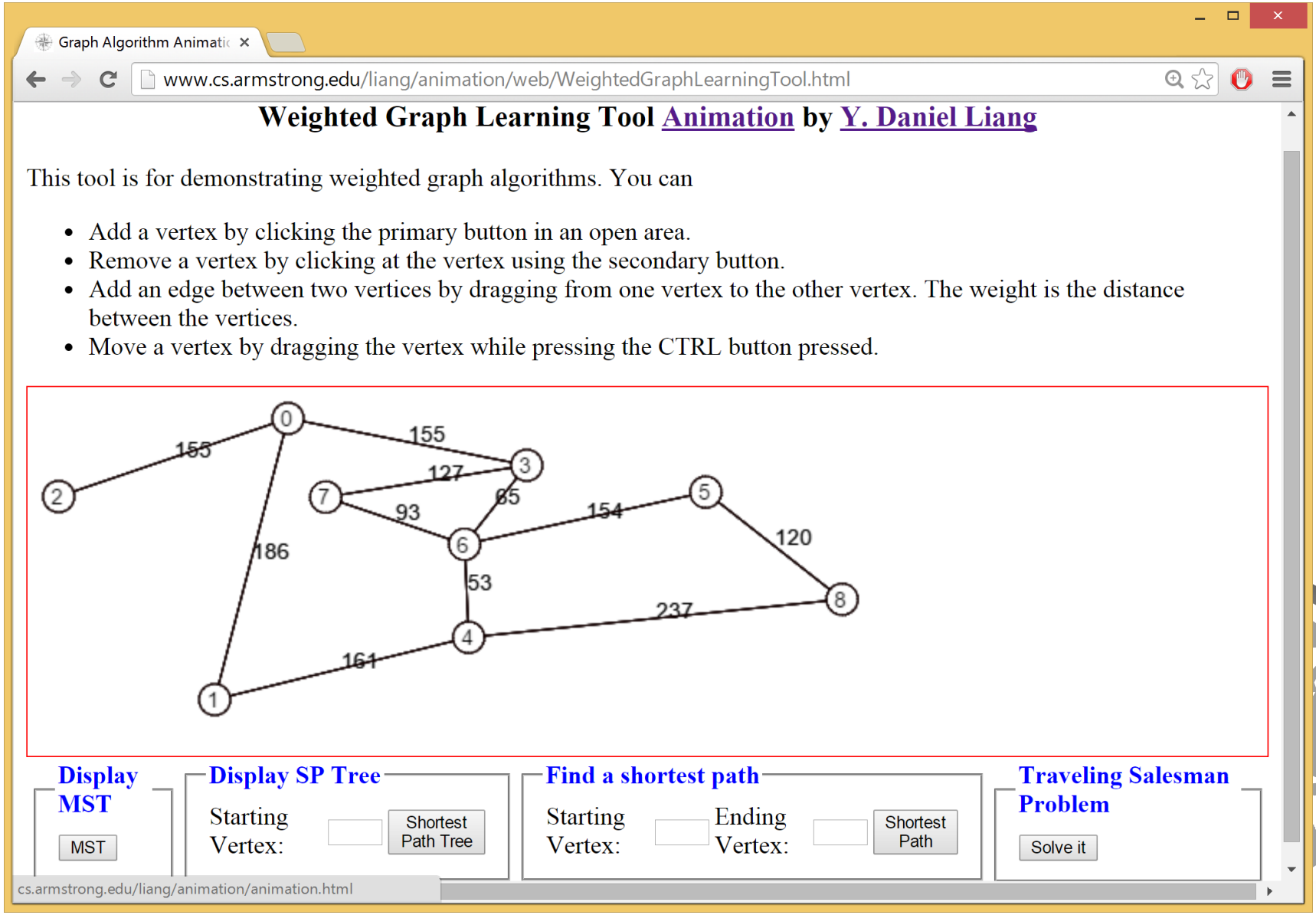
Chapter 29 Weighted Graphs and Applications

Objectives

- To represent weighted edges using adjacency matrices and adjacency lists (§29.2).
- To model weighted graphs using the WeightedGraph class that extends the AbstractGraph class (§29.3).
- To design and implement the algorithm for finding a minimum spanning tree (§29.4).
- To define the MST class that extends the Tree class (§29.4).
- To design and implement the algorithm for finding single-source shortest paths (§29.5).
- To define the ShortestPathTree class that extends the Tree class (§29.5).
- To solve the weighted nine tail problem using the shortest-path algorithm (§29.6).

Weighted Graph Animation

https://liveexample.pearsoncmg.com/dsanimation/WeightedGraphLearningTooleBook.html



Representing Weighted Graphs

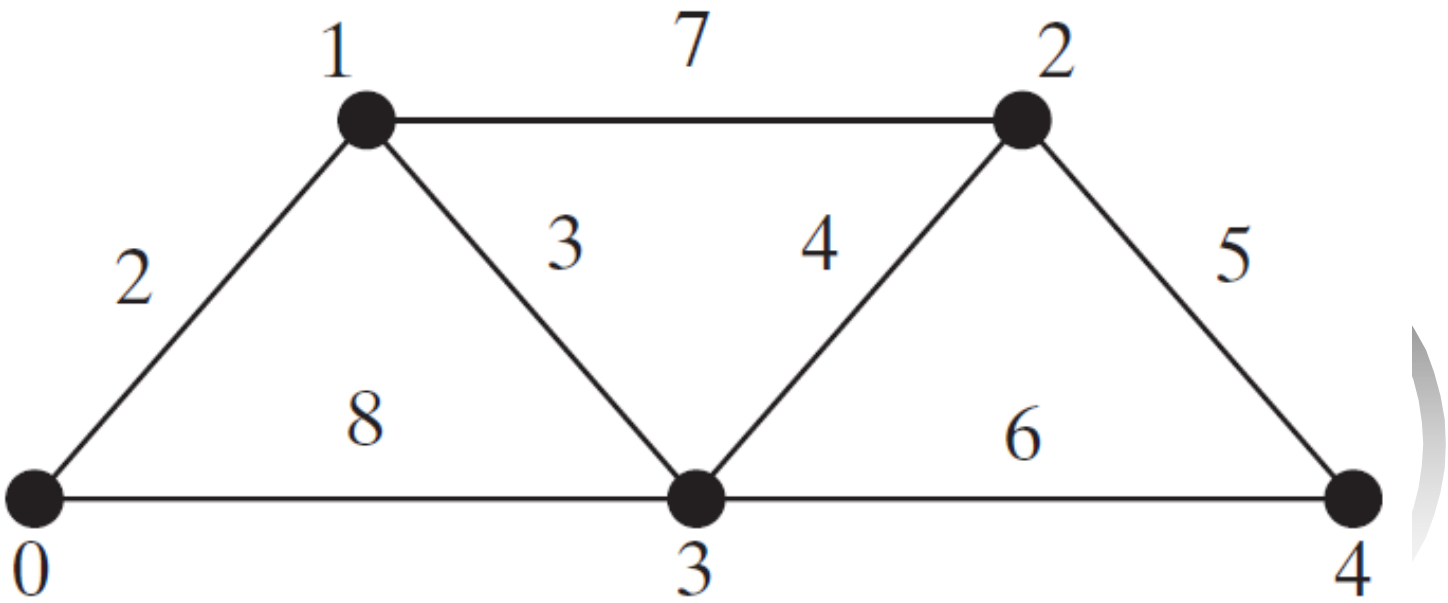
Representing Weighted Edges: Edge Array

Weighted Adjacency Matrices

Adjacency Lists

Representing Weighted Edges: Edge Array

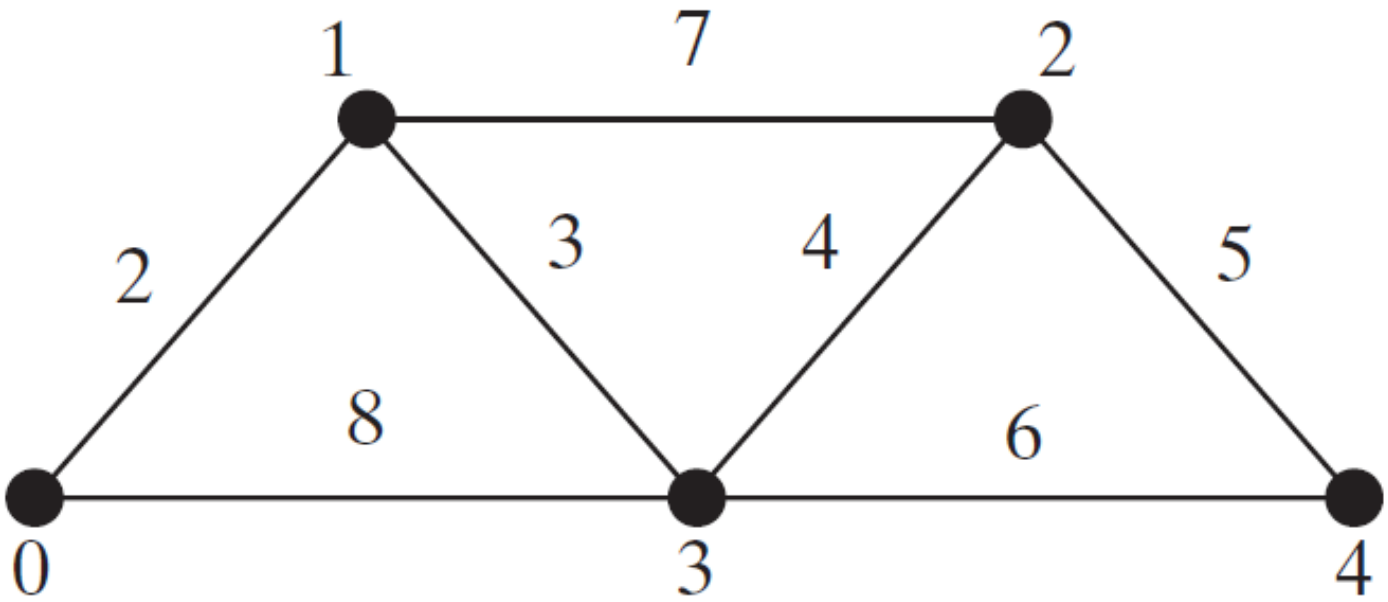
```
int[][] edges = {{0, 1, 2}, {0, 3, 8},
  {1, 0, 2}, {1, 2, 7}, {1, 3, 3},
  {2, 1, 7}, {2, 3, 4}, {2, 4, 5},
  {3, 0, 8}, {3, 1, 3}, {3, 2, 4}, {3, 4, 6},
  {4, 2, 5}, {4, 3, 6}
};
```



Representing Weighted Edges: Edge Array

```
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

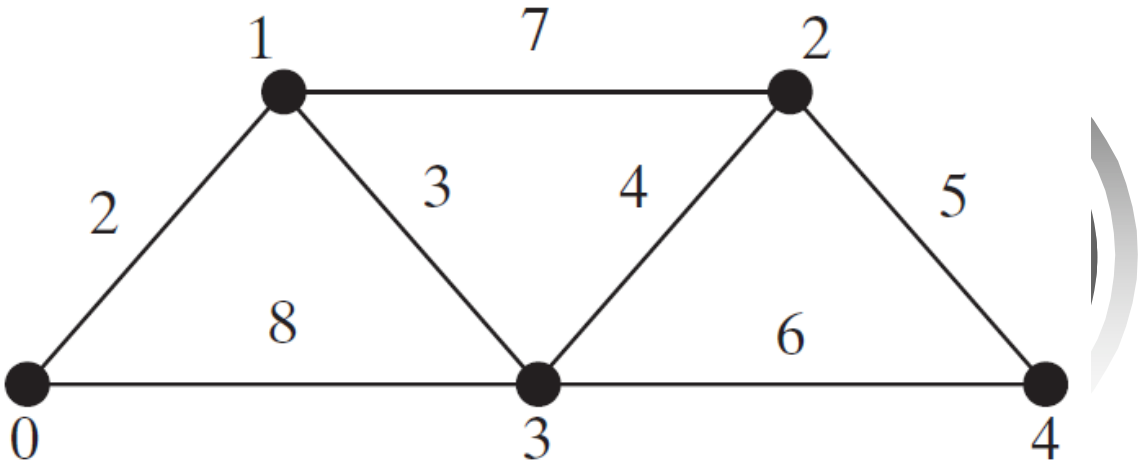


Edge Adjacency Lists

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

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

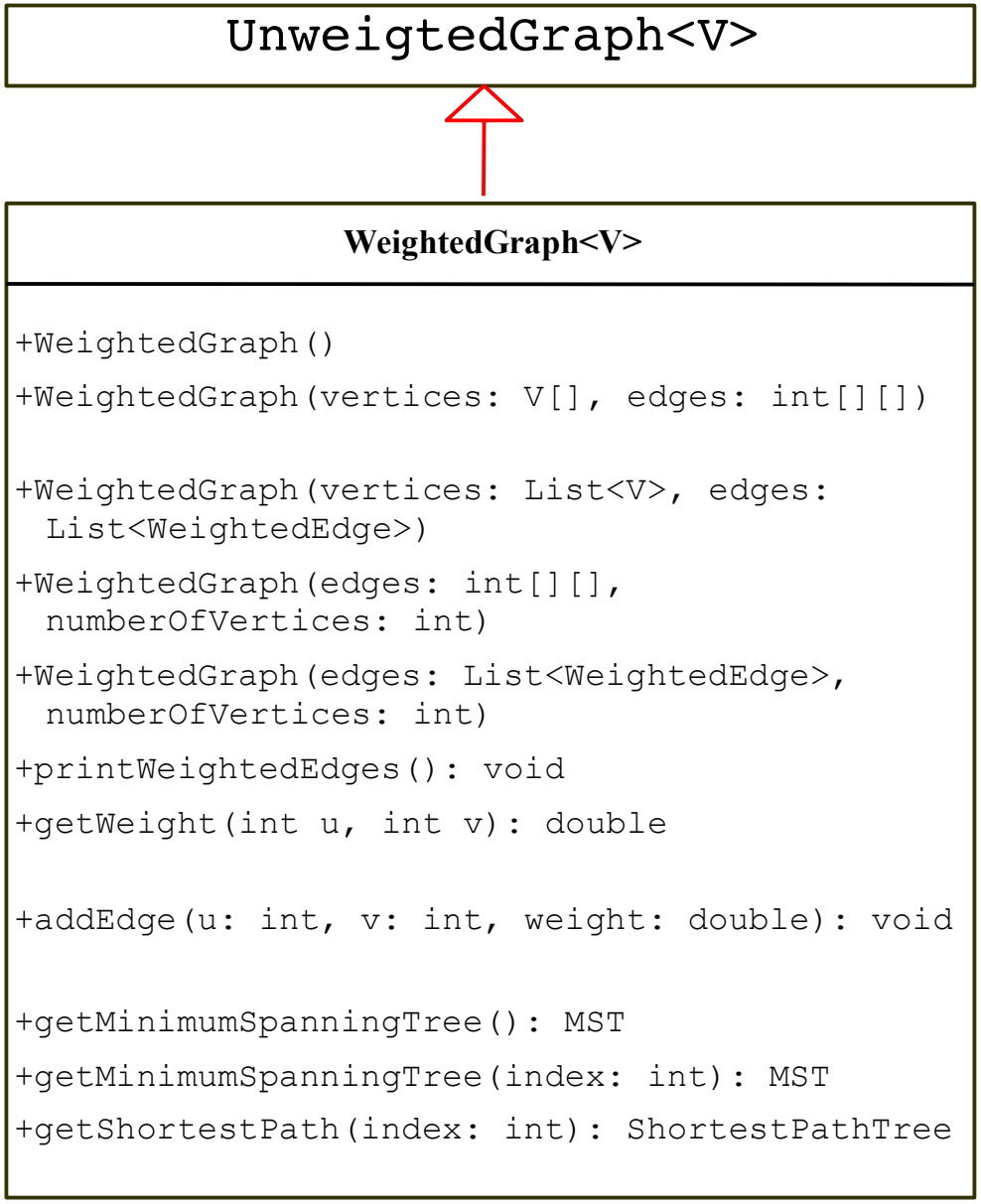
WeightedEdge



Edge Adjacency Lists

