Kennesaw State University

Department of Computer Science
CS 6045 - Adv. Algorithms

Freight Delivery System
(Final Report)

Team:

Carlos A. Cepeda

Freight Delivery System

(FDS)

Freight Delivery System

1. Abstract

The Freight Delivery System -FDS, which was defined on the proposal document, is analyzed and the proposed solution is described and successfully tested. In general, given the characteristics of the FDS problem and the already known Travelling Salesman Problem -TSP and the Shortest Path Problem-SPP, two main transformations were designed to reduce the FDS to a type of TSP as described on section 4. The two main transformations, which are already formally proved, consist on transform the initial partial connected graph of cities-nodes (including the client/cities-nodes), to a full connected graph of just client-nodes using already known algorithms to solve the SPP (it is proved that optimal solution to FDS will contain the shortest paths between client-nodes). The second transformation to make equivalent our problem to TSP is adding a "dummy" node connected with some specific weights as described on 4.2 and 4.3. With this last transformation any algorithm to solve TSP can be used and their solutions (with some small operations) will be the FDS solution. In addition, the algorithms for these two transformations are already implemented on Java and a GUI was implemented and tested on a graph with 31 cities and 3, 6, 10, 15, and 17 client nodes. Results shows the correctness of the designed algorithm.

2. Introduction

As we know, this project is based on a modified Travelling Salesman Problem combined with the Shortest Path Problem, which are an NP-Complete routing problems. In order to introduce the general idea of this problem, consider for example that there is a central bulk storage center at the destination city (let say Los Angeles) and starting on the Base City (let say Atlanta), we use a freight truck to pick up packages that need to be delivered to Los Angeles. Thus, given a map (graph) with the cities of a region where clients could be located (and where we can find the distance between cities), the objective is to maximize the profit, which, as it will be shown next, depends on the length of the paths taken along the route. Consequently, this algorithm is intended to find the route and the pickup sequence that maximize the profit.

3. Formal Problem Definition

- A graph (nodes and edges) is required. (See Figure 1)
- Each node "N_i" represent cities on a region where possible clients could be found.
- Each Edge $E_{i,j}$ (path from N_i to N_j) has the next properties:

  Average speed = S

  Distance or length = d
- The distance "d" between cities will be gotten using the web maps tools for calculating car millage distances between cities. The map/graph with the main routes and distances will be used. **(Note: regarding to the original proposal, this part change given that**

we consider it is going to be more real than calculating distances based on geographical coordinates.

- The minimal distance between the client nodes and the destination city "D" will be used to calculate the revenue (R).

- Each possible Client-package will have:

Weight = W
Volume = V
Location coordinates given by longitude and latitude.
Revenue $R = k_1 * W * V * D$, where k_1 is a constant that represent the price in dollars per pound, per cubic meter and per mile.
Position on the Node = Ci ("i" stand by the number of the node where the client is located)

- The Cost:
  Fixed Cost of operation (Cf)
  Cost of Gas (Cg) $Cg = price.gas.per.mille * d$
  Driver Salary $Ds = Totalhours * (Salary.per.hour) = \left(\frac{d}{S}\right)(Salary.per.hour)$
  Depreciation per millage $= k_2 * (distance)$; where k_2 is a constant that represent the depreciation factor for a given vehicle per mile.

- The possible paths can be described as follow:

$$PathEdges_i = \{E_{i,k}, E_{k+1,j}, \dots, E_{n,m}\}$$

$$PathNodes_i = \{N_i, N_{k,}, \dots, N_n\}$$

$$Path.Clients_i = \{C_i, C_{p,}, \dots, N_n\}$$

- The function to optimize is the profit as follow:

Profit = Total Revenue – Total Cost

$$TotalCost = Cf + \sum_{.path} E_{i,j}\{price.gas.per.mille * d + \left(\frac{d}{S}\right)(Salary.per.hour) + k_2 * (d)\}$$

$$\pm Total.Revenue = \sum_{Client.\in.path} C_i\{k_1 * W_i * V_i * D_i\}$$

Here it is important to note that the "d" and "S" are variables than depend of the edge, but if we calculate the cost per edge, it will be fixed during running time, so the problem will be reduced to minimize the cost. Consequently, in order to find the optimal route, we need just to find:

$$Minimum \sum_{.path} E_{i,j}\{\left(price.gas.per.mille + k_2 + \frac{Salary.per.hour}{S_i}\right) * d_{i,j}\}$$

- Restrictions:
  - Edges visited on the path must contain the Start node as start point and the Destination node as the last node.
  - Path.Clients are the clients along the path.
  - **(Note: we remove the restriction “All the edges on the path need to be connected” given that in order to work with real data, cities could be not connected to each other directly by a route).**

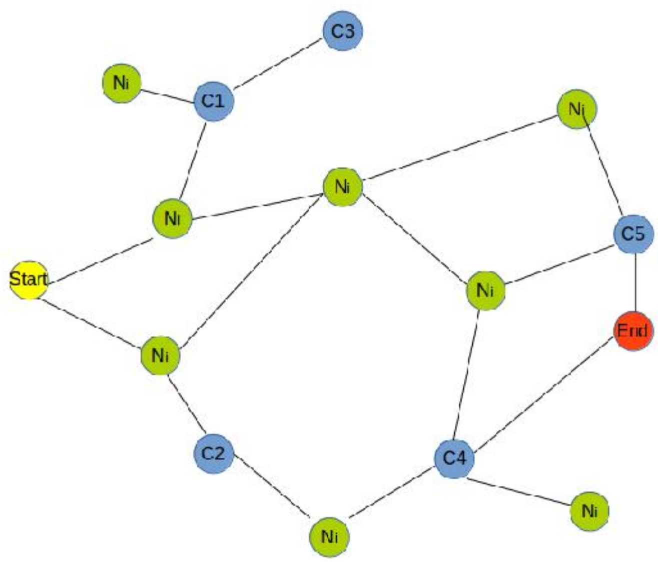

Figure 1. *Example Graph representing the problem*

4. Proposed Solution

The “Travelling Salesman Problem” is intended to find the shortest route that visit every node (city) exactly one time and come back to the starting point. This is a NP-hard problem and the worst case running time is known increase super-polynomially with the number of nodes to visit.

With this in mind, if we compare the TSP with the current problem proposed in this document, it is possible to see that there are two main differences as follow:

- First one is the number of nodes to visit. In our case, not every node has to be visited, just the nodes-cities where clients are located. However, we solve this difference with the process proposed on the section “4.1 Reducing to a Full connected graph”.
- Second, the problem proposed does not consider coming back to the starting node as the TSP problem requires.

Thus, after having analyzed the proposed problem, we proposed a solution that consist on two main steps and where we can reduce the solution to combining current algorithms to solve the “Shortest Path Problem – undirected graphs” and the “Travelling Salesman Problem”. The steps will be explained ahead.

4.1 Reducing to a Full connected graph:

Given that the optimal solution consider just the nodes where the clients are located, and the fact that any optimal solution will walk along paths that connect client-nodes with the shortest length, it is valid to reduce the initial partial connected graph into a full connected graph with just client nodes, where each path between each pair of client-nodes will have a length equivalent to the shortest length calculate from the initial graph.

Prove: Suppose that the optimal solution consider a path between client-node "j" and the client node "k" where the length is not the shortest length between these two nodes. If we replace that path for the shortest one, the total length of the route will be reduced, and given that the "Profit" increase if the lengths of the paths traversed decrease, the new route will have a higher profit, which is a contradiction.

Thus, we need to transform our initial graph to a full connected graph just with the client-nodes and where each path will be the shortest distance between each couple of nodes. The figure 2 shows this transformation. (Note: on figure 2, not every connection is shown – just for simplification small traces of the paths are displayed).

