

19AIE203 Data Structures & Algorithms 2 (1 2 0 3)

S3 B.Tech CSE(AI)

Project Report

Submitted by

Arjun Rathya R : (AM.EN.U4AIE19015)

Group Members

Abhiram Prasad : (AM.EN.U4AIE19001)

Devagopal AM : (AM.EN.U4AIE19025)

Vishal Menon : (AM.EN.U4AIE19070)



December 2020

Department of Computer science and Engineering

Amrita Vishwa Vidyapeetham

Amritapuri Campus

Kollam 690525

Kerala

Amrita Vishwa Vidyapeetham
Amritapuri Campus
Kollam 690525
Kerala



Department of Computer Science and Engineering

Certified that this is the bonafide project report for the course 19AIE203
Data Structures & Algorithms-2 submitted on its completion by Group-
Arjun Rathya R (AM.EN.U4AIE19015), Abhiram Prasad
(AM.EN.U4AIE19001), Devagopal AM (AM.EN.U4AIE19025) and
Vishal Menon (AM.EN.U4AIE19070) , third semester students of
B.Tech CSE(AI)

ABSTRACT

The project deals with helping Vivek who has moved to a city Jaipur where he wants to use the public transport system and is low in cash. The aim of the problem is to help Vivek find the most cost efficient way to go from the first station to the last station. Here we have helped Vivek on implementing the problem solution with Dijkstra's shortest path algorithm. We have implemented it by considering the above as a graph problem and handling it via adjacency lists. Considering the stations as nodes and the fares as the weight, we have calculated the shortest path (least fare) from source station to the destination. The solution is implemented with 5 java classes where 3 classes handle the Problem solution and two of them care about the Dataset Generation for implementing the algorithm. Eclipse IDE for java developers was used by us for implementing the code. The dataset generated is a text file that has a first line representing the number of Nodes and Edges and then the nodes and their fare details as 3 columns. We experimented our implementation with the sample datasets provided with the problem and the randomly generated ones. From a smaller dataset, with 4 nodes, we have implemented upto 50000 nodes(stations), 500000 edges and a max weight of 10000000 (fare). We were successful in our experimental results as it produced the shortest path(least fare) for the input datasets in very less time.

Contents

	page no
1. Introduction	5
2. Software and Hardware Requirements	11
3. Problem Statement.	12
4. Modules	16
5. Implementation Details.	22
6. Experimentation Result	26
7. Conclusion	35
8. References	37
9. Appendix	38

1. INTRODUCTION

This project deals with helping Vivek who is low in cash and needs help to figure out the most cost efficient way to go from the first station to the last station based on some fare policies. This resembles a real world application of finding the shortest path problem. And we approach this problem with Dijkstra's shortest path algorithm. So let's see the basic concepts that are bound to this topic.

GRAPH

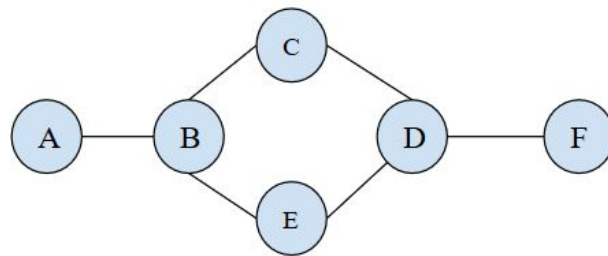
A graph can be defined as a group of vertices (nodes) and edges that are used to connect these vertices. A graph G can be defined as an ordered set $G(V, E)$ where $V(G)$ represents the set of vertices and $E(G)$ represents the set of edges which are used to connect these vertices.

Graphs are really helpful as they solve many real-life problems. They are used for representing networks. This may include paths in a city or telephone network or circuit network and can be extended with the case of social networks like linkedIn, Facebook etc. where, each person could be represented as a vertex(or node) and each node is a structure that contains many information like person id, name, gender, locale etc.

A graph can be directed or undirected.

UNDIRECTED GRAPH

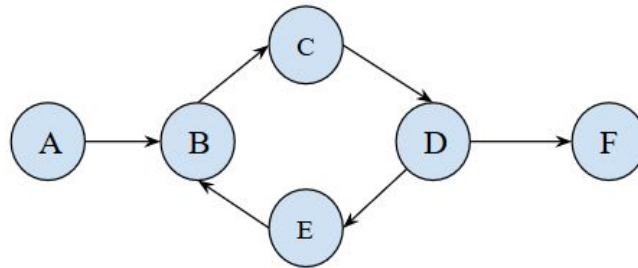
In an undirected graph, edges are not associated with the directions with them. Since its edges are not attached with any of the directions, if an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B . That is, when the graph is undirected the adjacency relation is symmetric.



UNDIRECTED GRAPH

DIRECTED GRAPH

In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node while node B is called terminal node.



DIRECTED GRAPH

TERMINOLOGIES

Path: It can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.

Cycle: It can be defined as the path which has no repeated edges or vertices except the first and last vertices.

Connected Graph: It is when some path exists between every two vertices (u, v) in V. There are no isolated nodes in the connected graph.

Complete Graph: It is when every node is connected with all other nodes. A complete graph contains $n(n-1)/2$ edges where n is the number of nodes in the graph.

Weighted Graph: Here each edge is assigned with some data such as length or weight. The weight of an edge e can be given as $w(e)$ which must be a positive (+) value indicating the cost of traversing the edge.

Adjacent Nodes: If two nodes u and v are connected via an edge e , then the nodes u and v are called as neighbours or adjacent nodes.

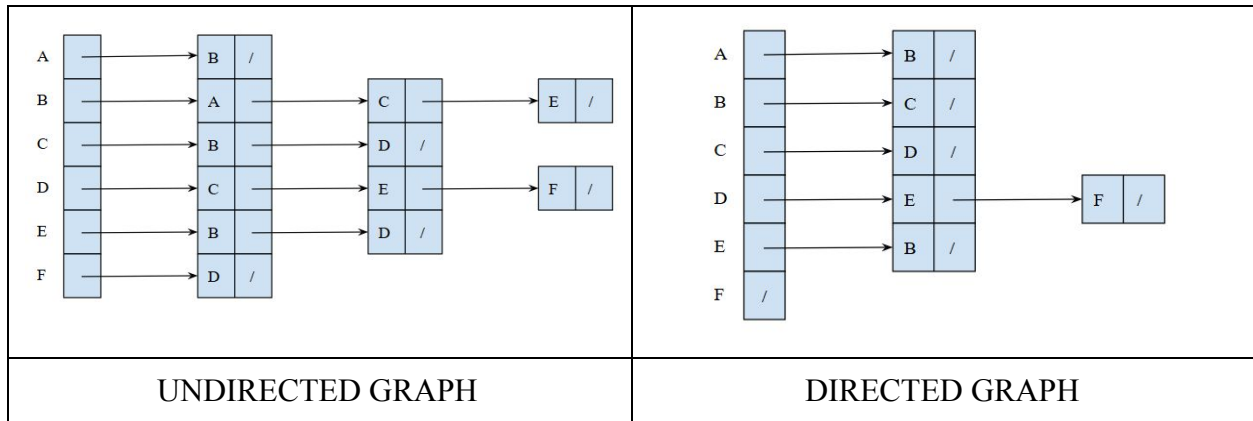
Degree of the Node: Degree of a node is the number of edges that are connected with that node. A node with degree 0 is called an isolated node. Degree of a vertex in an undirected graph is the number of edges incident on it and when it comes to directed graphs, it is the sum of indegree(number of arcs directed towards the vertex) and outdegree(number of arcs directed away from the vertex).

