Operations Research Final Project

Optimal Businessman's travelling problem

Hongyang Yu
Daniel Lu

Special Thanks to Professor Alan Frieze

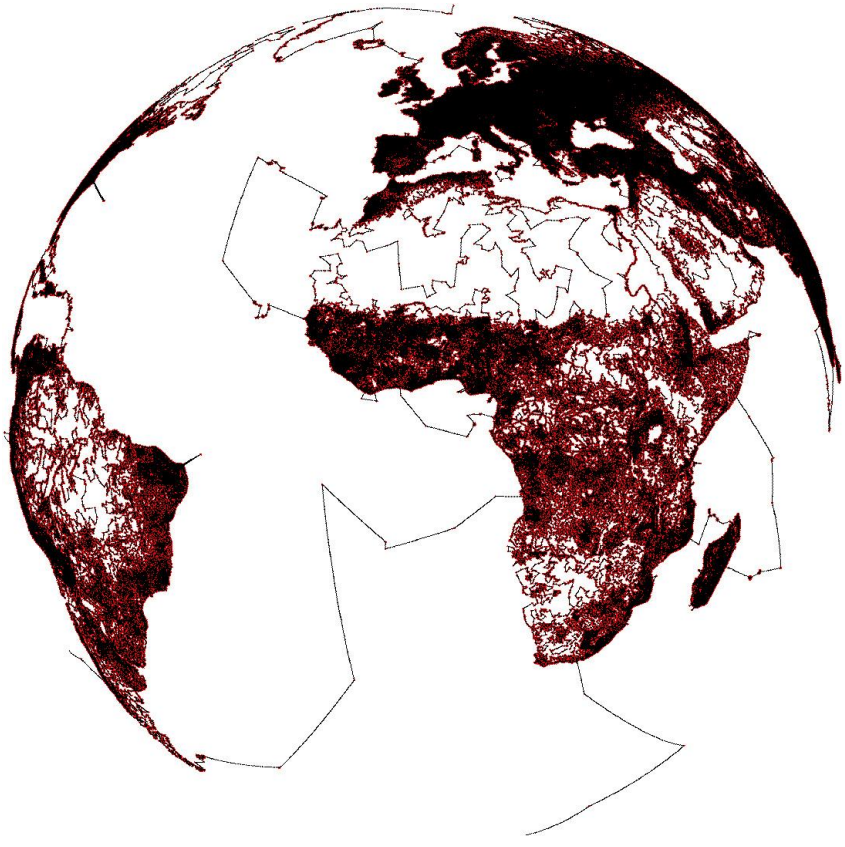

Table of Contents

Introduction

Inspired by the shortest path problem from lectures, we found it interesting to explore this idea and apply it to daily use. The shortest path problem is the problem of finding a path between two vertices in a graph such that sum of the weights of its constituent edges is minimized. In the original problem, the weights represent the distance between the vertices, and we aim to minimize the total length of the path. However, this simplified version, despite significant in theory, doesn't cooperate well with the real world problems. In reality, many other factors have to be considered besides the distance, for instance lodging cost, dining cost, total time travelled, etc. For a businessman on errand to travel from London to Shanghai, he could be asked to minimize the total cost of staying the night in a city, as well as the total distance (which is equivalent to total time, assuming no lay off time). Thus, we incorporate the extra constraints into the weights between the vertices. Since we have both the distance and costs non-negative, the arc length in the graph with cities as vertices are also non-negative, and therefore we are safe to apply Dijkstra's algorithm.

Problem Set-Up

In this project, we pick in total 30 cities and collect the travelling distances among each other. Moreover to accommodate the extra costs, we combine the lodging cost and dining cost and use the same hotel, same date, and same currency to standardize the problem. (We finalize by picking the date 12/25/2013 in Hilton hotel, all costs are in U.S dollar, and the combined costs is calculated by hotel cost $\times$ 1.5) Figure 1 shows the city names, the corresponding code names, and the combined costs. Figure 2 shows the airline distances among the cities, in hundreds of miles.

