**Home Work 5:**

1. Write a program to perform a topological sort using a queue.

a. Use Figure 9.81(p.417) in the text book as input.

Code for the problem:

java.util.\*;

public class TopologicalSort

{

public HashMap<String, Vertex> map = null;

public TopologicalSort()

{

map = new HashMap<String, Vertex>();

}

public void addVertex(String source, String destination)

{

Vertex sourceVertex = null; // checking for source vertex

if (!map.containsKey(source))

{

sourceVertex = new Vertex(source);

map.put(source, sourceVertex);

}

else

{

sourceVertex = map.get(source);

}

Vertex destinationVertex = null; // checking for destination vertex

if (!map.containsKey(destination))

{

destinationVertex = new Vertex(destination);

map.put(destination, destinationVertex);

}

else

{

destinationVertex = map.get(destination);

}

sourceVertex.AdjacentVertices.add(destinationVertex);

}

/\*\*

\* Sorts our map in a topological fashion

\* @return the sorted array

\* @throws Exception

\*/

public void topSort() throws Exception

{

Queue<Vertex> zeroQueue = new LinkedList<Vertex>();

HashMap<String, String> sortingTableInfo = new HashMap<String, String>();

// traversing through every vertex

for (Vertex v : map.values())

{

if(!map.containsKey(v.data));

sortingTableInfo.put(v.data, "");

// checking if the adjacent vertex has anything inside it

if(v.AdjacentVertices.isEmpty())

continue;

// traversing through adjacent vertex

for (Vertex x : v.AdjacentVertices)

{

x.indegree++; // increment the indegree for each adjacent vertex of v

}

}

// created the last two columns as per required table for problem

sortingTableInfo.put("Enqeue", "");

sortingTableInfo.put("Deqeue", "");

sortingTableInfo.put("Vertex", "");

// collecting all verticies of indegree 0

String tableinfo = "";

for (Vertex v : map.values()) // traversing through the map values

{

if (v.indegree == 0) // checking the indegree value to see if current vertex has any input coming to it

{

zeroQueue.add(v);

//

if(tableinfo.length() == 0)

tableinfo = v.data;

else

tableinfo = tableinfo + "," + v.data;

}

}

sortingTableInfo.put("Enqeue", tableinfo + "\t");

if(zeroQueue.isEmpty())

throw new Exception("This is a cyclic graph. Cannot perform topological sort");

int counter = 0;

while (!zeroQueue.isEmpty())

{

sortingTableInfo.put("Vertex", sortingTableInfo.get("Vertex") + (counter+1) + "\t");

for (Vertex info : map.values()) // traversing through the map values

{

String tableInfo = sortingTableInfo.get(info.data) + info.indegree + "\t"; // e.g) v1 0 \t ---

sortingTableInfo.put(info.data, tableInfo);

}

Vertex v = zeroQueue.remove();

v.topNum = counter++;

sortingTableInfo.put("Deqeue", sortingTableInfo.get("Deqeue") + v.data + "\t");

// checking if the adjacent vertex has anything inside it

if(v.AdjacentVertices.isEmpty())

continue;

String tableinfo2 = "";

// traversing through adjacent vertex

for (Vertex w : v.AdjacentVertices)

{

w.indegree--; // decrement the indegree for each adjacent vertex of v

if (w.indegree == 0)

{

zeroQueue.add(w);

//

if(tableinfo2.length() == 0)

tableinfo2 = w.data;

else

tableinfo2 = tableinfo2 + "," + w.data;

}

}

sortingTableInfo.put("Enqeue", sortingTableInfo.get("Enqeue") + tableinfo2 + "\t");

}

if(counter != map.keySet().size())

throw new Exception("This is a cyclic graph. Cannot perform topological sort");

// printing the

System.out.println("-------------------------------------------------------------------------------------------");

System.out.println(" \t\t\tINDEGREE BEFORE THE DEQEUE\t\t\t\t");

System.out.println("Vertex\t" + sortingTableInfo.get("Vertex"));

System.out.println("-------------------------------------------------------------------------------------------");

// printing the sorted table

for (Vertex info : map.values()) // traversing through the map values

{

String tableInfo = info.data + "\t" + sortingTableInfo.get(info.data);

System.out.println(tableInfo);

}

System.out.println("Enqeue\t" + sortingTableInfo.get("Enqeue"));

System.out.println("Deqeue\t" + sortingTableInfo.get("Deqeue"));

System.out.println("-------------------------------------------------------------------------------------------\n");

