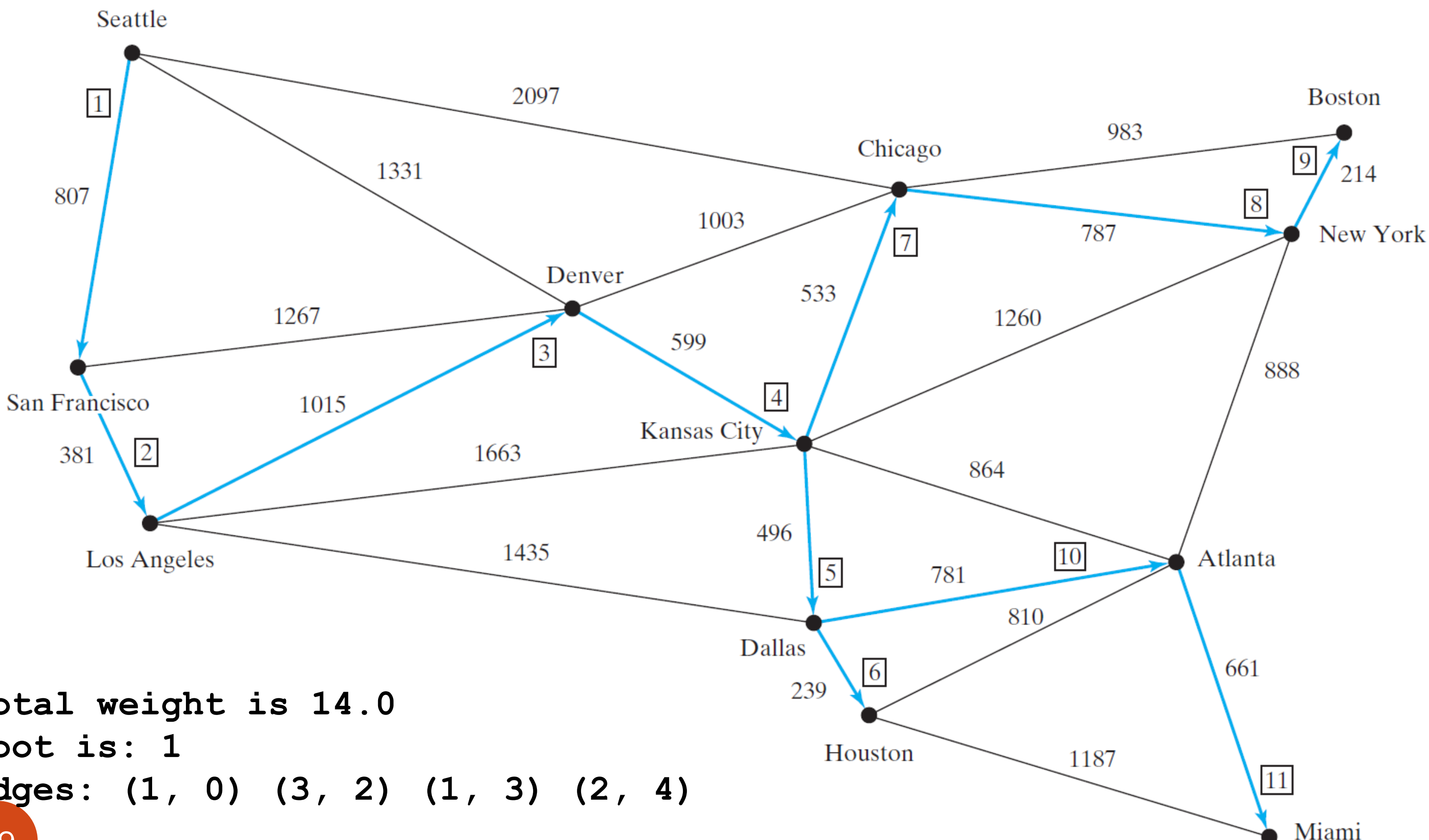
```
/** MST is an inner class in WeightedGraph */
public class MST extends Tree {
  private double totalWeight; // Total weight of all edges in the tree
  public MST(int root, int[] parent, List<Integer> searchOrder,
      double totalWeight) {
    super(root, parent, searchOrder);
    this.totalWeight = totalWeight;
  }
  public double getTotalWeight() {
    return totalWeight;
  }
}

/** 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();
```

```
    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();

  }
}
```

