**Programming Assignment Graphs CS-254**

**Dijkstra’s Shortest Paths**

Goals: Implement a graph as an adjacency **matrix**. Learn more about the Java Priority Queue and HashMap as data structures used in the algorithm.

1. Create a new Java project in eclipse and create the following classes from the UML
2. Add a GraphPathsDriver with a main method. Use the provided code to test your implementations. Add additional more complicated graphs to test it.
3. Implement the toString method to match the provided output of the graph. (20 points)
4. Implement the Dijkstra Algorithm and print out distances (20 points). Print out the paths from source to all other vertices (10 points). Match output provided on last page of this assignment.
5. Remember proper documentation.

|  |
| --- |
| GraphMatrix |
| * vertices: Vertex[] * edgeWeights: int[][] |
| + GraphMatrix(Vertex[] vertices, int[][] edgeWeights): //constructor  + Dijkstra(int sourceIndex): void // vertices with distance from source  + toString():String |

**GraphMatrix UML Details:**

**Instance Variables:**

**vertices** – instance variable to the array of all vertices in the graph.

**edgeWeights** – A 2D array of edge weights from Vertex to Vertex. The indices in the array match the indices of the Vertex in vertices.

**Methods:**

**Graph Matrix(Vertex[] vertices, int[][] edgeWeights) –** assign parameters to instance variables.

**toString()** - Print out vertex and its edges to other vertices with the weight of the edge. Here is an undirected graph output:

Adjacency matrix for graph:

A ->B, 1 ->C, 3

B ->A, 1 ->C, 1

C ->A, 3 ->B, 1

The Vertex class adds to the previous Vertex class by implementing Comparable and including the required compareTo method. This allows the Vertex objects to be compared for storage in a PriorityQueue. The distance variable and getter/setter is added to store the distance from the source Vertex in Dijkstra’s Algorithm.

|  |
| --- |
| Vertex **implements Comparable** |
| * name: String * color: int * visited: Boolean * **distance: int**   + WHITE: int //1  + GREEN: int //2  + BLACK: int //3 |
| + Vertex(name: String): //constructor  + getName():String  + getColor():int  + setColor(color: int):void  + isVisited(): boolean  + setVisited(visited: boolean): void  **+ getDistance():int**  **+ setDistance(distance: int):void**  **+ compareTo(Vertex o): int**  + toString():String |

**Vertex UML Details:**

**Instance Variables:**

**name** – name of the Vertex

**color** – color of Vertex. 1 is WHITE, 2 is GREEN, 3 is Black. Match the constants.

**visited** – boolean to store whether the vertex has been visited.

**distance** – int to store distance from source node in Dijkstra’s Algorithm.

**Methods:**

**getters and setters** – As usual.

**toString()** - return vertex name.

**compareTo(Vertex o) -**

**public** **int** compareTo(Vertex o) {

**int** value = 0;

**if**(distance<o.getDistance())

value = -1;

**else** **if**(distance>o.getDistance())

value = 1;

**return** value;

}

**Your code must implement the algorithm listed below. You must comment your code with exactly what part of the algorithm it is implementing. You must list Input, Output, and each line of the algorithm in comments by the code that implements it. Every method should have documentation in Javadocs. No credit for code that does not have algorithm comments to match code.**

In the following algorithm, the code

* u ← vertex in *Q* with min dist[u], searches for the vertex u in the vertex set Q that has the least dist[u] value.
* length(u, v) returns the length of the edge joining (i.e. the distance between) the two neighbor-nodes u and v.
* The variable alt on line 18 is the length of the path from the root node to the neighbor node v if it were to go through u. If this path is shorter than the current shortest path recorded for v, that current path is replaced with this alt path.
* The predecessor HashMap is populated with a previous node used to get the shortest route to the source.

**Algorithm:** Dijkstra(int *sourceIndex*):

Vertex source = vertices[sourceIndex]

create PriorityQueue<Vertex> Q

create a HashMap<Vertex,Vertex> predecessor

**for each** vertex *v* in *Graph*:

dist[*v*] ← INFINITY //v.setDistance(Integer.MAX\_INT)

if v is source

set dist[*source*] ← 0

add *v* to *Q*

//**end** for

**while** *Q* is not empty **do**

*u* ← vertex in *Q* with min distance

remove *u* from *Q*

*uIndex* ← index of u in vertices

**for each** neighbor *v* of *u* **do**

**if** v still in Q and edge from u to v

*alt* ← dist[*u*] + length(*u*, *v*)

**if** *alt* < dist[*v*]:

remove v from Q

dist[*v*] ← *alt*

*add v to Q*

set u to be predecessor to v

print out distance to each vertex from the source

print out path from source to each vertex

**for each** vertex current in vertices

String path = “”;

**if**(not sourceIndex)

path = “ -> “+path

**while**(predecessor contains key (current) )

**if**( predecessor of current is source

path = predecessor get current +path

**else**

path = “ -> “+predecessor get current + path

current = predecessor get current

**//end while**

**//end if**

print path

**//end for**

**Tester code for your main method:**

//Create vertices.

Vertex A = **new** Vertex("A");

Vertex B = **new** Vertex("B");

Vertex C = **new** Vertex("C");

//Store in an array of vertices

Vertex[] vertices = **new** Vertex[3];

vertices[0] = A;

vertices[1] = B;

vertices[2] = C;

//Create edge weight matrix and use to create graph.

**int**[][] distances = { {0, 1, 3}, {1, 0, 1}, {3, 1, 0} };

GraphMatrix graph1 = **new** GraphMatrix( vertices, distances);

//Print out graph for 20 points.

System.***out***.println(graph1);

//Call Dijkstra with Vertex A index. Uncomment once implemented. //Test with more challenging graphs.

//graph1.Dijkstra(0);

**Output for provided tester code:**

Adjacency matrix for graph:

A ->B, 1 ->C, 3

B ->A, 1 ->C, 1

C ->A, 3 ->B, 1

Distances from Vertex A

A

B, 1

C, 2

Shortest paths from Vertex A

A -> B, 1

A -> B, 1 -> C, 2