REPRESENTATION OF GRAPH

Graphs can generally be represented in two ways,

1. Adjacency List.
2. Adjacency Matrix.

Adjacency List is an array consisting of the address of all the linked lists. The first node of the linked list represents the vertex and the remaining lists connected to this node represents the vertices to which this node is connected. This representation can also be used to represent a weighted graph. The linked list can slightly be changed to even store the weight of the edge.



Adjacency Matrix is a 2D array of size $V \times V$ where V is the number of vertices in a graph. Let the 2D array be $adj[i][j]$, a slot $adj[i][j] = 1$ indicates that there is an edge from vertex i to vertex j . As mentioned before, the adjacency matrix for an undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If $adj[i][j] = w$, then there is an edge from vertex i to vertex j with weight w .

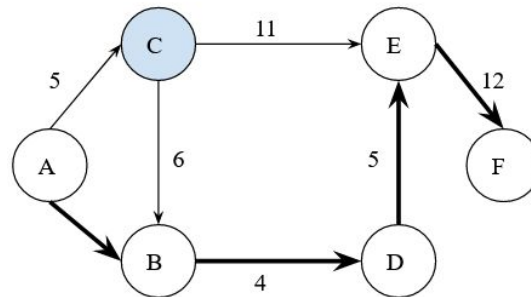
<table><tr><td></td><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td><td>F</td></tr><tr><td>A</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>B</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr><tr><td>C</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td></tr><tr><td>D</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr><tr><td>E</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td></tr><tr><td>F</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr></table>		A	B	C	D	E	F	A	0	1	0	0	0	0	B	1	0	1	0	1	0	C	0	1	0	1	0	0	D	0	0	1	0	1	1	E	0	1	0	1	0	0	F	0	0	0	1	0	0	<table><tr><td></td><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td><td>F</td></tr><tr><td>A</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>B</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>C</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr><tr><td>D</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><td>E</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>F</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>		A	B	C	D	E	F	A	0	1	0	0	0	0	B	0	0	1	0	0	0	C	0	0	0	1	0	0	D	0	0	0	0	1	1	E	0	1	0	0	0	0	F	0	0	0	0	0	0
	A	B	C	D	E	F																																																																																													
A	0	1	0	0	0	0																																																																																													
B	1	0	1	0	1	0																																																																																													
C	0	1	0	1	0	0																																																																																													
D	0	0	1	0	1	1																																																																																													
E	0	1	0	1	0	0																																																																																													
F	0	0	0	1	0	0																																																																																													
	A	B	C	D	E	F																																																																																													
A	0	1	0	0	0	0																																																																																													
B	0	0	1	0	0	0																																																																																													
C	0	0	0	1	0	0																																																																																													
D	0	0	0	0	1	1																																																																																													
E	0	1	0	0	0	0																																																																																													
F	0	0	0	0	0	0																																																																																													
UNDIRECTED GRAPH	DIRECTED GRAPH																																																																																																		

In our project we have focused on the implementation based on adjacency list where each node present in the graph stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed

then store the NULL in the pointer field of the last node of the list. Here for considering the weight, an extra field is considered for each node.

SHORTEST PATH PROBLEM

The shortest path problem is about finding a path between two vertices in a graph such that the total sum of the edges weights is minimum. The problem of finding the shortest path between two intersections on a road map may be modeled as a special case of the shortest path problem in graphs, where the vertices correspond to intersections and the edges correspond to road segments, each weighted by the length of the segment.



Shortest path (A, B, D, E, F) between vertices A and F in the weighted directed graph

Note: If P is a path that includes cycle C and P', a path that contains no cycle, then:

$$\text{weight}(P') = \text{weight}(P) - \text{weight}(C) < \text{weight}(P)$$

BFS can be used to find the shortest path only if the graph is unweighted.

DIJKSTRA'S ALGORITHM

This algorithm uses a technique of relaxation, i.e., for each vertex an attribute $v.d$ is maintained which represents an upper bound on the weight of shortest path from s to v . Let's call $v.d$ as the shortest path estimate.

<i>Initialise Single Source</i> (G, s) for each vertex $v \in G.V$ $v.d = \infty$ $s.d = 0$	<i>Relax</i> (u, v, w) if $v.d > u.d + w(u, v)$ $v.d = u.d + w(u, v)$
----------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------

<i>Dijkstra</i> (G, w, s) <i>Initialise Single Source</i> (G, s) $S = \emptyset$ $Q = G.V$ while $Q \neq \emptyset$ $u = \text{Extract Min}(Q)$ $S = S \cup \{u\}$ for each vertex $v \in G. \text{Adj}[u]$ $\text{Relax}(u, v, w)$

Dijkstra's Algorithm has several real-world use cases, these include Digital Mapping Services in Google Maps, Social Networking Applications, Telephone Network, IP routing to find Open shortest Path First, Flighting Agenda, Designate file server, Robotic Path etc..

For example, drones/robots which are automated and are used to deliver packages to a specific location use this algorithm so that when the source and destination is known, it moves by following the shortest path to keep delivering the package in a minimum amount of time.

Time Complexity of Dijkstra's Algorithm is $O(E \log V)$

2. SOFTWARE AND HARDWARE REQUIREMENTS

REQUIREMENT	MINIMUM	RECOMMENDED
Software	-	JAVA
Java Version	1.4.0	5.0 or greater
Memory	512 MB	1 GB or more
Free Disk Space	300 MB	1 GB or more
Processor Speed	800 MHz	1.Ghz or faster
Operating System	Windows 7	Windows 8 or higher
Keyboard	Compaq	-

IDE USED : Eclipse IDE for Java Developers - 2020-06

Eclipse is an integrated development environment (IDE) used in computer programming. It contains a base workspace and an extensible plug-in system for customizing the environment. Eclipse is written mostly in Java and its primary use is for developing Java applications, but it may also be used to develop applications in other programming languages via plug-ins. The package consists of essential tools for any Java developer, including a Java IDE, a CVS client, Git client, XML Editor, Mylyn, Maven integration and WindowBuilder and includes Code Recommenders Developer Tools, Eclipse Git Team Provider, Eclipse Java Development Tools, Maven Integration for Eclipse, Mylyn Task List, WindowBuilder Core, Eclipse XML Editors and Tools.