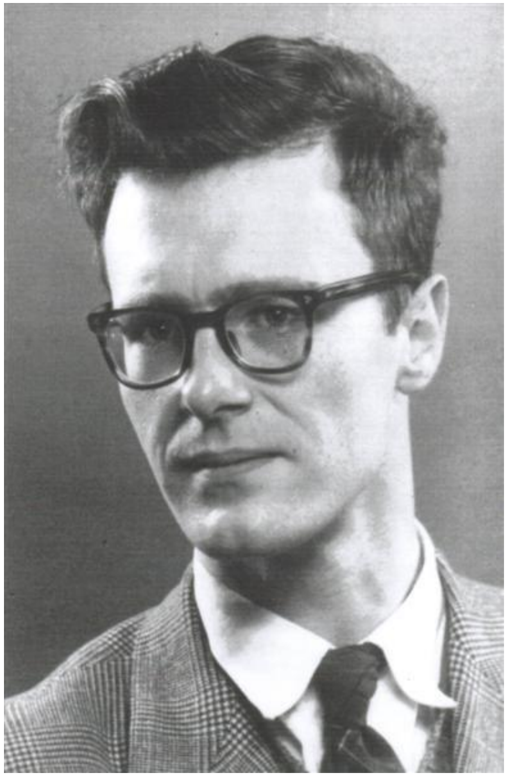
```
Total weight is 6513.0
Root is: Seattle
Edges: (Seattle, San Francisco) (San Francisco, Los Angeles)
 (Los Angeles, Denver) (Denver, Kansas City) (Kansas City, Chicago)
 (New York, Boston) (Chicago, New York) (Dallas, Atlanta)
 (Atlanta, Miami) (Kansas City, Dallas) (Dallas, Houston)
```




```
Total weight is 14.0
Root is: 1
Edges: (1, 0) (3, 2) (1, 3) (2, 4)
```

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

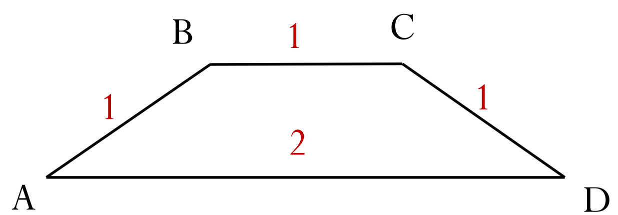
Single Source Shortest Path Algorithm

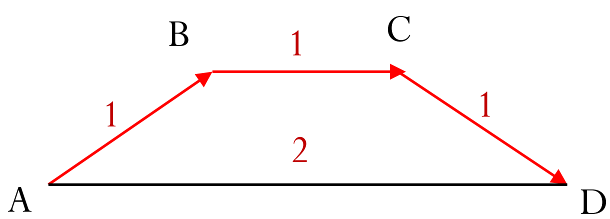
```
Input: a graph G = (V, E) with non-negative weights
Output: a shortest path tree with the source vertex s as the root
ShortestPathTree getShortestPath(s) {
 Let T be a set that contains the vertices whose
    paths to s are known; Initially T is empty;
 Set cost[s] = 0; and cost[v] = infinity for all other vertices in V;
 while (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;
      }
 }
}
```

This algorithm is very similar to Prim's for finding a minimum spanning tree:

- Both algorithms divide the vertices into two sets: T and V – T
  - In the case of Prim's algorithm, set T contains the vertices that are already added to the tree
  - In the case of Dijkstra's, set T contains the vertices whose shortest paths to the source have been found
- Both algorithms repeatedly find a vertex from V – T and add it to T
  - In the case of Prim's algorithm, the vertex is adjacent to some vertex in the set with the minimum weight on the edge
  - In Dijkstra's algorithm, the vertex is adjacent to some vertex in the set with the minimum total cost to the source

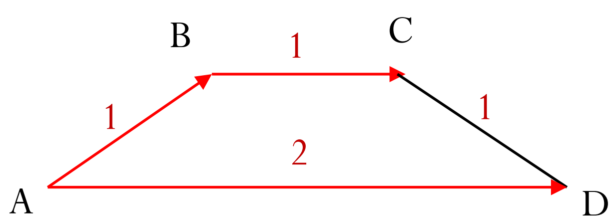
Minimum Spanning Tree vs. Shortest Path Algorithm





