

CS 5592, SPRING SEMESTER 2017

DESIGN AND ANALYSIS OF ALGORITHMS

DIJKSTRA'S SHORTEST UNCERTAIN PATH ALGORITHM *PROJECT REPORT*

TEAM MEMBERS

NIKHILA CHINTHAKINDHI – 16230697 – NCWF9

SESHA SAI PRASANNA CHENNUPATI – 16233063 – SPCGKD

SWAROOP CHITTIPROLU – 16230558 – SC6F6

Declaration of Academic Honesty

We hereby confirm that the project is solely our own work and that if any of the content from books, papers, or any other web resources have been copied or in any other way used, all references – have been acknowledged and fully cited.


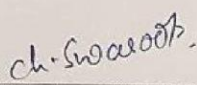
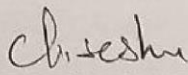
Team Sign-up Sheet
Dijkstra's Shortest Uncertain Path Algorithm
CS5592 Project, Spring Semester 2017
Due: Friday, April 14, 2017 (or earlier)
Each team has two or three members

sign up sheet
due
3/23/17
Thursday

(Choose a) Team name: **EVIL MASTER MINDS**

We are jointly committed to bring this project to a successful completion. We will work as a team, fully analyze and discuss various design options amongst ourselves and help each other with the implementation, the analysis of the collected data and with the writing of the report. Individually, each of us is prepared to help our team members, if needed, and go beyond our fair share.

We understand and we will adhere to the rules regarding student conduct. In particular, any and all material, including algorithms and programs, have been produced and written by members of our team, or are free and publicly available and will be appropriately cited. We understand that a violation of the code of conduct will result in a zero (0) for this assignment, and that the situation will be discussed and forwarded to the Academic Dean of the School for any potential follow up action. It could result in being expelled from the university. With our signatures, we accept full responsibility for the joint work for this project. (Please sign above each line)

1:	NIKHILA CHINTHAKINDHI		16230697
	First & Last Name	Signature	UMKC-Id
2:	Swaroop Chitiprolu		16230558
	First & Last Name	Signature	UMKC-Id
3:	Sesha Sai Prasanna Chennupati		16230673
	First & Last Name	Signature	UMKC-Id

ABSTRACT

Dijkstra's algorithm is used to find the shortest distances between nodes in a graph which can be interpreted as Road networks in Real world. Dijkstra's algorithm was developed by Edsger W.Dijkstra in the year 1956.

Basically, Dijkstra's algorithm comes in many variants, the original one is used to find the shortest distance between two nodes but the most commonly used variant of algorithm fixes a single node as the source node and finds the shortest paths from the source node to all other nodes in the graph.

For this project, we consider the nodes of the graph as cities and the edge costs represent the driving distances between the cities.

The working principle is very simple, at the start mark the distance to all the nodes as infinity. In this case, we want to find the shortest path between two cities let's say Chicago and Kansas City. In the first case, we make the distance from Chicago to all other nodes as infinity. This is done to show that the nodes have not yet been visited.

Now for each iteration, select a current node. For the first iteration, the node at which we are currently there will be the current node and the distance to it will be zero. For the next upcoming iterations, the current node will be the closet unvisited node to the starting point. From the current node update the distance to every unvisited node that is directly connected to it. The node is relabeled if the path to it through the current node is shorter than the previously known paths. After updating the distances to each neighboring node, mark the current node as visited and select the unvisited node with lowest distance as the current node.

Nodes that are marked as visited are labeled with the shortest path from the starting point and will not be revisited. Continue this process of updating the neighboring nodes with the shortest distances, then marking the current nodes as visited and moving onto the closest unvisited nodes until you have marked the destination as visited. Once you have marked the destination as visited in this case Kansas City you have determined the shortest path to it, from the starting point, and the path can be traced back.

CONTENTS

1. INTRODUCTION
2. CRITERIA IMPLEMENTED
 - 2.1. CRITERIA 1 – MEAN VALUE
 - 2.2. CRITERIA 2 – OPTIMIST
 - 2.3. CRITERIA 3 – PESSIMIST
 - 2.4. CRITERIA 4 – DOUBLE PESSIMIST
 - 2.5. CRITERIA 5 – STABLE
 - 2.6. CRITERIA 6 – MEAN+CSQUARE
3. EXPERIMENTAL IMPLEMENTATION
 - 3.1. PROGRAMMING LANGUAGE USED
 - 3.2. REASON FOR CHOOSING JAVA
 - 3.3. DATA STRUCTURES USED
 - 3.4. PROGRAMMATIC IMPLEMENTATION DETAILS
4. DATA COLLECTION AND INTERPRETATION OF RESULTS
 - 4.1. MANUAL INTERPRETATION
 - 4.2. PROGRAMMATIC INTERPRETATION WITH UNIT TEST CASES
AND RESULTS
5. CONCLUSION
6. EPILOUGE
7. APPENDIX
8. REFERENCES

1. INTRODUCTION

In the present-day world, any means of transportation like Road, Airways etc... are interpreted using graphs. Not only transportation but also all the communication networks like telephone, internet, routing, postal service etc.... are also modeled as graphs. The basic things we need to have if we want to establish a connection between two points are source, destination and path. But for an effective communication while communicating or to travel safely from one point to other two things effect one is time and the other is shortest path.

Let us explain this in terms of a road network between cities. For example, let's say we want to travel from Kansas City to Chicago. So, Kansas City will be our source and Chicago will be our destination. There will be many places between Kansas City and Chicago and let's consider them as intermediate nodes. Everyone chooses the shortest path to reach their destination fast. But to find the shortest path we must take a note that there are many paths between Chicago and Kansas City and the same path will not be the shortest at all times. We should also know that if every user chooses the shortest path then there would be a congestion in that path.

The most important thing we need to consider is that some shortest paths might be stopped due to an accident or some other issue and we should not recommend such paths to users which means we need to take the data dynamically.

So, finally we must give the user shortest of all paths and avoid congestion in shortest paths due to overflow of vehicles and if needed skip the congested paths and give the paths with less flow of vehicles. Though the path might be long but it gives the user to reach his destination fast.

Every Scenario we consider will have the following

- The size n of the system (the total number of cities), together with the index of two cities, a and b . Here n represents the number of cities between source and destination.
- An adjacency matrix E with edge-weights. The (i, j) th entry $E[i, j]$ represents the main characteristics of the random variable representing the traveling time on the edge between the two specific nodes, i and j , assuming that there are no other cars on the edge that might slow you down. Here I and j represent source and destinations.

We should not forget that we are not the only car on the road, and with more cars on the road will cause interference and extra delay because you share the road with more and more others. In other words: the delay along an edge is rarely a constant value, it is uncertain.

We still need to find some sort of shortest path, but shortest is now defined in a stochastic setting. We call such a path a Uncertain Shortest Path.

2. CRITERIAS IMPLEMENTED

The main objective of this project of this project is to find the shortest path and uncertain path between two nodes a and b from the above example we can say between Kansas City and Chicago. Sometimes it's enough to find the shortest path but in case of congestion the shortest path will not be enough due to high volume of vehicles in that path and we need to find the next shortest path which we call it as uncertain path.

So, in order to do this project, we actually have six criteria's, they are listed below.

2.1 CRITERIA 1- MEAN VALUE:

The main characteristics provided in this project are T (the type of distribution), together with two more parameters, α and β , whose interpretation depends on the type T.

Type	Name	Parameter 1 α	Parameter 2 β	Mean	Variance	c^2
1	Deterministic	at value	not used (ignore value)	α	0	0
2	Uniform $[\alpha, \beta]$	low value	high value	$\frac{\alpha+\beta}{2}$	$\frac{(\beta-\alpha)^2}{12}$	$\frac{1}{3}(\frac{\beta-\alpha}{\alpha+\beta})^2$
3	Exponential	mean rate	not used (ignore value)	$\frac{1}{\alpha}$	$\frac{1}{\alpha^2}$	1
4	Shifted Exponential	mean rate	shifted value	$\beta + \frac{1}{\alpha}$	$\frac{1}{\alpha^2}$	$(\frac{1}{1+\beta\alpha})^2$
5	Normal	location value	variance	α	β	$\frac{\beta}{\alpha^2}$
6	General	Mean value	squared coefficient of variation	α	$\alpha^2 \beta$	β

Fig: 1- Formulae for various types of distributions

So, from the above table we can see that there are different types of distributions given namely Deterministic, Uniform, Exponential etc....

We also have a sample input data given which is in the form “3, 4, 2, 10, 20” where 3 and 4 are the nodes and 2 is the type of distribution between them and 10, 20 are the alpha and beta values respectively.

So, first we need to find the Mean, variance and squared coefficient of variance for all the input values by substituting them in the equations given above. This should be done by using the code written and later be verified.