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

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



Defined in Figure 28.9.

Constructs an empty graph.

Constructs a weighted graph with the specified edges and the number of vertices in arrays.

Constructs a weighted graph with the specified edges and the number of vertices.

Constructs a weighted graph with the specified edges in an array and the number of vertices.

Constructs a weighted graph with the specified edges in a list and the number of vertices.

Displays all edges and weights.

Returns the weight on the edge from u to v. Throw an exception if the edge does not exist.

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.

Returns a minimum spanning tree starting from vertex 0.

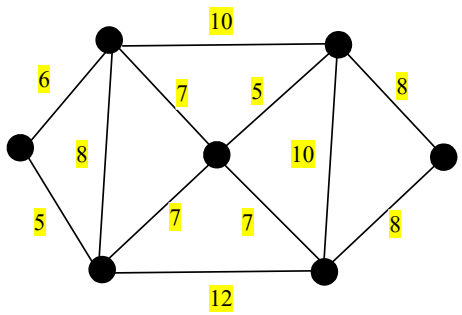
Returns a minimum spanning tree starting from vertex v.

Returns all single-source shortest paths.

WeightedGraph TestWeightedGraph Run

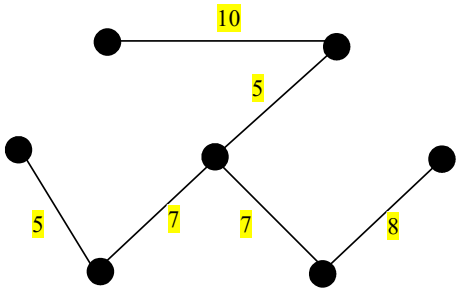
Minimum Spanning Trees

A graph may have many spanning trees. Suppose that the edges are weighted. A minimum spanning tree is a spanning tree with the minimum total weights. For example, the trees in Figures (b), (c), (d) are spanning trees for the graph in Figure (a). The trees in Figures (c) and (d) are minimum spanning trees.



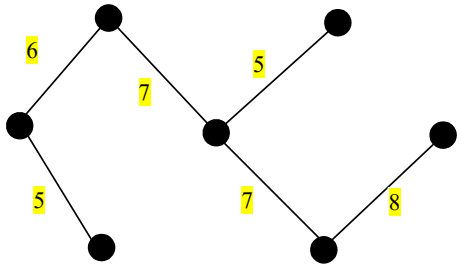
(a)

Total w: 42



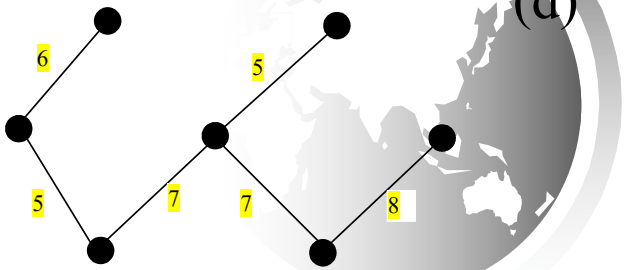
(b)

Total w: 38



(c)

Total w: 38

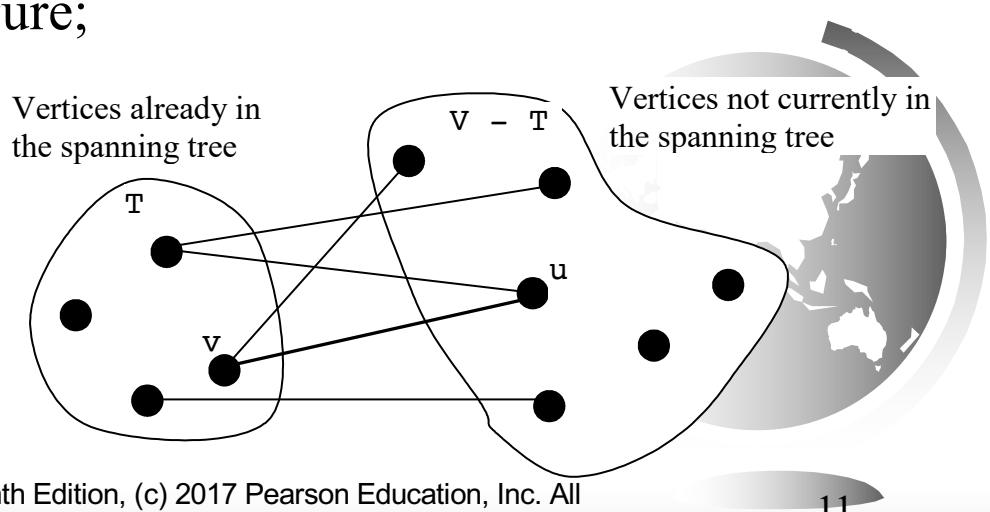


(d)

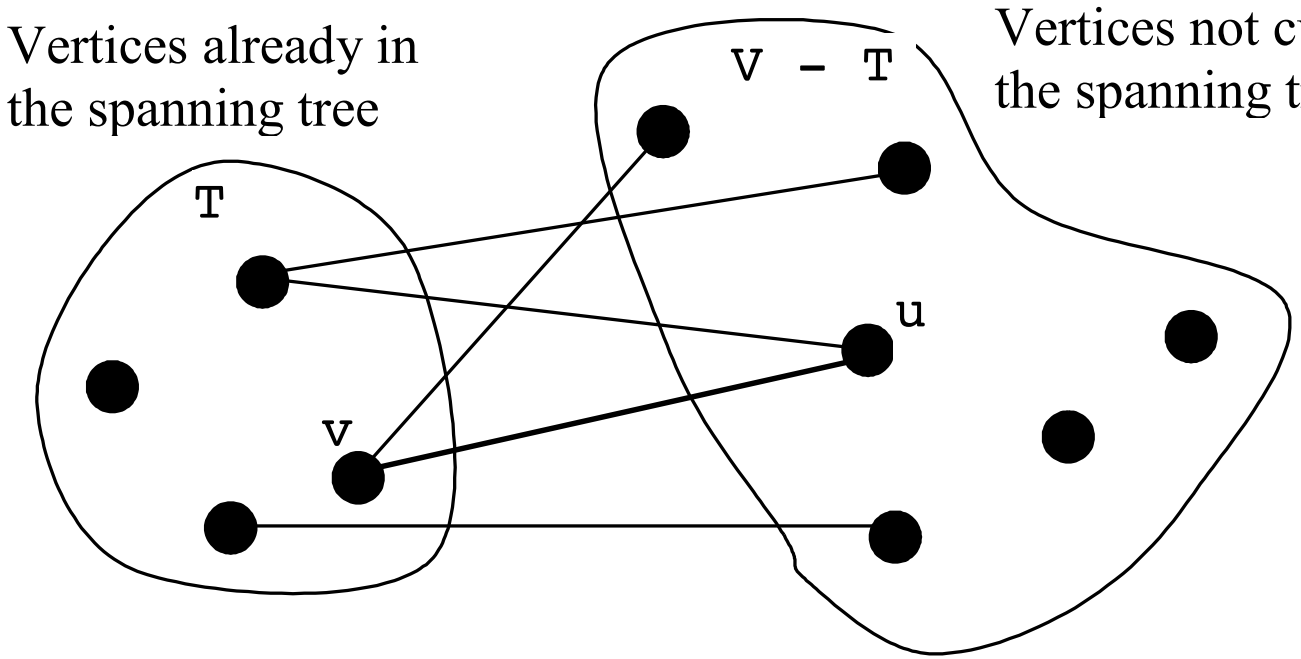
Prim's Minimum Spanning Tree Algorithm

Input: G = (V, E) with non-negative weights.