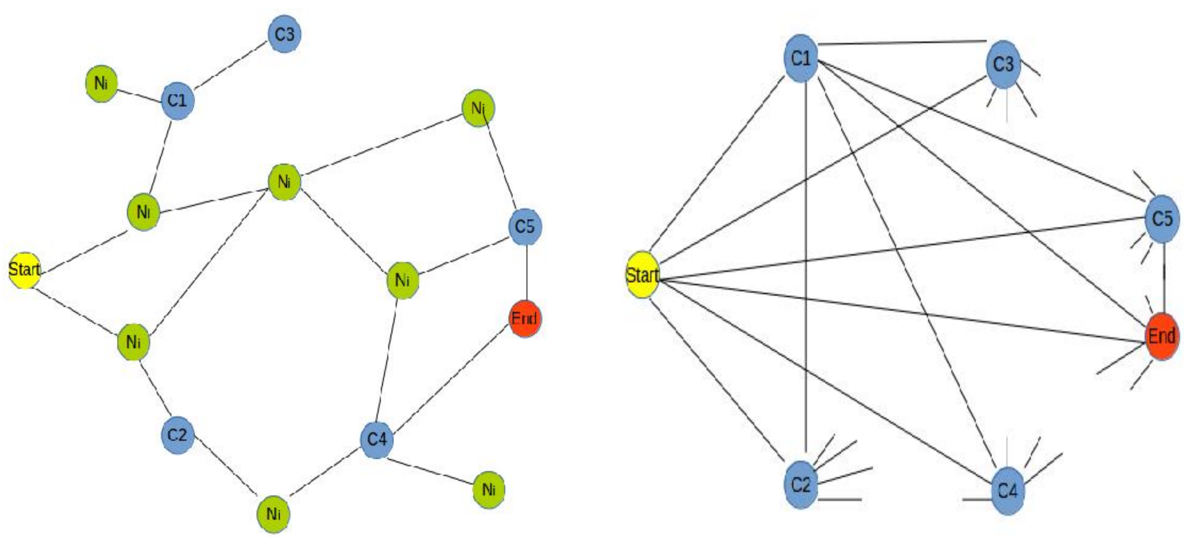

Figure 2. *Transforming the original graph to a full connected graph with just client nodes.*

With this idea in mind, I propose to use current algorithms to solve the "Shortest Path Problem" in order to find the shortest paths between client-nodes.

As it is known, The Shortest Path Problem consist of finding a path between two nodes such that the sum of the "weights" (in this case the length) of its constituent edges is minimized. There are several algorithms to solve this problem, just for mention Dijkstra's algorithm, Bellman–Ford algorithm, Floyd–Warshall algorithm, Johnson's algorithm, Viterbi algorithm, among others.

First, I decided to use at the beginning the Dijkstra's algorithm given that it is enough to solve the single-source shortest path problem. In fact, I am tried to use the Dijkstra's algorithm based on min-priority queue implemented by a Fibonacci heap. The Dijkstra's algorithm find the shortest path picking the unvisited node on the neighborhood with the lowest distance; then it calculates the distance through it to each unvisited neighbor, and updates the neighbor's distance if smaller.

The worst time for this algorithm is O(E + N logN), where E is the number of edges and "N" is the number of nodes on the graph. However, because we need to calculate the shortest path for each pair of client-nodes, it will requires to run this algorithm (C-1)+(C-2)+...+1 times (C represent the number of client-nodes). In other words, it will require (C-1)(C)/2 runs for the Dijkstra's algorithm. Thus, the worst case scenario is produced when "E" is maximum and the number of client-nodes reach the number of nodes on the original graph (C=N).

E maximum (full connected Graph) = (N-1)+(N-2)+...+1 = (N-1)(N)/2

Now, in big O notation, Worst Case is:

$$\frac{N^2-N}{2} * O\left(\frac{N^2-N}{2} + N * logN\right) = O(N^4)$$

However, it could be possible to use the Floyd–Warshall algorithm, which was designed to calculate all pairs shortest paths, has a better worst case scenario. It use techniques as dynamic programming where solution can be recursively calculate in terms of partial solutions that are re-used.

Finally, the worst case scenario for the previous algorithm is $O(N^3)$as proved on reference [1] pag 695, section 25.2 "The Floyd-Warshall algorithm".

4.2 Reducing to "Traveling Salesman Problem" - TSP

In order to make equivalent our problem (Freight Delivery System – FDS) to the TSP and then to use the already existing algorithms to solve the FDS, I propose the next methodology.

Given a full undirected connected graph of client-nodes (as a result of the process described on 4.1), we can add to the graph a "dummy" node as shown on the figure 5. Also, we need to consider that connections between this "dummy" node and other nodes have to have weighted paths following some rules that will be explained ahead.

First consider the next example of a possible full connected graph of a client nodes.

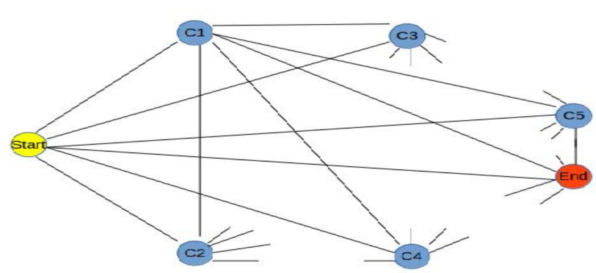

Figure 3. *Full connected graph resulting from 4.1*

Next, consider L1 as a possible solution for the TSP and L2 as solution to the problem that we are looking for.

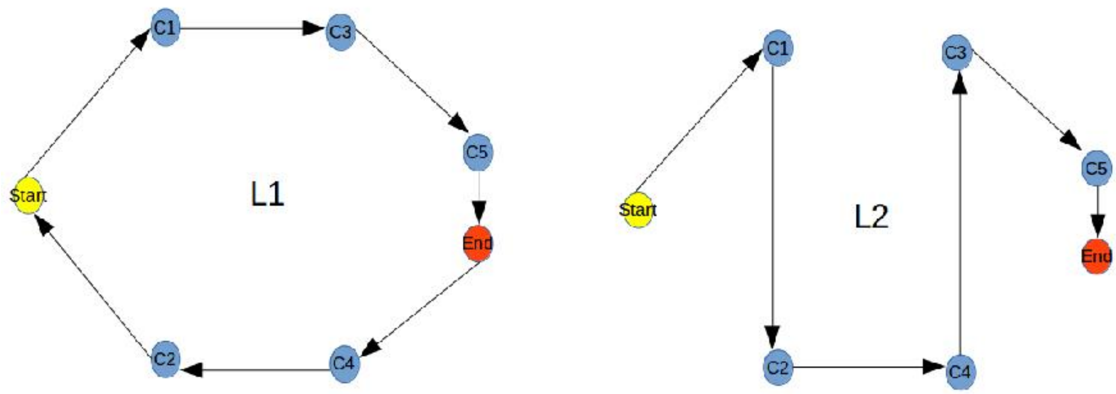

Figure 4. *a) Possible solutions for TSP b) Possible Solution to our problem*

Now, consider adding a "dummy" node -D-, with weights to the paths that connect to the client nodes equal to "α", weighted path that connect to the destination node equal to "β", and weighted path that connect to the starting node equal to "λ" as shown next:

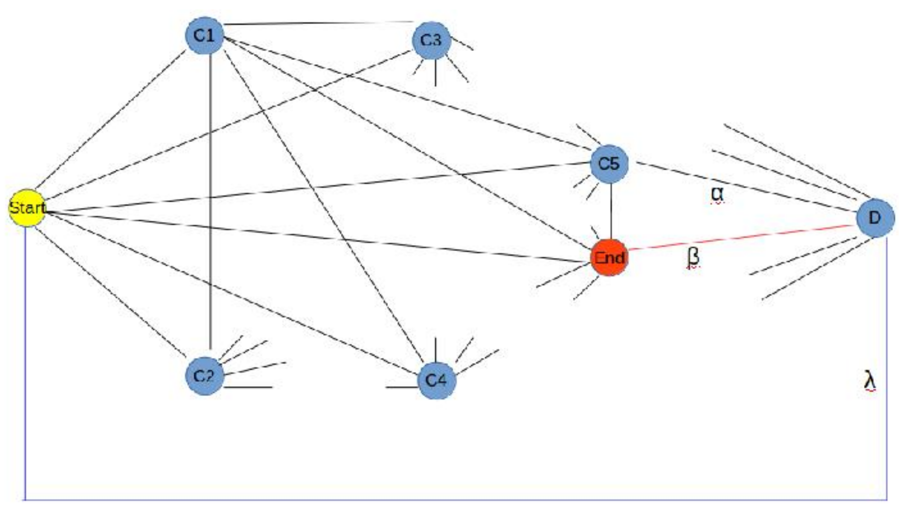

Figure 5. *Modified graph for applying TSP to solve the FDS*

Then, we can estimate possible approximate solutions to the TSP applied to the previous graph as follow:

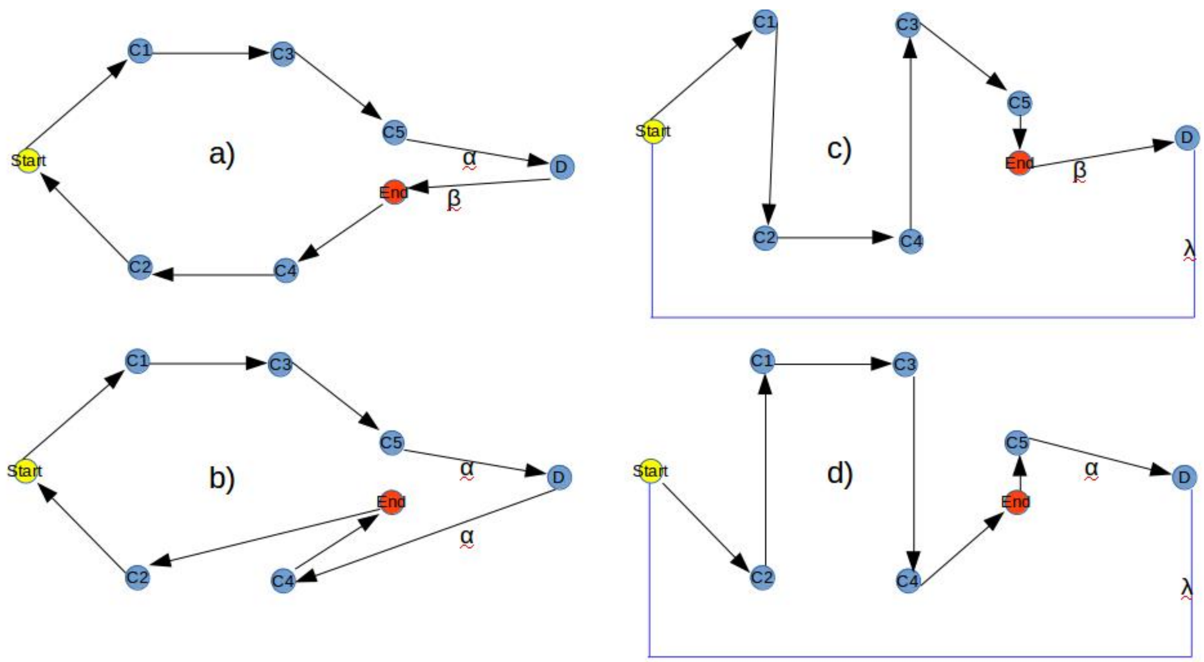

Figure 6. *Possible approximate TSP solutions to the graph on Figure 5.*

From the figure above, we can see that possible approximate solutions to the TSP are these four cases described as follow:

Case a). Approximate Solution: L1+α+β

Case b). Approximate Solution: L1+α+α

Case c). Approximate Solution: L2+β+λ

Case d). Approximate Solution: L2+α+λ

The before is true along as we make the following assumptions:

- L1 << α+β (1)
- L1 << α+α (2)
- L2 << β+λ (3)
- L2 << α+λ (4)

Now, due to we are looking for the solution to the case "C", we need to define the relations between these variables. Consequently the next inequalities can be established:

- L2+β+λ < L1+α+β (5)
- L2+β+λ < L1+2α (6)
- L2+β+λ < L2+α+λ (7)

As we now, it will not possible to solve these inequalities without reducing the number of variables. However, we can consider L1 and L2 first; here we don't know if L1 is bigger than L2 or vice-versa, so assuming that we know which is the biggest one, let say we name to this variable "L", we can solve the relations for the other variables:

- L+β+λ < L+α+β (8)
- L+β+λ < L+2α (9)
- L+β+λ < L+α+λ (10)

From (8): λ < α (11)

From (10): β < α (12)

From (9): β+λ < 2α, which always will be accomplished given (11) and (12).

Now, re-taken the problem about knowing which L1 or L2 is bigger, we can consider replacing L1 and L2 for the longest possible route that join every node of the graph shown in the figure 3. Remember that objective is to force the algorithm find the shortest path solution that go through the path "β" and then through the path "λ". With this in mind, we could replace "L" by let say the longest possible path, so any variation of L1 or L2 could be consider. Now, assume that we would need to solve "Longest Path Problem - LPP" for the graph on figure 3, but we know that LPP is NP-hard, which implies that it cannot be solved in polynomial time, so it is even harder than solving TSP. Fortunately, we don't need the exact solution to the LPP, we just need some length that we can guaranty that is bigger than L1 and bigger than L2, so making a reasonable assumption for "Longest Path - LP", so we can consider "LP" as the sum of every path on the graph, will could be even bigger than the solution to LPP.

Summarizing, we can redefine the graph on the figure 5 with the next variables as follow:

$LP = \sum_{path} E_{i,j}\{\}$ (15)

β = λ = LP (16)

α = 1000β (17)

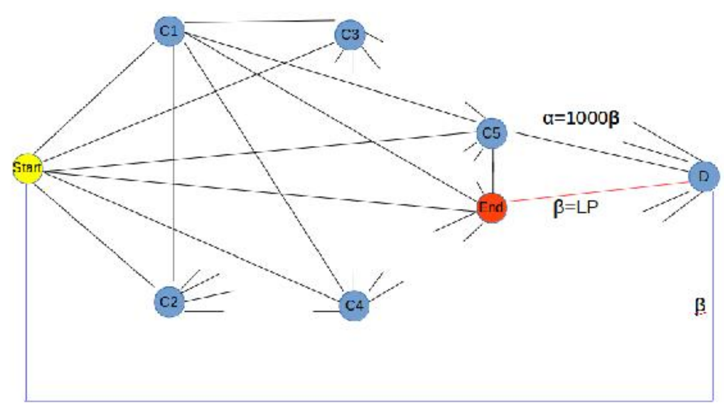

Figure 7. *Modified graph for applying TSP to solve the FDS with the defined variables*

Thus, with the previous transformation, we can apply the algorithms to solve the TSP to this configuration and the solution will contain the solution to the FDS just removing the two paths "β".

4.3 Applying the algorithms to solve TSP:

There are many algorithms to solve the TSP, which goes from the "brute force search", exact algorithms as the "Held-Karp Algorithm" and the "Branch and Bound", and heuristic algorithm as the "Nearest Neighbor algorithm" and the "Christofides algorithm", among others.

For the FDS, I am going to consider the "Held-Karp Algorithm" given that is a type of exactly algorithm that use dynamic programming and given that it is no too much complex to program and the number of variables that we consider are not too large.

