

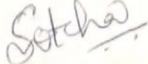


CS5592, Spring Semester 2017
Dijkstra's Shortest Uncertain Path Algorithm

I understand and have adhered to the rules regarding student conduct. In particular, any and all material, including algorithms and programs, have been produced and written by myself. Any outside sources that I have consulted are free, publicly available, and have been appropriately cited. I 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 follow up action. It could result in being expelled from the university.

1: Gopi Krishna Swargam		16222026
First & Last Name	Signature	UMKC-Id
2: Grebe Jason		16135499
First & Last Name	Signature	UMKC-Id
3: Swetcha Reddy Vangala		16232857
First & Last Name	Signature	UMKC-Id

Contents

- 1. Introduction**
- 2. Experimental Design**
- 3. Experimental Implementation**
- 4. Data Collection and Interpretation of Results**
- 5. Conclusion**
- 6. Epilogue**
- 7. Appendix**

1. INTRODUCTION

This is a study of how different distributions may affect the shortest path from across a graph. Because delay when traveling from one node to another node along a path might change over time. There are six different distribution types that may affect a delay that will be calculated using alpha and beta values. This delay may, in effect, cause the shortest path to alter, so that alternate routes are taken based on the criteria. We are analyzing this shortest path of “uncertain” delay utilizing Dijkstra’s Shortest Path Algorithm. We are using six different criteria to compare how these distribution types might affect the path taken for the values supplied.

2. EXPERIMENTAL DESIGN

The criteria used to find various shortest paths possible from city ‘a’ to city ‘b’ are

- a. Mean value
- b. Optimist
- c. Pessimist
- d. Double pessimist
- e. Stable

And the sixth criterion chosen by our team is ‘Sum of parameters alpha and beta ($\alpha + \beta$)’.

f. $\alpha + \beta$

a. Mean Value

The path with the smallest expected or mean value gives the shortest path from the city ‘a’ to city ‘b’. This path is optimal because not only the total weight of the path i.e., distance from city ‘a’ to city ‘b’ is the least, but also the edge costs of all the edges between ‘a’ and ‘b’ are also minimized in this criterion.

b. Optimist

The path whose ‘expected value - standard deviation’ is smallest among all the paths gives the shortest path from ‘a’ to ‘b’. Standard deviation provides a useful basis for interpreting the data in terms of probability. Standard deviation is a summary measure of the differences of each observation from the mean. Small edge costs result in small expected value. Therefore, the smaller the difference between expected value and each edge cost, the smaller the standard deviation. Hence, the least weighted optimist path.

c. Pessimist

The path whose ‘expected value + standard deviation’ is smallest among all the paths gives the shortest path from ‘a’ to ‘b’. Like the optimist path, the pessimist path is also obtained from the expected value and standard deviation. In the optimist criterion, there are chances of negative values for the difference between expected value and standard deviation. In the pessimist criterion, there is no such chance for the sum of expected value and standard deviation to be negative, so the least ‘expected value + standard deviation’ gives the shortest path.

d. Doubly Pessimist

The path whose 'expected value + 2 standard deviation' is smallest is the shortest path. This criterion is even more pessimistic than the pessimist criterion. When the path with smallest expected value is the shortest, whether the path is pessimist or double pessimist, it becomes the optimal path.

e. Stable

The path whose 'squared coefficient of variance' is smallest is the shortest path. Coefficient of variance is the relative standard deviation. It is the ratio of standard deviation to the mean. The smallest values of mean and standard deviation result in shortest path.

f. Sum of Alpha and Beta ($\alpha + \beta$)

This is the criteria that we added. The path with the smallest alpha and beta parameter values is the shortest path. Alpha and beta parameters are used to calculate expected value. The smaller the values of alpha and beta, the smaller the mean. Thus, smaller the edge cost and the shortest path.