Now we need to find the shortest distance using the mean we calculated and this exactly will be the shortest distance between two nodes without any approximations.

Now to understand about the next two cases we need to have a thorough understanding of Standard Deviation which is square root of variance.

Let us assume there are two points A and B and we need to travel between them. We have already made ‘n’ trips between them and in addition to travelling we have also measured the duration of each trip we made. The duration is likely to be different for each trip. But, however the measured duration is likely to be between optimistic and pessimistic values.

So, from the above we can say that Optimistic value is the least amount time taken to travel between two points and Pessimist value is the highest amount of time taken to travel between two points.

The maximum number of observations would be near Mean but there would be few observations near optimistic and pessimistic values. If we plot all the observations we made with it would look like a Bell curve where most of the observations would be near Mean.

Let us assume that we have to make one more trip from point A to point B and someone asked us what would be the estimated time for this trip? Now, we have a range of values we can provide or we can provide only the mean value. A good estimate would be the mean value but our next trip could be easily more or less than mean value due to different factors.

So, to be on the safe side it is better to give a range of values as estimate. So, for a given path it is safe to determine a Range of values for example 150 ± 30 minutes instead of giving single estimate.

Secondly, we can also determine the probability associated with the range of values. It means we can determine the probability of completing the trip in 150 ± 30 minutes. So, this is all about standard deviation and why we use it to find the distance alongside mean.

2.2 CRITERIA 2- OPTIMIST:

The optimist path is nothing but the path in which it takes a minimum amount of time to travel between two nodes. It is given by

$$\textit{Expected value} - \textit{Standard Deviation}$$

Here let's take an example. Let's consider a and b as source and destination nodes respectively. Now we need to find the optimistic path. Optimistic path indicates that the duration in traffic should be shorter than the actual travel time on most days, though occasional days with particularly good traffic conditions may be faster than this value.

Now if a person wants to go from point a to point b. The algorithm calculates the shortest path which will be different from the Actual shortest path and gives it to us. If the user uses this path me might not have any traffic while travelling. It calculates the optimistic path based on the traffic data provided dynamically. The optimistic model should be as accurate as possible to prevent too many mistakes and corrections. An overly optimistic model would assume that every state transition will lead to a state with the highest reward.

For example, in communication when a node is evaluating its options to find the best possible path to obtain an object it might not reserve all the resources along all candidate paths hoping that the selected path will not be occupied by other resources. By doing this, this node is running the risk of finding the resources reserved by other nodes when the path is selected. This is an Optimistic approach.

The optimistic path is much faster.

2.3 CRITERIA 3- PESSIMIST:

The Pessimistic path is nothing but the path in which it takes a maximum amount of time to travel between two nodes. It is given by

$$\textit{Expected value} + \textit{Standard Deviation}$$

Here let's take an example. Let's consider a and b as source and destination nodes respectively. Now we need to find the Pessimist path. if a person wants to go from point a to point b. The algorithm calculates the shortest path which will be different from the Actual shortest path and gives it to us. If the user uses this path he might not have any traffic while travelling. It calculates the Pessimist path based on the traffic data provided dynamically. The value of the pessimist path will be higher than the Actual value. It gives us the longest distance of all.

For example, in communication when a node is evaluating its options to find the best possible path to obtain an object it might reserve all the resources along all candidate paths so that other requests cannot grab the resources. By doing this, this node will have extra resources than the other nodes preventing them to use these resources. This is a pessimistic approach.

As the name, itself indicates it gives us the path in which a user can travel fast by taking less time. **The pessimist value is more reliable than optimistic.** It is better to use optimistic model than the pessimistic model. Using an optimistic model will lead to discovery of a more accurate model of the environment or to discovery of better paths. A pessimistic model would assume that no better path exists and would miss it.

The optimistic model should be as accurate as possible to prevent too many mistakes and corrections. An overly optimistic model would assume that every state transition will lead to a state with the highest reward. A model could aim to be more accurate at the risk to become pessimistic in some states. That is a risk we accept when we don't want to explore all states in unlimited state space.

2.4 CRITERIA 4- DOUBLE PESSIMIST:

The Double Pessimist path is nothing but the path in which it takes a maximum amount of time to travel between two nodes. It is given by

$$\textit{Expected value} + 2*\textit{Standard Deviation}$$

It's the same as Pessimist but here we multiply standard deviation by 2 and then add it to the expected value.

2.5 CRITERIA 5- STABLE:

In criteria 5 we used the squared coefficient of variation that is c^2 and found out the shortest path. Basically, the value of c^2 is very less and it is because of that it is called stable path. However, we manipulate the value of c^2 the result would be very minimum. So, we can say that the path obtained from this criterion is stable.

2.6 CRITERIA 6- MEAN+CSQUARE:

In this criterion, we calculate the value of MEAN+CSQUARE. By using this criterion, we can make sure that the path we obtained will be more stable than the previous one.

3. EXPERIMENTAL IMPLEMENTATION

3.1 Programming language used: JAVA

3.2 Reason for choosing Java:

The main reasons for choosing java to code our project are listed below:

- ✦ Basically, Java is well known for its Efficiency and Flexibility.
- ✦ It has garbage collection, this means that the Java compiler will catch many errors that dynamically typed languages won't (until runtime).
- ✦ In java, there's no need to deal with segmentation faults.
- ✦ It has many references, libraries and rich collections.

3.3 Data Structures used:

Graph and Tree Set are the data structures used. Building the graph from a set of edges takes $O(E \log V)$ for each pass. Vertices are stored in a Tree Set (self-balancing binary search tree) instead of a Priority Queue (a binary heap) in order to get $O(\log n)$ performance for removal of any element, not just the head. Decreasing the distance of a vertex is accomplished by removing it from the tree and later re-inserting it. With adjacency list representation, all vertices of a graph can be traversed in a total running time of $O((V+E) \log V)$.

3.4 Programmatic implementation details:

The programmatic implementation of this project is as follows:

- ✦ The main method of the program is as follows. It first call a method that deals with user input and then loops in the sub-program for 6 times to display shortest paths for 6 criterion. Functions performed by the loop includes calculating mean, variance and c square for all edges and passing those values to the graph. Start and end nodes from the input are passed to dijkstra and printpaths. After all graph calculations are performed, shortest paths are displayed. Two more tables are displayed here where in edges used by different criteria are shown and various paths are compared in the second table.

```

public static void main(String[] args) throws NumberFormatException, IOException {
    acceptInputs(); //Stores input in a map
    for (int i = 0; i < 6; i++) {
        criteriaCount++;
        hopCount = 0;
        calculateValues(); //Calculates mean, variance and cSquare for edges
        Graph.Edge[] GRAPH = new Graph.Edge[edgesMap.size()];
        int graphCount = 0;
        //Assign values to graph
        for (String key: edgesMap.keySet()) {
            GRAPH[graphCount] = new Graph.Edge(edgesMap.get(key).get(0), edgesMap.get(key).get(1), Double.parseDouble(edgesMap.get(key).get(2)
            graphCount++;
        }
        edgesMap.clear();
        Graph g = new Graph(GRAPH);

        g.dijkstra(START.trim()); //Passing start vertex
        g.printPath(END.trim()); //Passing end vertex

        System.out.println("Number of hops for the path traversed above is " + DijkstrasAlgorithm.hopCount);
    }
    linksUsedByPaths(); //Displays edges and criterias using those edges
    System.out.println("\n \t\t \u03BC \t\t \u03BC-\u03C3 \t\t \u03BC+\u03C3 \t\t \u03BC+2*\u03C3 \t\t CSq \t\t Mean+CSq");
    comparePathValues();
}

```

- ✦ AcceptInputs() method reads graph input provided by the user in the console. Start and end nodes are assigned from the line that has 3 fields in the input. Edges and their type with alpha, beta values are stored in hashmap. Errors are displayed for invalid inputs. These are shown in Test cases in section 4. Code for this method is as follows:

```

private static void acceptInputs() throws IOException {
    int inputCount = 0;
    System.out.println("Enter graph inputs");
    BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
    String line;
    while ((line = stdin.readLine()) != null && line.length() != 0) {
        String[] input = line.split(",");
        // Input's first line where start and end nodes are given
        if (input.length == 3) {
            inputStartEndCount++;
            if (inputStartEndCount == 1) {
                START = input[1];
                END = input[2];
            } else {
                System.err.printf("\n Start and end nodes are given more than once. Please verify your input");
                System.exit(1);
            }
        }
        // Input's other lines where edges with type, alpha, beta are provided
    } else if (input.length == 6) {
        char inputFirstElement = input[0].charAt(0);
        if (inputFirstElement == 'E') {
            node1 = input[1];
            node2 = input[2];
            type = Integer.parseInt(input[3]);
            alpha = Double.parseDouble(input[4]);
            beta = Double.parseDouble(input[5]);
            inputCount++;
            ArrayList < String > inputEdges = new ArrayList < String > ();
            inputEdges.add(node1);
            inputEdges.add(node2);
            inputEdges.add(Integer.toString(type));
            inputEdges.add(Double.toString(alpha));
            inputEdges.add(Double.toString(beta));
        }
    }
}