Output: a MST
MST 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 v in T and u in V – T with the smallest weight
      on the edge (u, v), as shown in the figure;
    add u to T;
  }
}

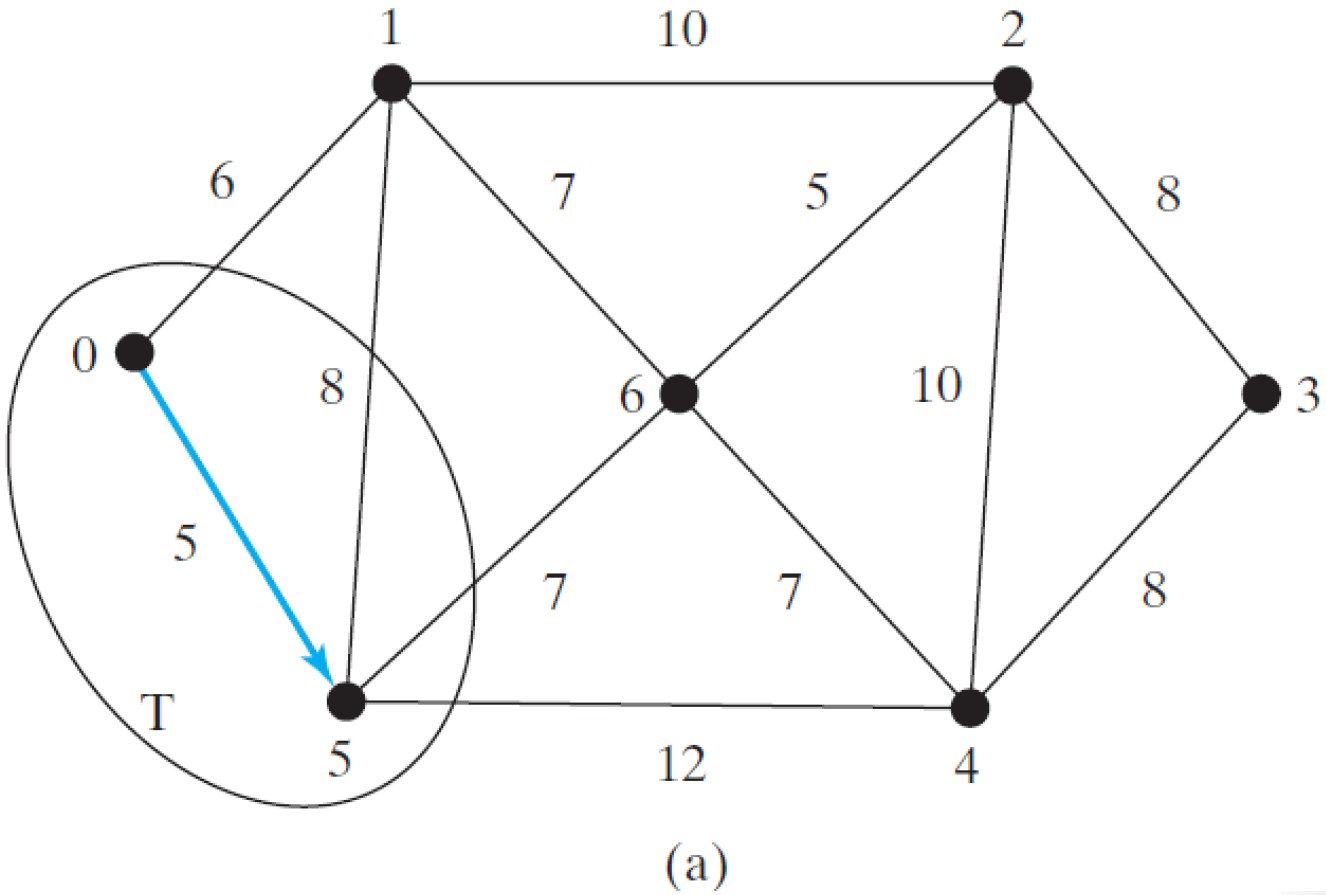
Find a vertex in V-T that has the smallest weighted edge connecting to a vertex in T.



Minimum Spanning Tree Algorithm

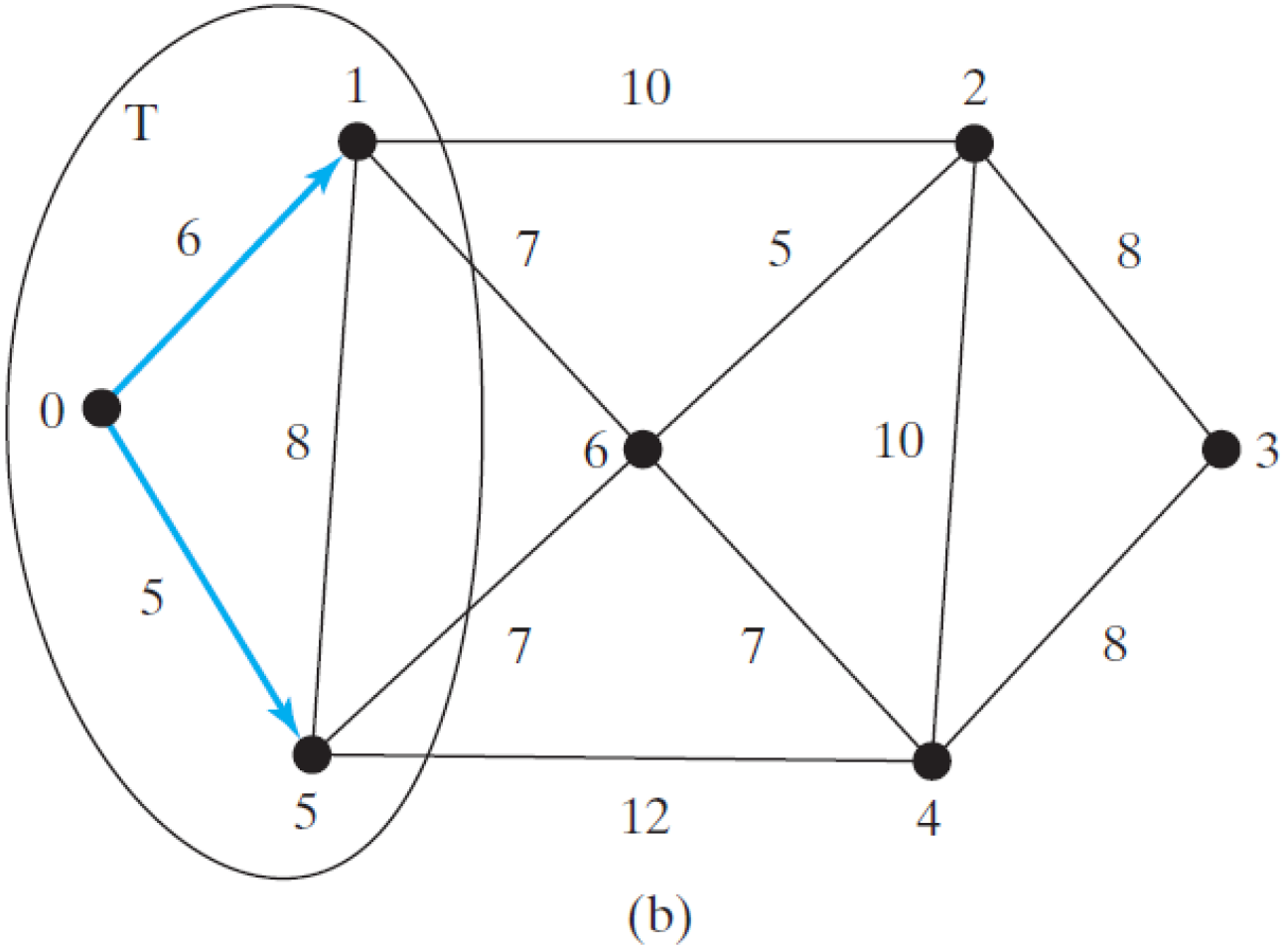


Minimum Spanning Tree Algorithm Example



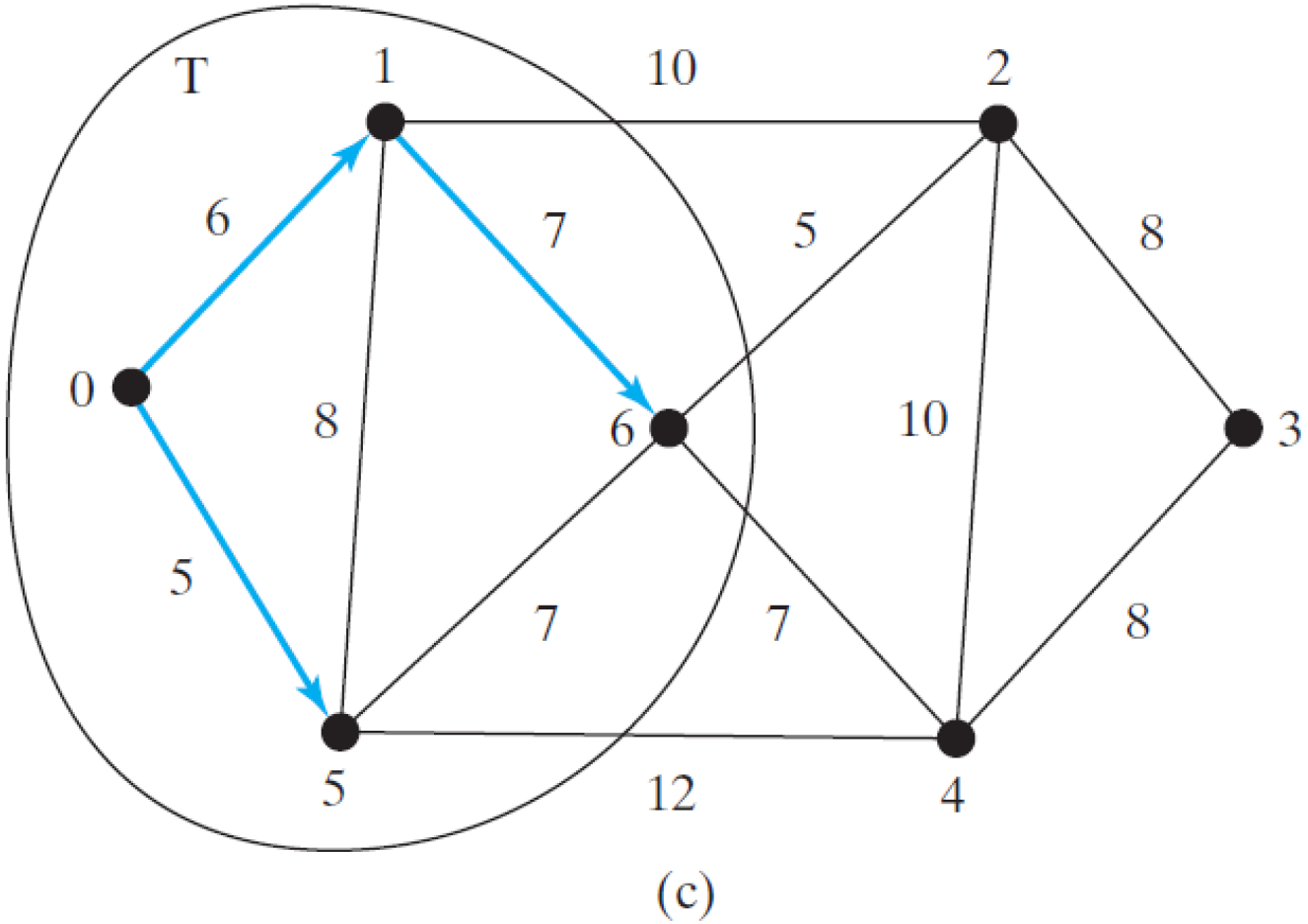
(a)

Minimum Spanning Tree Algorithm Example



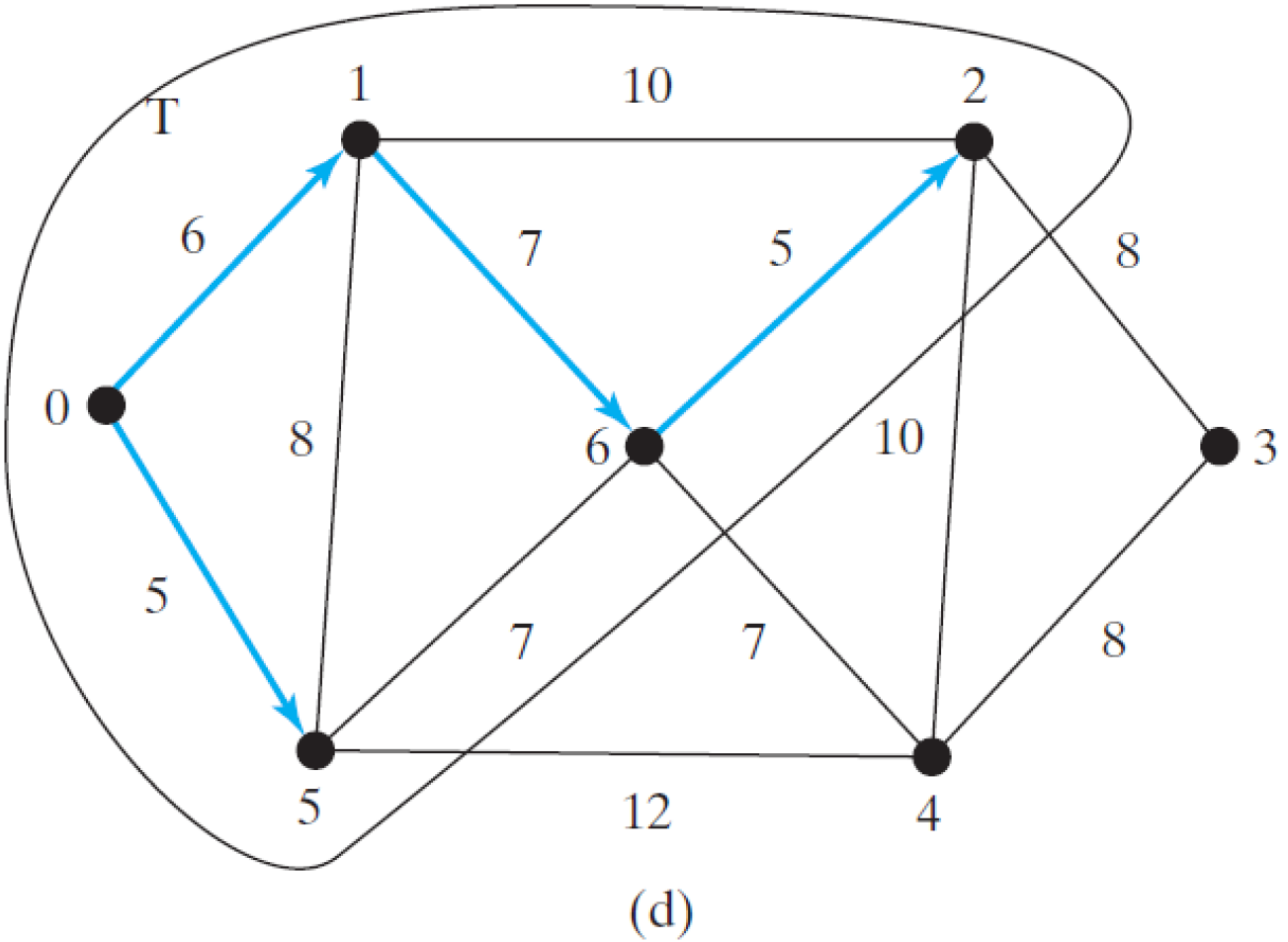
(b)

Minimum Spanning Tree Algorithm Example

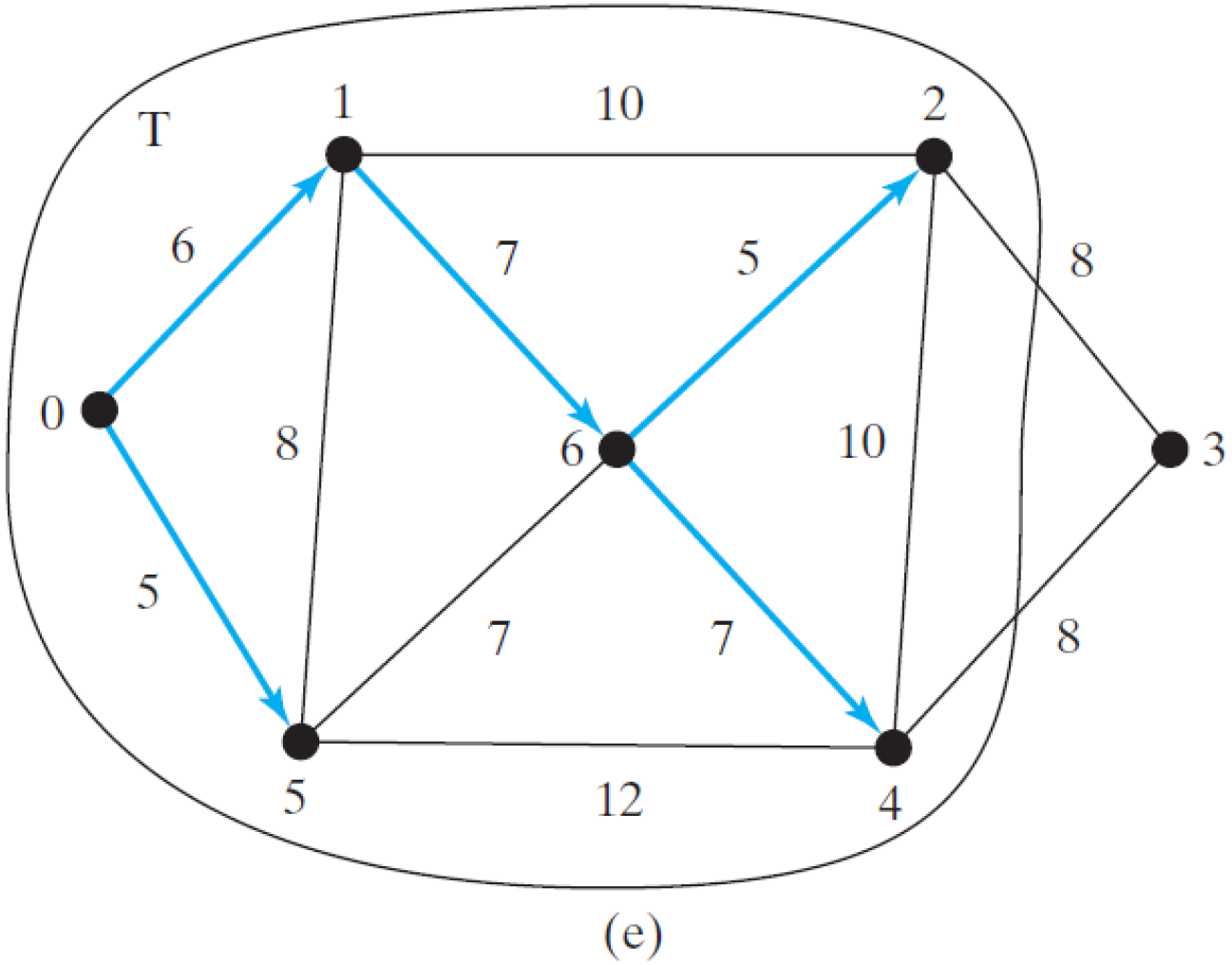


(c)

Minimum Spanning Tree Algorithm Example

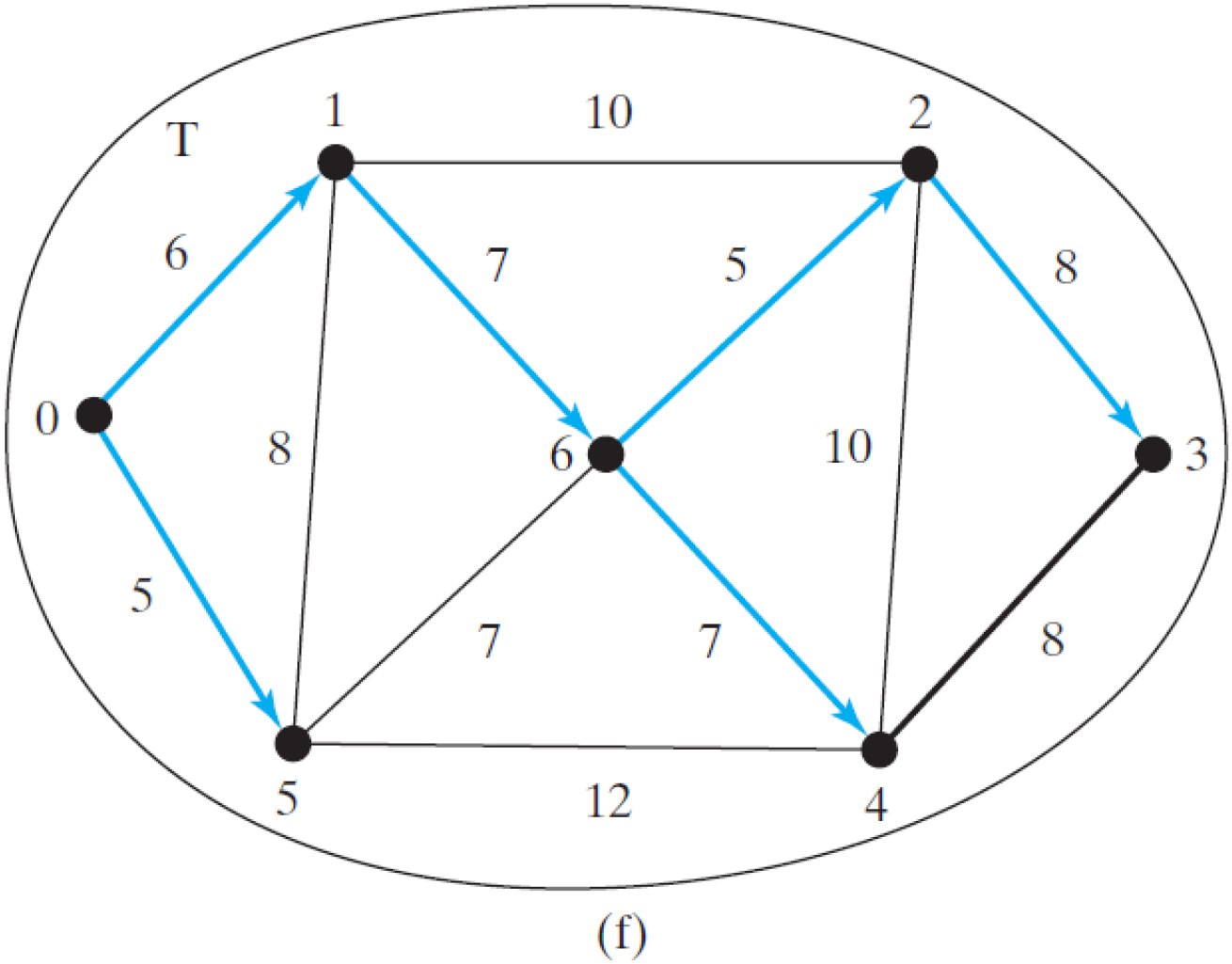


Minimum Spanning Tree Algorithm Example