3. EXPERIMENTAL IMPLEMENTATION

Upon receiving the requirements, we decided to implement our algorithm in Java. We chose this language, because all our team members have experience in programming in Java. We also know what data structures are required for implementation and realized Java included all the data structures we would want and need for our implementation. We also knew we could implement any new user defined data type using Java classes and associate their behavior with methods.

We knew we would need to implement an adjacency matrix or adjacency list to construct graphs to efficiently keep track of what nodes were connected, as well as, which direction to traverse the edge. Although an adjacency list would use less memory, we chose to implement an adjacency matrix. We have implemented adjacency matrices to construct the graphs, because it allows us to produce the edge weight between two nodes in $O(1)$ time, making our algorithm to execute more efficiently. However, our implementation could easily be modified to implement an adjacency list instead.

Data Structures Used

Knowing we were going to implement Dijkstra shortest path algorithm, we began analyzing and searching notes and documentation to find a very efficient pseudocode for Dijkstra's Shortest Path Algorithm. Based on our class room notes and online references we have developed our Dijkstra shortest path algorithm using Priority Queues, which ensure that shortest path is found in $O(E \log V)$ time.

Following are the data structures that we have used in our Dijkstra shortest path algorithm implementation: Adjacency Matrix, Priority Queue, Hash Set, Hash Map and Array.

- a. It is required to convert the given input file into a graph representation prior to calculating the shortest path. We have read the input from a file and calculated the corresponding the edge weights between the nodes in the given input file. We then construct the graph using Adjacency matrix representation.
- b. We have used the Java built-in Priority Queue data structure to obtain the next minimum distance node from the start node, as we try to find the shortest path from start to end. The priority queue needs to act as a min heap in our algorithm, so we have accommodated changes to Java priority queue by implementing the Java comparable interface and implementing the compareTo () method. This makes the Java priority queue acts as a min heap.

- c. To keep track of all the visited nodes and to ensure that the visited nodes are not visited again, we have used the Java built-in HashSet data structure.
- d. As we traverse through the nodes in the graph and reach the destination node, we have used the Java built-in HashMap data structure. For every node visited in our algorithm, the node and its parent node are entered as an entry in the HashMap. After the algorithm execution is completed, we have used the destination node to backtrack through the HashMap, constructing the shortest path along the way.
- e. It is possible to have duplicate entries for a node in the Priority Queue data structure. It is also possible we could add more entries to a node if the distance to that node from the start node is less than the existing entry in Priority Queue. To get the shortest distance from the start node to a particular node in the graph, we utilize an array. This allows us to retrieve the distance in $O(1)$ time.

We also knew we would need a way for calculating the mean, the standard deviation and the coefficient of variation, as these would be used by numerous distribution types during calculations. We have created three different methods, one for each of the aforementioned. The input for these methods are the distribution type number and the alpha and beta values, which return the mean, the standard deviation, or the coefficient of variation as output, depending on the method called. We have created one method to print the shortest path. This method takes a HashMap, start node, and end node as inputs. It then prints the shortest path by back tracing it. We have created another method, `getNeighbors()`, which returns the neighbors of a node. This is required in our implementation.

There were a few circumstances unforeseen during the primary implementation. One of these unforeseen circumstances was when we were going down a particular path that was believed to be the shortest path. But, because of significantly lower delay from a node that was not on the original shortest path, the path would have to backtrack and significantly change. For example, if you have a 12-node array, an edge from node 8 to node 11 could cause the shortest path to alter all the way back to the edge traversed. To alleviate this, we added an array to keep track of the smallest distance to a node that were previously visited. This assisted in backtracking to the node and correcting for changes in the shortest path.

