

Institute of Technology

School of Electrical and Computer Engineering

Data Structure

*Dijkstra’s Algorithm*

By: - Eyuel Worku ATR/5653/09

- Mikiyas Tesfaye ATR/6446/09

Instructor: - Menore T.

Due Date: - Jan 9, 2020

**Weighted Graphs**

A graph is a set of vertices that are connected by edges. A Weighted graph is a graph where each edge has a weight. These weights are usually a non-negative number that show the cost of that edge. In application, the weight may be a measure of the length of a route, the capacity if a line, the energy required to move between locations along a route, etc.

Graphs maybe either directed or undirected. In directed graphs each edge has a direction while in undirected graph the edges do not have direction. This means that the edges are two way.

In a graph there might be more than one path, sequence of vertices connected by edges, between two vertices. In a weighted graph the cost or weight of these paths is different. It is important to find the path with the least cost or the shortest path. To attain this, we can use different algorithms like Dijkstra’s, Prim’s and Kruskal’s. Next, we will see the Dijkstra’s Algorithm.

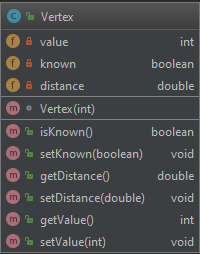
**Dijkstra’s Algorithm**

This algorithm is used to find the shortest distance between two vertices of a directed weighted graph. This algorithm does not work if one of the edges have a negative weight.

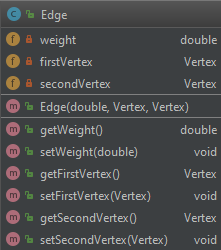
It starts off by making the distance of all the vertices infinity, except for the starting point of the path. It makes the distance of the starting point zero since the distance from a vertex to itself is zero. Then it goes through all the vertices and evaluate each edge. At every vertex, the algorithm evaluates the distance of each vertex connected to the current vertex and updates them if necessary. It also decides the next vertex whose edges are going to be evaluated. This is done simply by taking the edge with the least weight unless it runs into a dead end at which point it will take one of the unvisited vertices and procced.

**Implementation**

The Project was implemented using Java. The project took a total of four classes. One of the four classes is a demonstration classes that contains the main method. It instantiates the WeightedTree class and makes calls to its different functions. The second and the third classes were model classes that model the entities vertex and edge. These classes are named according to the entity they model. The fourth and the last class is a class that implements the weighted tree named according to the thing it implements. The UML Diagram and description for the model classes are presented below.

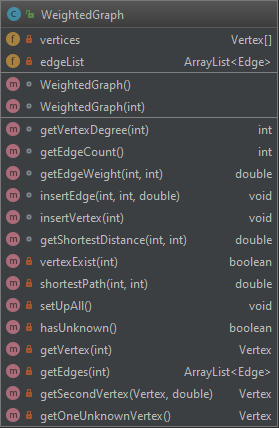


This class has three instance variables. The first variable has an integer data type that holds the value of the vertex. The second variable is a boolean that determines whether the vertex has been visited or not. The third variable holds the distance of the vertex from another vertex when calculating the shortest path.



This class has three instance variables. The first variable has double data type that holds the weight of the edge. The second and the third variables are type Vertex that hold the two vertices of the edge.

The UML Diagram and Pseudo code for the methods of the WeightedTree class are presented below.



This class holds an array of vertices and an Array List of edges. Its constructor is overloaded so that it can accept the size of the vertices array or it takes a default value of 50. The private methods are used by other methods within the class. The pseudo code for the methods in this class are given below.

int getVertexDegree(int vertexDegree)

int count = 0;

for each edge in edgeList

if(edge.getFirstVertex().getValue()==vertexDegree||

edge.getFirstVertex().getValue()==vertexDegree)

count++;

end for

return count;

end getVertexDegree

int getEdgeCount()

return edgeList.size();

end getEdgeCount

double getEdgeWeight(int firstValue, int secondValue)

if (vertex with value firstValue does not exist || vertex with value secondValue does

not exist) throw IllegalArgumentException

else if (firstValue equal to secondValue) return 0

else if (vertex with value firstValue and vertex with value secondValue are not

connected) return infinity

else

for each edge in edgeList

if (edge.getFirstVertex().getValue == firstValue

&& edge.getSecondVertex().getValue == secondValue)

return edge

end for

end getEdgeWeight

void insertEdge(int firstValue, int secondValue, double edgeWeight)

if (edgeWeight<0 || edgeWeight==infinity || if the vertices are the same || vertex with

firstValue does not exist || vertex with secondValue does not exist)

throw IllegalArgumentException

Boolean edgeExist = false;

for each edge in edgeList

if(edge.getFirstVertex().getValue() == firstValue ||

edge.getSecondVertex().getValue() == secondValue)

edge.setWeight(edgeWeight);

edgeExist = true;

end if

end for

if(edge does not Exist)

add into edgeList new Edge

end insertEdge

void insertVertex (int vertexValue)

if (vertexValue<=0 || vertex exists) throw IllegalArgumentException

else

add new Vertex on vetices array

end insertVertex

double getShortestDistance (int firstValue, int secondValue)

if (vertex with value firstValue does not exist || vertex with value secondValue does

not exist) throw IllegalArgumentException

else if (the two vertices are the same) return 0;

else if (the two vertices are not connected) return infinity

else return return value of call to shortestDistance calculating method

end getShortestDistance

double shortestPathWeight (int firstValue, int secondValue)

for each vertex in vertices

vertex.distance = infinity

end for

set distance of vertex with value firstValue = 0

Vertex currentVertex = vertex with value equal to first value

while (there is unknown vertex)

if (currentVertex == unknown)

currentVertex = one unknown vertex from the vertices array

ArrayList<Edges> edges = edges of the current vertex

Set current vertex as known

double nextPathDistance = infinity

for each edge in edges

if (the distance of the second vertex of the edge > edge.weight +

currentVertex.distance)

second vertex distance = currentVertex.distance + edge.weight

if (edge.weight<nextPathDistance)

nextPathDistance = edge.weight

end for

currentVertex = the second vertex of the current vertex connected by the edge

with weight nextPathDistance

end while

return vertex with value equal secondVertex.distance

end shortestPathWeight