```

```

        inputMap.put(Integer.toString(inputCount), inputEdges);
    } else {
        System.err.printf("\n Invalid input graph edge");
        System.exit(1);
    }
} else {
    System.err.printf("\n Improper graph input. Please verify and try again.");
    System.exit(1);
}
}
}
}

```

- ✦ CalculateValues() calculates mean, variance and c square for 6 criterion by looping in for 6 times as shown below and stores that result into hashmap. This hashmap is used to send edges and their corresponding weights data into Graph:

```

private static void calculatevalues() throws NumberFormatException, IOException {
    if (edgesMap.isEmpty()) {
        graphEdgesCount = 0;
        for (Entry < String, ArrayList < String >> entry: inputMap.entrySet()) {
            ArrayList < String > value = entry.getValue();
            double edgeWeight = 0;
            double standDeviation=0;
            int inputType = Integer.parseInt(value.get(2));
            switch (inputType) {
                case 1:
                    node1 = value.get(0);
                    node2 = value.get(1);
                    alpha = Double.parseDouble(value.get(3));
                    beta = Double.parseDouble(value.get(4));
                    mean = alpha;
                    variance = 0;
                    cSquare = 0;
                    standDeviation= (double) Math.sqrt(variance);
                    edgeWeight=criteriaBasedEdgeWeightsCal(node1, node2, criteriaCount,mean,cSquare,standDeviation);

                    ArrayList < String > alEdges1 = new ArrayList < String > ();
                    alEdges1.add(node1);
                    alEdges1.add(node2);
                    alEdges1.add(Double.toString(edgeWeight));
                    graphEdgesCount++;
                    edgesMap.put(Integer.toString(graphEdgesCount), alEdges1);
                    break;

```

This switch case is repeated for 6 times. criteriaBasedEdgeWeightsCal method is called here that provides edge weights for various criteria.

```

private static double criteriaBasedEdgeWeightsCal(String node1, String node2, int criteriaCount, double mean, double cSquare2, double
    double tempEdgeWeight=0;
    switch (criteriaCount) {
    case 1:
        tempEdgeWeight = mean;
        break;
    case 2:
        tempEdgeWeight = mean-standDeviation;
        break;
    case 3:
        tempEdgeWeight = mean + standDeviation;
        break;
    case 4:
        tempEdgeWeight = mean + 2 * standDeviation;
        break;
    case 5:
        tempEdgeWeight = cSquare;
        break;
    case 6:
        tempEdgeWeight = mean + cSquare;
        break;
    default:
        break;
    }
    double tempValueStore=0;
    if(DijkstrasAlgorithm.criteriaCount==6){
        ArrayList < String > alValues = new ArrayList < String > ();
        valuesCount++;
        alValues.add(node1);
        alValues.add(node2);
        tempValueStore=mean;

        tempValueStore=mean;
        alValues.add(Double.toString(tempValueStore));
        tempValueStore=mean-standDeviation;
        alValues.add(Double.toString(tempValueStore));
        tempValueStore=mean+standDeviation;
        alValues.add(Double.toString(tempValueStore));
        tempValueStore=mean+ 2* standDeviation;
        alValues.add(Double.toString(tempValueStore));
        tempValueStore= cSquare ;
        alValues.add(Double.toString(tempValueStore));
        tempValueStore=mean + cSquare ;
        alValues.add(Double.toString(tempValueStore));
        valuesMap.put(Integer.toString(valuesCount), alValues);
    }
    return tempEdgeWeight;
}
}

```

- ✦ Now, the values obtained from above methods are stored in a map and assigned to Graph as shown in main method which in turn is implemented as follows:

```

class Graph {
    private static Map < String, Vertex > graph; // mapping of vertex names to Vertex objects, built from a set of Edges

    /** One edge of the graph (only used by Graph constructor) */
    public static class Edge {
        public final String v1, v2;
        public final double dist;
        public Edge(String v1, String v2, double d) {
            this.v1 = v1;
            this.v2 = v2;
            this.dist = d;
        }
    }

    /** One vertex of the graph, complete with mappings to neighbouring vertices */
    public static class Vertex implements Comparable < Vertex > {
        public final String name;
        public double dist = Double.MAX_VALUE; // MAX_VALUE assumed to be infinity
        public Vertex previous = null;
        public final Map < Vertex,
            Double > neighbours = new HashMap < > ();

        public Vertex(String name) {
            this.name = name;
        }
    }

    private void printPath() {
        if (this == this.previous) {
            switch(DijkstraAlgorithm.criteriaCount){
                case 1:
                    DijkstraAlgorithm.shortestPathVar1 = DijkstraAlgorithm.shortestPathVar1.concat(this.name).concat(",");
                    break;
                case 2:
                    DijkstraAlgorithm.shortestPathVar2 = DijkstraAlgorithm.shortestPathVar2.concat(this.name).concat(",");
                    break;
                case 3:
                    DijkstraAlgorithm.shortestPathVar3 = DijkstraAlgorithm.shortestPathVar3.concat(this.name).concat(",");
                    break;
                case 4:
                    DijkstraAlgorithm.shortestPathVar4 = DijkstraAlgorithm.shortestPathVar4.concat(this.name).concat(",");
                    break;
                case 5:
                    DijkstraAlgorithm.shortestPathVar5 = DijkstraAlgorithm.shortestPathVar5.concat(this.name).concat(",");
                    break;
                case 6:
                    DijkstraAlgorithm.shortestPathVar6 = DijkstraAlgorithm.shortestPathVar6.concat(this.name).concat(",");
                    break;
                default:
                    break;
            }
        }
        System.out.printf("%s", this.name);
    }
    else if (this.previous == null) {
        System.out.printf("%s(unreached)", this.name);
    }
    else {
        this.previous.printPath();
        DijkstraAlgorithm.hopCount++;
        switch(DijkstraAlgorithm.criteriaCount){

```



```

        DijkstrasAlgorithm.shortestPathVar1 = DijkstrasAlgorithm.shortestPathVar1.concat(this.name).concat(",");
        break;
    case 2:
        DijkstrasAlgorithm.shortestPathVar2 = DijkstrasAlgorithm.shortestPathVar2.concat(this.name).concat(",");
        break;
    case 3:
        DijkstrasAlgorithm.shortestPathVar3 = DijkstrasAlgorithm.shortestPathVar3.concat(this.name).concat(",");
        break;
    case 4:
        DijkstrasAlgorithm.shortestPathVar4 = DijkstrasAlgorithm.shortestPathVar4.concat(this.name).concat(",");
        break;
    case 5:
        DijkstrasAlgorithm.shortestPathVar5 = DijkstrasAlgorithm.shortestPathVar5.concat(this.name).concat(",");
        break;
    case 6:
        DijkstrasAlgorithm.shortestPathVar6 = DijkstrasAlgorithm.shortestPathVar6.concat(this.name).concat(",");
        break;
    default:
        break;
    }
    System.out.printf(" -> %s(%f)", this.name, this.dist);
}
}

public int compareTo(Vertex other) {
    if (dist == other.dist)
        return name.compareTo(other.name);

    return Double.compare(dist, other.dist);
}

@Override public String toString() {
    return "(" + name + ", " + dist + ")";
}
}

```

- ✦ The following code builds the required graph, searches for start and end vertices, and sets edges with weights.

```

/** Builds a graph from a set of edges */
public Graph(Edge[] edges) {
    graph = new HashMap < > (edges.length);

    //one pass to find all vertices
    for (Edge e: edges) {
        if (!graph.containsKey(e.v1)) graph.put(e.v1, new Vertex(e.v1));
        if (!graph.containsKey(e.v2)) graph.put(e.v2, new Vertex(e.v2));
    }

    //another pass to set neighbouring vertices
    for (Edge e: edges) {
        //graph.get(e.v1).neighbours.put(graph.get(e.v2), e.dist);
        graph.get(e.v1).neighbours.put(graph.get(e.v2), e.dist);
        graph.get(e.v2).neighbours.put(graph.get(e.v1), e.dist); // for undirected graph
    }
}