The latest Eclipse IDE 2020-12 can be downloaded from here :

<https://www.eclipse.org/eclipseide/>

3. PROBLEM AND SOLUTION

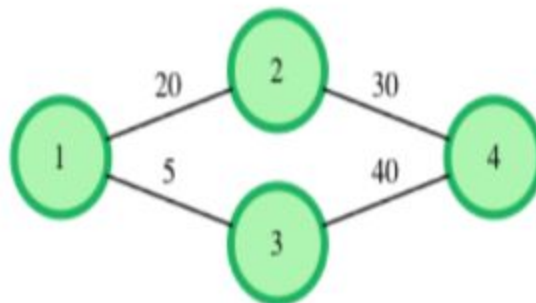
PROBLEM

Vivek has just moved to a new city called Jaipur. He wants to use the public transport system. The fare rules are as follows:

1. Each pair of connected stations has a fare assigned to it regardless of direction of travel.
2. If a passenger travels from station A to station B, he only has to pay the difference between the fare from A to B and the cumulative fare that he has paid to reach station A [fare(A,B) - total fare to reach station A]. If the difference is negative, he can travel free of cost from A to B.

Vivek is low on cash and needs your help to figure out the most cost efficient way to go from the first station to the last station.

For example, there are **g_nodes = 4** stations with undirected connections at the costs indicated:



Travel from station **1→2→4** costs **20** for the first segment(**1→2**) then the cost differential, an additional **30-20 = 10** for the remainder. The total cost is 30. Travel from station **1→3→4** costs **5** for the first segment, then an additional **40-5 = 35** for the remainder, a total cost of **40**. The lower priced option costs **30**.

Complete the program in the editor below. It should print the cost of the lowest price route from station 1 to station **g_nodes**. If there is no route, print NO PATH EXISTS.

Function Description

Complete the `getCost` function in the editor below. It should print the cost of the lowest priced route from station 1 to station **g_nodes**, or if there is no route, print NO PATH EXISTS. There is no expected return value from the function.

`getCost` has the following parameters:

- **g_nodes**: an integer that represents the number of stations in the network.
- **g_from**: an array of integers that represent end stations of a bidirectional connection.
- **g_to**: an array of integers that represents end stations of a bidirectional connection, where **g_from[i]** is connected to **g_to[i]** at cost **g_weight[i]**.
- **g_weight**: an array of integers that represent the cost of travel between associated stations.

Input Format

The first line contains two space-separated integers, **g_nodes** and **g_edges**, the number of stations and the number of connections between them.

Each of the next **g_edges** lines contains three separated integers, **g_from** and **g_weight**, the starting and ending stations that are connected and the fare between them.

Constraints

- $1 \leq \text{g_nodes} \leq 50000$
- $1 \leq \text{g_edges} \leq 500000$
- $1 \leq \text{g_weight}[i] \leq 10^7$

Sample Input 2



Sample Output 2

85

Explanation 2

Travel starts at node **1** and there are two paths to node **3** that cost either **50** or **70**. Taking the route from **3** through **4** to **5** brings the cost up to **90**, while going directly from **3** to **5** costs only **85**.

Output Format

The minimum fare to be paid to reach station **g_nodes** from station **1**. If the station **g_nodes** cannot be reached from station 1, print NO PATHS EXISTS.

SOLUTION

The problem is identified as finding the shortest path. Here Vivek needs to find the most cost effective path from station A which acts as a node to the last station (last node). Here the weight of the path is considered as the fare for his travel. So for finding the shortest path, we have implemented Dijkstra's algorithm with some basic modifications to obtain the final fare.

The implemented algorithm runs through the dataset finding the shortest weight to reach the final node. It's clearly explained in the Modules section. In brief, it works on the basis of a queue where the weight of the source vertex is initially made zero and the weight of the neighbor nodes are determined on the basis of a relaxation technique. This falls under a while loop where it runs till the Q is empty. Finally the shortest path estimate of the last node is determined. If its value tends to be the highest possible integer value representing infinity, then the algorithm returns that no path exists from the source vertex to the final vertex.

Coming to datasets, we have used a large number of datasets for experimenting our implementation. The smaller ones are previewed in the experimented results and the bigger ones are submitted with proper names in a zip file. The algorithm for Test class is implemented in a way that it could accept a dataset that contains the first line with the number of nodes and the number of edges and the remaining with the node values and weights. Bigger datasets were produced as a result of our requirement with the help of Pair and Data classes.

The required dataset for solving our problem is Undirected Weighted Graphs.

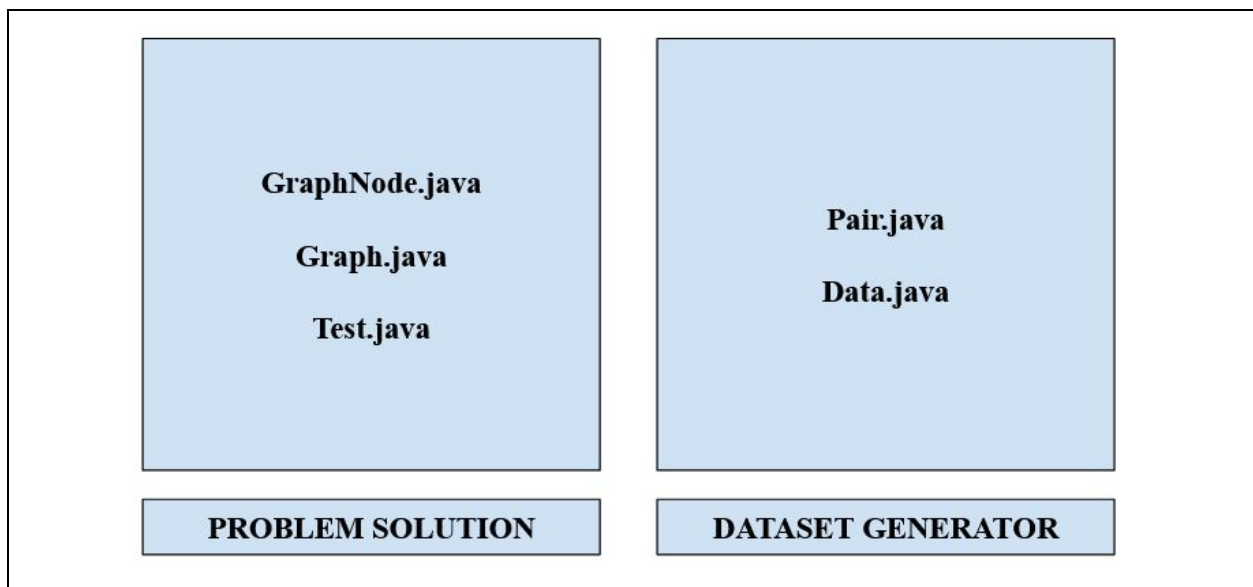
The smallest dataset experimented with this algorithm is the base example provided for the explanation of the question. It was successful and we were able to obtain the minimum cost as 30. This was again confirmed with the second example where we got 85 as the minimum cost required for Vivek to travel to the last station.