Minimum Spanning Tree

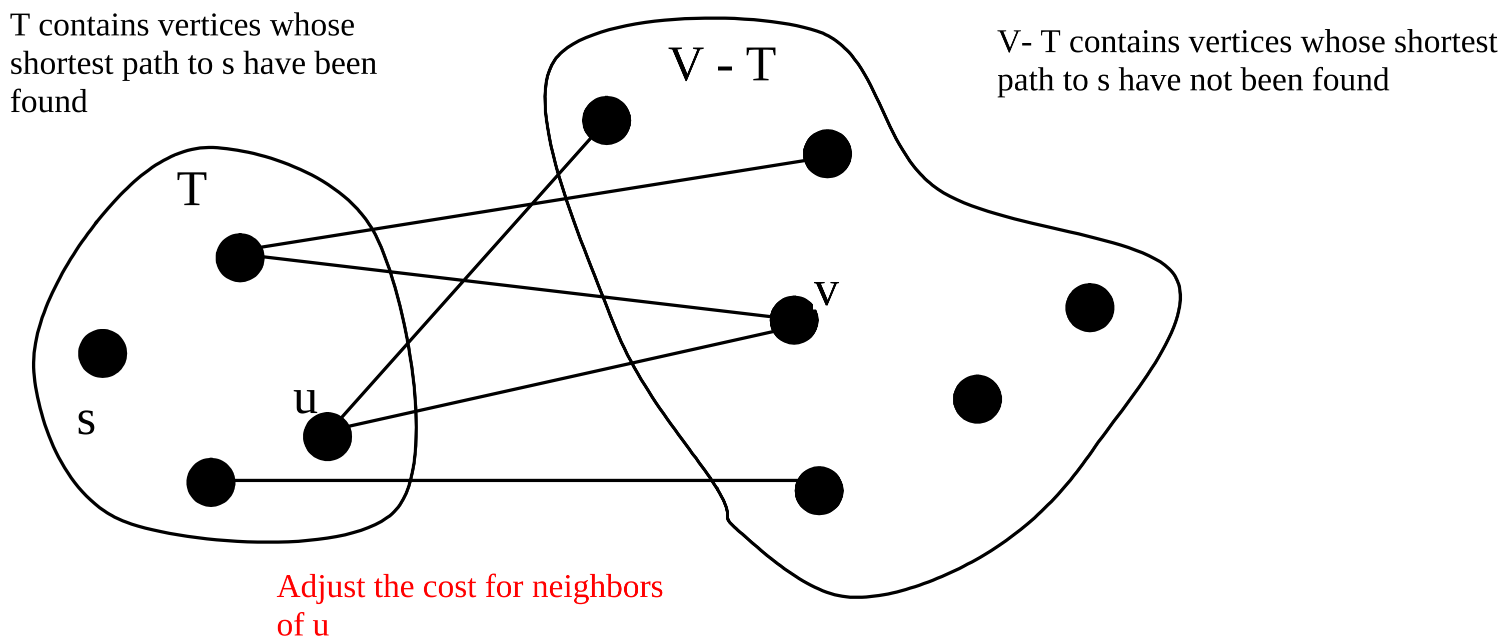
cost[v] = w(u, v)



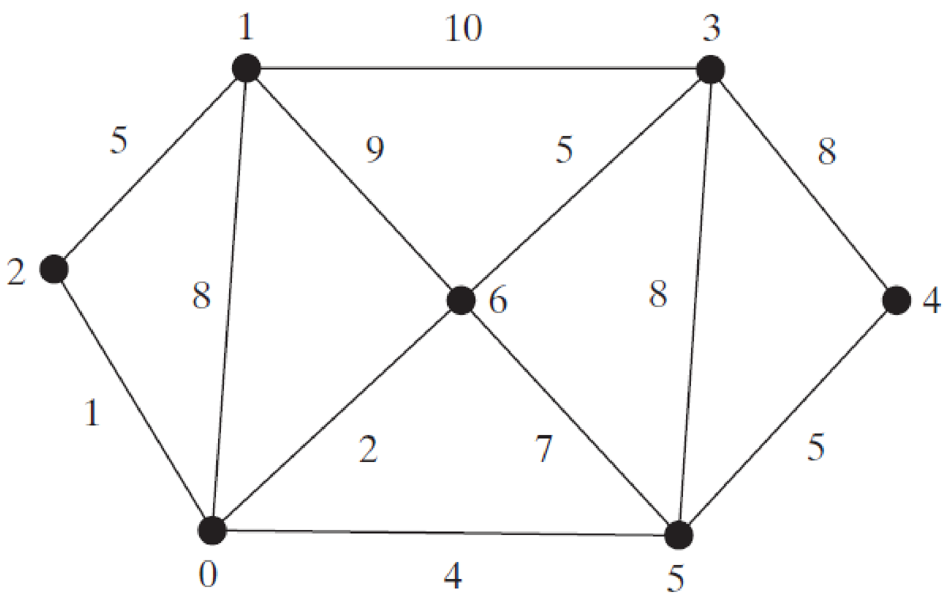
Shortest Path from node A

cost[v] = w(u, v) + cost[u]

Single Source Shortest Path Algorithm



SP Algorithm Example (Step 0)