The "Held-Karp Algorithm" is based on the property that "e*very subpath of a path of minimum distance is itself of minimum distance".* Thus, finding the solutions to the subproblems starting by the smallest ones, progressively will find the solution. Next, a brief description of the "Held-Karp Algorithm" is as follow (taken from [3]):

Suppose directed graph G=(V,E) with "n" vertices and a non-negative length function $\ell: E \rightarrow R+$. For any vertices s and t, and any subset X of vertices that excludes s and t, let L(s,X,t) denote the length of the shortest Hamiltonian path from s to t in the induced subgraph $G[X \cup \{s,t\}]$. The Bellman-Held-Karp algorithm is based on the following recurrence:

$$L(s,X,t)=\{\ell(s,t) \text{ if } X=\emptyset \qquad [3]$$

$$L(s,X,t)=\{\min_{v\in X}(L(s,X\backslash\{v\},v)+\ell(v,t)) \quad \text{otherwise} \qquad [3]$$

Thus, for any vertex s, the length of the optimal traveling salesman tour is $L(s,V\backslash\{s\},s)$.

Time Complexity for TSP:

As we know, the force brute search for TSP takes n!, which is approximately:

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n.$$

For the "Held-Karp Algorithm", it is also known that the Worst Time Complexity is:

$O(2^N * N^2)$ and the space $O(2^N * N)$

Note: demonstrations of these equations are not displayed given that can be found on several textbooks or on-line documents.

4.4 Summarizing Time Complexity

As it was explained, the proposed solution for the FDS consist on two main steps. From section "**4.1",** applying the "Dijkstra's algorithm" takes $O(N^4)$and the space $O(N^2)$. Also, from section **"4.3"**, the "Held-Karp Algorithm" will take $O(2^N * N^2)$ and the space $O(2^N * N)$.

Consequently, given that these two main processes need to be applied on sequence, the Worst Time Complexity will be $O(N^4)$+$O(2^N * N^2)$, which is $O(2^N * N^2)$and maximum space $O(2^N * N)$.

5. Implementation

After having designed the process to solve the FDS, I started the programming phase on JAVA, which is a language that is familiar to me. **Note: as a group we had decided to use C++, however due to I am not longer part of a group, I change the programming language to use.**

5.1 Input Data sources

A. Region of interest and Distances between cities:

I started by selecting an appropriate number of cities to conform our graph. Thus, a fast intuitive analysis shows that given the time complexity $O(2^N * N^2)$, solving graphs for more than 20 client nodes will take more than 1 day running the algorithms (assuming 1 millisecond per operation).

In addition, I consider enough 31 as a total number of cities (possible cities with clients plus cities without clients). Thus, I choose the next region and cities shows on the next figure:

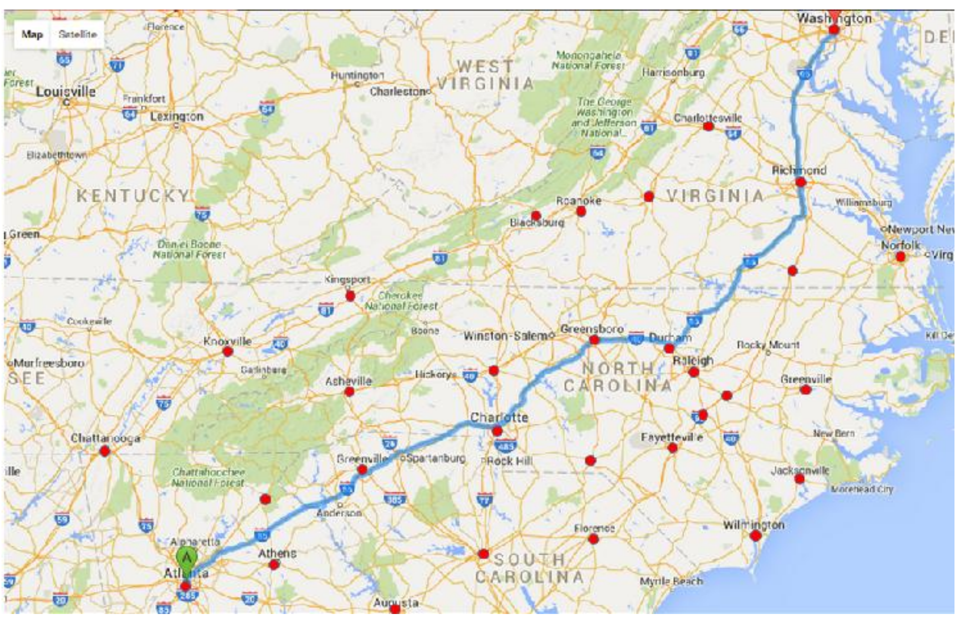

Figure 8. Map of the region of interest.

As we can see, the choose region goes from Atlanta to Washington, and I selected a total of 31 cities along the main route. The distances between cities were gotten using Google Maps finding the minimum distances between cities. It is important to mention that any path between cities that cross other of the chosen cities was obviated, so the initial graph build is a no full connected graph.

Next Figure displayed the paths between the chosen cities.

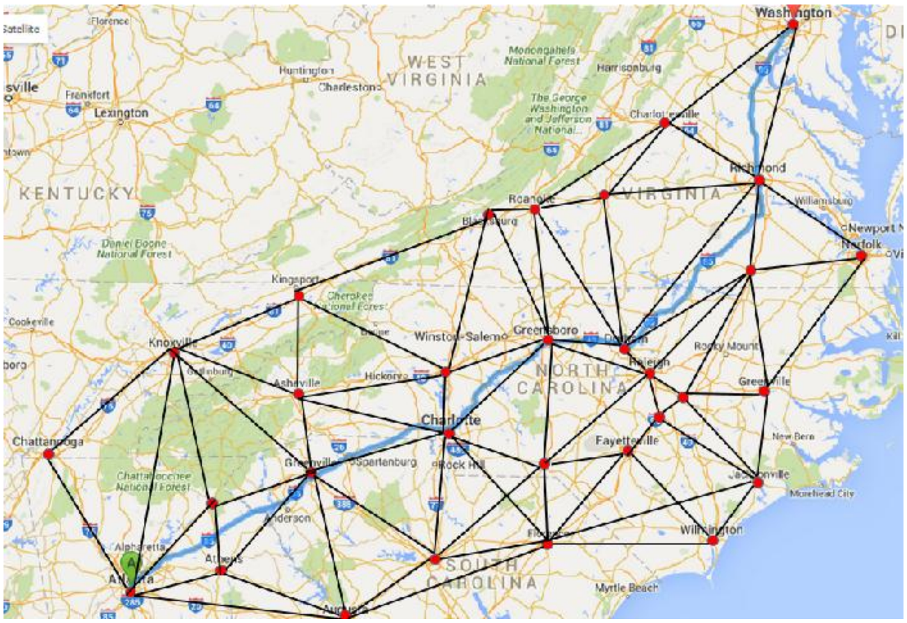

Figure 9. Map of the region of interest – connection routes.