Azores AZ $72	Baghdad BD $32	Berlin BN $107	Bombay BY $139	Buenos Aires BS $209	Cairo CO $125
Cape town CN $372	Chicago CH $95	Guam GM $140	Honolulu HU $229	Istanbul IL $204	Juneau JU $109
London LN $387	Manila MA $60	Melbourne ME $240	Mexico City MY $142	Montreal ML $122	Moscow MW $183
New Orleans NS $119	New York NY $169	Panama City PY $119	Paris PS $305	Rio de Janeiro RO $229	Rome RE $339
San Francisco SF $199	Santiago SO $159	Seattle SE $119	Shanghai SI $377	Sydney SY $239	Tokyo TO $291

Figure 1

	AZ	BD	BN	BY	BS	CO	CN	CH	GM	HU	IL	JU	LN	MA	ME	MY	ML	MW	NS	NY	PY	PS	RO	RE	SF	SO	SE	SI	SY	TO
AZ	0	39	22	59	54	33	57	32	89	73	29	46	16	83	120	45	24	32	36	25	38	16	43	21	50	57	46	72	121	73
BD	39	0	20	20	81	8	49	64	63	84	10	61	25	49	81	81	58	16	72	60	78	24	69	18	75	88	68	44	83	52
BN	22	20	0	39	74	18	60	44	71	73	11	46	6	61	99	61	37	10	51	40	59	5	62	7	57	78	51	51	100	56
BY	59	20	39	0	93	27	51	81	48	80	30	69	45	32	61	97	75	31	89	78	97	44	83	38	84	100	77	31	63	42
BS	54	81	74	93	0	73	43	56	104	76	76	77	69	111	72	46	56	84	49	53	33	69	12	69	64	7	69	122	73	114
CO	33	8	18	27	73	0	45	61	71	88	8	63	22	57	87	77	54	18	68	56	71	20	61	13	75	80	68	52	90	60
CN	57	49	60	51	43	45	0	85	88	115	52	103	60	75	64	85	79	63	83	78	70	58	38	52	103	49	102	81	69	92
CH	32	64	44	81	56	61	85	0	74	43	55	23	40	81	97	17	8	50	8	7	23	41	53	48	19	53	17	70	92	63
GM	89	63	71	48	104	71	88	74	0	38	69	51	75	16	35	75	77	61	77	80	90	76	116	76	58	98	57	19	33	16
HU	73	84	73	80	76	88	115	43	38	0	81	28	72	53	55	38	49	70	42	50	53	75	83	80	24	69	27	49	51	39
IL	29	10	11	30	76	8	52	55	69	81	0	55	16	57	91	71	48	11	62	50	68	14	64	9	67	81	61	49	93	56
JU	46	61	46	69	77	63	103	23	51	28	55	0	44	59	81	32	26	46	29	29	45	47	76	53	15	73	9	49	77	40
LN	16	25	6	45	69	22	60	40	75	72	16	44	0	67	105	56	33	16	46	35	53	2	57	9	54	72	48	57	106	60
MA	83	49	61	32	111	57	75	81	16	53	57	59	67	0	39	88	82	51	87	85	103	67	113	65	70	109	67	12	39	19
ME	120	81	99	61	72	87	64	97	35	55	91	81	105	39	0	84	104	90	93	104	90	104	82	99	79	70	82	50	4	51
MY	45	81	61	97	46	77	85	17	75	38	71	32	56	88	84	0	23	67	9	21	15	57	48	64	19	41	23	80	81	70
ML	24	58	37	75	56	54	79	8	77	49	48	26	33	82	104	23	0	44	14	3	25	34	51	41	25	54	23	70	100	65
MW	32	16	10	32	84	18	63	50	61	70	11	46	16	51	90	67	44	0	58	47	67	16	72	15	59	88	52	42	90	47
NS	36	72	51	89	49	68	83	8	77	42	62	29	46	87	93	9	14	58	0	12	16	48	48	55	19	45	21	77	89	69
NY	25	60	40	78	53	56	78	7	80	50	50	29	35	85	104	21	3	47	12	0	22	36	48	43	26	51	24	73	100	68
PY	38	78	59	97	33	71	70	23	90	53	68	45	53	103	90	15	25	67	16	22	0	54	33	59	33	31	37	93	88	84
PS	16	24	5	44	69	20	58	41	76	75	14	47	2	67	104	57	34	16	48	36	54	0	57	7	56	72	50	57	105	61
RO	43	69	62	83	12	61	38	53	116	83	64	76	57	113	82	48	51	72	48	48	33	57	0	57	66	18	69	113	84	115
RE	21	18	7	38	69	13	52	48	76	80	9	53	9	65	99	64	41	15	55	43	59	7	57	0	63	74	57	57	101	61
SF	50	75	57	84	64	75	103	19	58	24	67	15	54	70	79	19	25	59	19	26	33	56	66	63	0	59	7	61	74	52
SO	57	88	78	100	7	80	49	53	98	69	81	73	72	109	70	41	54	88	45	51	31	72	18	74	59	0	64	117	71	107
SE	46	68	51	77	69	68	102	17	57	27	61	9	48	67	82	23	23	52	21	24	37	50	69	57	7	64	0	57	77	48
SI	72	44	51	31	122	52	81	70	19	49	49	49	57	12	50	80	70	42	77	73	93	57	113	57	61	117	57	0	49	11
SY	121	83	100	63	73	90	69	92	33	51	93	77	106	39	4	81	100	90	89	100	88	105	84	101	74	71	77	49	0	48
TO	73	52	56	42	114	60	92	63	16	39	56	40	60	19	51	70	65	47	69	68	84	61	115	61	52	107	48	11	48	0