Algorithm/Pseudocode

```
dijkstraShortestPath (Graph G, Start Node, EndNode):
//HashSet tracks the visited nodes in the graph
//Priority Queue has the nodes with shortest distances to reach them
//HashMap has node and parent nodes of every node inserted in to Priority Queue
Initialize Priority Queue, HashSet, HashMap

Enqueue {StartNode, 0} onto the Priority Queue
while Priority Queue is not empty:
    currentNode = dequeue from Priority Queue
    if (currentNode is not in HashSet)
        add currentNode to HashSet
        if currentNode == EndNode
            return HashMap
        for each of currentNode neighbor's n not in HashSet:
            if path through currentNode to n is less than currentNode entry
            in Priority Queue
                update currentNode as n's parent in HashMap
                enqueue {n, distance} into the Priority Queue
```

Above is the pseudo code of the Dijkstra shortest path algorithm that we have implemented. We have passed the graph G (which is an adjacency matrix), start node, and end node as inputs to our algorithm. Once we

arrive at the end node, our algorithm returns the HashMap, which can be backtracked from end node to start node. This can then print the shortest path.

For our convenience, we have created a user-defined data type, Path, which has two variables, vertex and distance. These are required from the starting node to reach any other particular node. We are storing the objects of Path into the priority queue.

- a. Initially we insert the Path object with the start vertex and its distance to reach from the start vertex equal to zero.
- b. As we enter the while loop, we check if the priority queue is empty. If it is not empty, we dequeue a Path object from the priority queue that is acting like a min heap.
- c. If the dequeued Path object is not visited, we check to see if the Path object is the end node. If it is end node, we return the HashMap, which can be used to back trace and print the shortest path. If the Path object is not the destination node, we add unvisited neighbors. This is done by checking if the neighbors have already been visited, via a HashSet. The HashSet keeps track of the visited nodes and adds dequeued path object as parent nodes for the neighbors in the HashMap.
- d. Repeat steps 2 to 4 until we reach the target node.

4. DATA COLLECTION AND INTERPRETATION OF RESULTS

Once we have developed our algorithm, we have written the code in Java. This code was run on input files taken from blackboard, as well as, input files we generated ourselves. Given a graph G , if the number of vertices is V and number of edges is E our algorithm runs in $O(E \log V)$ time.

Test Cases

- a. Given a valid input file which has edges from start node to end node our algorithm is generating the correct shortest path and the number of hops traversed across this path.
- b. If the input file is missing an edge leading to the end node, our program does not generate any shortest path.
- c. If there are any duplicate entries for an edge in the input file, the edge weights are updated accordingly. Thus, it takes the last entry for the edge, ignoring all previous entries.
- d. Considering the alpha and beta values to be positive, the edge weights are also positive; our algorithm would work only for positive edge weights. Given additional time and slight modifications to the code, it would not be too difficult to allow for negative edge weights and protection from negative cycles.
- e. For every edge which is a new line in the input file, if there are any extra spaces, our program trims such extra spaces and reads the input correctly.

Below are performance measure tables for input3 and input4 files in blackboard

Performance Measures

Input3

	$\mu - \sigma$	μ	$\mu + \sigma$	$\mu + 2 \sigma$	$C^2(Y)$	hops	Shortest path
Mean Value	69.91709623	130.999801	192.0825	253.1652105	2.267162639	6	1->3->5->7->8->10->12
Optimist	69.91709623	130.999801	192.0825	253.1652105	2.267162639	6	1->3->5->7->8->10->12
Pessimist	140	141	142	143	2.07E-03	6	1->2->4->7->9->10->12
Doubly Pess	140	141	142	143	2.07E-03	6	1->2->4->7->9->10->12
Stable	140	141	142	143	0.002066116	6	1->2->4->7->9->10->12
AlphaBeta	106	136.000009	166	196.000027	0.553426499	6	1->3->4->6->9->10->12

Input4

	$\mu - \sigma$	μ	$\mu + \sigma$	$\mu + 2 \sigma$	$C^2(Y)$	hops	Shortest path
Mean Value	29	136.999719	244.9994	352.999157	5	6	1->3->5->6->8->11->12
Optimist	29	136.999719	244.9994	352.999157	5	6	1->3->5->6->8->11->12
Pessimist	140.9585481	152.999988	165.0414	177.0828678	0.114128097	6	1->2->4->6->9->11->12
Doubly Pess	140.9585481	152.999988	165.0414	177.0828678	0.114128097	6	1->2->4->6->9->11->12
Stable	140.9585481	152.999988	165.0414	177.0828678	0.114128097	6	1->2->4->6->9->11->12
AlphaBeta	29	136.999719	244.9994	352.999157	5	6	1->3->5->6->8->11->12