(e)

Minimum Spanning Tree Algorithm Example



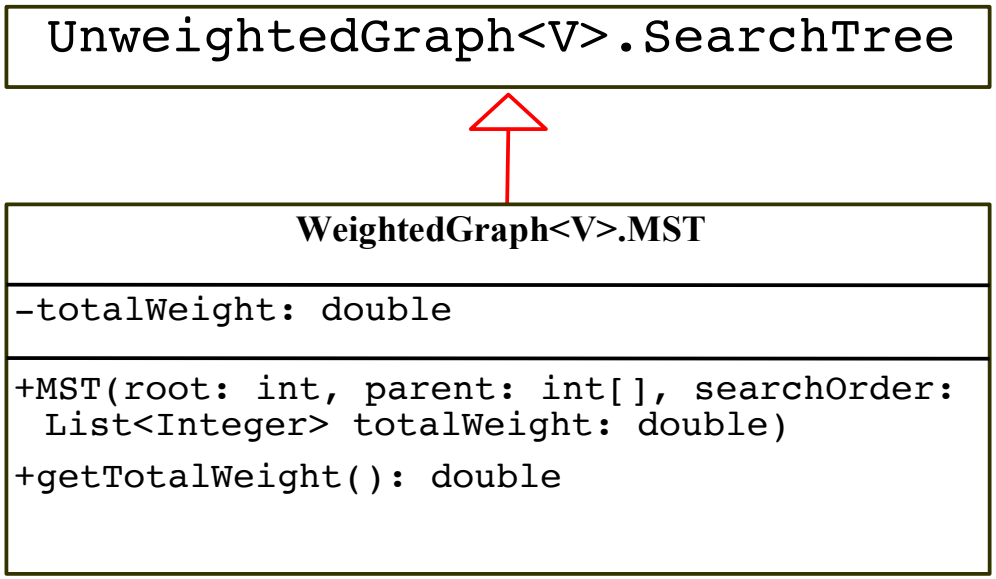
(f)

Refined Version of Prim's Minimum Spanning Tree Algorithm

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

```
 1  MST getMinimumSpanngingTree(s) {
 2    Let T be a set that contains the vertices in the spanning tree;
 3    Initially T is empty;
 4    Set cost[s] = 0; and cost[v] = infinity for all other vertices in V;
 5
 6    while (size of T < n) {
 7      Find u not in T with the smallest cost[u];
 8      Add u to T;
 9      for (each v not in T and (u, v) in E)
10        if (cost[v] > w(u, v)) {
11          cost[v] = w(u, v); parent[v] = u;
12        }
13    }
```

Implementing MST Algorithm



Total weight of the tree.