Figure 2

In this problem, we have a connected directed graph$G = (V, A)$, where set V includes all of the cities, and set A includes all of the pairwise edges. Different from the original version, the businessman has a budget constraint which must be less than or equal to L (a constant) for the total costs of the stays in any city he has stays for the night, excluding the departure city. Hence every arc length is computed by adding the distance between two cities to some constant weight λ times the max of the combined cost at the destined city minus the budget constraint and zero.

For instance, the arc length between two cities A and B is

$$l(A,B) = Dist(A,B) + \lambda \times (Combined\ Cost\ of\ Stays\ between\ A\ and\ B(incl.B) - L)^+,$$

Similarly, we have the arc length from B to A as,

$$l(B,A) = Dist(A,B) + \lambda \times (Combined\ Cost\ of\ Stays\ between\ B\ and\ A(incl.A) - L)^+$$

Following the definition above, the length of a Path $P = (x_0, x_1, x_2, \dots x_k)$, not additive as in the original Dijkstra's algorithm, is the addition of the distances plus λ times the max of the combined cost of all of the cities, excluding the departure city, minus the budget constraint and zero.

$$Distance(x_0, x_1) + Distance(x_1, x_2) + \cdots Distance(x_{k-1}, x_k) + \lambda \times (Combined\ Cost\ x_1 + \cdots Combined\ Cost\ x_k - L)^+$$

Therefore, given a departure city and arrival city, we are essentially minimizing the total distance travelled while keeping the whole trip's cost under the budget, and our final goal is to find the minimum λ such that both constraints are satisfied. The λ in this problem plays the role of balancer, when λ has small value, the length of an arc is dominated by the distance constraint, and the budget constraint has little effect. As a result, when we are minimizing the total weight, we are finding the shortest path between two cities, but the budget constraint could be violated. On the other hand, if the λ has large value, the length of an arc is dominated by budget constraint, and the distance constraint has little effect. Consequently, when we are minimizing the total weight, we are finding a path such that the budget constraint is met, while we're not guaranteed the path we find is the shortest. Hence, finding the appropriate λ can give us the shortest path given each day's constraint is satisfied. Later in the paper, we will implement the Dijkstra's algorithm efficiently and taking advantage of bisection method to find the λ.