As per the problem, we have checked with a randomly generated dataset that consists of 50000 nodes, 500000 edges and a maximum weight of 10000000. We

were able to find the shortest path or the minimum fare for this dataset also within a fraction of time. We obtained a result of 2592097.

4. MODULES

Our project is implemented using 5 java classes where 3 of them focus on the solution to our problem and 2 of them care about the dataset generation.



GraphNode.java, **Graph.java** and **Test.java** are the 3 classes proposed for solving our problem, that is helping Vivek find the most cost efficient way.

GraphNode.java

GraphNode.java contains an integer label, 2 associated arraylists adjLists and weights, a boolean variable visited and finally another integer d. The graph in our solution is represented as an adjacency list and for considering their corresponding weights we have 2 arraylists. Integer label is used for keeping a track of the node

ID and the boolean variable visited is used to help determine whether a node is visited or not. Integer d helps in handling the weight values to get the shortest path in our problem.

Initially we have created a GraphNode constructor, where the label could be passed as a parameter. We are also allocating memory for both the arraylists here and variable d is initialised here with the maximum possible value for any integer variable in Java.

The class consists of a **print()** that traverses till the size of the adjacency list of a graph node and prints the respective label. When it reaches the end it displays null. This functionality is associated with the print method in the Graph.java class where it's called with the help of an object in the driver class, Test.java.

Then the class consists of a **reset()** that initially sets the value of visited as false for all the node neighbors. Here the shortest path estimates are also initialised to the maximum integer value.

Then we implemented the core algorithm for solving our problem in this class, Dijkstra's algorithm. This algorithm is implemented with slight modifications to produce our expected result.

Dijkstra() is implemented such that a double ended queue is passed to it as a parameter. The function initially checks if the queue is empty or not with a while condition and proceeds if it's not empty until it becomes empty. A variable x of type GraphNode is returned with the head value of the queue and the head is removed. A for loop is created that iterates till the size of x's adjacency list. Here another GraphNode variable y is created. It holds a neighbor of the current node. The weight variable gets the weight of the edge from the arraylist weights. After checking whether node y is visited or not, the method progresses.

If y is not visited, then the variable value is changed to true making it mean that the node is visited. The node y is then added to the queue. The value of weight is then subtracted with the shortest path estimate of the current node and compared with zero. If this results in a less than case, then w ends up as zero otherwise w equals the difference between weight and the shortest path estimate of the current node. The shortest path estimate of node y represented as y.d is updated then as the sum

of w and the shortest path estimate of the current node x . This is the case when the node isn't visited.

But if the node was a visited one, then the difference between weight and the shortest path estimate of current node x is compared with zero. Further with one more such check the shortest path estimate is updated. This technique is known as **relaxation** technique. In this way the algorithm works for determining the most cost efficient path for helping Vivek.

Graph.java

In Graph.java we are creating a collection of graph nodes of type GraphNode. With the help of a parameterised constructor, we are able to create a collection of graph nodes with size of the array passed as argument. For allocating memory for each object in the array a for loop is created.

The **addEdge()** helps in creating an edge from one node to another and assigning weights to them. Here three parameters are passed on to it where the first one represents the source vertex, the second one the destination vertex and the third one the weight of the edge. The array list of source vertex is accessed for this implementation.

The **print()** allows the user to print the adjacency list representation of the graph in the console. Here the print() takes the help of the print() implemented within the GraphNode.java as mentioned earlier. Even though this method is defined, it hasn't been used as we are focusing on bigger datasets. This is helpful when it comes to analysing small datasets and their adjacency lists.

The **reset()** resets the collection of graph nodes with visited value false and shortest path estimates to max integer value representing infinity. Then for considering and implementing the same for the adjacency list, it calls the reset() of GraphNode.java which is explained above.

With the **getCost()**, we are actually finding the shortest path from a source vertex s which is passed to it as a parameter. The function then calls the Dijkstra() passing the source vertex as the argument.

Here, in **Dijkstra()** , after calling the `reset()` , a double ended queue is declared and the node is added to it. The method then calls the Dijkstra method of the `GraphNode.java` for the actual implementation of the algorithm. It's here the shortest path is determined.

Then the `getCost()` checks if the last node's shortest path estimate tends to infinity or not by comparing it with the maximum possible integer value. If it satisfies, the method prints NO PATH EXISTS else prints the respective starting node, destination node and the shortest path estimate.

Test.java

`Test.java` acts as the driver class for the program code finding solution. Here, the path of the source file is requested from the user initially. But on executing the code after generating the dataset using `Pair.java` and `Data.java`, it is fine to proceed with entering "dataset.txt" in the user input requesting query, as the destination for the generated file is not specified in the code.

With the help of `BufferedReader`, the dataset is read by the console and is stored to a string array after splitting them on the basis of a white space. The first part of the array that holds the number of vertices is converted to an integer with the help of `Integer.parseInt()`. The graph is created based on this integer which represents the number of nodes.

Then with a while loop that traverses until the end of the dataset is created such that on each iteration with the values from the dataset, the `addEdge()` method can be implemented. This is done by passing the read details to the `addEdge` method directly with a decrement to make the index value of node start from zero.

However this is accommodated at the end with a proper increment in the `getCost` method of the `Graph.java` class. Atlast, the `getCost(0)` is called to find the shortest path from node 0 to the last node.

Pair.java, and **Data.java** are the 2 classes proposed for generating sample datasets for our problem. We are generating weighted undirected graphs here.

Pair.java

Pair.java class is created for pairing the randomly generated node values. It consists of a constructor that helps in this pairing process.

With **get_first()** and **get_second()** the values are returned. This is helpful for Data.java to access them at the time of appending the values to the writing file.