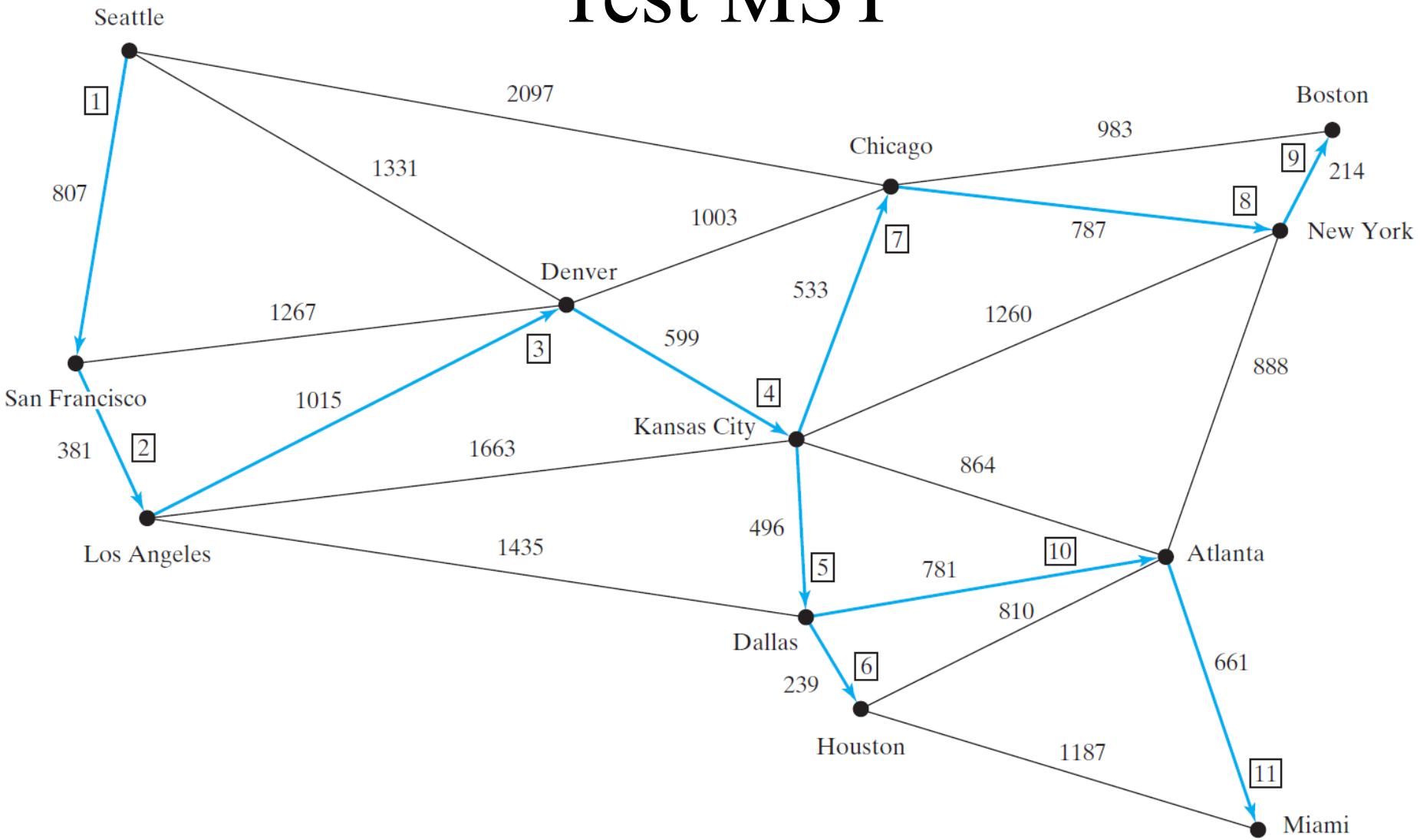
Constructs an MST with the specified root, parent array, `searchOrder`, and total weight for the tree.

Returns the `totalWeight` of the tree.

Time Complexity

Note that testing whether a vertex **i** is in **T** by invoking **T.conatins(i)** takes **O(n)** time, since **T** is a list. Therefore, the overall time complexity for this implemention is $O(n^3)$. Interested readers may see Programming Exercise 29.20 for improving the implementation and reduce the complexity to $O(n^2)$.

Test MST



TestMinimumSpanningTree Run

Shortest Path

§29.1 introduced the problem of finding the shortest distance between two cities for the graph in Figure 29.1. The answer to this problem is to find a shortest path between two vertices in the graph.