Finally, a matrix was built with the distances between cites gotten as explained before. The distances that correspond to the paths not considered on the previous map were fixed to a big numbers (1000 Millions). Given that the matrix is too big, just a picture of this one will be displayed ahead.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Atlanta	1	0	###	72	143	###	###	###	###	118	###	85	###	###	###	###	###	###	###	###	###	###	###	###	###	###	###	###	###	###	###	###
Asheville	2	###	0	###	###	###	###	###	###	###	###	###	###	###	###	###	64	###	###	###	82	116	###	###	###	###	###	###	107	###	###	###
Athens	3	72	###	0	95	###	###	###	###	###	###	50	###	###	###	###	93	###	###	###	###	###	###	###	###	###	###	###	###	###	###	###
Augusta	4	143	###	95	0	###	###	###	###	###	74	###	###	###	###	147	###	###	###	###	###	###	###	###	###	###	###	###	###	###	###	###
Benson	5	###	###	###	###	0	###	###	###	###	###	###	###	###	30	###	###	###	###	97	###	###	###	###	31	###	###	###	###	###	41	###
Blacksburg	6	###	###	###	###	###	0	###	###	###	###	###	###	###	###	###	###	144	###	###	150	###	###	###	###	###	37	###	125	###	###	###
Charlotte	7	###	###	###	###	###	###	0	###	###	91	###	###	###	###	114	100	87	###	###	###	###	###	###	###	###	###	71	46	###	###	###
Charlottesvill	8	###	###	###	###	###	###	###	0	###	###	###	###	###	###	###	###	###	###	###	###	###	65	###	###	65	111	###	###	###	###	116
Chattanooga	9	118	###	###	###	###	###	###	###	0	###	###	###	###	###	###	###	###	###	###	###	109	###	###	###	###	###	###	###	###	###	###
Columbia	10	###	###	###	74	###	###	91	###	###	0	###	###	###	###	80	104	###	###	###	###	###	###	###	###	###	###	###	###	###	###	###
Cornelia	11	85	###	50	###	###	###	###	###	###	###	0	###	###	###	###	76	###	###	###	###	215	###	###	###	###	###	###	###	###	###	###
Durham	12	###	###	###	###	###	###	###	###	###	###	###	0	109	###	###	###	57	###	###	###	###	121	###	30	145	130	###	###	###	###	###
Emporia	13	###	###	###	###	###	###	###	###	###	###	###	109	0	###	###	###	###	###	###	###	###	###	79	###	64	###	###	###	###	79	###
Fayetteville	14	###	###	###	###	30	###	###	###	###	###	###	###	###	0	88	###	91	###	###	###	###	###	###	###	###	###	60	###	###	###	###
Florence	15	###	###	###	147	###	###	114	###	###	80	###	###	###	88	0	###	###	###	170	###	###	###	###	###	###	###	66	###	122	###	###
Greenville	16	###	64	93	###	###	###	100	###	###	104	76	###	###	###	###	0	###	###	###	###	###	###	###	###	###	###	###	###	###	###	###
Greensboro	17	###	###	###	###	###	144	87	###	###	###	###	57	###	91	###	###	0	###	###	###	###	###	###	81	###	99	81	72	###	###	###
Greenville2	18	###	###	###	###	###	###	###	###	###	###	###	###	###	###	###	###	###	0	73	###	###	###	119	###	###	###	###	###	###	36	###
Jacksonville	19	###	###	###	###	97	###	###	###	###	###	###	###	###	###	170	###	###	73	0	###	###	###	###	###	###	###	###	###	58	86	###
Kingsport	20	###	82	###	###	###	150	###	###	###	###	###	###	###	###	###	###	###	###	###	0	96	###	###	###	###	###	###	146	###	###	###
Knoxville	21	###	116	###	###	###	###	###	###	109	###	215	###	###	###	###	###	###	###	###	96	0	###	###	###	###	###	###	###	###	###	###
Lynchburg	22	###	###	###	###	###	###	###	65	###	###	###	121	###	###	###	###	###	###	###	###	###	0	###	###	113	54	###	###	###	###	###
Norfolk	23	###	###	###	###	###	###	###	###	###	###	###	###	79	###	###	###	###	119	###	###	###	###	0	###	97	###	###	###	###	###	###
Raleigh	24	###	###	###	###	31	###	###	###	###	###	###	30	###	###	###	###	81	###	###	###	###	###	###	0	###	###	###	###	###	50	###
Richmond	25	###	###	###	###	###	###	###	65	###	###	###	145	64	###	###	###	###	###	###	###	###	113	97	###	0	###	###	###	###	###	111
Roanoke	26	###	###	###	###	###	37	###	111	###	###	###	130	###	###	###	###	99	###	###	###	###	54	###	###	###	0	###	###	###	###	###
Rockingham	27	###	###	###	###	###	###	71	###	###	###	###	###	###	60	66	###	84	###	###	###	###	###	###	###	###	###	0	###	###	###	###
Statesville	28	###	107	###	###	###	125	46	###	###	###	###	###	###	###	###	###	72	###	###	146	###	###	###	###	###	###	###	0	###	###	###
Wilmington	29	###	###	###	###	###	###	###	###	###	###	###	###	###	###	122	###	###	###	58	###	###	###	###	###	###	###	###	###	0	###	###
Wilson	30	###	###	###	###	41	###	###	###	###	###	###	###	79	###	###	###	###	36	86	###	###	###	###	50	###	###	###	###	###	0	###
Washington	31	###	###	###	###	###	###	###	116	###	###	###	###	###	###	###	###	###	###	###	###	###	###	###	###	111	###	###	###	###	###	0

Figure 10. Matrix of distances between cities.

5.2 Implementation and The Pseudocode:

For the implementation I use Java as the programming language. I designed a GUI interface that receive all the general parameters-constants and the paths to for files with information required explained as follow:

File 1: Lists of Cities. A .CSV file containing the name of the cities to include on the Graph

Line1: Atlanta

Line2: Augusta

…

File 2: Symmetric Matrix of distances in miles between cities. (Set distances between nodes without direct connections to a considerably big number).

Line1: 0,23,30,1000000…

Line2: 23,0,40,43…

…

File 3. Symmetric Matrix of speeds in miles per hour between cities. (Set speeds between nodes without direct connections to “one”).

Line1: 0,45,55,1…

Line2: 45,0,45,55…

…

File 4. Contains the information about the clients. Each line contains three fields, the first one is the City where is located the client, second is the weight in lbs, and third is the volume in in^3.

Line1: Augusta,40,3000

Line2: Asheville, 50,6000

…

The parameters that I use to calculate are:

- K1 = 0.000003
- K2 = 0.1
- Fixed Cost of operation (FC) = 500
- $gas/mile = 0.17
- Salary per hour = 20

Two main programs need to be executed as explained on section 4. The first one is the Dijkstra’s algorithm to calculate all pairs shortest paths and then transform the original graph with all the cities-nodes to a full connected graph with just the client-cities. Given that this algorithm is already designed and even more already implemented in several languages, I used the implementation on Java language by Tushar Roy (see reference [4]). Next I shows the Pseudocode taken from [2].

```
function Dijkstra(Graph, source):

     create vertex set Q

     for each vertex v in Graph:             // Initialization
         dist[v] ← INFINITY                  // Unknown distance from source to v
         prev[v] ← UNDEFINED                 // Previous node in optimal path from source
         add v to Q                          // All nodes initially in Q (unvisited nodes)

     dist[source] ← 0                        // Distance from source to source

     while Q is not empty:
         u ← vertex in Q with min dist[u]    // Source node will be selected first
         remove u from Q

         for each neighbor v of u:           // where v is still in Q.
             alt ← dist[u] + length(u, v)
             if alt < dist[v]:               // A shorter path to v has been found
                 dist[v] ← alt
                 prev[v] ← u

     return dist[], prev[]
```

The second main code correspond to solving the TSP. I choose to use the “Held-Karp Algorithm” as explained on sections 4.2 and 4.3. Again, the pseudocode and the implementation are already designed in different languages, thus I have made small modifications to the implementation on Java language by Tushar Roy (see reference [5]). The pseudocode from reference [5] is shows next.

```
function algorithm TSP (G, n)
  for k := 2 to n do
    C({1, k}, k) := d_{1,k}
  end for

  for s := 3 to n do
    for all S ⊆ {1, 2, . . . , n}, |S| = s do
      for all k ∈ S do
        {C(S, k) = min_{m≠1,m≠k,m∈S} [C(S − {k}, m) + d_{m,k} ]}
      end for
    end for
  end for

  opt := min_{k≠1} [C({1, 2, 3, . . . , n}, k) + d_{k,1}]
  return (opt)
end
```

6. Results

The input parameters and the GUI is shows next:

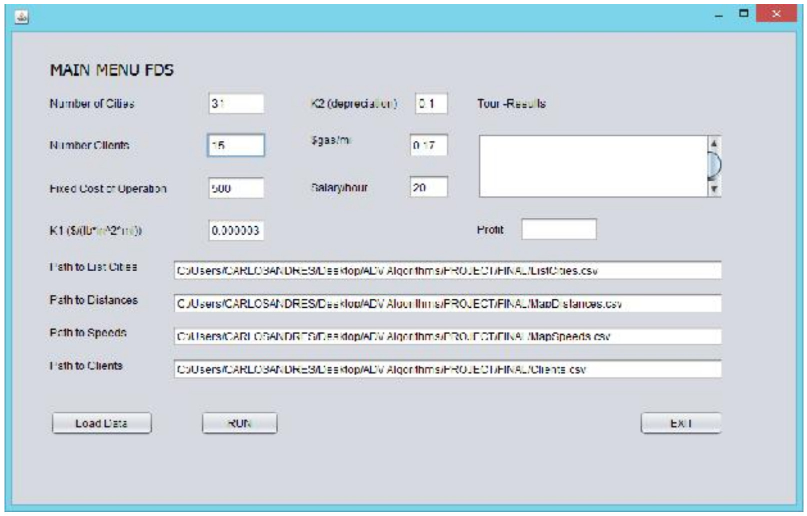

Figure 11. GUI with the input parameters.