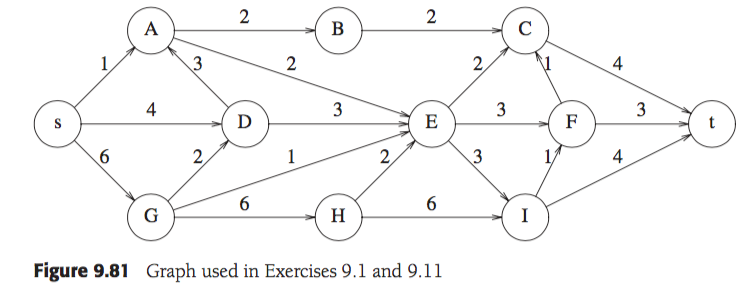
System.out.println("Number of vertices: " + map.keySet().size());

System.out.println("List in sorted order:");

System.out.println(sortingTableInfo.get("Deqeue"));

}

}



Test case based on Figure 9.81:

@Test

public void TopologicalSortTest() throws Exception

{

TopologicalSort ts = new TopologicalSort();

// creating the input graph

ts.addVertex("A", "B");

ts.addVertex("A", "E");

ts.addVertex("B", "C");

ts.addVertex("C", "t");

ts.addVertex("D", "A");

ts.addVertex("D", "E");

ts.addVertex("E", "C");

ts.addVertex("E", "F");

ts.addVertex("E", "I");

ts.addVertex("F", "C");

ts.addVertex("F", "t");

ts.addVertex("G", "D");

ts.addVertex("G", "E");

ts.addVertex("G", "H");

ts.addVertex("H", "E");

ts.addVertex("H", "I");

ts.addVertex("I", "F");

ts.addVertex("I", "t");

ts.addVertex("s", "A");

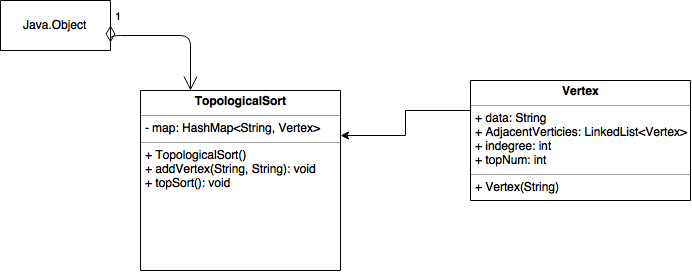
ts.addVertex("s", "D");

ts.addVertex("s", "G");

// print out the nodes and links

ts.topSort();

}

UML diagram for the problem

b. Print the sorting table, similar to Figure 9.6 (p.364).

Output for the code:

———————————————————————————————————————————————————

INDEGREE BEFORE THE DEQEUE

Vertex 1 2 3 4 5 6 7 8 9 10 11

———————————————————————————————————————————————————

A 2 1 1 0 0 0 0 0 0 0 0

B 1 1 1 1 1 0 0 0 0 0 0

C 3 3 3 3 3 3 2 1 1 0 0

s 0 0 0 0 0 0 0 0 0 0 0

t 3 3 3 3 3 3 3 3 2 1 0

D 2 1 0 0 0 0 0 0 0 0 0

E 4 4 3 2 1 0 0 0 0 0 0

F 2 2 2 2 2 2 2 1 0 0 0

G 1 0 0 0 0 0 0 0 0 0 0

H 1 1 0 0 0 0 0 0 0 0 0

I 2 2 2 2 1 1 1 0 0 0 0

Enqeue s G D,H A B,E I F C t

Deqeue s G D H A B E I F C t

———————————————————————————————————————————————————-

c. Print the nodes in sorted order, starting with vertex s.

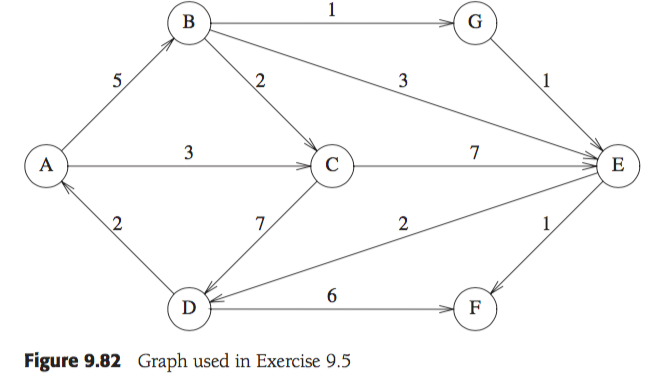
Number of vertices: 11

List in sorted order:

s G D H A B E I F C t

1. Write a program to find the unweighted shortest path from a given vertex to all other vertices.

a. Use Figure 9.82(p.418) in the text book as input.