(a)

cost

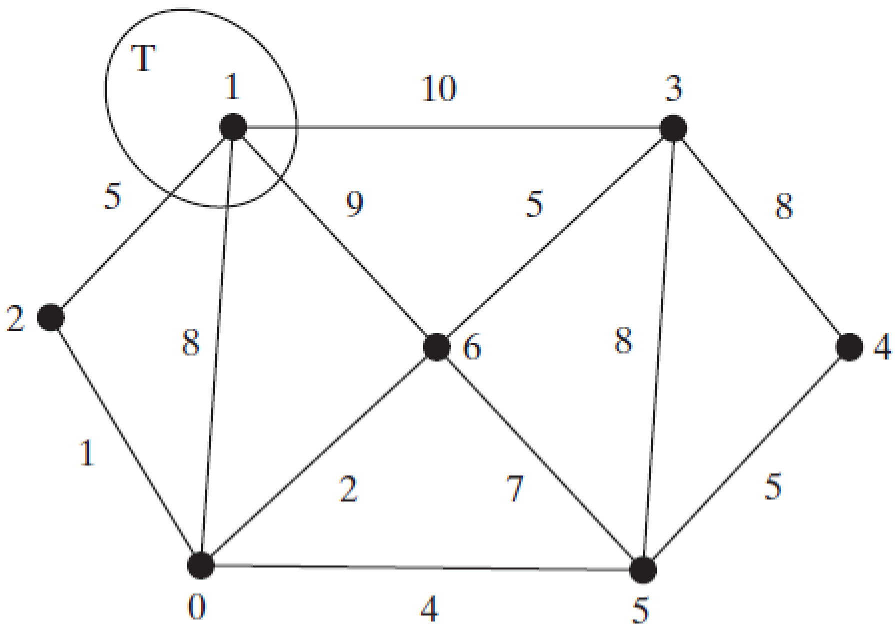
0	1	2	3	4	5	6
∞	0	∞	∞	∞	∞	∞

parent

0	1	2	3	4	5	6
	–1					

(b)

SP Algorithm Example (Step 1)



(a)

cost

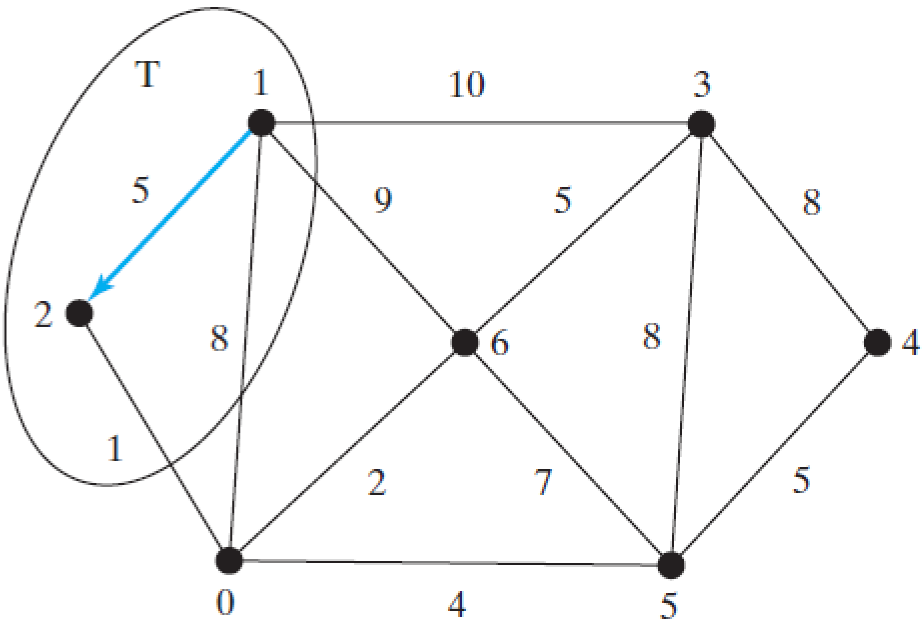
0	1	2	3	4	5	6
8	0	5	10	∞	∞	9

parent

0	1	2	3	4	5	6
1	–1	1	1			1

(b)

SP Algorithm Example (Step 2)



(a)