```


- ✦ Start node is passed to the following method that runs dijkstra's

```
/** Runs dijkstra using a specified source vertex */
public void dijkstra(String startName) {
    if (!graph.containsKey(startName)) {
        System.err.printf("\n Graph doesn't contain start vertex " + startName);
        System.exit(1);
    }
    final Vertex source = graph.get(startName);
    NavigableSet < Vertex > q = new TreeSet < > ();

    // set-up vertices
    for (Vertex v: graph.values()) {
        v.previous = v == source ? source : null;
        v.dist = v == source ? 0 : Double.MAX_VALUE;
        q.add(v);
    }

    dijkstra(q);
}
```

- ✦ The following is the implementation of dijkstras algorithm using binary heap:

```
private static void linksUsedByPaths() {
    String inputEdges="";
    String criteria1="";
    String criteria2="";
    String criteria3="";
    String criteria4="";
    String criteria5="";
    String criteria6="";
    System.out.println("\nEdges \t\t MV \t Op \t Ps \t DP \t St \t Own");
    for (String key: inputMap.keySet()) {
        inputEdges="";
        inputEdges=inputMap.get(key).get(0) + "," + inputMap.get(key).get(1);
        criteria1="";
        criteria2="";
        criteria3="";
        criteria4="";
        criteria5="";
        criteria6="";
        if (shortestPathVar1.contains(inputEdges)) {
            criteria1="*";
        }
        if (shortestPathVar2.contains(inputEdges)) {
            criteria2="*";
        }
        if (shortestPathVar3.contains(inputEdges)) {
            criteria3="*";
        }
        if (shortestPathVar4.contains(inputEdges)) {
            criteria4="*";
        }
        if (shortestPathVar5.contains(inputEdges)) {
            criteria5="*";
        }
    }
}
```

```

/** Implementation of dijkstra's algorithm using a binary heap. */
private void dijkstra(final NavigableSet < Vertex > q) {
    Vertex u, v;
    while (!q.isEmpty()) {

        u = q.pollFirst(); // vertex with shortest distance (first iteration will return source)
        if (u.dist == Double.MAX_VALUE) break; // we can ignore u (and any other remaining vertices) since they are unreachable

        //look at distances to each neighbour
        for (Map.Entry < Vertex, Double > a: u.neighbours.entrySet()) {
            v = a.getKey(); //the neighbour in this iteration

            final double alternateDist = u.dist + a.getValue();
            if (alternateDist < v.dist) { // shorter path to neighbour found
                q.remove(v);
                v.dist = alternateDist;
                v.previous = u;
                q.add(v);
            }
        }
    }
}

```

- ✦ The below method lists all edges and highlights the edges used by mean values, optimist, pessimist, double pessimist, stable and own paths.

```

    }
    if (shortestPathVar6.contains(inputEdges)) {
        criteria6="**";
    }
    System.out.println("Edge(" + inputMap.get(key).get(0) + "," + inputMap.get(key).get(1) + ") "
        + "\t" + criteria1
        + "\t" + criteria2
        + "\t" + criteria3
        + "\t" + criteria4
        + "\t" + criteria5
        + "\t" + criteria6);
}
}

```

- ✦ Various path values are compared using the method shown below:

```

private static void comparePathValues() {
    String tempCompare="";
    String shortestPathVar="";
    float criteria1=0;
    float criteria2=0;
    float criteria3=0;
    float criteria4=0;
    float criteria5=0;
    float criteria6=0;
    for(int i=1;i<7;i++){

        switch (i) {
            case 1:
                shortestPathVar="";
                shortestPathVar=shortestPathVar1;
                System.out.println("\nMean\t");
                break;
            case 2:
                shortestPathVar="";
                shortestPathVar=shortestPathVar2;
                System.out.println("Opt\t");
                break;
            case 3:
                shortestPathVar="";
                shortestPathVar=shortestPathVar3;
                System.out.println("Pmst\t");
                break;
            case 4:
                shortestPathVar="";
                shortestPathVar=shortestPathVar4;
                System.out.println("DPess\t");
                break;

            case 5:
                shortestPathVar="";
                shortestPathVar=shortestPathVar5;
                System.out.println("Stbl\t");

                break;
            case 6:
                shortestPathVar="";
                shortestPathVar=shortestPathVar6;
                System.out.println("M+SQq\t");

                break;
            default:
                break;
        }
        for (String key: valuesMap.keySet()) {
            tempCompare=valuesMap.get(key).get(0) + "," + valuesMap.get(key).get(1);
            if(shortestPathVar.contains(tempCompare)){
                criteria1= (float) (criteria1+ Double.parseDouble(valuesMap.get(key).get(2))) ;
                criteria2= (float) (criteria2+ Double.parseDouble(valuesMap.get(key).get(3))) ;
                criteria3= (float) (criteria3+ Double.parseDouble(valuesMap.get(key).get(4))) ;
                criteria4= (float) (criteria4+ Double.parseDouble(valuesMap.get(key).get(5))) ;
                criteria5= (float) (criteria5+ Double.parseDouble(valuesMap.get(key).get(6))) ;
                criteria6= (float) (criteria6+ Double.parseDouble(valuesMap.get(key).get(7))) ;
            }

        }
        System.out.println("\t\t" + criteria1 + "\t\t" + criteria2+ "\t\t" + criteria3+ "\t\t" + criteria4+ "\t\t" + criteria5
            + "\t\t" + criteria6 );
    }
}

```

4. DATA COLLECTION AND INTERPRETATION OF RESULTS

4.1 MANUAL DATA INTERPRETATION:

Input data considered here for manual interpretation is shown below. First line in the input represents total number of nodes, start node and end node. Other lines represent E as Edge, node 1, node 2, type, alpha and beta.

```
12, 1, 12,  
E,1,2,1,24.000000,22.000000,  
E,1,3,4,0.100000,13.000000,  
E,2,4,1,25.000000,13.000000,  
E,2,5,1,30.000000,13.000000,  
E,3,4,4,0.111111,11.000000,  
E,3,5,2,16.000000,28.000000,  
E,4,6,4,0.100000,15.000000,  
E,4,7,1,25.000000,15.000000,  
E,5,6,2,19.000000,37.000000,  
E,5,7,3,0.047619,9.000000,  
E,6,8,5,22.000000,1.000000,  
E,6,9,5,23.000000,1.000000,  
E,7,8,3,0.045455,5.000000,  
E,7,9,5,22.000000,1.000000,  
E,8,10,2,12.000000,28.000000,  
E,8,11,5,28.000000,1.000000,  
E,9,10,1,22.000000,1.000000,  
E,9,11,1,30.000000,1.000000,  
E,10,12,1,23.000000,1.000000,  
E,11,12,5,28.000000,1.000000,
```

Mean, variance and cSquare for the input shown above are calculated from alpha and beta provided using the formulae shown in figure 1, Section 2.1.

MEAN VALUE PATH:

From the mean values calculated, the following is the graph:

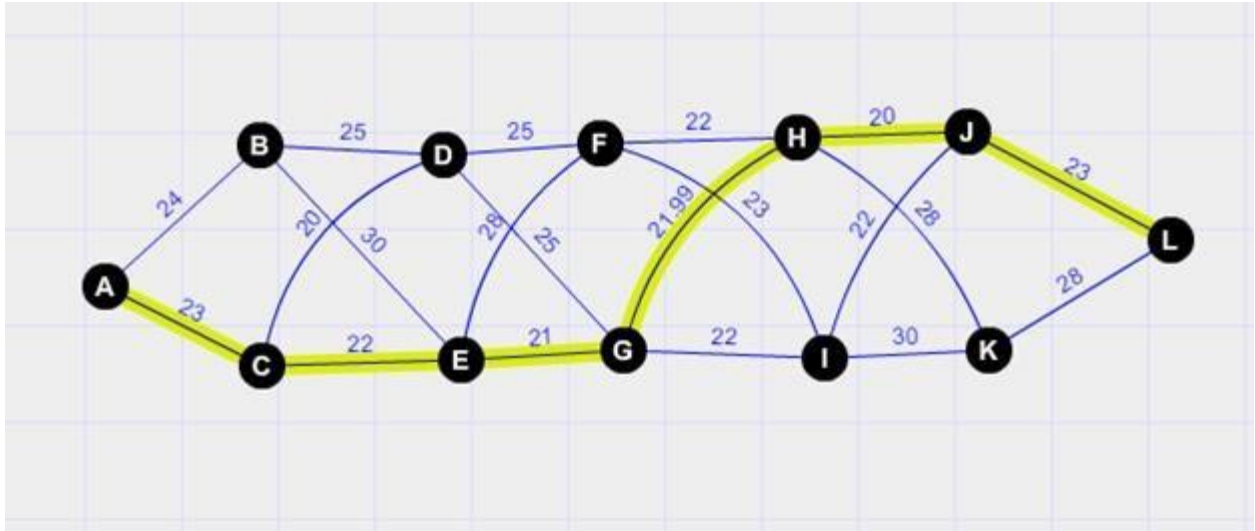


Fig 2- Dijkstra's shortest Mean value path calculated from given inputs

OPTIMIST PATH:

From the values calculated, the following is the graph for optimistic path:

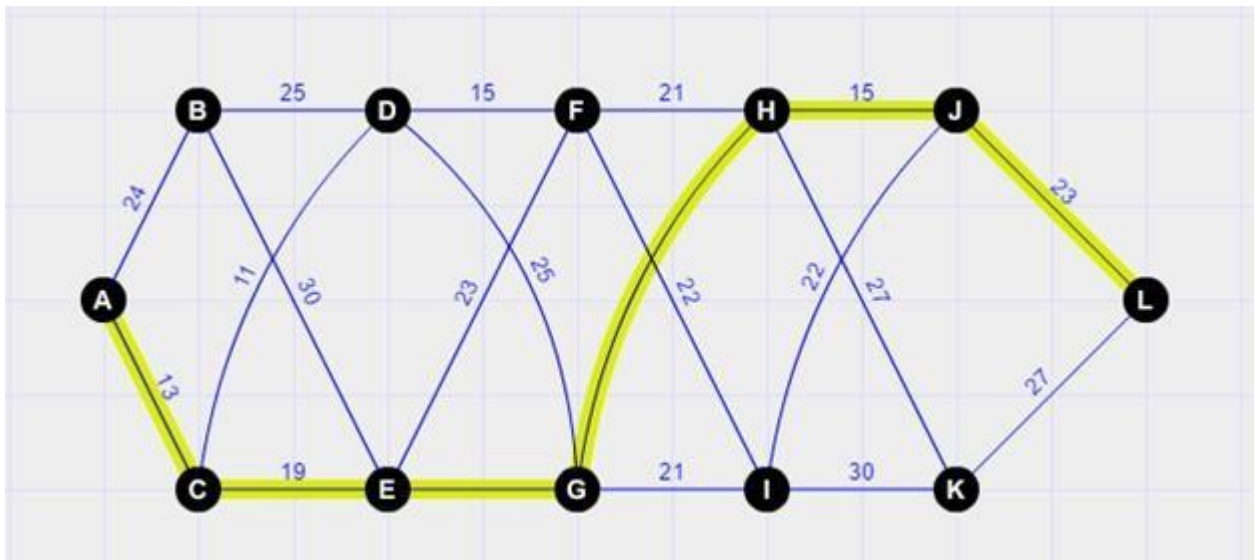


Fig 3- Dijkstra's shortest optimist path calculated from given inputs

PESSIMIST PATH:

From the values calculated, the following is the graph for optimistic path:

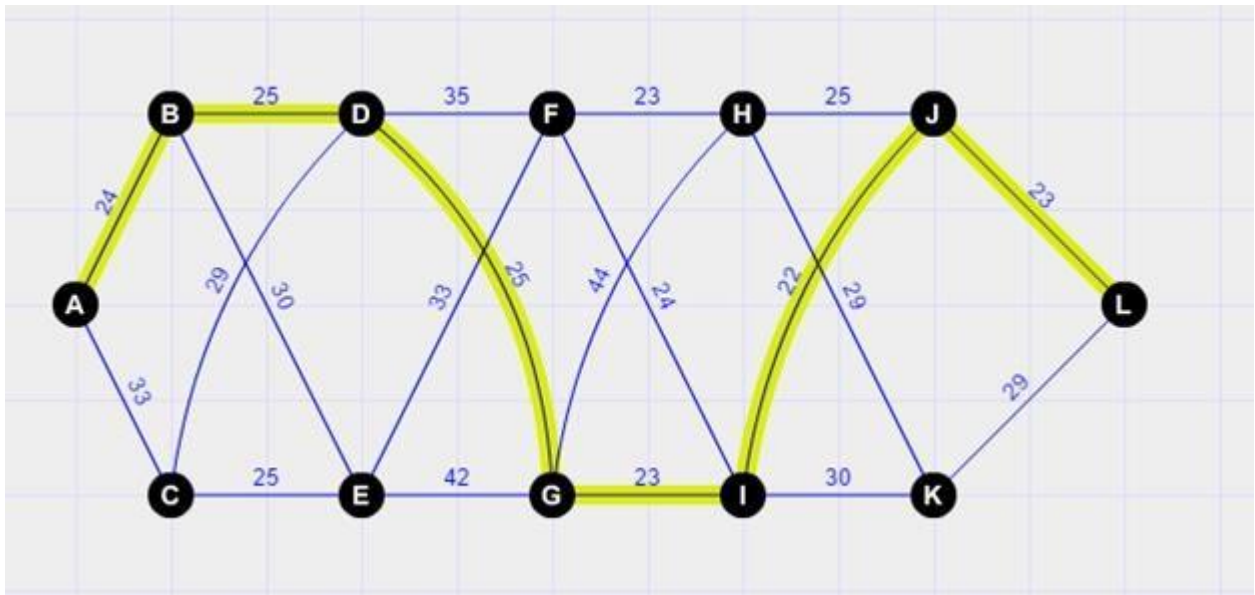


Fig 4- Dijkstra's shortest pessimist path calculated from given inputs

DOUBLE PESSIMIST:

From the values calculated, the following is the graph for optimistic path:

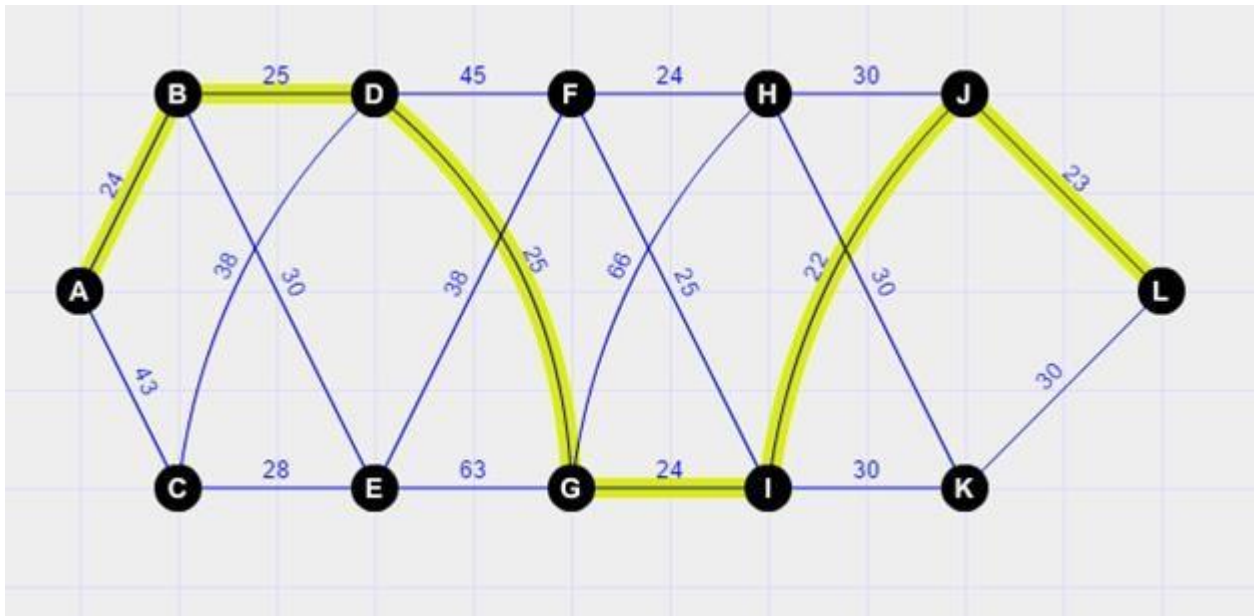


Fig 5- Dijkstra's shortest double pessimist path calculated from given inputs

STABLE:

From the values calculated, the following is the graph for optimistic path:

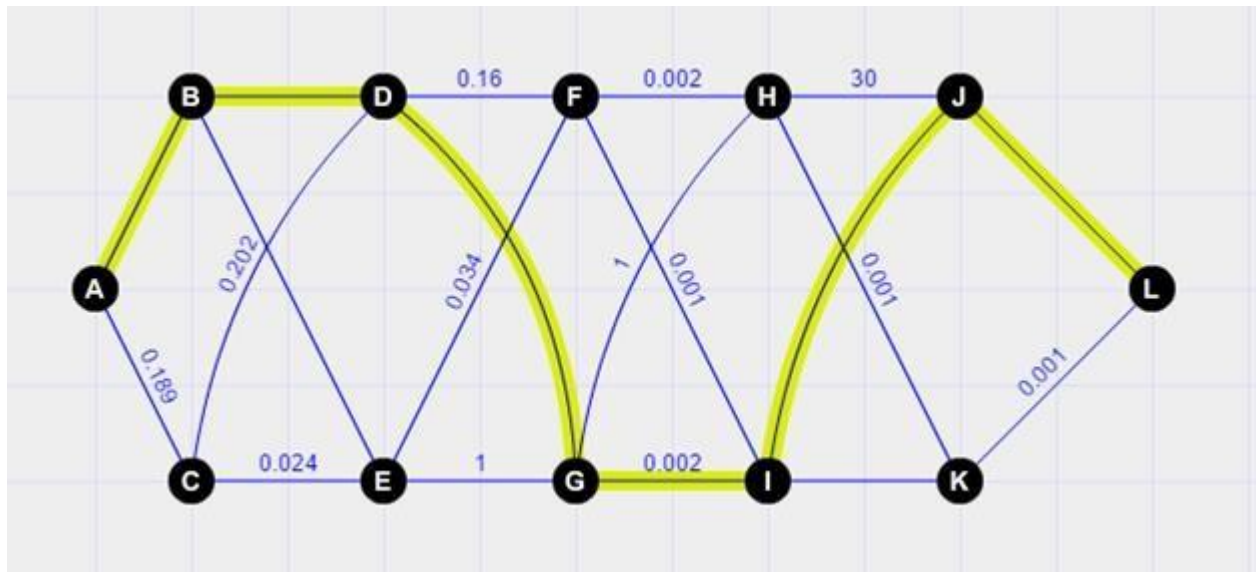


Fig 6- Dijkstra's shortest stable path calculated from given inputs

MEAN+CSQUARE:

From the values calculated, the following is the graph for optimistic path:

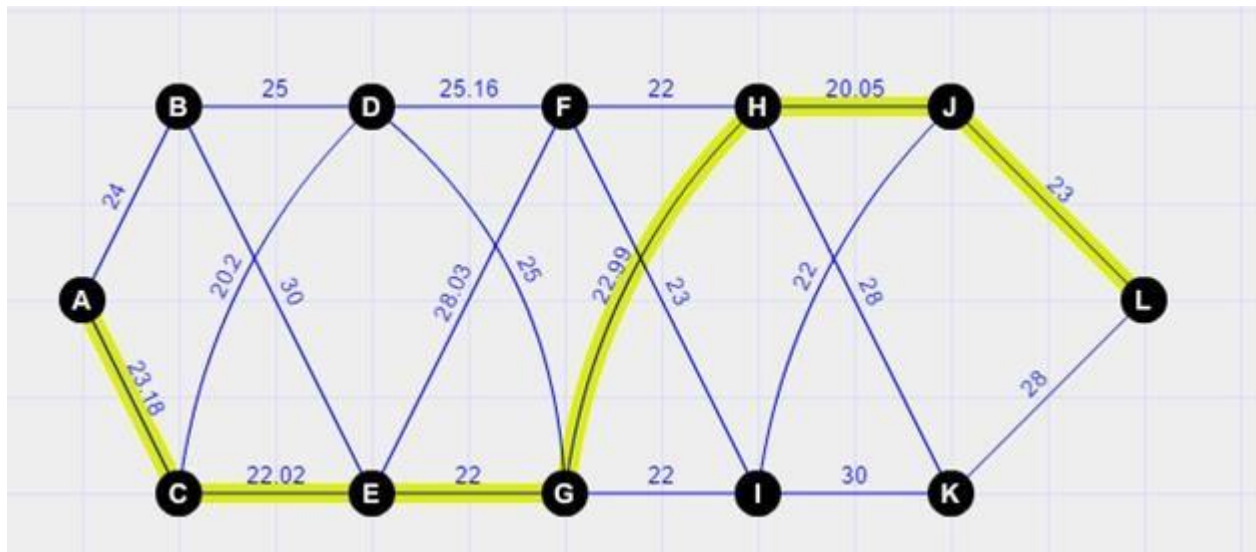


Fig 7- Dijkstra's shortest mean+csquare path calculated from given inputs

4.2 PROGRAMMATIC IMPLEMENTATION USING UNIT TEST CASES:

UNIT TEST CASES:

<i>S.No</i>	<i>Test Case</i>	<i>Description</i>	<i>Expected Result</i>	<i>Actual result</i>	<i>Pass/Fail</i>
1	Output	When a valid input is provided, then shortest paths are shown for the graph with 6 criterion namely mean value, optimist, pessimist, double- pessimist, stable and own paths. Also links that are used by these paths are shown.	Shortest paths for 6 criterion has to be shown along with the links used by these paths	Shortest paths for 6 criterion are shown along with the links used by these paths	Pass
2	Improper Input	One Input line has to contain 3 fields with total edges, starting edge and ending edge. Other lines has to contain 6 fields with E representing edge, vertex 1, vertex2, type, alpha and beta details. If starting vertex and ending vertex are given more than once in the input, then the program throws error.	Error stating "Start and end nodes are given more than once. Please verify your input"	Error stating "Start and end nodes are given more than once. Please verify your input"	Pass
3	Improper Input	One Input line has to contain 3 fields with total edges, starting edge and ending edge. Other lines has to contain 6 fields with E representing edge, vertex 1, vertex2, type, alpha and beta details. If any line contains number of fields not equal to 3 or 5, then the program throws error.	Error stating "Improper graph input. Please verify and try again"	Error stating "Improper graph input. Please verify and try again"	Pass
4	Improper Input	One Input line has to contain 3 fields with total edges, starting edge and ending edge. Other lines has to contain 6 fields with E representing edge, vertex 1, vertex2, type, alpha and beta details. If the first character in the input that represents edge is not 'E', then the program throws error.	Error stating "Invalid edge. First letter of input has to be 'E' indicating Edge"	Error stating "Invalid edge. First letter of input has to be 'E' indicating Edge"	Pass
5	Improper Input	One Input line has to contain 3 fields with total edges, starting edge and ending edge. Other lines has to contain 6 fields with E representing edge, vertex 1, vertex2, type, alpha and beta details. If the input provided has start vertex that is not present in the graph, then the program throws error.	Error stating "Graph doesn't contain start vertex"	Error stating "Graph doesn't contain start vertex"	Pass

6	Improper Input	<p>One Input line has to contain 3 fields with total edges, starting edge and ending edge. Other lines has to contain 6 fields with E representing edge, vertex 1, vertex2, type, alpha and beta details.</p> <p>If the input provided has end vertex that is not present in the graph, then the program throws error.</p>	Error stating "Graph doesn't contain end vertex"	Error stating "Graph doesn't contain end vertex"	Pass
---	----------------	--	--	--	------

UNIT TEST CASES RESULTS:

TEST CASE 1:

INPUT3:

```

Console
<terminated> DijkstrasAlgorithm [Java Application] C:\Program Files\Java\jre1.8.0_101\bin\javaw.exe (Apr 17, 2017, 7:47:16 PM)
Enter graph inputs
12, 1, 12,
E,1,2,1,24.000000,22.000000,
E,1,3,4,0.100000,13.000000,
E,2,4,1,25.000000,13.000000,
E,2,5,1,30.000000,13.000000,
E,3,4,4,0.111111,11.000000,
E,3,5,2,16.000000,28.000000,
E,4,6,4,0.100000,15.000000,
E,4,7,1,25.000000,15.000000,
E,5,6,2,19.000000,37.000000,
E,5,7,3,0.047619,9.000000,
E,6,8,5,22.000000,1.000000,
E,6,9,5,23.000000,1.000000,
E,7,8,3,0.045455,5.000000,
E,7,9,5,22.000000,1.000000,
E,8,10,2,12.000000,28.000000,
E,8,11,5,28.000000,1.000000,
E,9,10,1,22.000000,1.000000,
E,9,11,1,30.000000,1.000000,
E,10,12,1,23.000000,1.000000,
E,11,12,5,28.000000,1.000000,

```

OUTPUT:

Mean Value Path:
Dijkstra's shortest path for the given input with least total 'expected value' is
1 -> 3(23.000000) -> 5(45.000000) -> 7(66.000021) -> 8(87.999801) -> 10(107.999801) -> 12(130.999801)
Number of hops for the path traversed above is 6

Optimist Path:
Dijkstra's shortest path for the given input with least 'expected value - standard deviation' is
1 -> 3(13.000000) -> 5(31.535898) -> 7(31.535898) -> 8(31.535898) -> 10(46.917096) -> 12(69.917096)
Number of hops for the path traversed above is 6

Pessimist Path:
Dijkstra's shortest path for the given input with least 'expected value + standard deviation' is
1 -> 2(24.000000) -> 4(49.000000) -> 7(74.000000) -> 9(98.000000) -> 10(119.000000) -> 12(142.000000)
Number of hops for the path traversed above is 6