Data.java

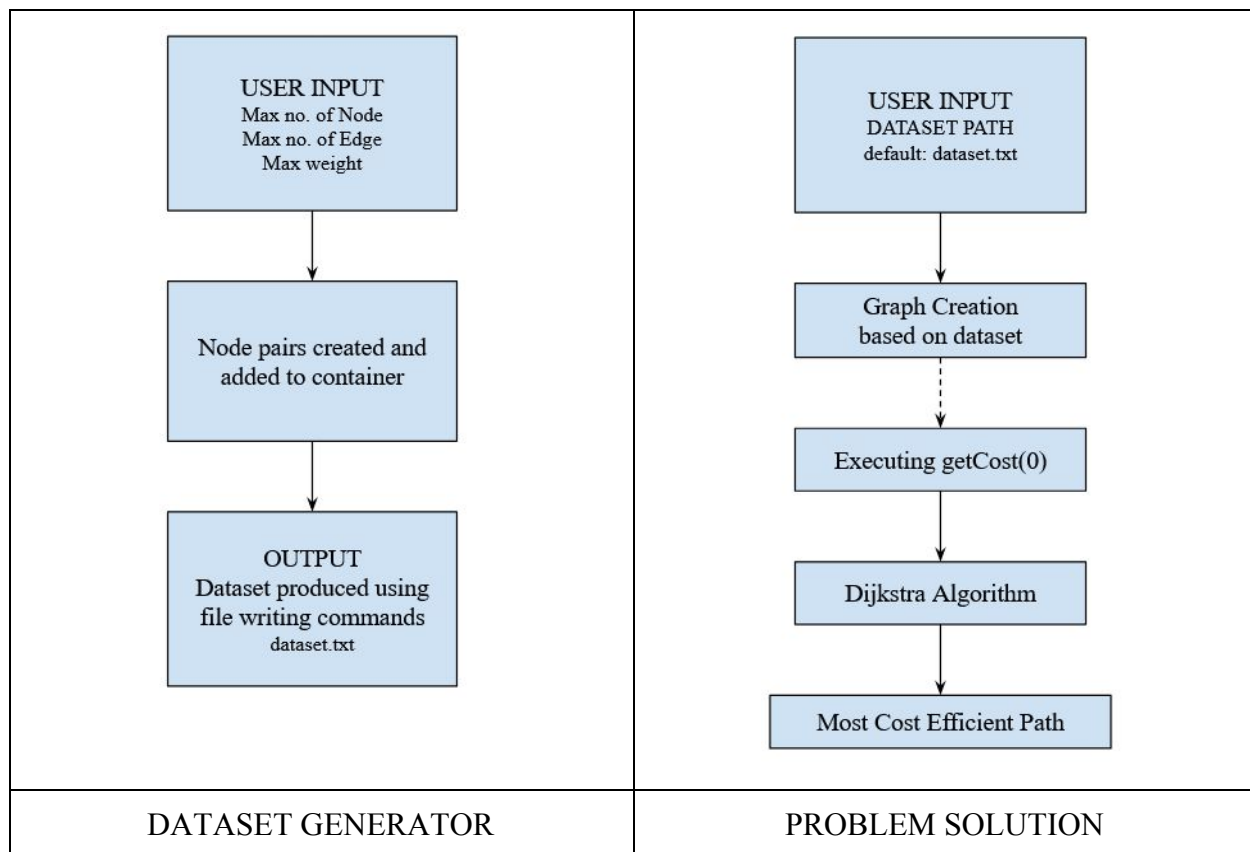
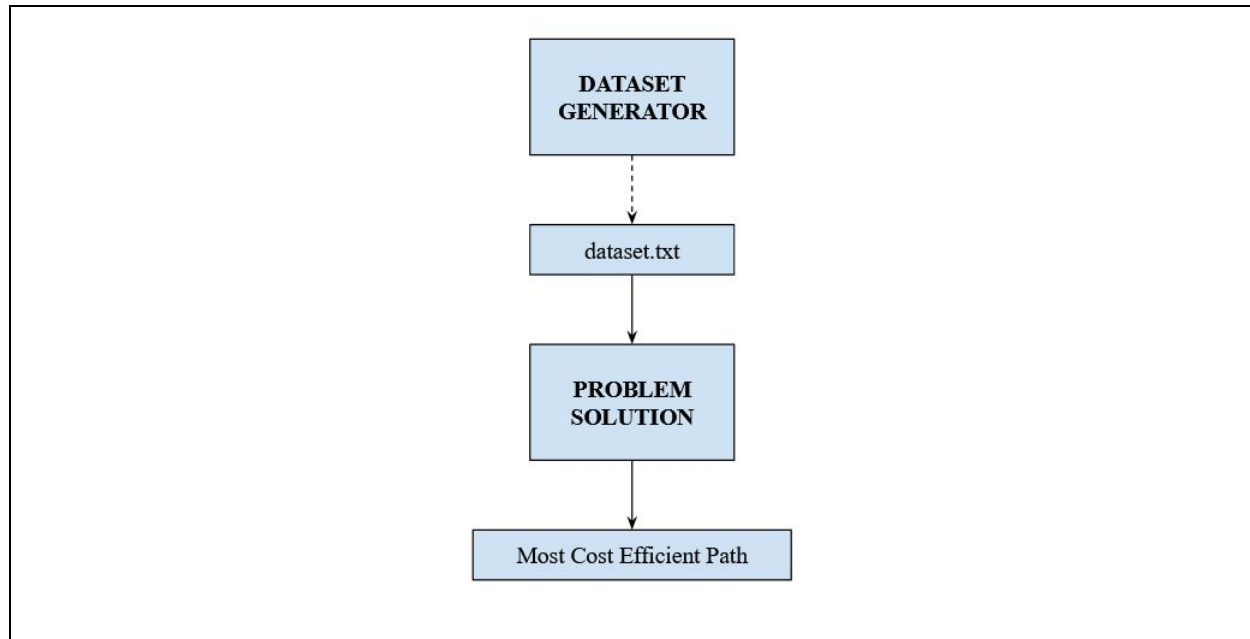
Data.java acts as the driver class for the dataset generating program. Initially three scanner objects are created for accessing the maximum number of edges, maximum number of nodes and maximum possible weight from the user.

A random number generator is created followed by the creation of a container of type Pair that stores values in an empty set called HashSet. The number of vertices and edges are randomised then.

With a while loop condition, the number of edges is again randomised for generating a dense graph. Now two variables a and b are created for handling the generated node values. They are paired and added to the container.

The BufferedWriter helps produce the output to a text file with its write() and append() functionalities. Since we haven't mentioned any path, the file is stored as "dataset.txt" in the java project folder.

PROGRAM FLOW



5. IMPLEMENTATION

GraphNode.java

```
GraphNode(int l){
    label = l;
    adjList = new ArrayList<GraphNode>();
    weights = new ArrayList<Integer>();
    d = Integer.MAX_VALUE;
}
```

```
public void reset() {

    for(int j = 0; j<adjList.size(); j++){
        adjList.get(j).visited = false;
        adjList.get(j).d = Integer.MAX_VALUE;
    }
}
```

```
public void Dijkstra(ArrayDeque<GraphNode> q) {
    while(!q.isEmpty()) {
        GraphNode x = q.remove();
        for(int j=0; j<x.adjList.size(); j++) {
            GraphNode y = x.adjList.get(j);
            int weight = x.weights.get(j);
            if(!y.visited) {
                y.visited = true;
                q.add(y);
                int w = 0;
                if(weight-x.d<0)
                    w = 0;
                else
                    w = weight-x.d;
                y.d = w+x.d;
            }
            else if (y.visited) {
                int w = 0;
                if(weight-x.d<0)
                    w = 0;
                else
                    w = weight-x.d;
                if(y.d>(w+x.d))
                    y.d = w+x.d;
            }
        }
    }
}
```