8

cost

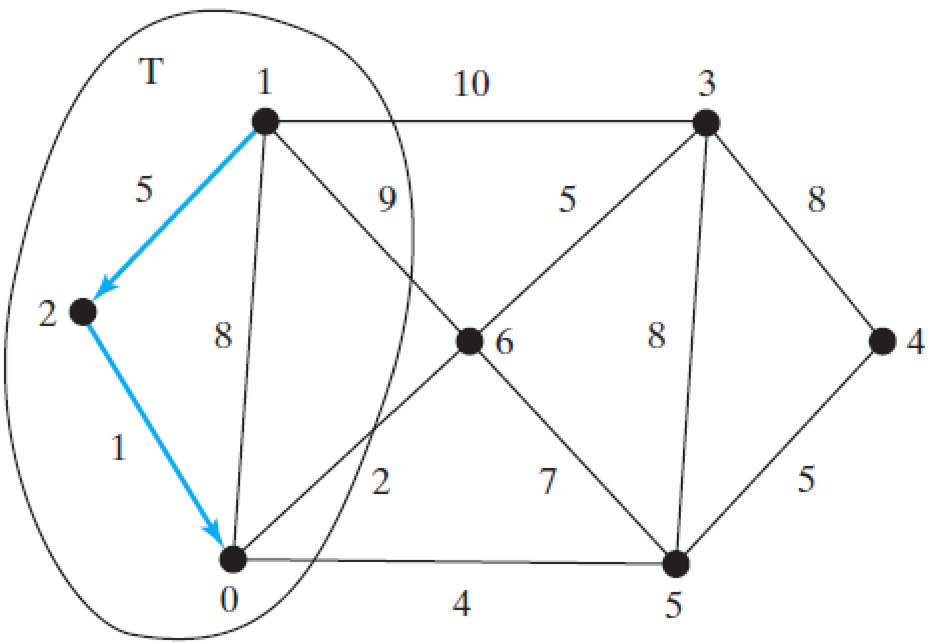
6	0	5	10	∞	∞	9
0	1	2	3	4	5	6

parent

2	–1	1	1			1
0	1	2	3	4	5	6

(b)

SP Algorithm Example (Step 3)



(a)

9

cost

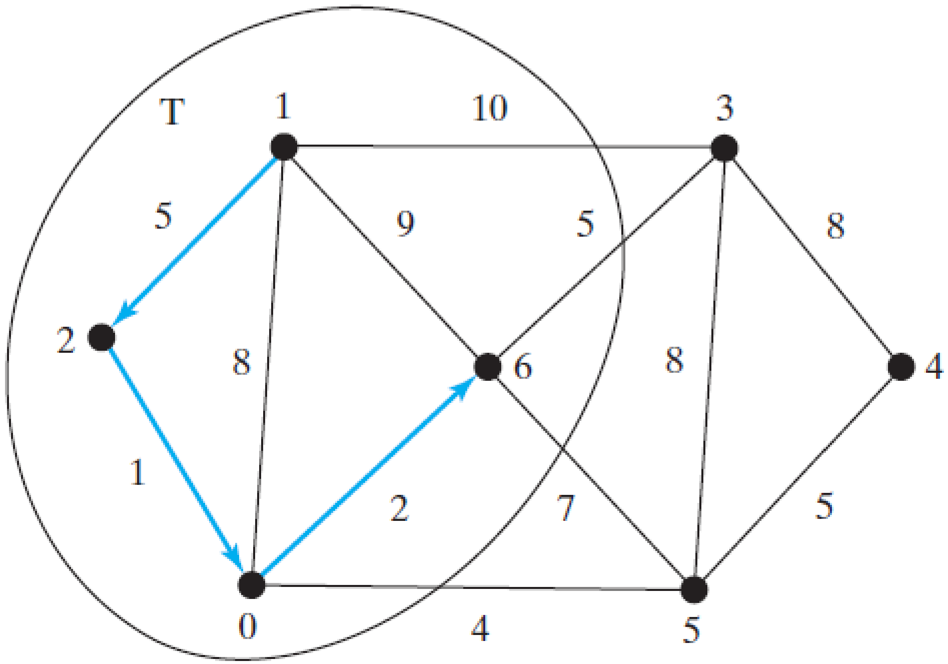
6	0	5	10	∞	10	8
0	1	2	3	4	5	6

parent

2	–1	1	1		0	0
0	1	2	3	4	5	6

(b)

SP Algorithm Example (Step 4)



(a)