From the above performance measure table, we can see that the hop count is always 6 for both input files supplied, irrespective of the criteria used to generate the shortest path. For input3 the shortest paths for Mean and Optimist criteria are same and shortest paths for Pessimist, Doubly Pessimist and Stable criteria are same and the shortest path for our own criteria is different from all the other shortest paths.

For input4 the shortest paths for Pessimist, Doubly Pessimist and Stable are same, whereas the shortest paths for Mean, Optimist and our own criteria are same.

Below is the table showing the presence of common links in different paths and the number of times each link is seen. N standing for No and Y standing for Yes.

Input3

Edge	Mean Value	Optimist	Pessimist	Doubly Pess	Stable	AlphaBeta	No. of paths
(1,2)	N	N	Y	Y	Y	N	3
(1,3)	Y	Y	N	N	N	Y	3
(2,4)	N	N	Y	Y	Y	N	3
(3,4)	N	N	N	N	N	Y	1
(3,5)	Y	Y	N	N	N	N	2
(4,6)	N	N	N	N	N	Y	1
(4,7)	N	N	Y	Y	Y	N	3
(5,7)	Y	Y	N	N	N	N	2
(6,9)	N	N	N	N	N	Y	1
(7,8)	Y	Y	N	N	N	N	2
(7,9)	N	N	Y	Y	Y	N	3
(8,10)	Y	Y	N	N	N	N	2
(9,10)	N	N	Y	Y	Y	Y	4
(10,12)	Y	Y	Y	Y	Y	Y	6

Input4

Edge	Mean Value	Optimist	Pessimist	Doubly Pess	Stable	AlphaBeta	No. of paths
(1,2)	N	N	Y	Y	Y	N	3
(1,3)	Y	Y	N	N	N	Y	3
(2,4)	N	N	Y	Y	Y	N	3
(3,5)	Y	Y	N	N	N	Y	3
(4,6)	N	N	Y	Y	Y	N	3
(5,6)	Y	Y	N	N	N	Y	3
(6,8)	Y	Y	N	N	N	Y	3
(6,9)	N	N	Y	Y	Y	N	3
(8,11)	Y	Y	N	N	N	Y	3
(9,11)	N	N	Y	Y	Y	N	3
(11,12)	Y	Y	Y	Y	Y	Y	6

From the above table for input3, we can see that link (10,12) is used in shortest paths for all criteria. Link (9,10) is used in four criteria. The rest are used in three or less of the criteria.

For input4, we can see that link (11,12) is used in shortest paths for all criteria. All other links are used in 3 criteria.

5. CONCLUSION

Our program implements a robust, high functionality implementation of Dijkstra's Shortest Path Algorithm. It correctly and efficiently calculates the shortest path for inputs given. This could be used in many areas, such as logistics, networking or data science. This high functionality could also lead to more reusability and utility. However, there is no way to select a particular input file to create the graph. Given more time, we could port the code to either a Java applet or stand-alone jar file that would allow a user to browse their files to select an input file.

The program uses modules to separate certain aspects. For example, the Path class is separated from the RandomVariableDistribution class and the Dijkstra class. This could lead to more reusability for the code, as it could be used in other projects that may have conception in the future.

With the current level of documentation, the program is readable by a trained programmer. However, someone with less Java and/or programming experience might struggle to follow certain portions of the code. With additional documentation, this readability greatly increases, so that anyone would be able to decipher the code. This would naturally lead to the ability to explain the code in detail to others and the ability for anyone to maintain the code.

It is a highly portable. This code could be used in any programming language that utilizes the same, or similar, data structures. Small changes might be needed. For example, there is no HashMap data structure in PHP. But, in PHP, arrays can have a key value structure. This code could easily be ported to most any object-oriented language.