Dijkstra's Algorithm & Implementation

In this project, we decide to code our programs in Java.

To solve the problem, there are two different algorithms we must consider. The first is the extended Dijkstra's algorithm, described as follows.

Define our measure of distance as

$l(A,B) = Dist(A,B) + \lambda \times (Combined\ Cost\ of\ Stays\ Between\ A\ and\ B\ (incl.B) - L)^{+}$.

Dijkstra's algorithm will first select a starting node and assign some initial distance values and will try to improve them step by step.

1. Assign to every node a tentative distance value; set it to zero for our initial node and to infinity for all other nodes.
2. Mark all nodes unvisited. Set the initial node as current. Create a set of the unvisited nodes called the *unvisited set* consisting of all the nodes.
3. For the current node, consider all of its unvisited neighbors and calculate their *tentative* distances. In our case, we need to keep track of the actual distance of the path as well as the total cost accumulated from staying overnight in cities during the trip. Call the first distance d_1 and the second distance d_2, respectively. We then test if the new distance to an unvisited neighbor is less than the previously recorded tentative distance of *B*, then overwrite that distance. Even though a neighbor has been examined, it is not marked as "visited" at this time, and it remains in the *unvisited set*.
4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again.

5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the *unvisited set* is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
6. Select the unvisited node that is marked with the smallest tentative distance, and set it as the new "current node" then go back to step 3.

A complete implementation of our algorithm is attached in Appendix.

Bisection Algorithm

Since Dijkstra's algorithm in our problem requires a specific value for the λ parameter in order to run, we wish to find the minimum λ which gives a different path between any two given cities. In the context of our problem, finding such a λ implies that we are finding the Lagrange multiplier such that the businessman is indifferent between traveling a longer physical distance and traveling a shorter distance while staying overnight at different cities.

The algorithm to find the λ is as follows.

1. Initialize $\lambda = 0.001$.
2. While Dijkstra's algorithm continues to give the same cost for two predefined cities (in this case Azores and Manila), $\lambda \leftarrow \lambda \cdot 2$.
3. Now we have two different λ that gives two different total costs. To find the minimum λ, we apply the bisection algorithm and stop at a given threshold of 0.001.

The code for this part can also be found in the appendix.

Results & Discussion

Having implemented the revised Dijkstra's algorithm and the bisection method from the previous sections, in this section, we will display and explain some of the outputs and discuss improvements that could be made to expand this paper.

Example 1: Departure city: Azores Arrival city: Manila

Firstly, we set $\lambda = 0$, in this way, we are only considering minimizing the travelling length between the two cities while ignoring the budget constraint.

Output: Path: [Azores, Paris, Berlin, Manila]

Distance of this path is: 82.0 (thousand miles)

Moreover, we can calculate the total cost associated with this route, which in this case is $472.

Hence, if we have a budget higher than this number, we can achieve the shortest path of 82 thousand miles.

Now, let's consider a shorted budget of only $400. Our search of λ gives us 0.0835.

And the path from the output: Path: [Azores, Manila]

Distance of this path is: 83.0

From this output, we are guaranteed the direct flight from Azores to Manila with distance 83(thousand miles) is the shortest and cheapest path under the budget L = $400. And our trip's cost is $60.

To double check the algorithm's validity, we can consider another route from Azores to Paris and then to Manila. In this case, we have a travelling distance of 83(thousand miles), the same as our output from the program. However, this trip's cost adds to $365, which is way higher than the direct flight of only $60.

Example 2 Departure city: Azores Arrival city: San Francisco

We follow the same step as in example 1, and get the following results.

When $\lambda = 0$,

Output: Path: [Azores, Montreal, San Francisco]

Distance of this path is: 49.0(thousand miles)

And the cost with the route is $321, which means any budget greater than this will achieve this shortest path.

Consider a new budget of $200. We get $\lambda = 0.0455$.