cost

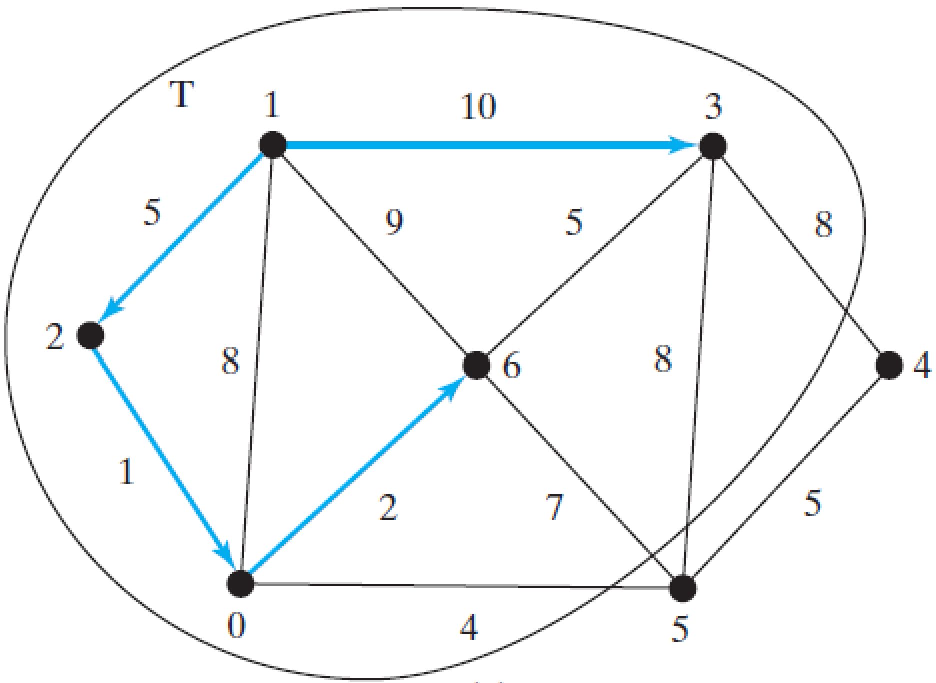
6	0	5	10	∞	10	8
0	1	2	3	4	5	6

parent

2	–1	1	1		0	0
0	1	2	3	4	5	6

(b)

SP Algorithm Example (Step 5)



(a)

cost

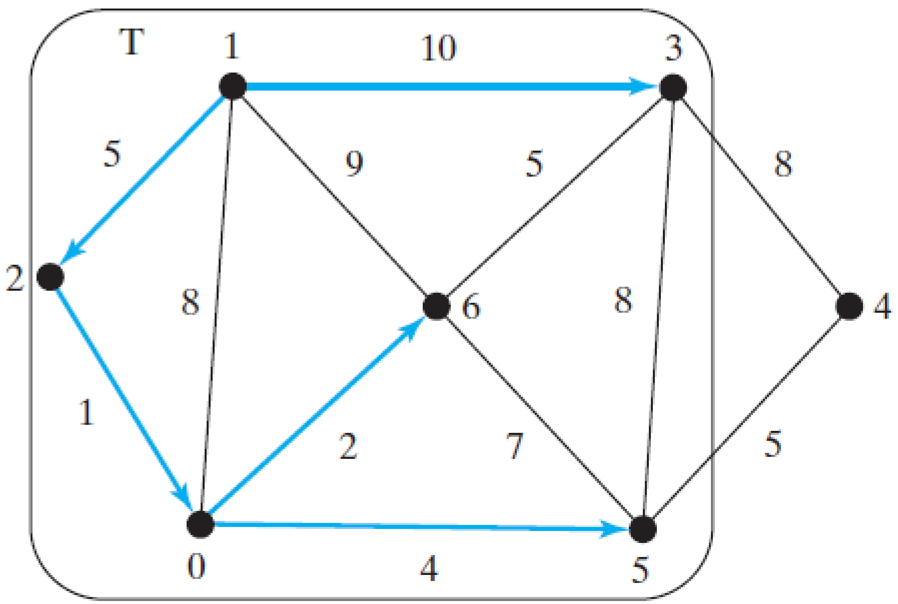
6	0	5	10	18	10	8
0	1	2	3	4	5	6

parent

2	–1	1	1	3	0	0
0	1	2	3	4	5	6

(b)

SP Algorithm Example (Step 6)



(a)