Code for the problem:

import java.util.\*;

/\*\*

\*

\* @author Adish

\*

\*/

public class UnweightedShortestPath

{

public HashMap<String, Vertex> map = null;

public UnweightedShortestPath()

{

map = new HashMap<String, Vertex>();

}

public void addVertex(String source, String destination)

{

Vertex sourceVertex = null; // checking for source vertex

if (!map.containsKey(source))

{

sourceVertex = new Vertex(source);

map.put(source, sourceVertex);

}

else

{

sourceVertex = map.get(source);

}

Vertex destinationVertex = null; // checking for destination vertex

if (!map.containsKey(destination))

{

destinationVertex = new Vertex(destination);

map.put(destination, destinationVertex);

}

else

{

destinationVertex = map.get(destination);

}

sourceVertex.AdjacentVertices.add(destinationVertex);

}

public void Unweighted(String vertexName)

{

Queue<Vertex> q = new LinkedList<Vertex>();

for (Vertex v : map.values())

{

v.distance = Integer.MAX\_VALUE; // INFINITY

v.known = false;

}

Vertex s = map.get(vertexName);

s.distance = 0;

q.add(s);

PrintMap("Initial State", q);

while(!q.isEmpty())

{

Vertex v = q.remove();

for (Vertex w : v.AdjacentVertices)

{

if (w.distance == Integer.MAX\_VALUE)

{

w.distance = v.distance + 1;

w.previous = v;

q.add(w);

}

}

PrintMap(v.data + " Dequeued", q);

System.out.print("Path: ");

PrintPath(v);

System.out.println();

}

}

private void PrintPath(Vertex v)

{

if (v.previous != null)

{

PrintPath(v.previous);

System.out.print(" to ");

}

System.out.print(v.data);

}

private void PrintMap(String message, Queue<Vertex> q)

{

System.out.println("----------------------------");

System.out.println(" \t" + message);

System.out.println("\t---------------");

System.out.println(" V\tKnown\tDv\tPv");

System.out.println("----------------------------");

// printing the sorted table

for (Vertex info : map.values()) // traversing through the map values

{

String tableInfo = info.data + "\t" +

((info.known) ? "T" : "F" + "\t") +

((info.distance == Integer.MAX\_VALUE) ? "\*" : info.distance) + "\t" +

((info.previous != null) ? info.previous.data : "0");

System.out.println(tableInfo);

}

String tableinfo = "";

for (Vertex v : q) // traversing through the map values

{

if(tableinfo.length() == 0)

tableinfo = v.data;

else

tableinfo = tableinfo + "," + v.data;

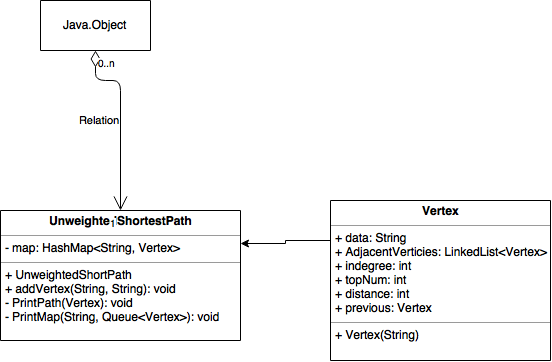
}

System.out.println("Q:\t\t" + tableinfo);

System.out.println("----------------------------");

}

}

UML diagram for problem:

b. Print the intermediate tables (such as Figure 9.19).