Double Pessimist Path:
Dijkstra's shortest path for the given input with least 'expected value + 2 * standard deviation' is
1 -> 2(24.000000) -> 4(49.000000) -> 7(74.000000) -> 9(98.000000) -> 10(120.000000) -> 12(143.000000)
Number of hops for the path traversed above is 6

Stable Path:
Dijkstra's shortest path for the given input with least total 'squared coefficient of variation' is
1 -> 2(0.000000) -> 4(0.000000) -> 7(0.000000) -> 9(0.002066) -> 10(0.002066) -> 12(0.002066)
Number of hops for the path traversed above is 6

Mean+CSquare Path:
Dijkstra's shortest path for the given input with least 'expected value + squared coefficient' is
1 -> 3(23.189036) -> 5(45.213829) -> 7(67.213850) -> 8(90.213630) -> 10(110.266964) -> 12(133.266964)
Number of hops for the path traversed above is 6

Links used by paths:

Edges	MV	Op	Ps	DP	St	Own
Edge(6,8)						
Edge(6,9)						
Edge(7,8)	*	*				*
Edge(7,9)			*	*	*	
Edge(8,10)	*	*				*
Edge(8,11)						
Edge(9,10)			*	*	*	
Edge(9,11)						
Edge(10,12)	*	*	*	*	*	*
Edge(1,2)			*	*	*	
Edge(1,3)	*	*				*
Edge(2,4)			*	*	*	
Edge(2,5)						
Edge(3,4)						
Edge(3,5)	*	*				*
Edge(4,6)						
Edge(4,7)			*	*	*	
Edge(5,6)						
Edge(11,12)						
Edge(5,7)	*	*				*
	μ	$\mu-\sigma$	$\mu+\sigma$	$\mu+2*\sigma$	CSq	Mean+CSq
Mean						
Opt	130.9998	69.9171	192.0825	253.1652	2.2671626	133.26695
Pmst	130.9998	69.9171	192.0825	253.1652	2.2671626	133.26695
DPess	141.0	140.0	142.0	143.0	0.0020661156	141.00208
Stbl	141.0	140.0	142.0	143.0	0.0020661156	141.00208
M+SQq	141.0	140.0	142.0	143.0	0.0020661156	141.00208
	130.9998	69.9171	192.0825	253.1652	2.2671626	133.26695

INPUT4:

```
Console
<terminated> DijkstrasAlgorithm [Java Application] C:\Program Files\Java\jre1.8.0_101\bin\javaw.exe (Apr 17, 2017, 10:10:14 PM)
Enter graph inputs
12, 1, 12
E,1,2,5,25.000000,1.000000,
E,1,3,3,0.043478,10.000000,
E,2,4,5,30.000000,1.000000,
E,2,5,5,29.000000,1.000000,
E,3,4,2,13.000000,29.000000,
E,3,5,3,0.045455,5.000000,
E,4,6,2,14.000000,28.000000,
E,4,7,2,19.000000,37.000000,
E,5,6,3,0.050000,10.000000,
E,5,7,5,26.000000,1.000000,
E,6,8,3,0.045455,9.000000,
E,6,9,4,0.166667,16.000000,
E,7,8,3,0.050000,5.000000,
E,7,9,3,0.041667,9.000000,
E,8,10,2,23.000000,35.000000,
E,8,11,3,0.047619,8.000000,
E,9,10,2,19.000000,37.000000,
E,9,11,1,26.000000,37.000000,
E,10,12,4,0.166667,16.000000,
E,11,12,1,29.000000,16.000000,
```

OUTPUT:

Mean Value Path:

Dijkstra's shortest path for the given input with least total 'expected value' is

1 -> 3(23.000138) -> 5(44.999918) -> 6(64.999918) -> 8(86.999698) -> 11(107.999719) -> 12(136.999719)
Number of hops for the path traversed above is 6

Optimist Path:

Dijkstra's shortest path for the given input with least 'expected value - standard deviation' is

1 -> 3(0.000000) -> 5(0.000000) -> 6(0.000000) -> 8(0.000000) -> 11(0.000000) -> 12(29.000000)
Number of hops for the path traversed above is 6

Pessimist Path:

Dijkstra's shortest path for the given input with least 'expected value + standard deviation' is

1 -> 2(27.000000) -> 4(57.000000) -> 6(82.041452) -> 9(110.041428) -> 11(136.041428) -> 12(165.041428)
Number of hops for the path traversed above is 6

Double Pessimist Path:

Dijkstra's shortest path for the given input with least 'expected value + 2 * standard deviation' is

1 -> 2(27.000000) -> 4(59.000000) -> 6(88.082904) -> 9(122.082868) -> 11(148.082868) -> 12(177.082868)
Number of hops for the path traversed above is 6

Stable Path:

Dijkstra's shortest path for the given input with least total 'squared coefficient of variation' is

1 -> 2(0.001600) -> 4(0.002711) -> 6(0.039748) -> 9(0.114128) -> 11(0.114128) -> 12(0.114128)
Number of hops for the path traversed above is 6

Mean+CSquare Path:

Dijkstra's shortest path for the given input with least 'expected value + squared coefficient' is

1 -> 3(24.000138) -> 4(45.048513) -> 6(66.085550) -> 9(88.159918) -> 10(116.194357) -> 12(138.268724)
Number of hops for the path traversed above is 6

Links used by paths:

Edges	MV	Op	Ps	DP	St	Own
Edge (6, 8)	*	*				
Edge (6, 9)			*	*	*	*
Edge (7, 8)						
Edge (7, 9)						
Edge (8, 10)						
Edge (8, 11)	*	*				
Edge (9, 10)						*
Edge (9, 11)			*	*	*	
Edge (10, 12)						*
Edge (1, 2)			*	*	*	
Edge (1, 3)	*	*				*
Edge (2, 4)			*	*	*	
Edge (2, 5)						
Edge (3, 4)						*
Edge (3, 5)	*	*				
Edge (4, 6)			*	*	*	*
Edge (4, 7)						
Edge (5, 6)	*	*				
Edge (11, 12)	*	*	*	*	*	
Edge (5, 7)						

	μ	$\mu-\sigma$	$\mu+\sigma$	$\mu+2*\sigma$	CSq	Mean+CSq
Mean	136.99971	29.0	244.99944	352.99915	5.0	141.99971
Opt	136.99971	29.0	244.99944	352.99915	5.0	141.99971
Fmst	152.99998	140.95856	165.04143	177.08286	0.11412806	153.11412
DFess	152.99998	140.95856	165.04143	177.08286	0.11412806	153.11412
Stbl	152.99998	140.95856	165.04143	177.08286	0.11412806	153.11412
M+Sqq	137.0001	88.14359	185.85663	234.71317	1.2686105	138.26872

TEST CASE 2:

```

Console
<terminated> DijkstrasAlgorithm [Java Application] C:\Program Files\Java\jre1.8.0_101\bin\javaw.exe (Apr 17, 2017, 7:50:17 PM)
Enter graph inputs
12, 1, 12,
E,1,2,1,24.000000,22.000000,
12, 1, 12,
E,1,3,4,0.100000,13.000000,
E,2,4,1,25.000000,13.000000,
E,2,5,1,30.000000,13.000000,
E,3,4,4,0.111111,11.000000,
E,3,5,2,16.000000,28.000000,
E,4,6,4,0.100000,15.000000,
E,4,7,1,25.000000,15.000000,
E,5,6,2,19.000000,37.000000,
E,5,7,3,0.047619,9.000000,
E,6,8,5,22.000000,1.000000,
E,6,9,5,23.000000,1.000000,
E,7,8,3,0.045455,5.000000,
E,7,9,5,22.000000,1.000000,
E,8,10,2,12.000000,28.000000,
E,8,11,5,28.000000,1.000000,
E,9,10,1,22.000000,1.000000,
E,9,11,1,30.000000,1.000000,
E,10,12,1,23.000000,1.000000,
E,11,12,5,28.000000,1.000000,
Start and end nodes are given more than once. Please verify your input

```

TEST CASE 3:

```
Console
<terminated> DijkstrasAlgorithm [Java Application] C:\Program Files\Java\jre1.8.0_101\bin\javaw.exe (Apr 17, 2017, 7:51:31 PM)
Enter graph inputs
12, 1, 12,
E,1,2,1,24.000000,22.000000,
E,1,3,4,0.100000,13.000000,0.900,10.9
E,2,4,1,25.000000,13.000000,
E,2,5,1,30.000000,13.000000,
E,3,4,4,0.111111,11.000000,
E,3,5,2,16.000000,28.000000,
E,4,6,4,0.100000,15.000000,
E,4,7,1,25.000000,15.000000,
E,5,6,2,19.000000,37.000000,
E,5,7,3,0.047619,9.000000,
E,6,8,5,22.000000,1.000000,
E,6,9,5,23.000000,1.000000,
E,7,8,3,0.045455,5.000000,
E,7,9,5,22.000000,1.000000,
E,8,10,2,12.000000,28.000000,
E,8,11,5,28.000000,1.000000,
E,9,10,1,22.000000,1.000000,
E,9,11,1,30.000000,1.000000,
E,10,12,1,23.000000,1.000000,
E,11,12,5,28.000000,1.000000,|
Improper graph input. Please verify and try again.
```