Graph.java

```
public Graph(int n) {
    size = n;
    node = new GraphNode[size];
    for(int i=0; i<n; i++) {
        node[i] = new GraphNode(i);
    }
}
```

```
public void addEdge(int from, int to, int w) {
    node[from].adjList.add(node[to]);
    node[to].adjList.add(node[from]);
    node[from].weights.add(w);
    node[to].weights.add(w);
}
```

```
public void reset() {
    for(int i=0; i<size; i++) {
        node[i].visited = false;
        node[i].d = Integer.MAX_VALUE;
        node[i].reset();
    }
}
```

```
public void Dijkstra(int i) {
    reset();
    ArrayDeque<GraphNode> q = new ArrayDeque<GraphNode>();
    q.add(node[i]);
    node[i].visited = true;
    node[i].d = 0;
    node[i].Dijkstra(q);
}
```

```

public void getCost(int s) {
    Dijkstra(s);
    if(node[size-1].d==Integer.MAX_VALUE) {
        System.out.println("NO PATH EXIST");
    }
    else {
        System.out.println();
        System.out.println("*****");
        System.out.println("STATION CODE           : "+(node[s].label+1));
        System.out.println();
        System.out.println("DESTINATION CODE       : "+(node[size-1].label+1));
        System.out.println();
        System.out.println("MOST EFFICIENT PATH WILL COST : "+(node[size-1].d));
        System.out.println("*****");
        //System.out.println("Shortest path from "+(node[s].label+1)+" to "+(node[size-1].label+1)+": "+(node[size-1].d));
    }
}

```

Test.java

```

BufferedReader br = new BufferedReader(new FileReader(file));
String line1 = br.readLine();
String[] str = line1.trim().split(" ");
int n = Integer.parseInt(str[0]);
Graph g = new Graph(n);

```

```

String st;
while ((st = br.readLine())!= null) {
    String[] a = st.split(" ");
    g.addEdge(Integer.parseInt(a[0])-1,Integer.parseInt(a[1])-1,Integer.parseInt(a[2]));
}

System.out.println();
g.getCost(0);

```

Pair.java

```

Pair(int p,int q){
    this.a=p;
    this.b=q;
}

public int get_first() {
    return a;
}

public int get_second() {
    return b;
}

```


Data.java

```

int max_vertex,max_edges,max_weight;

System.out.print("Enter the max no. of Vertices: ");
max_vertex=mva.nextInt();
System.out.print("Enter the max no. of Edges   : ");
max_edges=mea.nextInt();
System.out.print("Enter the max Weight       : ");
max_weight=mwa.nextInt();

```

```

Random rand = new Random();
Set<Pair> container = new HashSet<Pair>();
int num = 1+ rand.nextInt(max_vertex);
int num_edge = 1+rand.nextInt(max_edges);
while(num_edge> (num*(num-1))/2)
    num_edge = 1+rand.nextInt(max_edges);
for (int j=1; j<=num_edge; j++) {
    int a = 1+ rand.nextInt(num);
    int b = 1+ rand.nextInt(num);
    Pair p = new Pair(a,b);
    Pair reverse_p = new Pair(b,a);
    while (container.contains(p) || container.contains(reverse_p)){
        a = 1+ rand.nextInt(num);
        b = 1+ rand.nextInt(num);
        p = new Pair(a,b);
        reverse_p = new Pair(b,a);
    }
    container.add(p);
}

```

```

FileWriter file = new FileWriter("dataset.txt");
BufferedWriter output = new BufferedWriter(file);

output.write(Integer.toString(num));
output.append(" ");
output.append(Integer.toString(num_edge));
output.append("\n");

for (Pair value : container) {
    int wt = 1+ rand.nextInt(max_weight);
    output.append(Integer.toString(value.get_first()));
    output.append(" ");
    output.append(Integer.toString(value.get_second()));
    output.append(" ");
    output.append(Integer.toString(wt));
    output.append("\n");
}
output.close();

```

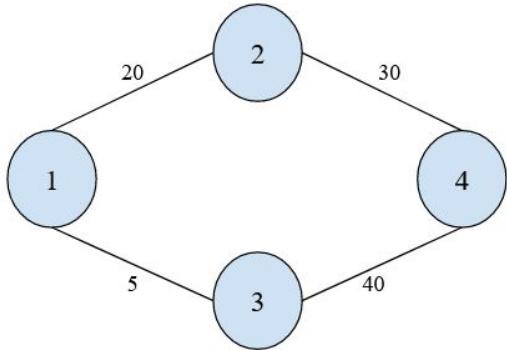
6. EXPERIMENTATION RESULT

Input 1

Number of Vertices: 4

Number of Edges : 4

Maximum Weight : 50

DATASET PREVIEW	GRAPH PREVIEW
<pre> 4 4 1 2 20 2 4 30 4 3 40 3 1 5 </pre>	 <pre> graph LR 1((1)) --- 20 2((2)) 2 --- 30 4((4)) 4 --- 40 3((3)) 3 --- 5 1 </pre>

Output 1

OUTPUT PREVIEW
<pre> ***** HELPING VIVEK FIND THE MOST COST EFFICIENT WAY ***** PLEASE ENTER THE PUBLIC TRANSPORT SYSTEM FARE DATASET: dataset.txt ***** STATION CODE : 1 DESTINATION CODE : 4 MOST EFFICIENT PATH WILL COST : 30 ***** </pre>

Input 2

Number of Vertices: 7

Number of Edges : 9

Maximum Weight : 50

DATASET PREVIEW	GRAPH PREVIEW
<pre> 7 9 1 2 10 2 3 15 3 4 19 3 7 20 4 2 18 4 1 20 4 5 25 5 6 30 6 7 35 </pre>	

Output 2

OUTPUT PREVIEW
<pre> ***** HELPING VIVEK FIND THE MOST COST EFFICIENT WAY ***** PLEASE ENTER THE PUBLIC TRANSPORT SYSTEM FARE DATASET: dataset.txt ***** STATION CODE : 1 DESTINATION CODE : 7 MOST EFFICIENT PATH WILL COST : 20 ***** </pre>

Input 3

Number of Vertices: 5

Number of Edges : 6

Maximum Weight : 100

DATASET PREVIEW	GRAPH PREVIEW
5 6 1 2 30 2 3 50 3 4 70 4 5 90 1 3 70 3 5 85	<pre> graph TD 1((1)) --- 30 2((2)) 1 --- 70 3((3)) 2 --- 50 3 3 --- 70 4((4)) 3 --- 85 5((5)) 4 --- 90 5 </pre>

Output 3

OUTPUT PREVIEW
<pre> ***** HELPING VIVEK FIND THE MOST COST EFFICIENT WAY ***** PLEASE ENTER THE PUBLIC TRANSPORT SYSTEM FARE DATASET: dataset.txt ***** STATION CODE : 1 DESTINATION CODE : 5 MOST EFFICIENT PATH WILL COST : 85 ***** </pre>

Input 4

Number of Vertices: 7

Number of Edges : 7

Maximum Weight : 50

DATASET PREVIEW	GRAPH PREVIEW
<pre> 7 7 1 2 10 2 3 15 3 4 19 4 2 18 4 1 20 4 5 25 6 7 35 </pre>	

Output 4

OUTPUT PREVIEW
<pre> ***** HELPING VIVEK FIND THE MOST COST EFFICIENT WAY ***** PLEASE ENTER THE PUBLIC TRANSPORT SYSTEM FARE DATASET: dataset.txt NO PATH EXIST </pre>