Our route for this constraint is Path: [Azores, San Francisco]

Distance of this path is: 50.0(thousand miles)

Hence, we are guaranteed the shortest and cheapest path under this constraint is a direct flight from Azores to San Francisco with cost of $199.

Further Improvement

Despite that the algorithms implemented have shed us light upon solving a more realistic version of shortest path problem, we still have a lot to improve. One of the restrictions is that our data set size is limited. By including more cities and giving fewer direct flights available, we are expecting to see λ to change more dramatically and by feeding in different budget constraint we can obtain more diversified routes. In addition, we can throw in more constraints to the weight, and lessen more assumptions. For instance, instead of ignoring the lay-off time, we can add in the cost of this waiting time and the commuting cost from airport to hotel as well. Hence, we introduce more λ's and can try various ways other than bisection method to get those numbers faster and more efficiently.

Conclusion

In this paper, we have achieved a more realistic version of shortest path problem with one more budget constraints. To accomplish this, we've modified the weight of a path by adding the extra term λ times the max of the combined cost of all of the cities, excluding the departure city, minus the budget constraint and zero. We then revised the Dijkstra's algorithm and implemented the bisection method to find λ. Despite limited in data size and assumptions constraints, the approach of solving this single budget constraint problem sheds us light and paves a good way towards a more complicated multi-constraints shortest path problem.

Appendix

Graph Implementation

```
package graph;

public class Vertex implements Comparable<Vertex>
{
    private final String name;
    private Edge[] adjacencies;
    private double minDistance = Double.POSITIVE_INFINITY;
    private double minDistance2 = 0.0;
    private Vertex previous;
    private double cost;

    public Vertex(String argName, double cost) {
       name = argName;
       this.cost = cost;
    }

    public void setAdjacencies(Edge[] e) {
       adjacencies = e;
    }

    @Override
    public int compareTo(Vertex other) {
        return Double.compare(minDistance, other.minDistance);
    }

    public String getName() {
       return name;
    }

    public double getCost() {
       return cost;
    }

       public Edge[] getAdjacencies() {
              return adjacencies;
       }

       public Vertex getPrevious() {
              return previous;
       }

       public String toString() {
              return name;
       }

       public void setMinDistance(double minDist) {
              minDistance = minDist;
       }

       public double getMinDistance() {
              return minDistance;
       }

       public void setMinDistance2(double minDist) {
              minDistance2 = minDist;
       }

       public double getMinDistance2() {
              return minDistance2;
       }
```

```
        public void setPrevious(Vertex u) {
                previous = u;
        }
}
```

```
package graph;

public class Edge
{
    private final Vertex target;
    private final double weight;
    public Edge(Vertex argTarget, double argWeight) {
       target = argTarget;
       weight = argWeight;
    }

    public Vertex getTarget() {
       return target;
    }

    public double getWeight() {
       return weight;
    }
}
```

Djikstra Implementation

```
package algorithm;

import graph.Edge;
import graph.Vertex;

import java.util.PriorityQueue;
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class Dijkstra
{
        private static double lambda;
        private static double thresh;

        public Dijkstra(double _lambda, double _thresh) {
                lambda = _lambda;
                thresh = _thresh;
        }

        public void setLambda(double _lambda) {
                lambda = _lambda;
        }

        public void setThresh(double _thresh) {
                thresh = _thresh;
        }

        public double getLambda() {
                return lambda;
        }

        public double getThresh() {
                return thresh;
        }

        public static double getL2Weight(double cost) {
```

```
                return lambda*Math.max(0, cost-thresh);
        }

    public void computePaths(Vertex source)
    {
        source.setMinDistance(0.);
        PriorityQueue<Vertex> vertexQueue = new PriorityQueue<Vertex>();
        vertexQueue.add(source);

        while (!vertexQueue.isEmpty()) {
                Vertex u = vertexQueue.poll();

            // Visit each edge exiting u
            for (Edge e : u.getAdjacencies())
            {
                Vertex v = e.getTarget();
                // weight and the l2 weight
                double weight = e.getWeight() + getL2Weight(u.getMinDistance2());
                double distanceThroughU;
                if(u.getPrevious() == null) {
                      distanceThroughU = u.getMinDistance() + weight;
                } else {
                      distanceThroughU = u.getMinDistance() + weight -
getL2Weight(u.getPrevious().getMinDistance2());
                }

                            if (distanceThroughU < v.getMinDistance()) {
                                vertexQueue.remove(v);
                                v.setMinDistance2(u.getMinDistance2() + v.getCost()) ;
                                v.setMinDistance(distanceThroughU) ;
                                v.setPrevious(u);
                                vertexQueue.add(v);
                            }
            }
        }
    }

    public List<Vertex> getShortestPathTo(Vertex target)
    {
        List<Vertex> path = new ArrayList<Vertex>();
        for (Vertex vertex = target; vertex != null; vertex = vertex.getPrevious())
            path.add(vertex);
        Collections.reverse(path);
        return path;
    }
}
```