Also, it was used to find all pairs shortest weights from our original graph. Results are shown next:

0	141	45	82	242	262	158	339	58	128	53	253	315	221	156	101	213	296	264	190	130	309	364	264	337	275	204	190	236	271	401
141	0	98	146	179	133	93	223	129	100	88	146	208	163	150	40	106	213	240	51	67	193	258	157	230	159	139	61	220	188	290
45	98	0	60	220	219	115	296	113	106	31	210	272	199	144	58	170	274	242	149	155	265	322	221	294	232	161	147	214	249	358
82	146	60	0	160	207	103	284	150	45	91	198	239	139	84	105	158	214	182	197	212	254	282	182	279	220	125	135	154	189	343
242	179	220	160	0	144	109	149	308	125	214	43	79	21	76	166	73	54	61	202	246	112	122	22	119	118	63	118	102	29	183
262	133	219	207	144	0	104	90	208	161	209	101	163	140	169	161	83	178	205	86	146	60	186	122	125	26	136	72	259	153	157
158	93	115	103	109	104	0	181	222	57	105	95	157	88	65	57	55	162	163	116	160	151	207	106	179	117	46	32	135	137	243
339	223	296	284	149	90	181	0	298	238	286	106	77	170	220	238	126	152	181	176	236	37	98	127	37	64	179	162	222	127	67
58	129	113	150	308	208	222	298	0	196	121	275	337	239	234	169	235	342	332	122	52	258	337	236	333	234	258	100	304	317	355
128	100	106	45	125	161	57	238	196	0	108	152	205	105	50	60	112	180	148	151	157	208	248	148	236	174	91	89	120	155	300
53	88	31	91	214	209	105	286	121	108	0	200	262	193	158	48	160	267	256	139	124	255	312	211	284	222	151	137	228	242	348
253	146	210	198	43	101	95	106	275	152	200	0	62	64	119	152	40	77	104	169	213	69	112	21	84	75	93	85	145	52	148
315	208	272	239	79	163	157	77	337	205	262	62	0	100	155	214	102	75	104	231	275	105	50	81	40	137	142	147	145	50	104
221	163	199	139	21	140	88	170	239	105	193	64	100	0	55	145	57	75	82	186	230	133	143	43	140	119	42	102	125	50	204
156	150	144	84	76	169	65	220	234	50	158	119	155	55	0	110	94	130	98	181	217	188	198	98	195	156	41	97	70	105	259
101	40	58	105	166	161	57	238	169	60	48	152	214	145	110	0	112	219	208	91	107	208	264	163	236	174	103	89	180	194	300
213	106	170	158	73	83	55	126	235	112	160	40	102	57	94	112	0	107	134	129	173	96	152	51	124	62	53	45	164	82	188
296	213	274	214	54	178	162	152	342	180	267	77	75	75	130	219	107	0	46	236	280	146	68	56	115	152	117	152	87	25	179
264	240	242	182	61	205	163	181	332	148	256	104	104	82	98	208	134	46	0	263	307	173	114	83	144	179	124	179	41	54	208
190	51	149	197	202	86	116	176	122	151	139	169	231	186	181	91	129	236	263	0	60	146	272	180	211	112	162	84	251	211	243
130	67	155	212	246	146	160	236	52	157	124	213	275	230	217	107	173	280	307	60	0	206	325	224	271	172	206	128	287	255	303
309	193	256	254	112	60	151	37	258	208	256	69	105	133	188	208	96	146	173	146	206	0	126	90	65	34	149	132	214	121	104
364	258	322	282	122	186	207	98	337	248	312	112	50	143	198	264	152	68	114	272	325	125	0	124	61	160	185	197	155	93	125
264	157	221	182	22	122	106	127	236	148	211	21	81	43	98	163	51	56	83	180	224	90	124	0	105	90	85	96	124	31	169
337	230	294	279	119	125	179	37	333	236	284	84	40	140	195	236	124	115	144	211	271	65	61	105	0	99	177	169	185	90	64
275	159	232	220	118	26	117	64	234	174	222	75	137	119	156	174	62	152	179	112	172	34	160	96	99	0	115	98	220	127	131
204	139	161	125	63	136	46	179	258	91	151	93	142	42	41	103	53	117	124	162	206	149	185	85	177	115	0	78	111	92	241
190	61	147	135	118	72	32	162	100	89	137	85	147	102	97	89	45	152	179	84	128	132	197	96	169	98	78	0	167	127	229
236	220	214	154	102	259	135	222	304	120	228	145	145	125	70	180	164	87	41	251	287	214	155	124	185	220	111	167	0	95	249
271	188	249	189	29	153	137	127	317	155	242	52	50	50	105	194	82	25	54	211	255	121	93	31	90	127	92	127	95	0	154
401	290	358	343	183	157	243	67	355	300	348	148	104	204	259	300	188	179	208	243	303	104	125	169	64	131	241	229	249	154	0

Figure 12. Matrix of shortest weights between cities.

Regarding to the implementation of the processes described on sections 4.2 (transforming to a TSP) and 4.3 (the "Held-Karp Algorithm" by Tushar Roy [5]), these were tested on a subset of 15 client nodes.

Next, the output of the program is displayed:

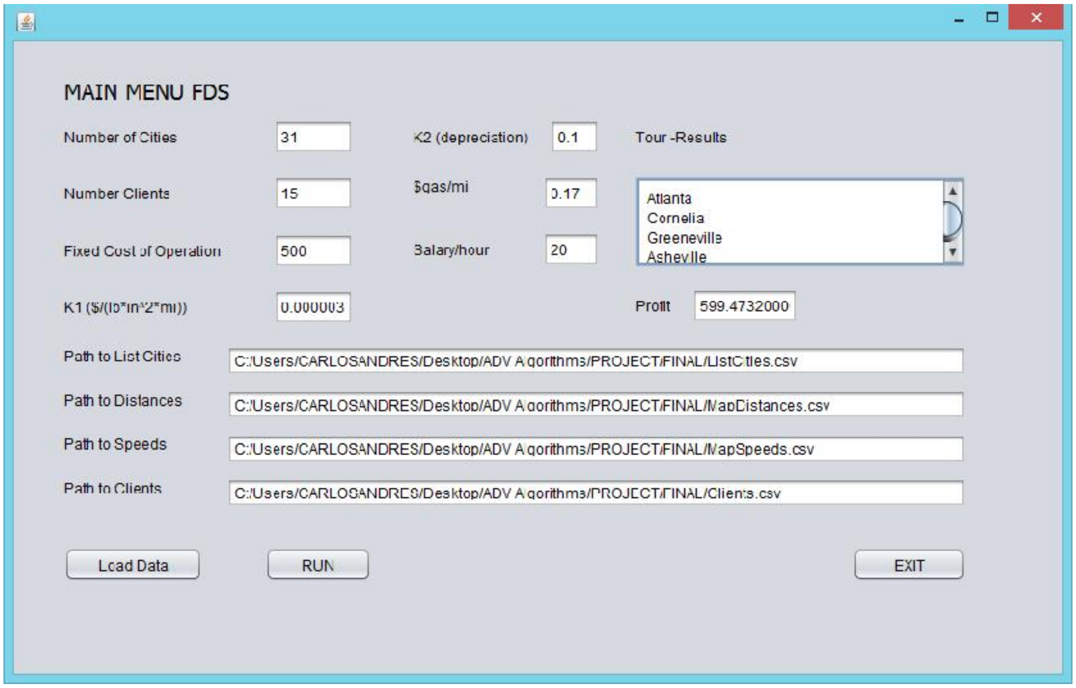

Figure 13. Output results for sample FDS.