Input 5

Number of Vertices: 50000

Number of Edges : 500000

Maximum Weight : 10000000

Dataset Submitted with Report : sampleinput5.txt

Output 5

OUTPUT PREVIEW	
<pre> ***** HELPING VIVEK FIND THE MOST COST EFFICIENT WAY ***** PLEASE ENTER THE PUBLIC TRANSPORT SYSTEM FARE DATASET: dataset.txt ***** STATION CODE : 1 DESTINATION CODE : 50000 MOST EFFICIENT PATH WILL COST : 2592097 ***** </pre>	

Input 6

Random Dataset Generation with pair.java and data.java

Dataset Submitted with Report : sampleinput6.txt

DATASET GENERATION PREVIEW
<pre> Enter the max no. of Vertices: 20 Enter the max no. of Edges : 2000 Enter the max Weight : 10000 ***** File Created Successfully!!! ***** </pre>

Uses Random Function and Generates 11 Vertices and 21 Edges

Output 6

OUTPUT PREVIEW	
<pre> ***** HELPING VIVEK FIND THE MOST COST EFFICIENT WAY ***** PLEASE ENTER THE PUBLIC TRANSPORT SYSTEM FARE DATASET: dataset.txt ***** STATION CODE : 1 DESTINATION CODE : 11 MOST EFFICIENT PATH WILL COST : 1139 ***** </pre>	

Input 7

Random Dataset Generation with pair.java and data.java

Dataset Submitted with Report : sampleinput7.txt

DATASET GENERATION PREVIEW	
<pre> Enter the max no. of Vertices: 100 Enter the max no. of Edges : 10000 Enter the max Weight : 100000 ***** File Created Successfully!!! ***** </pre>	

Uses Random Function and Generates 45 Vertices and 560 Edges

Output 7

OUTPUT PREVIEW	
<pre> ***** HELPING VIVEK FIND THE MOST COST EFFICIENT WAY ***** PLEASE ENTER THE PUBLIC TRANSPORT SYSTEM FARE DATASET: dataset.txt ***** STATION CODE : 1 DESTINATION CODE : 45 MOST EFFICIENT PATH WILL COST : 15006 ***** </pre>	

Input 8

Random Dataset Generation with pair.java and data.java

Dataset Submitted with Report : sampleinput8.txt

DATASET GENERATION PREVIEW	
<pre> Enter the max no. of Vertices: 500 Enter the max no. of Edges : 60000 Enter the max Weight : 100000 ***** File Created Successfully!!! ***** </pre>	

Uses Random Function and Generates 443 Vertices and 36525 Edges

Output 8

OUTPUT PREVIEW	
<pre> ***** HELPING VIVEK FIND THE MOST COST EFFICIENT WAY ***** PLEASE ENTER THE PUBLIC TRANSPORT SYSTEM FARE DATASET: dataset.txt ***** STATION CODE : 1 DESTINATION CODE : 443 MOST EFFICIENT PATH WILL COST : 2230 ***** </pre>	

Input 9

Random Dataset Generation with pair.java and data.java

Dataset Submitted with Report : sampleinput9.txt

DATASET GENERATION PREVIEW	
<pre> Enter the max no. of Vertices: 20000 Enter the max no. of Edges : 500000 Enter the max Weight : 10000000 ***** File Created Successfully!!! ***** </pre>	

Uses Random Function and Generates 19311 Vertices and 175437 Edges

Output 9

OUTPUT PREVIEW	
<pre> ***** HELPING VIVEK FIND THE MOST COST EFFICIENT WAY ***** PLEASE ENTER THE PUBLIC TRANSPORT SYSTEM FARE DATASET: dataset.txt ***** STATION CODE : 1 DESTINATION CODE : 19311 MOST EFFICIENT PATH WILL COST : 2099816 ***** </pre>	

Input 10

Random Dataset Generation with pair.java and data.java

Dataset Submitted with Report : sampleinput10.txt

DATASET GENERATION PREVIEW	
<pre> Enter the max no. of Vertices: 45000 Enter the max no. of Edges : 500000 Enter the max Weight : 10000000 ***** File Created Successfully!!! ***** </pre>	

Uses Random Function and Generates 38177 Vertices and 118396 Edges

Output 10

OUTPUT PREVIEW	
<pre> ***** HELPING VIVEK FIND THE MOST COST EFFICIENT WAY ***** PLEASE ENTER THE PUBLIC TRANSPORT SYSTEM FARE DATASET: dataset.txt ***** STATION CODE : 1 DESTINATION CODE : 38177 MOST EFFICIENT PATH WILL COST : 5297487 ***** </pre>	

7. CONCLUSION

In general, we have done the implementation of Dijkstra's Algorithm for finding the shortest path in our project. Vivek, who was in need of help for finding the most cost efficient path for travelling from station A to the last station was thus helped with experimenting the proposed algorithm.

For implementing this work, we have gone through the concepts of Graph data structures, its implementation, adjacency lists, adjacency matrices and more as per needs and the concept of Shortest path was studied in detail for proposing this code.

The implementation has helped me study how to handle weights while dealing with nodes and edges. It also helped me refresh my knowledge with `BufferedReader` classes. The idea of implementation of these topics which was taken well in the class have helped me understand these topics and the real reasons why a particular data structure is used. And this has helped me well enough to judge and make decisions whether to follow a code or not based on the time complexities.

The dataset generation code and problem solution code both helped us all to understand more on the concepts of Object Oriented Programming as well as its necessity.

The algorithm was experimented with a variety of test cases, datasets. Both manually created ones and datasets generated from our `Data.java` and `Pair.java` classes produced a good and accurate result. It was successful in dealing with the maximum constraints provided with the problem.

This algorithm can be extended as a real world application for calculating fare policies as well as for many other applications.