Output for the code

----------------------------

Initial State

---------------

V Known Dv Pv

----------------------------

A F 0 0

B F \* 0

C F \* 0

D F \* 0

E F \* 0

F F \* 0

G F \* 0

Q: A

----------------------------

----------------------------

A Dequeued

---------------

V Known Dv Pv

----------------------------

A T 0 0

B F 1 A

C F 1 A

D F \* 0

E F \* 0

F F \* 0

G F \* 0

Q: C,B

----------------------------

Path: A

----------------------------

C Dequeued

---------------

V Known Dv Pv

----------------------------

A T 0 0

B F 1 A

C T 1 A

D F 2 C

E F 2 C

F F \* 0

G F \* 0

Q: B,E,D

----------------------------

Path: A to C

----------------------------

B Dequeued

---------------

V Known Dv Pv

----------------------------

A T 0 0

B T 1 A

C T 1 A

D F 2 C

E F 2 C

F F \* 0

G F 2 B

Q: E,D,G

----------------------------

Path: A to B

----------------------------

E Dequeued

---------------

V Known Dv Pv

----------------------------

A T 0 0

B T 1 A

C T 1 A

D F 2 C

E T 2 C

F F 3 E

G F 2 B

Q: D,G,F

----------------------------

Path: A to C to E

----------------------------

D Dequeued

---------------

V Known Dv Pv

----------------------------

A T 0 0

B T 1 A

C T 1 A

D T 2 C

E T 2 C

F F 3 E

G F 2 B

Q: G,F

----------------------------

Path: A to C to D

----------------------------

G Dequeued

---------------

V Known Dv Pv

----------------------------

A T 0 0

B T 1 A

C T 1 A

D T 2 C

E T 2 C

F F 3 E

G T 2 B

Q: F

----------------------------

Path: A to B to G

----------------------------

F Dequeued

---------------

V Known Dv Pv

----------------------------

A T 0 0

B T 1 A

C T 1 A

D T 2 C

E T 2 C

F T 3 E

G T 2 B

Q:

----------------------------

Path: A to C to E to F

c. Print the final path.

Path: A

Path: A to C

Path: A to B

Path: A to C to E

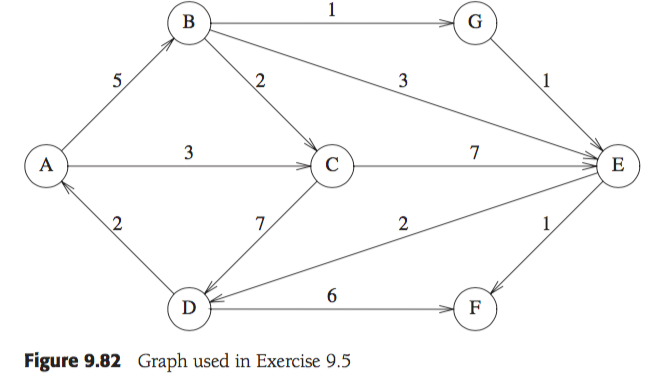
Path: A to C to D

Path: A to B to G

Path: A to C to E to F

1. Write a program to find the weighted shortest path from a given vertex to all other vertices.

a. Use Figure 9.82(p.418) in the text book as input.



Vertex A is distinguished.

Code for the problem:

import java.util.HashMap;

import java.util.HashSet;

import java.util.LinkedList;

import java.util.Queue;

import java.util.Set;

/\*\*

\*

\* @author Adish

\*

\*/

public class WeightedShortestPath

{

public HashMap<String, Vertex> map = null;

public HashMap<String, Integer> mapWeight = null;

public WeightedShortestPath()

{

map = new HashMap<String, Vertex>();

mapWeight = new HashMap<String, Integer>();

}

public void addVertex(String source, String destination, int weight)

{

Vertex sourceVertex = null; // checking for source vertex

if (!map.containsKey(source))

{

sourceVertex = new Vertex(source);

map.put(source, sourceVertex);

}

else

{

sourceVertex = map.get(source);

}

Vertex destinationVertex = null; // checking for destination vertex

if (!map.containsKey(destination))

{

destinationVertex = new Vertex(destination);

map.put(destination, destinationVertex);

}

else

{

destinationVertex = map.get(destination);

}

sourceVertex.AdjacentVertices.add(destinationVertex);

String weightKey = source + destination;

mapWeight.put(weightKey, weight);

}

public void Dijkstra(String vertexName)

{

Set<Vertex> q = new HashSet<Vertex>();

for (Vertex v : map.values())

{

v.distance = Integer.MAX\_VALUE; // INFINITY

v.known = false;

}

Vertex s = map.get(vertexName);

s.distance = 0;

q.add(s);

PrintMap("Initial State", q);

while(!q.isEmpty())

{

Vertex v = getMinimum(q);

v.known = true;

q.remove(v);

for (Vertex w : v.AdjacentVertices)

{

if (w.distance > (v.distance + getDistance(v, w) ))

{

w.distance = getDistance(v, w);

w.previous = v;

q.add(w);

}

}

PrintMap(v.data + " Dequeued", q);

System.out.print("Path: ");

PrintPath(v);

System.out.println();

}

}

private int getDistance(Vertex v, Vertex w)

{

return mapWeight.get(v.data + w.data);

}

private void PrintPath(Vertex v)

{

if (v.previous != null)

{

PrintPath(v.previous);

System.out.print(" to ");

}

System.out.print(v.data);

}

private Vertex getMinimum(Set<Vertex> vertexes)

{

Vertex minimum = null;

for (Vertex vertex : vertexes)

{

if (minimum == null)

minimum = vertex;

else

{

if(vertex.distance < minimum.distance)

{

minimum = vertex;

}

}

}

return minimum;

}

private void PrintMap(String message, Set<Vertex> q)

{

System.out.println("----------------------------");