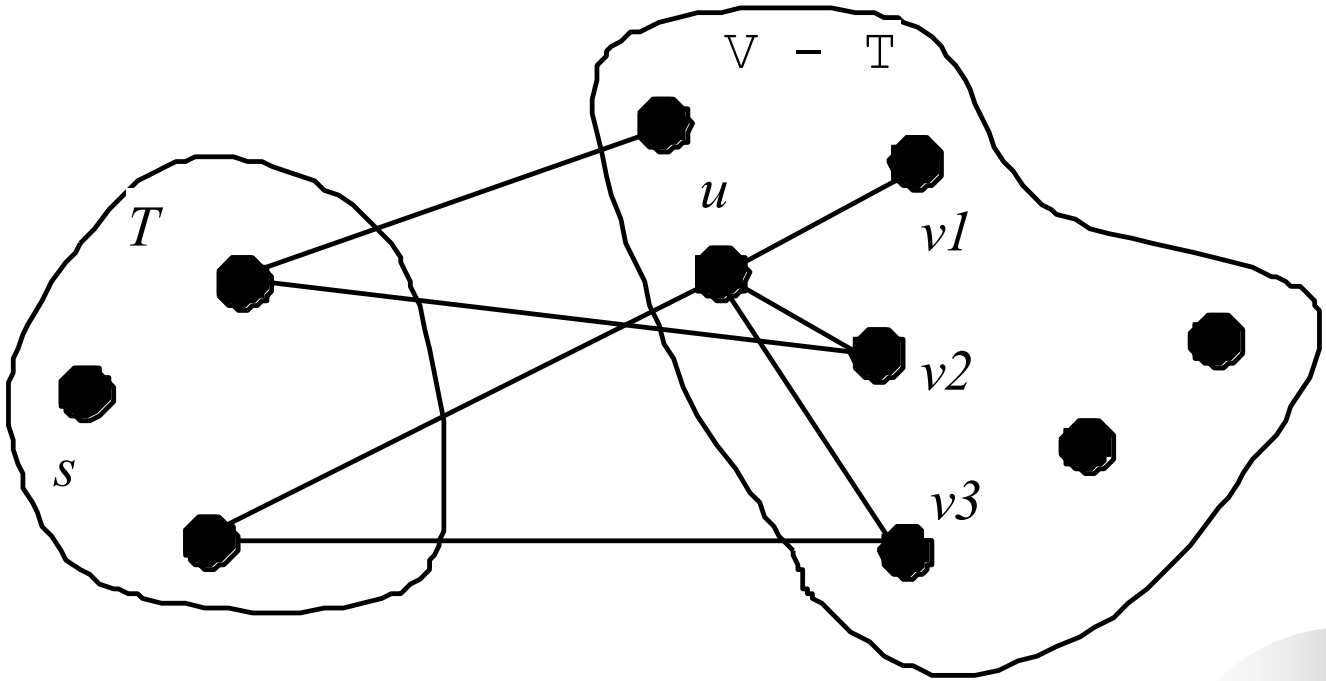
Single Source Shortest Path Algorithm

Input: a graph G = (V, E) with non-negative weights

Output: a shortest path tree with the source vertex s as the root

```
 1  ShortestPathTree getShortestPath(s) {
 2    Let T be a set that contains the vertices whose
 3      paths to s are known; Initially T is empty;
 4    Set cost[s] = 0; and cost[v] = infinity for all other vertices in V;
 5
 6    while (size of T < n) {
 7      Find u not in T with the smallest cost[u];
 8      Add u to T;
 9      for (each v not in T and (u, v) in E)
10        if (cost[v] > cost[u] + w(u, v)) {
11          cost[v] = cost[u] + w(u, v); parent[v] = u;
12        }
13    }
14  }
```

Single Source Shortest Path Algorithm

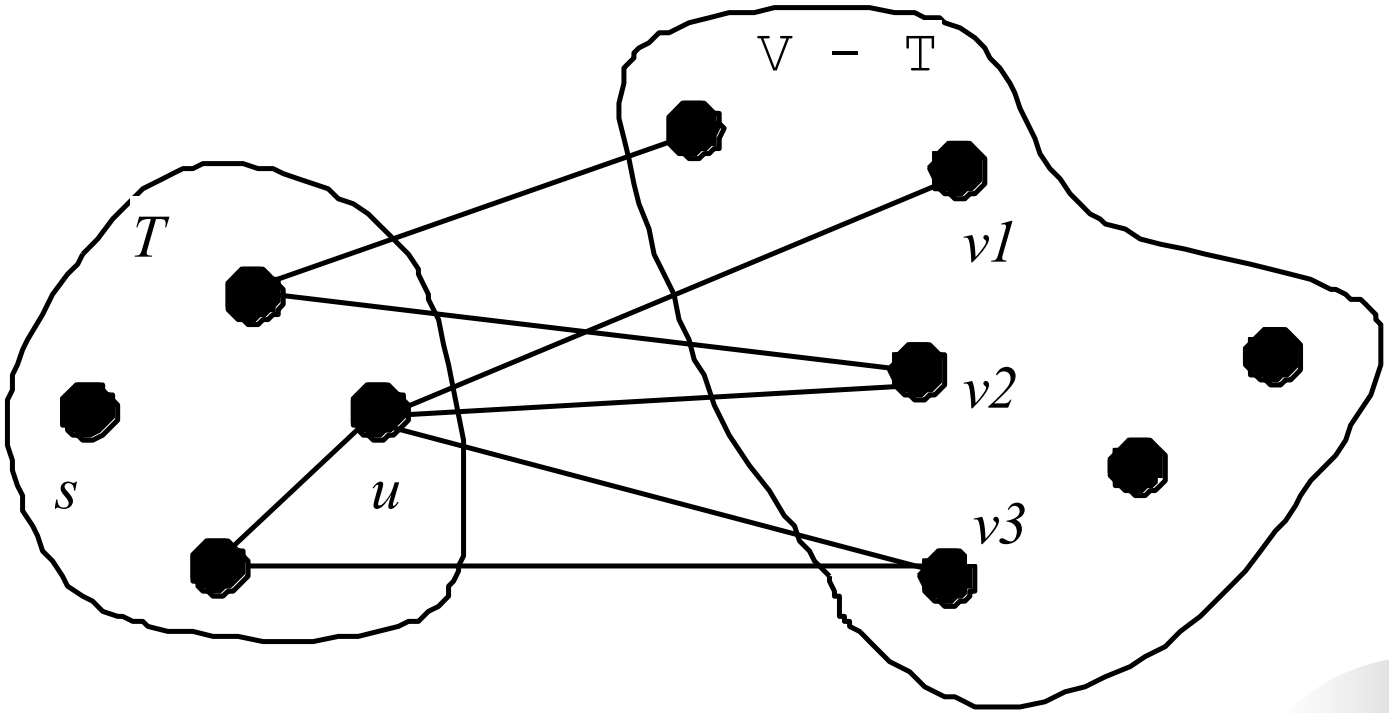


T contains vertices whose shortest path to s are known

V-T contains vertices whose shortest path to s are not known yet

Before moving u to T

Single Source Shortest Path Algorithm

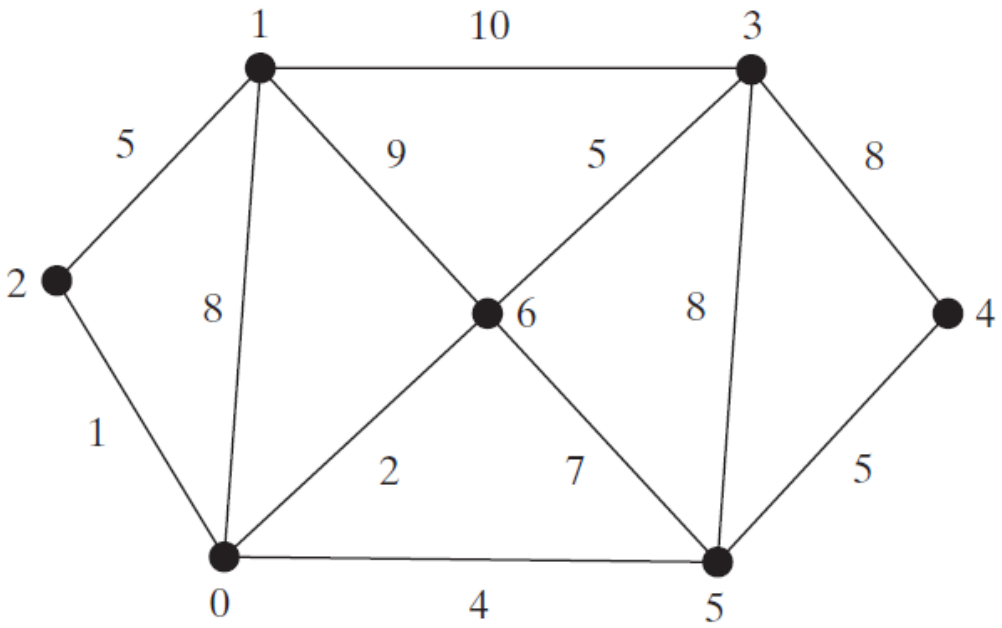


T contains vertices whose shortest path to s are known

V-T contains vertices whose shortest path to s are not known yet

After moving u to T

SP Algorithm Example (Step 0)



(a)

cost

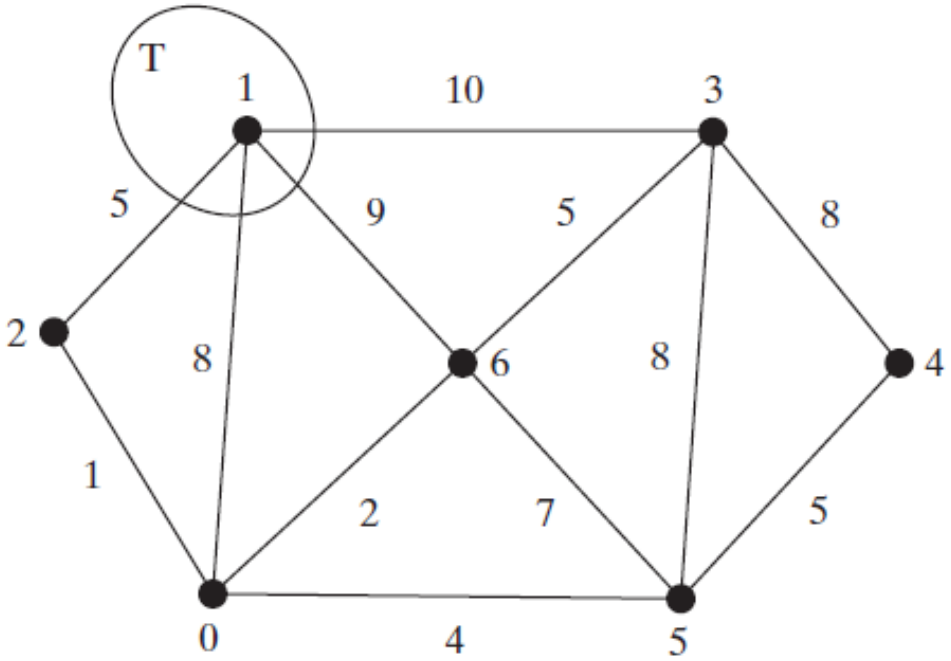
∞	0	∞	∞	∞	∞	∞
0	1	2	3	4	5	6

parent

	–1					
0	1	2	3	4	5	6

(b)

SP Algorithm Example (Step 1)



(a)

cost

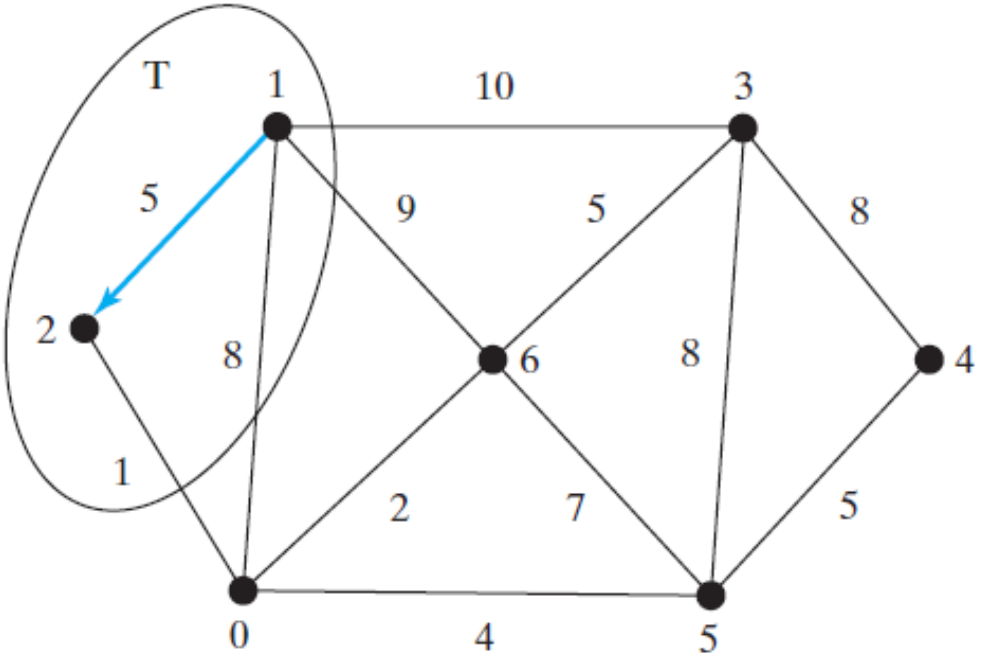
8	0	5	10	∞	∞	9
0	1	2	3	4	5	6