8. REFERENCES

No.	REFERENCES
1	Class Notes and Lecture Videos
2	https://www.javatpoint.com/ds-graph
3	https://www.quora.com/What-is-the-indegree-and-outdegree-of-a-graph
4	https://www.geeksforgeeks.org/comparison-between-adjacency-list-and-adjacency-matrix-representation-of-graph/
5	https://www.oreilly.com/library/view/eclipse-ide-pocket/0596100655/ch01.html
6	https://www.geeksforgeeks.org/applications-of-dijkstras-shortest-path-algorithm/
7	https://www.hackerearth.com/practice/algorithms/graphs/shortest-path-algorithms/tutorial/
8	https://www.eclipse.org/downloads/packages/release/kepler/sr1/eclipse-ide-java-developers
9	https://www.geeksforgeeks.org/test-case-generation-set-4-random-directed-undirected-weighted-and-unweighted-graphs/

9. APPENDIX

PROBLEM SOLUTION

GraphNode.java

```
import java.util.*;

public class GraphNode {

    int label;
    ArrayList<GraphNode> adjList;
    ArrayList<Integer> weights;
    boolean visited;
    public int d;

    GraphNode(int l){
        label = l;
        adjList = new ArrayList<GraphNode>();
        weights = new ArrayList<Integer>();
        d = Integer.MAX_VALUE;
    }

    public void print() {
        for(int j=0; j<adjList.size(); j++) {
            System.out.print(adjList.get(j).label+"-->");
        }
        System.out.print("null");
    }

    public void reset() {

        for(int j = 0; j<adjList.size(); j++){
            adjList.get(j).visited = false;
            adjList.get(j).d = Integer.MAX_VALUE;
        }
    }

    public void Dijkstra(ArrayDeque<GraphNode> q) {
```

```

while(!q.isEmpty()) {
    GraphNode x = q.remove();
    for(int j=0; j<x.adjList.size(); j++) {
        GraphNode y = x.adjList.get(j);
        int weight = x.weights.get(j);
        if(!y.visited) {
            y.visited = true;
            q.add(y);
            int w = 0;
            if(weight-x.d<0)
                w = 0;
            else
                w = weight-x.d;
            y.d = w+x.d;
        }
        else if (y.visited) {
            int w = 0;
            if(weight-x.d<0)
                w = 0;
            else
                w = weight-x.d;
            if(y.d>(w+x.d))
                y.d = w+x.d;
        }
    }
}
}
}
}

```

Graph.java

```

import java.util.*;

public class Graph {
    GraphNode[] node;
    int size;

    public Graph(int n) {
        size = n;
    }
}

```

```

        node = new GraphNode[size];
        for(int i=0; i<n; i++) {
            node[i] = new GraphNode(i);
        }
    }

    public void addEdge(int from, int to, int w) {
        node[from].adjList.add(node[to]);
        node[to].adjList.add(node[from]);
        node[from].weights.add(w);
        node[to].weights.add(w);
    }

    public void print() {
        for(int i=0; i<this.size; i++) {
            System.out.print(i+": ");
            node[i].print();
            System.out.println();
        }
    }

    public void reset() {
        for(int i=0; i<size; i++) {
            node[i].visited = false;
            node[i].d = Integer.MAX_VALUE;
            node[i].reset();
        }
    }

    public void Dijkstra(int i) {
        reset();
        ArrayDeque<GraphNode> q = new ArrayDeque<GraphNode>();
        q.add(node[i]);
        node[i].visited = true;
        node[i].d = 0;
        node[i].Dijkstra(q);
    }

    public void getCost(int s) {
        Dijkstra(s);
    }

```



```

        if(node[size-1].d==Integer.MAX_VALUE) {
            System.out.println("NO PATH EXIST");
        }
        else {
            System.out.println();

System.out.println("*****
*****
*****");
            System.out.println("STATION CODE          :
" +(node[s].label+1));
            System.out.println();
            System.out.println("DESTINATION CODE      :
" +(node[size-1].label+1));
            System.out.println();
            System.out.println("MOST EFFICIENT PATH WILL COST :
" +(node[size-1].d));

System.out.println("*****
*****
*****");
            //System.out.println("Shortest path from " +(node[s].label+1)+
to " +node[size-1].label+1+": " +node[size-1].d);
        }
    }
}

```

Test.java

```

import java.util.*;
import java.io.*;

public class Test {
    private static Scanner in;
    public static void main(String[]args) throws IOException{

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

```

```

        System.out.println();
        System.out.println("
FIND THE MOST COST EFFICIENT WAY
HELPING VIVEK
");

System.out.println("*****
*****
*****");
        in = new Scanner(System.in);
        System.out.println();
        System.out.print("PLEASE ENTER THE PUBLIC TRANSPORT
SYSTEM FARE DATASET: ");
        String s = in.nextLine();

        File file = new File(s);

        @SuppressWarnings("resource")
        BufferedReader br = new BufferedReader(new FileReader(file));
        String line1 = br.readLine();
        String[] str = line1.trim().split(" ");
        int n = Integer.parseInt(str[0]);
        Graph g = new Graph(n);

        String st;
        while ((st = br.readLine()) != null) {
            String[] a = st.split(" ");

            g.addEdge(Integer.parseInt(a[0])-1,Integer.parseInt(a[1])-1,Integer.parseInt(a[2]));
        }

        //            g.print();

        System.out.println();
        g.getCost(0);
    }
}

```

DATASET GENERATOR

Pair.java

```
public class Pair{
    int a;
    int b;

    Pair(int p,int q){
        this.a=p;
        this.b=q;
    }

    public int get_first() {
        return a;
    }

    public int get_second() {
        return b;
    }
}
```

Data.java

```
import java.io.*;
import java.util.*;

public class Data {
    private static Scanner mva;
    private static Scanner mea;
    private static Scanner mwa;

    public static void main(String[]args) throws IOException {

        mva = new Scanner(System.in);
        mea = new Scanner(System.in);
        mwa = new Scanner(System.in);

        int max_vertex,max_edges,max_weight;

        System.out.print("Enter the max no. of Vertices: ");
```

```

max_vertex=mva.nextInt();
System.out.print("Enter the max no. of Edges  : ");
max_edges=mea.nextInt();
System.out.print("Enter the max Weight    : ");
max_weight=mwa.nextInt();

```

```

Random rand = new Random();

```

```

Set<Pair> container = new HashSet<Pair>();
int num = 1+ rand.nextInt(max_vertex);
int num_edge = 1+rand.nextInt(max_edges);

```

```

while(num_edge> (num*(num-1))/2)
    num_edge = 1+rand.nextInt(max_edges);

```

```

for (int j=1; j<=num_edge; j++) {
    int a = 1+ rand.nextInt(num);
    int b = 1+ rand.nextInt(num);
    Pair p = new Pair(a,b);
    Pair reverse_p = new Pair(b,a);

    while (container.contains(p) || container.contains(reverse_p)){
        a = 1+ rand.nextInt(num);
        b = 1+ rand.nextInt(num);
        p = new Pair(a,b);
        reverse_p = new Pair(b,a);
    }
    container.add(p);
}

```

```

FileWriter file = new FileWriter("dataset.txt");
BufferedWriter output = new BufferedWriter(file);

```

```

output.write(Integer.toString(num));
output.append(" ");
output.append(Integer.toString(num_edge));
output.append("\n");

```

```

for (Pair value : container) {

```

```

        int wt = 1+ rand.nextInt(max_weight);
        //System.out.format("%d %d %d\n", value.get_first(),
value.get_second(), wt);
        output.append(Integer.toString(value.get_first()));
        output.append(" ");
        output.append(Integer.toString(value.get_second()));
        output.append(" ");
        output.append(Integer.toString(wt));
        output.append("\n");
    }
    output.close();

    System.out.println();

    System.out.println("*****
*");
    System.out.println("  File Created Successfully!!!");

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

    }

}

```