The program is quite robust. The program uses exception handling and error handling in case reading a file fails. Because computer memory is very inexpensive today, the program easily runs in memory for small sets. However, it could come into errors while reading very, very large data sets perhaps with millions of nodes. But, it is impossible to predict all errors that could arise, and given the time period, no additional error checking can be done.

Our implementation runs in $O(E \log V)$ for calculating Dijkstra's Shortest Path. This does not include the backtracking to display the path.

Overall, our implementation is highly functional, reusable, portable, efficient and maintainable. Given additional time, some changes could be made to make the implementation even better.

6. EPILOGUE

We started the project by doing some work on the white board. We figured out that the project has two main sub tasks. The first subtask is reading the input from the input file and constructing the graph accordingly. The second subtask is passing the graph to the Dijkstra algorithm to generate and print the shortest path.

There are two ways to represent the graph, an adjacency list or an adjacency matrix. We decided to go for an adjacency matrix representation of graph from the beginning. Though it uses more memory, it is comparatively faster.

For the second subtask, producing the Dijkstra algorithm, we decided to use priority queue data structure, as it would make the algorithm run in $O(E \log V)$ time, which we understand is the fastest implementation for Dijkstra. But, as we started writing the algorithm, we realized we required other data structures, like HashSet,

to mark the nodes that are already visited, and HashMap, to store the parent nodes of the nodes that we had already visited. Again, the use of HashSet and HashMap would allow us to do the get and put operations in $O(1)$ time.

During this project, we learned that dividing the main task into subtasks is very important in seeing how the data flows from one subtask to another. This also allowed us to see that all the design decisions were made prior to coding, and confirm we are using appropriate data structures to ensure the code runs efficiently with efficient memory usage.

If this project were to be done again with different requirements, we would probably keep our algorithm the same, as we feel we have done very good job in coming up with an algorithm which has best time complexity. However, we might re-think some of the data structures or create appropriate user defined data types to read and organize the input. This, of course, would depend on the requirements.

We have done this project in Java. Rather than writing the code for different data structures that are utilized, we have learned and used the Java's built in APIs, data structures, and classes to ease the implementation. This also assists in making it easier for our algorithm to be ported to another language.

For future students, we recommend they start the project by having as much discussion and debate about the problem as possible. This way they can clear their doubts and then divide the main project in to different subtasks. Then, they can work on each subtask separately, while testing various data structures for efficiency. Finally, integrating the code all of the subtasks and testing the code thoroughly.

7. APPENDIX

Program listing

```
package daaProject;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Deque;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map.Entry;
import java.util.PriorityQueue;
import java.util.Scanner;

public class Dijkstra {
    //Helper method which returns mean value for given distribution, alpha and beta values
    public static double mean(int d, double a, double b){
        double m = 0.0;
        switch(d){
            case 1:
                m = a;
```

```

        break;
    case 2:
        m = (a + b)/2;
        break;
    case 3:
        m = (1/a);
        break;
    case 4:
        m = (b + 1/a);
        break;
    case 5:
        m = a;
        break;
    case 6:
        m = a;
        break;
    }
    return m;
}

```

```

//Helper method which returns standard deviation for given distribution, alpha and beta values
public static double sd(int d, double a, double b){
    double s = 0.0;
    double v;
    switch(d){
    case 1:
        break;
    case 2:
        v = Math.pow(b-a, 2)/12;
        s = Math.sqrt(v);
        break;
    case 3:
        v = 1/Math.pow(a, 2);
        s = Math.sqrt(v);
        break;
    case 4:
        v = 1/Math.pow(a, 2);
        s = Math.sqrt(v);
        break;
    case 5:
        v = b;
        s = Math.sqrt(v);
        break;
    case 6:
        v = Math.pow(a, 2) * b;
        s = Math.sqrt(v);
        break;
    }
    return s;
}