TEST CASE 4:

```
Console
<terminated> DijkstrasAlgorithm [Java Application] C:\Program Files\Java\jre1.8.0_101\bin\javaw.exe (Apr 17, 2017, 7:53:41 PM)
Enter graph inputs
12, 1, 12,
E,1,2,1,24.000000,22.000000,
A,1,3,4,0.100000,13.000000,
E,2,4,1,25.000000,13.000000,
E,2,5,1,30.000000,13.000000,
E,3,4,4,0.111111,11.000000,
E,3,5,2,16.000000,28.000000,
E,4,6,4,0.100000,15.000000,
E,4,7,1,25.000000,15.000000,
E,5,6,2,19.000000,37.000000,
E,5,7,3,0.047619,9.000000,
E,6,8,5,22.000000,1.000000,
E,6,9,5,23.000000,1.000000,
E,7,8,3,0.045455,5.000000,
E,7,9,5,22.000000,1.000000,
E,8,10,2,12.000000,28.000000,
E,8,11,5,28.000000,1.000000,
E,9,10,1,22.000000,1.000000,
E,9,11,1,30.000000,1.000000,
E,10,12,1,23.000000,1.000000,
E,11,12,5,28.000000,1.000000,|
Invalid input graph edge. First letter of input has to be 'E' indicating Edge
```


TEST CASE 5:

```
Console [X]
<terminated> DijkstrasAlgorithm [Java Application] C:\Program Files\Java\jre1.8.0_101\bin\javaw.exe (Apr 17, 2017, 7:59:14 PM)
```

```
Enter graph inputs
```

```
12, a, 12,
E,1,2,1,24.000000,22.000000,
E,1,3,4,0.100000,13.000000,
E,2,4,1,25.000000,13.000000,
E,2,5,1,30.000000,13.000000,
E,3,4,4,0.111111,11.000000,
E,3,5,2,16.000000,28.000000,
E,4,6,4,0.100000,15.000000,
E,4,7,1,25.000000,15.000000,
E,5,6,2,19.000000,37.000000,
E,5,7,3,0.047619,9.000000,
E,6,8,5,22.000000,1.000000,
E,6,9,5,23.000000,1.000000,
E,7,8,3,0.045455,5.000000,
E,7,9,5,22.000000,1.000000,
E,8,10,2,12.000000,28.000000,
E,8,11,5,28.000000,1.000000,
E,9,10,1,22.000000,1.000000,
E,9,11,1,30.000000,1.000000,
E,10,12,1,23.000000,1.000000,
E,11,12,5,28.000000,1.000000,
```

```
|
Graph doesn't contain start vertex a
```

TEST CASE 6:

```
Console [X]
<terminated> DijkstrasAlgorithm [Java Application] C:\Program Files\Java\jre1.8.0_101\bin\javaw.exe (Apr 17, 2017, 7:59:58 PM)
```

```
Enter graph inputs
```

```
12, 1, z,
E,1,2,1,24.000000,22.000000,
E,1,3,4,0.100000,13.000000,
E,2,4,1,25.000000,13.000000,
E,2,5,1,30.000000,13.000000,
E,3,4,4,0.111111,11.000000,
E,3,5,2,16.000000,28.000000,
E,4,6,4,0.100000,15.000000,
E,4,7,1,25.000000,15.000000,
E,5,6,2,19.000000,37.000000,
E,5,7,3,0.047619,9.000000,
E,6,8,5,22.000000,1.000000,
E,6,9,5,23.000000,1.000000,
E,7,8,3,0.045455,5.000000,
E,7,9,5,22.000000,1.000000,
E,8,10,2,12.000000,28.000000,
E,8,11,5,28.000000,1.000000,
E,9,10,1,22.000000,1.000000,
E,9,11,1,30.000000,1.000000,
E,10,12,1,23.000000,1.000000,
E,11,12,5,28.000000,1.000000,
```

```
|
Graph doesn't contain end vertex z
```

5. CONCLUSION

From the above diagrams, we can obtain the conclusions for different criteria's

MEAN VALUE PATH

Here we obtained the shortest path for the given input values along with their expected values. So, from the figure above we can see that the shortest path is from

1- 3 (23.0000000) - 5 (45.000000) - 7 (66.000021) - 8 (87.999801) - 10 (107.999801)
- 12 (130.999801)

So, the total cost would be 130.999801

Number of hops for the path traversed above is 6

the obtained path will be the Actual path between points 1 and 12 in this case

OPTIMIST PATH

Using Dijkstra's shortest path algorithm we derived the shortest path for the given input along with their "expected value - standard deviation".

from the above figure, we can see the shortest path which is

1 -> 3(13.000000) -> 5(31.535898) -> 7(31.535898) -> 8(31.535898) -> 10(46.917096) ->
12(69.917096)

so, the total cost would be 69.917096

Number of hops for the path traversed above is 6

So, from the result we can interpret that the cost of optimist path is less than the cost of mean value path.

PESSIMIST PATH

we found the shortest path for the given input with least "expected value + standard deviation". the shortest path obtained would be

1 -> 2(24.000000) -> 4(49.000000) -> 7(74.000000) -> 9(97.000000) -> 10(119.000000) ->
12(142.000000)

the total cost would be 142.000000

Number of hops for the path traversed above is 6

here in this case the total cost to travel from 1 to 12 is larger than mean value path as we added standard deviation to it. we can assume it as the worst-case scenario

DOUBLE PESSIMIST PATH

Dijkstra's algorithm is used to find the shortest path from node 1 to 12 for the given edge weights with least "expected value+2* standard deviation"

1 -> 2(24.000000) -> 4(49.000000) -> 7(74.000000) -> 9(98.000000) -> 10(120.000000) -> 12(143.000000)

the total cost in this case would be 143.000000 Number of hops for the path traversed above is 6 the total cost is greater than the pessimist path in this case.

STABLE PATH

we used Dijkstra's algorithm to find the shortest path for the given input values with least total "squared coefficients of variation"

1 -> 2(0.000000) -> 4(0.000000) -> 7(0.000000) -> 9(0.002066) -> 10(0.002066) -> 12(0.002066)

the total cost would be 0.002066

Number of hops for the path traversed above is 6

EXPECTED VALUE + SQUARED COEFFICIENT PATH:

Dijkstra's shortest path for the given input with least 'expected value + squared coefficient' is

1 -> 3(23.189036) -> 5(45.213829) -> 7(67.213850) -> 8(90.213630) -> 10(110.266964) -> 12(133.266964)

the total cost would be 133.266964

Number of hops for the path traversed above is 6

In this case, the total cost from 1 to 12 is almost equal to the mean value path.

6. EPILOGUE

We learned a lot from this project. Firstly, the project is very interesting. Until this project was given to us we thought that shortest paths are very easy to find. Of course, we knew that there will be congestion in the paths but we didn't know that these criteria are possible. It made the project very interesting.

We used java to implement our algorithm and we came to know about some new functions and libraries while working on this project. We actually had a problem with data types. At the start, we used float as the datatype for the variables and round function to round up the values. Then we observed that there are some errors in the outputs and realized that we shouldn't use round function as it is rounding of the values for example if we have 0.0011 then it is rounding of to 0. So, the outputs were not correct, then we replaced the float data type with double. And it worked perfect.

From this project, we did learn a lot of things like what is Dijkstra's uncertain path algorithm, How to handle congestions in the obtained shortest path, But we can say that finding the shortest path and seeing that there will be no congestion at the same time is a bit difficult and this is what we learned from this project.

If we were given this project again and of course with some modifications and extensions we will try to implement it more effectively which means we will try to eliminate the congestion in the paths and we will also try to improve the efficiency of the code that is we will try to decrease the time complexity.

7. APPENDIX

Generally, inputs are in the form E, 1, 2, 1, 24.000000, 22.000000, Here E represents Edge

'1' and '2' represents source and destinations respectively

'1' represents the type of distribution

'24.000000' and '22.000000' represents the alpha and beta values.

8. REFERENCES

<https://www.wikipedia.org/>

www.stackoverflow.com

<https://www.quora.com/>

https://rosettacode.org/wiki/Dijkstra%27s_algorithm