18

cost

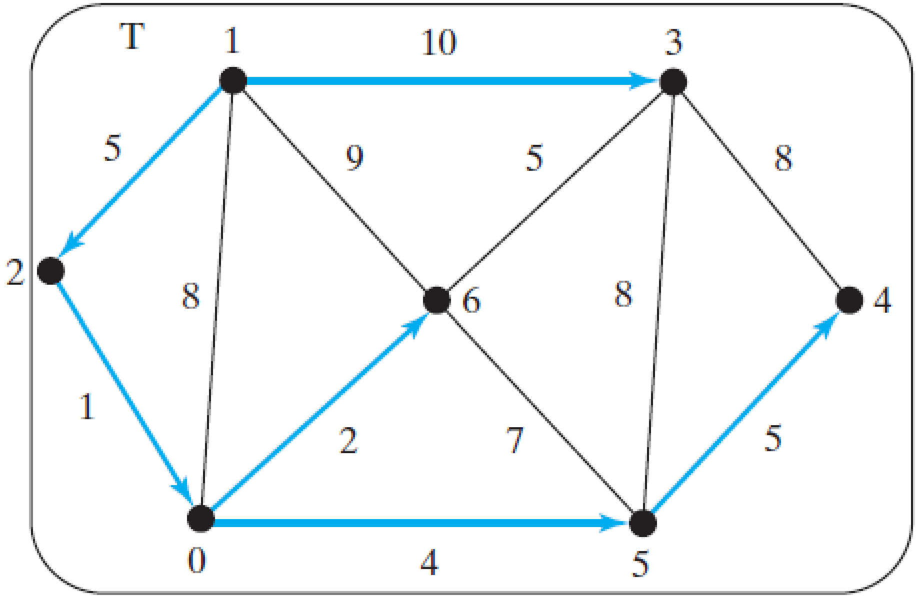
6	0	5	10	15	10	8
0	1	2	3	4	5	6

parent

2	–1	1	1	5	0	0
0	1	2	3	4	5	6

(b)

SP Algorithm Example (Step 7)



(a)

cost

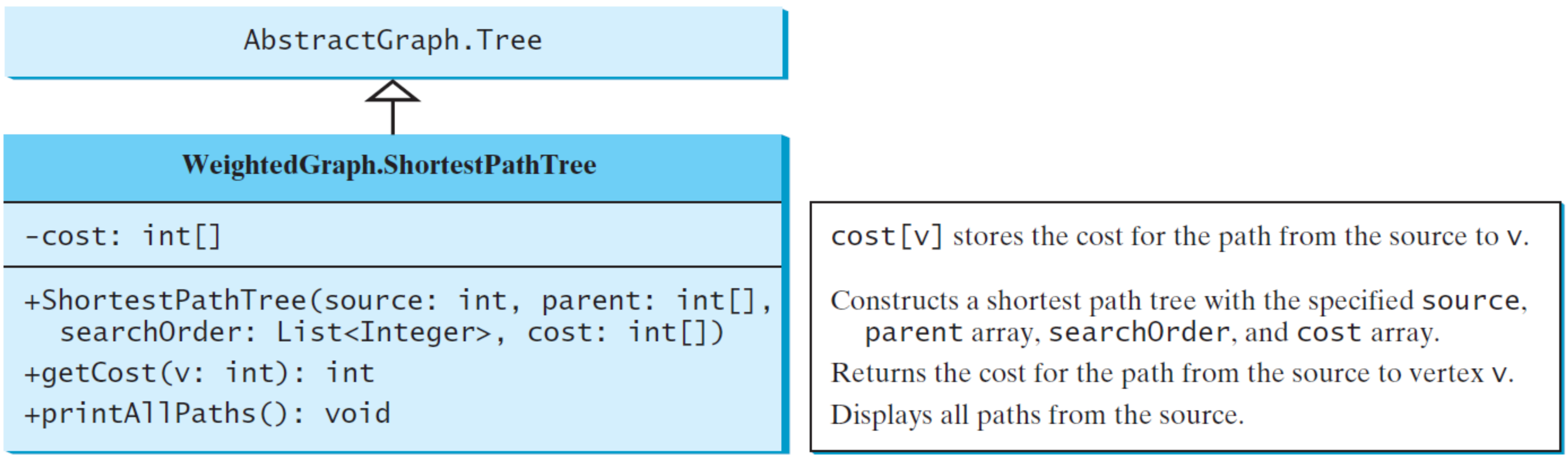
6	0	5	10	15	10	8
0	1	2	3	4	5	6

parent

2	–1	1	1	5	0	0
0	1	2	3	4	5	6

(b)