Our Program with Bisection Method

```
package main;

import graph.Edge;
import graph.Vertex;

import java.util.List;
import java.util.Scanner;

import algorithm.Dijkstra;
import util.ParseData;

public class Main {

        public static int cityIndex(String city, String[] cities) {
                for(int i = 0; i < cities.length; i++) {
                        if(cities[i].equals(city))
                                return i;
```

```
		}
		return -1;
	}

	private static double costPath(String startCity, String endCity, String[] name,
int[] px, int[][] dist, double lambda) {
		double cost = 0.0;

		Vertex[] vertices = new Vertex[30];
		int index = 0;

		for(int i = 0; i < 30; i++) {
			vertices[i] = new Vertex(name[i], px[i]);
		}

		for(int i = 0; i < 30; i++) {
			Edge[] e = new Edge[29];
			for(int j = 0; j < 30; j++) {
				int d = dist[i][j];
				if(d != 0) {
					//System.out.println(vertices[j].getName());
					e[index] = new Edge(vertices[j], d);
					index++;
				}
				vertices[i].setAdjacencies(e);
			}
			index = 0;
		}

		// Read the starting and ending vertices

		int startIndex = cityIndex(startCity, name);
		int endIndex = cityIndex(endCity, name);

			System.out.println(lambda);
		Dijkstra dk = new Dijkstra(lambda, 400);
		dk.computePaths(vertices[startIndex]);
		List<Vertex> path = dk.getShortestPathTo(vertices[endIndex]);
		//System.out.println("Path: " + path);

		double expenseCost = 0.0;
		double distCost = 0.0;

	    for(int i = 1; i < path.size(); i++) {
		int pIndex = cityIndex(path.get(i-1).getName(), name);
		int ind = cityIndex(path.get(i).getName(), name);
		distCost += dist[pIndex][ind];
		if(i != path.size() - 1) {
			expenseCost += px[ind];
		}
	    }
	    cost = expenseCost + distCost;
	    return cost;
	}


	private static double computeLambda(String[] name, int[] px, int[][] dist) {
		String startCity = "Azores";
		String endCity = "Manila";
		double lambda = 0.001;
		double cost;
		do {
			lambda *= 2;
			cost = costPath(startCity, endCity, name, px, dist, lambda);
	    } while (cost >= 494.0);
```

```
        double lambdaA = lambda/2;
        double lambdaB = lambda;
        while(lambdaB-lambdaA > 0.001) {
                lambda = (lambdaA + lambdaB)/2.0;
                cost = costPath(startCity, endCity, name, px, dist, lambda);
                if(cost >= 494.0) {
                        lambdaA = lambda;
                } else {
                        lambdaB = lambda;
                }
        }

    return lambda;
}

public static void main(String[] args)
{

ParseData pd = new ParseData();
// names of the cities
        String[] name = pd.parseName();
        // cost of the cities
        int[] px = pd.parseCost();
        // distances between cities
        int[][] dist = pd.parseDist();

        Vertex[] vertices = new Vertex[30];
        int index = 0;

        for(int i = 0; i < 30; i++) {
                vertices[i] = new Vertex(name[i], px[i]);
        }

        for(int i = 0; i < 30; i++) {
                Edge[] e = new Edge[29];
                for(int j = 0; j < 30; j++) {
                        int d = dist[i][j];
                        if(d != 0) {
                                //System.out.println(vertices[j].getName());
                                e[index] = new Edge(vertices[j], d);
                                index++;
                        }
                        vertices[i].setAdjacencies(e);
                }
                index = 0;
        }

        // Read the starting and ending vertices

        Scanner in = new Scanner(System.in);
        System.out.print("Please enter your starting destination: ");
        String startCity = in.nextLine();
        int startIndex = cityIndex(startCity, name);
        System.out.print("Please enter your ending destination: ");
        String endCity = in.nextLine();
        int endIndex = cityIndex(endCity, name);
        System.out.print("Please enter your budget: ");
        double budget = in.nextDouble();
        Dijkstra d = new Dijkstra(0.083, budget);

        d.computePaths(vertices[startIndex]);
        List<Vertex> path = d.getShortestPathTo(vertices[endIndex]);
        System.out.println("Path: " + path);

        double expenseCost = 0.0;
        double distCost = 0.0;
```

```
            double cost;
        for(int i = 1; i < path.size(); i++) {
            int pIndex = cityIndex(path.get(i-1).getName(), name);
            int ind = cityIndex(path.get(i).getName(), name);
            distCost += dist[pIndex][ind];
            if(i != path.size() - 1) {
                    expenseCost += px[ind];
            }
        }
        cost = expenseCost + distCost;
        System.out.println("Distance of this path is:  " + distCost);
        System.out.println("Cost of this path is: " + cost);

        in.close();

        System.out.println(computeLambda(name, px, dist));
    }
}
```