```

//Helper method which returns coefficient of variation for given distribution, alpha and beta values

```
public static double c2(int d, double a, double b){
    double c = 0.0;
    switch(d){
    case 1:
        break;
    case 2:
        c = (Math.pow(b-a, 2) * 1)/(Math.pow(a+b, 2) * 3);
        break;
    case 3:
        c = 1.0;
        break;
    case 4:
        c = 1/(Math.pow(1+b*a, 2));
        break;
    case 5:
        c = b/(Math.pow(a, 2));
        break;
    case 6:
        c = b;
        break;
    }
    return c;
}
```

//Helper method which prints the adjacency matrix

```
public static void printMatrix(double[][] m){
    for(int i=1; i<m.length; i++){
        for(int j=1; j<m[0].length; j++){
            System.out.print(m[i][j]+" ");
        }
        System.out.println();
    }
}
```

//Helper method which returns the neighbors of a node as a list

```
public static ArrayList<Integer> getNeighbors(double[][] m, int r){
    ArrayList<Integer> neighbors = new ArrayList<Integer>();

    for(int i=1; i<m[r].length; i++){
        if(m[r][i] >= 0 ){
            neighbors.add(i);
        }
    }

    return neighbors;
}
```

```

//Helper method which generates the shortest path for the given input graph
public static HashMap<Integer,Integer> dijkstraShortestPath(double[][] m, int startVertex, int endVertex){
    HashMap<Integer,Integer> shortestPath = new HashMap<Integer,Integer>();

    PriorityQueue<Path> nextVertices = new PriorityQueue<Path>();
    HashSet<Integer> visitedVertices = new HashSet<Integer>();

    double[] verticeLeastDistance = new double[m.length];

    for(int i=0; i<verticeLeastDistance.length; i++){
        verticeLeastDistance[i] = Double.MAX_VALUE;
    }

    Path pObj = new Path();
    pObj.vertex = startVertex;
    pObj.distance = 0;

    nextVertices.add(pObj);

    while(!nextVertices.isEmpty()){
        Path currObj = nextVertices.remove();
        if(!visitedVertices.contains(currObj.vertex)){
            visitedVertices.add(currObj.vertex);
            if(currObj.vertex == endVertex){
                System.out.println("Shortest Distance "+currObj.distance);
                return shortestPath;
            }
            for(Integer n : getNeighbors(m,currObj.vertex)){
                if(!visitedVertices.contains(n)){
                    double newDistance = m[currObj.vertex][n]+currObj.distance;

                    if(newDistance < verticeLeastDistance[n]){
                        verticeLeastDistance[n] = newDistance;
                        shortestPath.put(n, currObj.vertex);
                    }

                    pObj = new Path();
                    pObj.vertex = n;
                    pObj.distance = newDistance;
                    nextVertices.add(pObj);
                }
            }
        }
    }

    return shortestPath;
}

```

```

//Helper method which prints the shortest path
public static void printPath(HashMap<Integer,Integer> shortestPath, int startVertex, int endVertex, String
graphName, RandomVariableDistribution[][] rvd){
    Deque<Integer> stack = new ArrayDeque<Integer>();
    stack.push(endVertex);
    int currVertex = endVertex;
    int hopCount = 0;
    double m = 0;
    double std = 0;
    double cc = 0;
    while(currVertex != startVertex)
    {
        RandomVariableDistribution rv = rvd[shortestPath.get(currVertex)][currVertex];
        m += mean(rv.distribution,rv.alpha,rv.beta);
        std += sd(rv.distribution,rv.alpha,rv.beta);
        cc += c2(rv.distribution,rv.alpha,rv.beta);
        stack.push(shortestPath.get(currVertex));
        currVertex = shortestPath.get(currVertex);
        hopCount++;
    }

    System.out.println("Shortest path: "+graphName);
    System.out.println("Hop Count "+hopCount);
    System.out.println("Mean - std "+(m-std));
    System.out.println("Mean "+m);
    System.out.println("Mean + std "+(m+std));
    System.out.println("Mean + 2 * std "+(m+2*std));
    System.out.println("C2 "+cc);

    while(!stack.isEmpty()){
        System.out.print(stack.pop()+"->");
    }
    System.out.println();
}

//Helper method which initializes the adjacency matrix
public static void initMat(double[][] mat){
    for(int i=0;i < mat.length; i++){
        for(int j=0; j < mat[0].length; j++){
            mat[i][j] = -1;
        }
    }
}

public static void main(String[] args) {
    //Scanner sc = new Scanner(System.in);
    String fileName = "input.txt";
    //Creating the adjacency matrices for all the criteria
    double[][] adjMatMean = null;
    double[][] adjMatOpt = null;
}