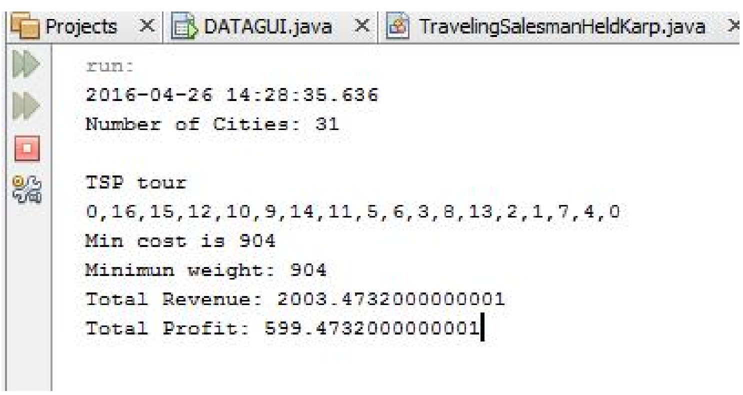

Figure 14. Output from console for sample FDS.

Remember that we need to remove the "dummy node and its connections from the result, which in this case is the node 15. Here the program automatically delete the cost related to the dummy node.

Next a graph of the solution to the FDS problem with 15 client nodes is shown:

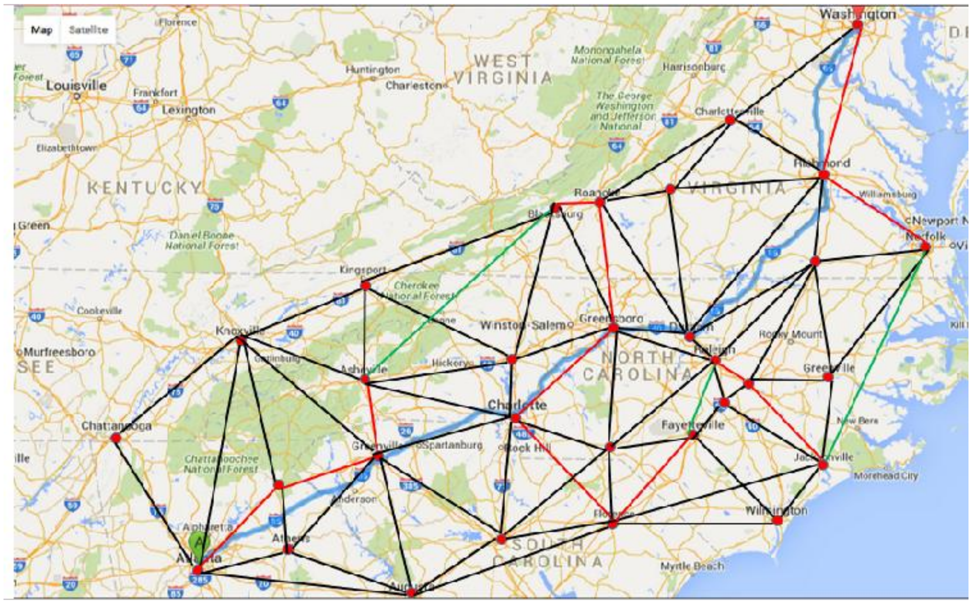

Figure 15. Map of the solution for sample FDS.

Finally, the following figures shows the behavior of the revenue and profit regarding to the number of client nodes visited.

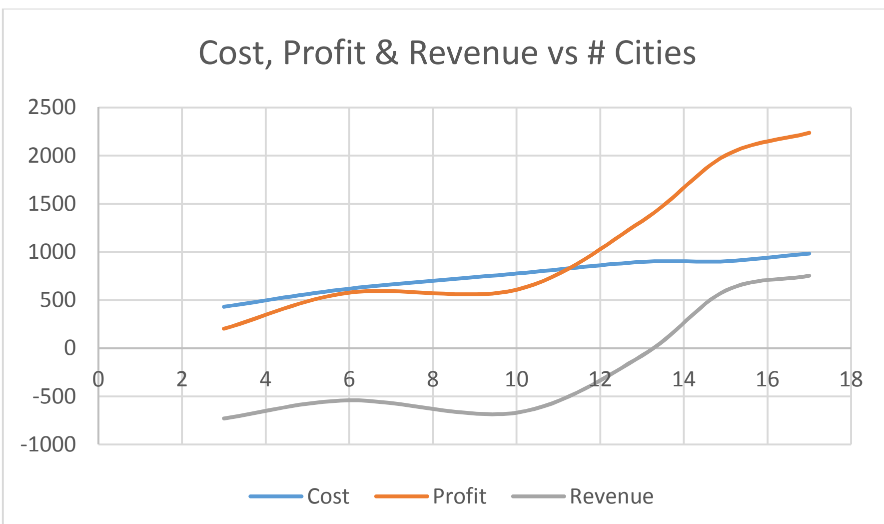

Figure 16. Behavior of Variable Cost, Profit and Revenue regarding to number of client nodes.

7. Testing Correctness with real data:

In order to prove that the algorithm (the second transformation in specific) is working properly, I calculate all possible path combinations for a subgraph of 6 cities and then I got the cost for every combination. Note that city “0” will remain the same as the starting city and city “5” will remain as destination city.

Path	Cities						Cost
1	0	1	2	3	4	5	831
2	0	1	2	4	3	5	831
3	0	1	3	2	4	5	895
4	0	1	3	4	2	5	705
5	0	1	4	3	2	5	595
6	0	1	4	2	3	5	785
7	0	2	1	3	4	5	941
8	0	2	1	4	3	5	831
9	0	2	3	1	4	5	895
10	0	2	3	4	1	5	849
11	0	2	4	3	1	5	959
12	0	2	4	1	3	5	895
13	0	3	2	1	4	5	831
14	0	3	2	4	1	5	849
15	0	3	1	2	4	5	941
16	0	3	1	4	2	5	705
17	0	3	4	1	2	5	641
18	0	3	4	2	1	5	895
19	0	4	2	3	1	5	749
20	0	4	2	1	3	5	731
21	0	4	3	2	1	5	685
22	0	4	3	1	2	5	541
23	0	4	1	3	2	5	495
24	0	4	1	2	3	5	621

In addition, the results using the algorithm proposed is shown next:

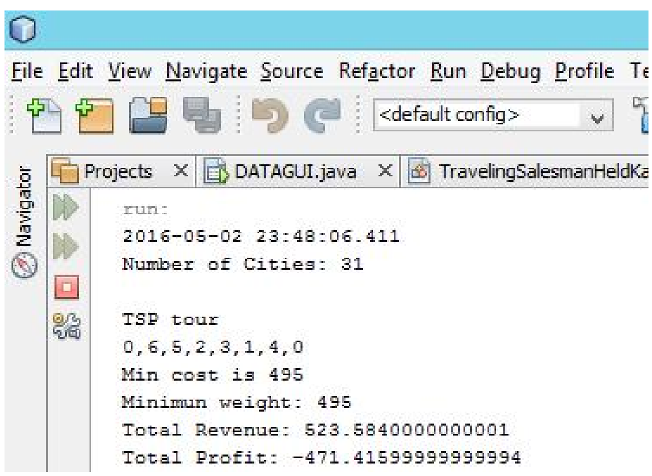

Figure 17. Solution to FDS for subgraph.