parent

1	–1	1	1			1
0	1	2	3	4	5	6

(b)

SP Algorithm Example (Step 2)



(a)

cost

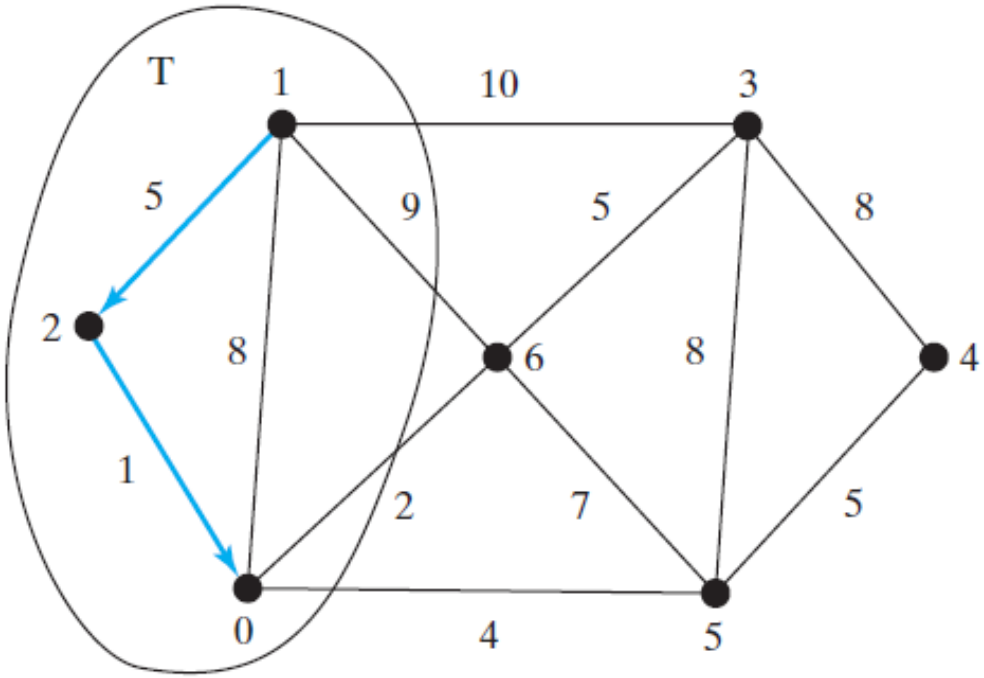
6	0	5	10	∞	∞	9
0	1	2	3	4	5	6

parent

2	–1	1	1			1
0	1	2	3	4	5	6

(b)

SP Algorithm Example (Step 3)



(a)

cost

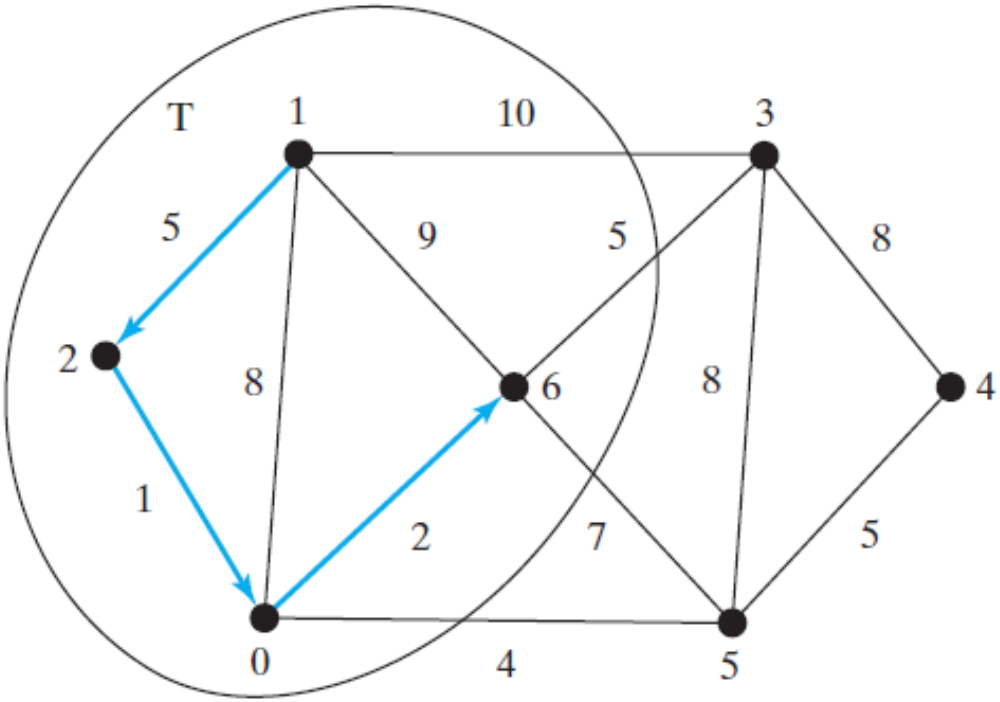
6	0	5	10	∞	10	8
0	1	2	3	4	5	6

parent

2	–1	1	1		0	0
0	1	2	3	4	5	6

(b)

SP Algorithm Example (Step 4)



(a)

cost

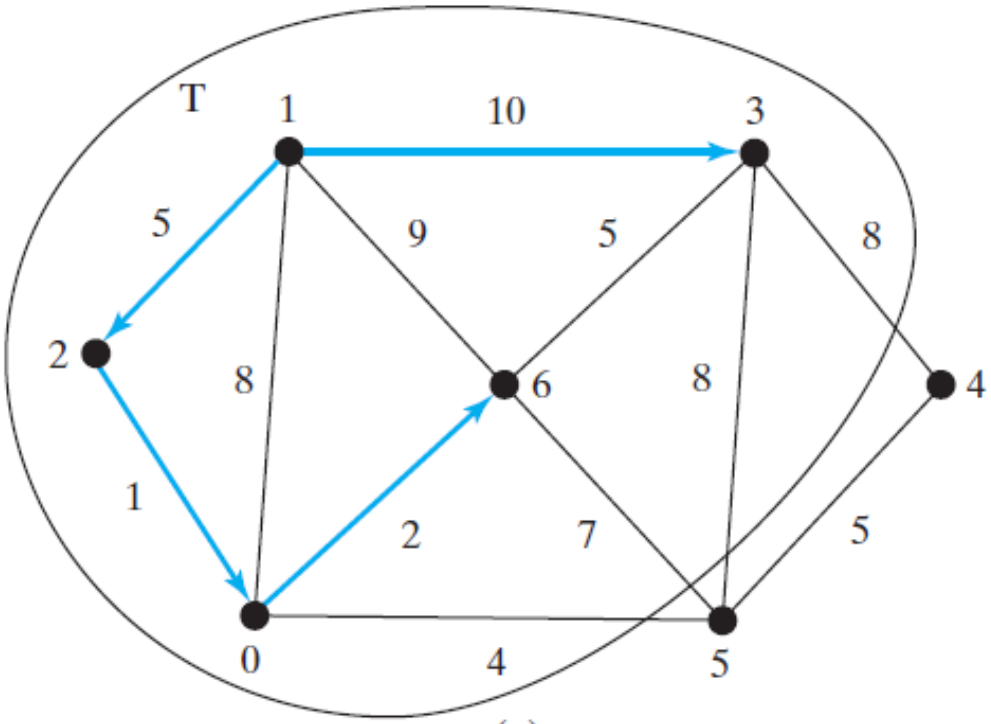
6	0	5	10	∞	10	8
0	1	2	3	4	5	6

parent

2	–1	1	1		0	0
0	1	2	3	4	5	6

(b)

SP Algorithm Example (Step 5)



(a)

cost

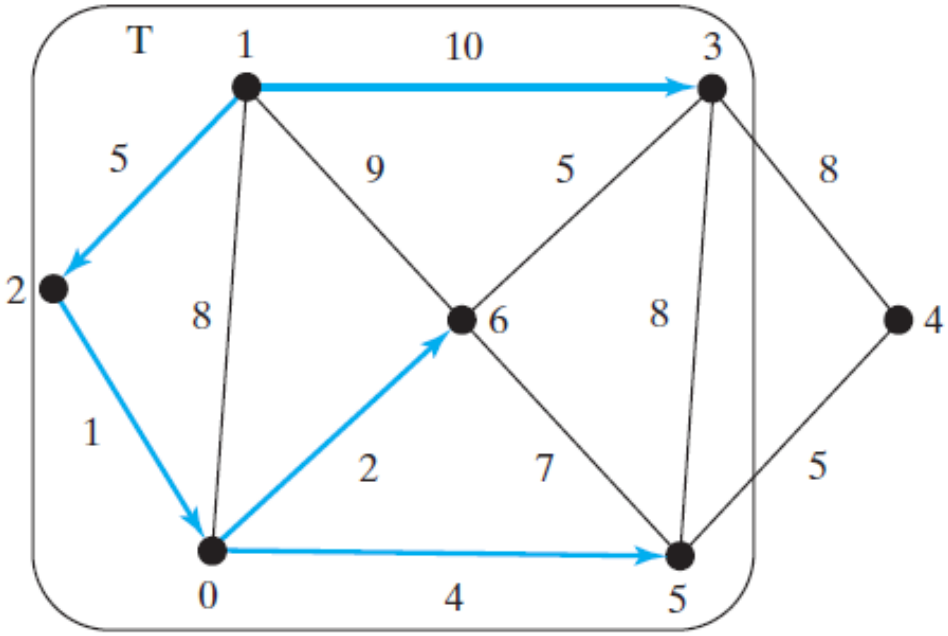
6	0	5	10	18	10	8
0	1	2	3	4	5	6

parent

2	–1	1	1	3	0	0
0	1	2	3	4	5	6

(b)

SP Algorithm Example (Step 6)



(a)

cost

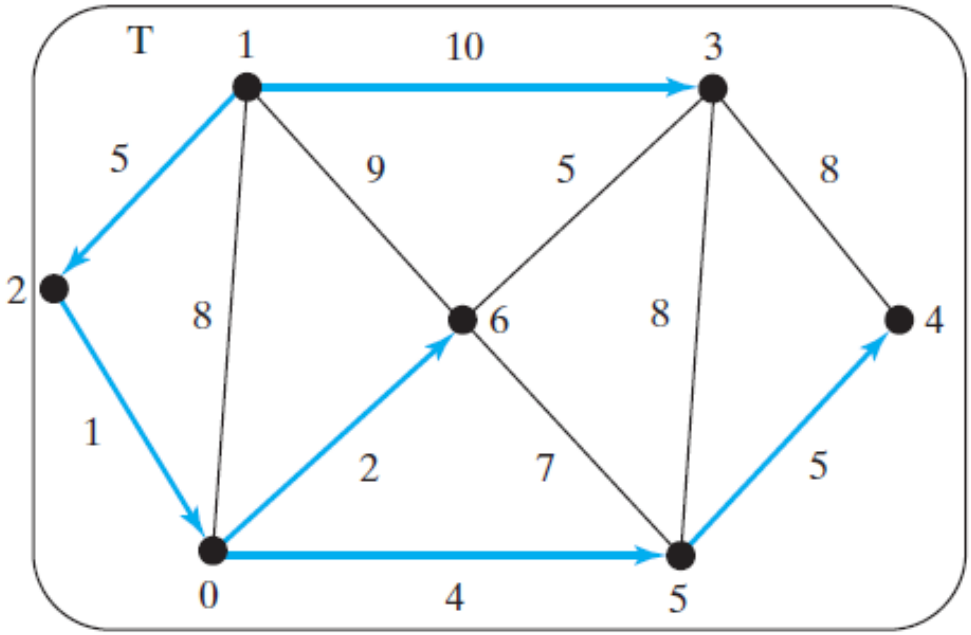
6	0	5	10	15	10	8
0	1	2	3	4	5	6

parent

2	–1	1	1	5	0	0
0	1	2	3	4	5	6

(b)

SP Algorithm Example (Step 7)



(a)

cost

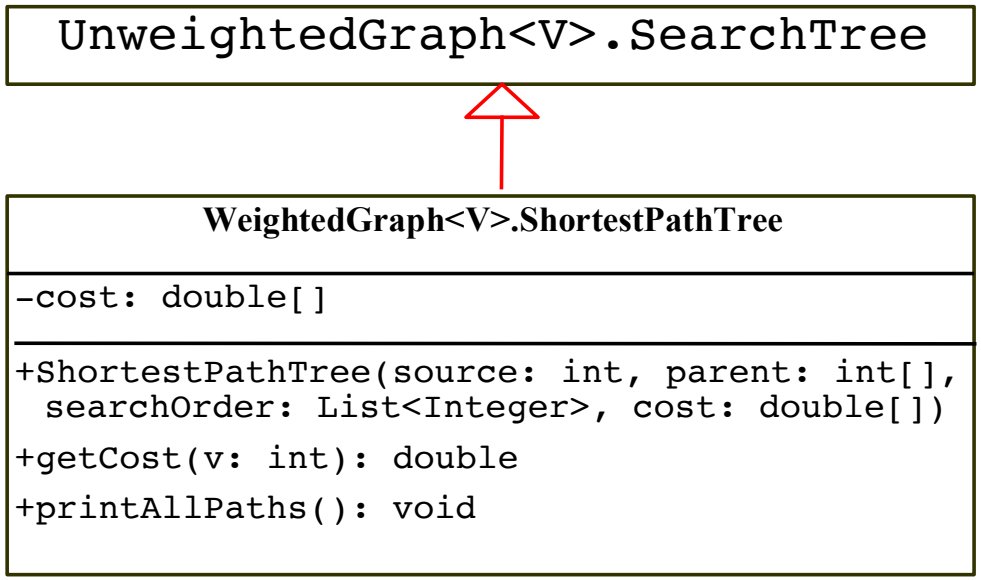
6	0	5	10	15	10	8
0	1	2	3	4	5	6

parent

2	–1	1	1	5	0	0
0	1	2	3	4	5	6

(b)

SP Algorithm Implementation



`cost[v]` stores the cost for the path from the source to `v`.

Constructs a shortest path tree with the specified `source`, `parent` array, `searchOrder`, and `cost` array.

Returns the cost for the path from the source to vertex `v`.

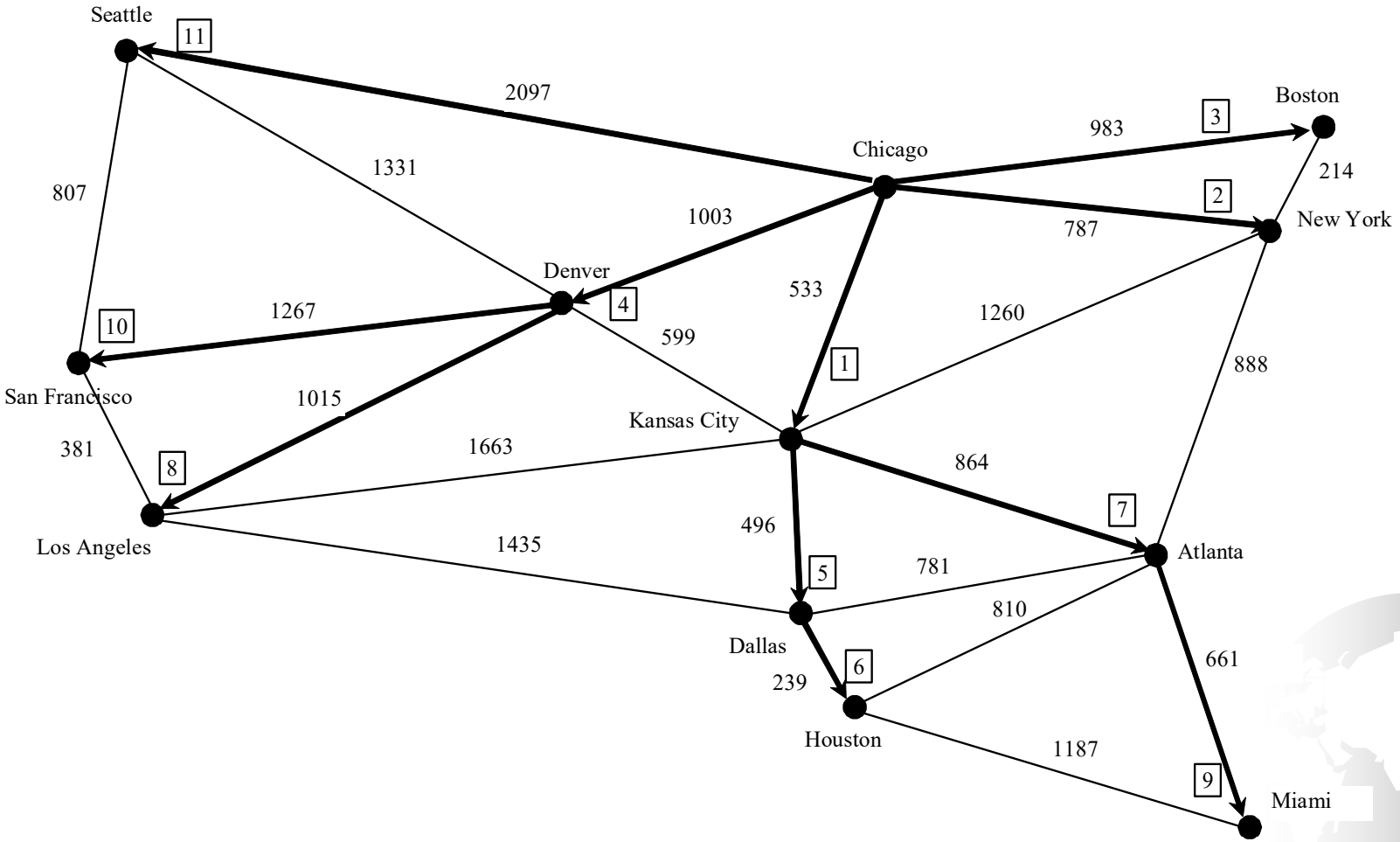
Displays all paths from the source.

TestShortestPath Run

Time Complexity

Same as the Prim's MST, O(n^3) and can be reduced to O(n^2).