```

```

double[][] adjMatPest = null;
double[][] adjMatDPest = null;
double[][] adjMatStable = null;
double[][] adjMatRisk = null;

RandomVariableDistribution[][] rvd = null;
int startVertex=0, endVertex=0, nodes;

String line = null;

try {
    // FileReader reads text files in the default encoding.
    FileReader fileReader = new FileReader(fileName);

    // Always wrap FileReader in BufferedReader.
    BufferedReader bufferedReader = new BufferedReader(fileReader);
    line = bufferedReader.readLine();
    String[] parts = line.split(",");
    nodes = 1 + Integer.parseInt(parts[0].trim());
    startVertex = Integer.parseInt(parts[1].trim());
    endVertex = Integer.parseInt(parts[2].trim());
    //Allocating memory and initializing the adjacency matrix
    adjMatMean = new double[nodes][nodes];
    initMat(adjMatMean);
    adjMatOpt = new double[nodes][nodes];
    initMat(adjMatOpt);
    adjMatPest = new double[nodes][nodes];
    initMat(adjMatPest);
    adjMatDPest = new double[nodes][nodes];
    initMat(adjMatDPest);
    adjMatStable = new double[nodes][nodes];
    initMat(adjMatStable);
    adjMatRisk = new double[nodes][nodes];
    initMat(adjMatRisk);
    rvd = new RandomVariableDistribution[nodes][nodes];
    //reading the input file and calculating the edge weights and storing in adjacency matrices
    while((line = bufferedReader.readLine()) != null) {
        parts = line.split(",");
        int r = Integer.parseInt(parts[1].trim());
        int c = Integer.parseInt(parts[2].trim());
        int dist = Integer.parseInt(parts[3].trim());
        double a = Double.parseDouble(parts[4].trim());
        double b = Double.parseDouble(parts[5].trim());
        RandomVariableDistribution rv = new RandomVariableDistribution(dist,a,b);
        rvd[r][c] = rv;
        double m = mean(dist,a,b);
        double std = sd(dist,a,b);
        double cc = c2(dist,a,b);
        adjMatMean[r][c] = m;
        adjMatOpt[r][c] = m - std;
    }
}

```

```

        adjMatPest[r][c] = m + std;
        adjMatDPest[r][c] = m + 2 * std;
        adjMatStable[r][c] = cc;
        adjMatRisk[r][c] = a + b;
    }

    // Always close files.
    bufferedReader.close();
}
catch(FileNotFoundException ex) {
    System.out.println("Unable to open file '" + fileName + "'");
}
catch(IOException ex) {
    System.out.println("Error reading file '" + fileName + "'");
}

//Passing the adjacency matrix to the Dijkstra algorithm and printing the path
printPath(dijkstraShortestPath(adjMatMean,startVertex,endVertex),startVertex,endVertex,"Mean",r,vd);

printPath(dijkstraShortestPath(adjMatOpt,startVertex,endVertex),startVertex,endVertex,"Optimist",r,vd);

printPath(dijkstraShortestPath(adjMatPest,startVertex,endVertex),startVertex,endVertex,"Pessimist",r,vd);

printPath(dijkstraShortestPath(adjMatDPest,startVertex,endVertex),startVertex,endVertex,"DoublePessimist",r,vd);

printPath(dijkstraShortestPath(adjMatStable,startVertex,endVertex),startVertex,endVertex,"Stable",r,vd);

printPath(dijkstraShortestPath(adjMatRisk,startVertex,endVertex),startVertex,endVertex,"AlphaBeta",r,vd);

}

}