System.out.println(" \t" + message);

System.out.println("\t---------------");

System.out.println(" V\tKnown\tDv\tPv");

System.out.println("----------------------------");

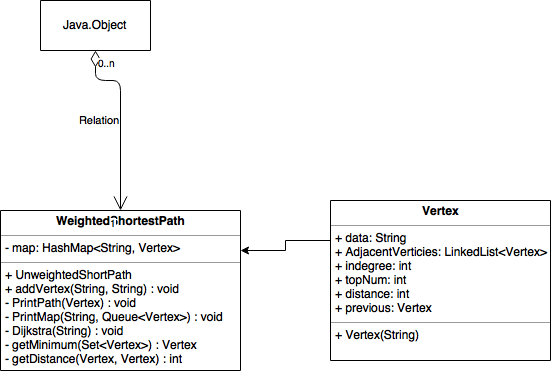
// printing the sorted table

for (Vertex info : map.values()) // traversing through the map values

{

String tableInfo = info.data + "\t" +

((info.known) ? "T\t" : "F" + "\t") +

((info.distance == Integer.MAX\_VALUE) ? "\*" : info.distance) + "\t" +

((info.previous != null) ? info.previous.data : "0");

System.out.println(tableInfo);

}

String tableinfo = "";

for (Vertex v : q) // traversing through the map values

{

if(tableinfo.length() == 0)

tableinfo = v.data;

else

tableinfo = tableinfo + "," + v.data;

}

System.out.println("Q:\t\t" + tableinfo);

System.out.println("----------------------------");

}

}

UML diagram for the Problem:

b. Print the intermediate tables (such as Figures 9.21-9.25).

@Test

public void WeightedShortPathTest() throws Exception

{

WeightedShortestPath ts = new WeightedShortestPath();

// creating the input graph

ts.addVertex("A", "B", 5);

ts.addVertex("A", "C", 3);

ts.addVertex("B", "G", 1);

ts.addVertex("B", "E", 3);

ts.addVertex("B", "C", 2);

ts.addVertex("C", "E", 7);

ts.addVertex("G", "E", 1);

ts.addVertex("C", "D", 7);

ts.addVertex("E", "D", 2);

ts.addVertex("E", "F", 1);

ts.addVertex("D", "F", 6);

ts.addVertex("D", "A", 2);

// print out the nodes and links

ts.Dijkstra("A");

}

----------------------------

Initial State

---------------

V Known Dv Pv

----------------------------

A F 0 0

B F \* 0

C F \* 0

D F \* 0

E F \* 0

F F \* 0

G F \* 0

Q: A

----------------------------

----------------------------

A Dequeued

---------------

V Known Dv Pv

----------------------------

A T 0 0

B F 5 A

C F 3 A

D F \* 0

E F \* 0

F F \* 0

G F \* 0

Q: C,B

----------------------------

Path: A

----------------------------

C Dequeued

---------------

V Known Dv Pv

----------------------------

A T 0 0

B F 5 A

C T 3 A

D F 7 C

E F 7 C

F F \* 0

G F \* 0

Q: D,B,E

----------------------------

Path: A to C

----------------------------

B Dequeued

---------------

V Known Dv Pv

----------------------------

A T 0 0

B T 5 A

C T 3 A

D F 7 C

E F 7 C

F F \* 0

G F 1 B

Q: D,E,G

----------------------------

Path: A to B

----------------------------

G Dequeued

---------------

V Known Dv Pv

----------------------------

A T 0 0

B T 5 A

C T 3 A

D F 7 C

E F 1 G

F F \* 0

G T 1 B

Q: D,E

----------------------------

Path: A to B to G

----------------------------

E Dequeued

---------------

V Known Dv Pv

----------------------------

A T 0 0

B T 5 A

C T 3 A

D F 2 E

E T 1 G

F F 1 E

G T 1 B

Q: D,F

----------------------------

Path: A to B to G to E

----------------------------

F Dequeued

---------------

V Known Dv Pv

----------------------------

A T 0 0

B T 5 A

C T 3 A

D F 2 E

E T 1 G

F T 1 E

G T 1 B

Q: D

----------------------------

Path: A to B to G to E to F

----------------------------

D Dequeued

---------------

V Known Dv Pv

----------------------------

A T 0 0

B T 5 A

C T 3 A

D T 2 E

E T 1 G

F T 1 E

G T 1 B

Q:

----------------------------

Path: A to B to G to E to D

c. Print the final path

Path: A

Path: A to C

Path: A to B

Path: A to B to G

Path: A to B to G to E

Path: A to B to G to E to F

Path: A to B to G to E to D