SP Algorithm Example



The Weighted Nine Tail Problem

The nine tail problem is to find the minimum number of the moves that lead to all coins face down. Each move flips a head coin and its neighbors. The weighted nine tail problem assigns the number of the flips as a weight on each move. For example, you can move from the coins in (a) to (b) by flipping the three coins. So the weight for this move is 3.

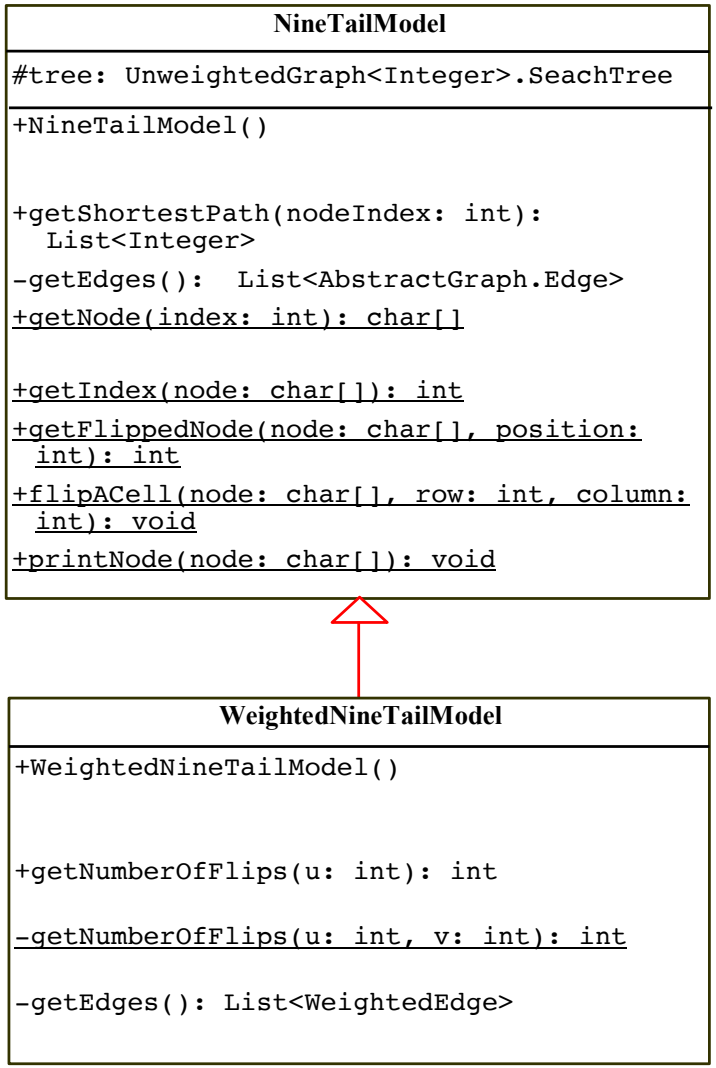
(a)

H	H	H
T	T	T
H	H	H

(b)

T	T	H
H	T	T
H	H	H

WeightedNineTailModel



A tree rooted at node 511.

Constructs a model for the nine tails problem and obtains the tree.

Returns a path from the specified node to the root. The path returned consists of the node labels in a list.

Returns a list of `Edge` objects for the graph.

Returns a node consisting of nine characters of Hs and Ts.

Returns the index of the specified node.

Flips the node at the specified position and returns the index of the flipped node.

Flips the node at the specified row and column.

Displays the node on the console.

Constructs a model for the weighted nine tails problem and obtains a `ShortestPathTree` rooted from the target node.

Returns the number of flips from node u to the target node 511.

Returns the number of different cells between the two nodes.

Gets the weighted edges for the weighted nine tails problem.

NineTailModel | WeightedNineTailModel | WeightedNineTail | Run