Utility to Parse Files

```
package util;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.StringTokenizer;

public class ParseData {

        public int[] parseCost() {
                int[] ret = new int[30];
                Object sCurrentLine;
                int index = 0;

                BufferedReader tb;
                try {
                        tb = new BufferedReader(new FileReader("ha30_prices.txt"));
                        while ((sCurrentLine = tb.readLine()) != null) {
                                ret[index] = Integer.parseInt((String)sCurrentLine);
                                index++;
                        }
                        tb.close();
                } catch (FileNotFoundException e) {
                        e.printStackTrace();
                } catch (NumberFormatException e) {
                        e.printStackTrace();
                } catch (IOException e) {
                        e.printStackTrace();
                }
                return ret;
        }

        public String[] parseName() {
                String[] ret = new String[30];
                Object sCurrentLine;
                int index = 0;

                try {
                        BufferedReader tb = new BufferedReader(new
FileReader("ha30_name.txt"));
                        while ((sCurrentLine = tb.readLine()) != null) {
                                ret[index] = (String)sCurrentLine;
                                index++;
                        }
```

```
                        tb.close();
                } catch (FileNotFoundException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
                return ret;
        }

        public int[][] parseDist() {

                int[][] ret = new int[30][30];
                Object sCurrentLine;
                String delims = " ";
                // reset index to 0 after every row
                int i = 0, j = 0;

                try {
                        BufferedReader tb = new BufferedReader(new
FileReader("ha30_dist.txt"));
                        while ((sCurrentLine = tb.readLine()) != null) {
                               StringTokenizer st = new StringTokenizer((String)sCurrentLine,
delims);
                            while (st.hasMoreTokens()) {
                                ret[i][j] = Integer.parseInt(st.nextToken());
                                j++;
                            }
                            i++;
                            j = 0;
                        }
                        tb.close();
                } catch (FileNotFoundException e) {
                        e.printStackTrace();
                } catch (NumberFormatException e) {
                        e.printStackTrace();
                } catch (IOException e) {
                        e.printStackTrace();
                }
                return ret;
        }
}
```