SP Algorithm Implementation

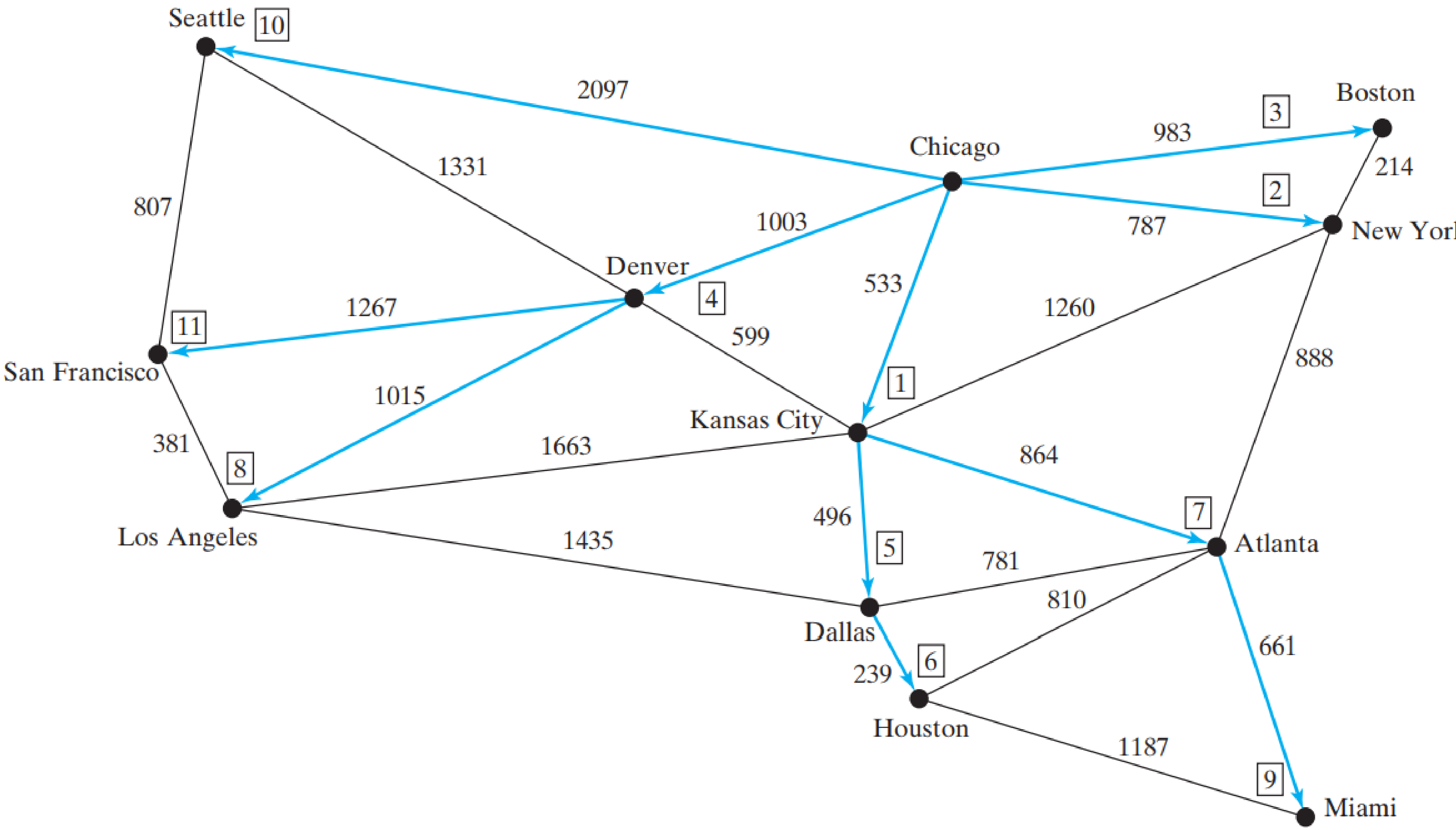


```
/** Find single source shortest paths */
public ShortestPathTree getShortestPath(int sourceVertex) {
  // cost[v] stores the cost of the path from v to the source
  double[] cost = new double[getSize()];
  for (int i = 0; i < cost.length; i++)
    cost[i] = Double.POSITIVE_INFINITY; // Initial cost set to infinity
  cost[sourceVertex] = 0; // Cost of source is 0
  // 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);
}
```

```java
  /** ShortestPathTree is an inner class in WeightedGraph */
  public class ShortestPathTree extends Tree {
    private double[] cost; // cost[v] is the cost from v to source

    /** Construct a path */
    public ShortestPathTree(int source, int[] parent,
        List<Integer> searchOrder, double[] cost) {
      super(source, parent, searchOrder);
      this.cost = cost;
    }

    /** Return the cost for a path from the root to vertex v */
    public double getCost(int v) {
      return cost[v];
    }

    /** Print paths from all vertices to the source */
    public void printAllPaths() {
      System.out.println("All shortest paths from " +
        vertices.get(getRoot()) + " are:");
      for (int i = 0; i < cost.length; i++) {
        printPath(i); // Print a path from i to the source
        System.out.println("(cost: " + cost[i] + ")"); // Path cost
      }
    }
  }

}
```

```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();
```

```
    // 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();
  }

}
```


Seattle
10
2097
Boston
3
983
Chicago
214
1331
2
807
1003
787
New York
Denver
4
533
1260
1267
11
599
888
San Francisco
1
1015
Kansas City
381
1663
8
864
Los Angeles
7
496
Atlanta
1435
5
781
810
Dallas
661
6
239
Houston
1187
9
Miami

```
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)
```