```

Output for Input3 file on blackboard

```

Shortest Distance 130.99980100222098
Shortest path: Mean
Hop Count 6
Mean - std 69.91709623134523
Mean 130.99980100222098
Mean + std 192.08250577309673
Mean + 2 * std 253.16521054397248
C2 2.267162638587282
1->3->5->7->8->10->12->
Shortest Distance 69.91709623134524
Shortest path: Optimist
Hop Count 6
Mean - std 69.91709623134523

```


Mean 130.99980100222098
 Mean + std 192.08250577309673
 Mean + 2 * std 253.16521054397248
 C2 2.267162638587282
 1->3->5->7->8->10->12->
 Shortest Distance 142.0
 Shortest path: Pessimist
 Hop Count 6
 Mean - std 140.0
 Mean 141.0
 Mean + std 142.0
 Mean + 2 * std 143.0
 C2 0.002066115702479339
 1->2->4->7->9->10->12->
 Shortest Distance 143.0
 Shortest path: DoublePessimist
 Hop Count 6
 Mean - std 140.0
 Mean 141.0
 Mean + std 142.0
 Mean + 2 * std 143.0
 C2 0.002066115702479339
 1->2->4->7->9->10->12->
 Shortest Distance 0.002066115702479339
 Shortest path: Stable
 Hop Count 6
 Mean - std 140.0
 Mean 141.0
 Mean + std 142.0
 Mean + 2 * std 143.0
 C2 0.002066115702479339
 1->2->4->7->9->10->12->
 Shortest Distance 110.311111
 Shortest path: AlphaBeta
 Hop Count 6
 Mean - std 106.00000000000001
 Mean 136.00000900000902
 Mean + std 166.00001800001803
 Mean + 2 * std 196.00002700002702
 C2 0.5534264987426223
 1->3->4->6->9->10->12->

Output for Input4 file on blackboard

Shortest Distance 136.99971900524895
 Shortest path: Mean
 Hop Count 6
 Mean - std 29.0
 Mean 136.99971900524895
 Mean + std 244.9994380104979
 Mean + 2 * std 352.99915701574685
 C2 5.0

```

1->3->5->6->8->11->12->
Shortest Distance 29.0
Shortest path: Optimist
Hop Count 6
Mean - std 29.0
Mean 136.99971900524895
Mean + std 244.9994380104979
Mean + 2 * std 352.99915701574685
C2 5.0
1->3->5->6->8->11->12->
Shortest Distance 165.0414278843754
Shortest path: Pessimist
Hop Count 6
Mean - std 140.95854811567264
Mean 152.99998800002402
Mean + std 165.0414278843754
Mean + 2 * std 177.08286776872677
C2 0.11412809705921377
1->2->4->6->9->11->12->
Shortest Distance 177.08286776872677
Shortest path: DoublePessimist
Hop Count 6
Mean - std 140.95854811567264
Mean 152.99998800002402
Mean + std 165.0414278843754
Mean + 2 * std 177.08286776872677
C2 0.11412809705921377
1->2->4->6->9->11->12->
Shortest Distance 0.11412809705921377
Shortest path: Stable
Hop Count 6
Mean - std 140.95854811567264
Mean 152.99998800002402
Mean + std 165.0414278843754
Mean + 2 * std 177.08286776872677
C2 0.11412809705921377
1->2->4->6->9->11->12->
Shortest Distance 87.232007
Shortest path: AlphaBeta
Hop Count 6
Mean - std 29.0
Mean 136.99971900524895
Mean + std 244.9994380104979
Mean + 2 * std 352.99915701574685
C2 5.0
1->3->5->6->8->11->12->

```

References

www.coursera.com