As we can see, the program shows that the minimum cost was 495 and the optimal path after removing the dummy node and reading backwards the path is 0,4,1,3,2,5, which is exactly the path corresponding to the combination 23 on the table before that has the minimum cost.

I tried with different cities, and results are always correct, but it happened that more than

one optimal solution could exist (two or more paths with the same minimum cost). My program just shows one of the optimal paths, but the algorithm could be modified to include all possible solutions.

8. Current Milestone and Timeline

Note: Text on red color means removed and text in blue color means that an added milestone.

Activity	Start Date	End Date	Milestone	State
1. Writing project proposal				100%
1.1 Reviewing literatures related with the topic	20-Jan-16	1-Feb-16	Reviewed literature	Done
1.2 Defining the problem and set objectives	2-Feb-16	4-Feb-16	objectives and problem	Done
1.3 Analyze the problem and proposed methods	6-Feb-16	10-Feb-16	proposed methods	Done
1.4 Set the simulation setting, environment and data source	11-Feb-16	14-Feb-16	proposed simulation	Done
1.5 Set milestones and timeline	1-Feb-16	2-Feb-16	milestone and timeline	Done
2. Designing the Algorithm				100%
2.1 Define the methods of the algorithm	16-Feb-16	2/30/2016	Algorithm	Done
2.2 Design the simulation settings and environment	19-Feb-16	20-Feb-16	simulation	Removed
2.3 Choosing dependencies and coding standards	Feb 21 2016	21-Feb-16	standards and dependencies	Removed
2.2 Design the logical/mathematical prove	1-Mar-16	15-Mar-16	Math prove	Done
3. Input Data				100%

3.1 Getting the distances between cities	10-Mar-16	14-Mar-16	Generate Graphs	Done
3.2 Generating data for client nodes as weight, volume, localization, etc	1-Apr-16	5-Apr-16	Client Data	Done
3.3 Create the input JSON data	2/23/2016 - 3/5/2016	2/26/2016 - 3/10/2016	JSON data	Removed
4. writing the code				100%
4.1 Write algorithm classes with the necessary functions	2/28/2016 - 3/15/2016	3/8/2016 - 4/5/2016	classes created	Done
3.3 Write the main runner	9-Mar-16	13-Mar-16	Simulation output data	Removed
5. Testing the code				Done
5.1 Unit testing	3/15/2016 - 4/5/2016	4/10/2016 - 4/15/2016	unit test result	Done
5.2 Integration testing	4/11/2016 - 4/16/2016	4/26/2016 - 4/20/2016	integration test result	Done
6. Writing Document				100%
6.1 Writing algorithm document and submission	3/24/2016 - 2/1/2016	4/26/2016 - 5/10/2016	Final document	Done

9. Performance

As we expected, the behavior is exponential and results for simulations with 3,6,10 and 15 client nodes are shown next:

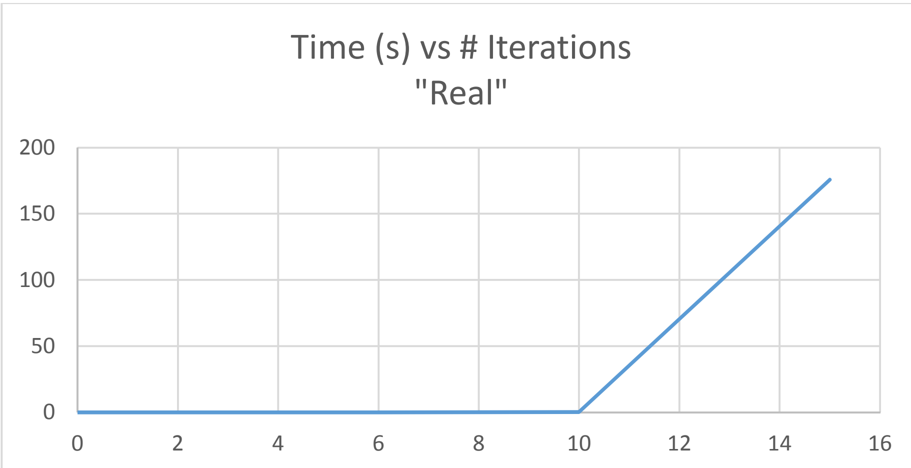

Figure 18. Behavior of Execution time (s) vs Number of client nodes.

In addition, the next figure shows a comparison between the real simulation, the Worst Case $O(2^N * N^2)$, and the Brute force behavior (factorial). Note: Iterations were normalized to CPU cycles time.

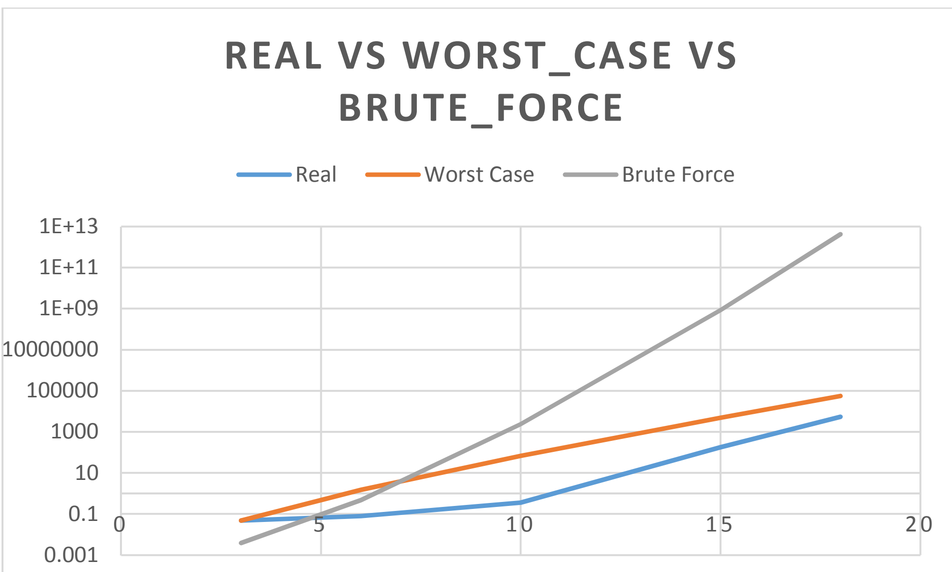

Figure 19. Performance in time (s) for Real Execution, Worst-case and Brute Force

vs Number of client nodes.

10. Conclusions and Future Work

- Future work can be made to complete printing the path for shortest distances (in the first section).
- Other algorithms for solve the TSP with smaller time complexity can be implemented to reduce the time for solving the FDS problem.
- The procedure to solve the FDS shows good results and we could shows that it works as expected.
- Reducing the problems to another already known problem can be a good election for reducing time for finding/building the correct algorithm.
- For more than 20 nodes it is necessary to use other faster algorithms to solve the TSP.
- The optimal path could be not unique (more than one optimal path with the same cost), so as a future work, the code could be change in order to shows every possible optimal paths.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. Introduction to Algorithms. Third edition. The MIT Press. Cambridge, Massachussets. Englan. ISBN 978-0-262-03384-8. 2009.

[2] Dijkstra's algorithm, Wikipedia. Web: https://en.wikipedia.org/wiki/Dijkstra's_algorithm, retr. March 7, 2016.

[3] Time complexity of Bellman-Held-Karp algorithm for TSP, by Suresh Venkat. StackExchange Inc. Web: http://cstheory.stackexchange.com/questions/3666/time-complexity-of-bellman-held-karp-algorithm-for-tsp-take-2, retr. March 10, 2016

[4] Tushar Roy. Web: https://github.com/mission-peace/interview/blob/master/src/com/interview/graph/DijkstraShortestPath.java, retr. March 15, 2016

[5] Tushar Roy. Web: https://github.com/mission-peace/interview/blob/master/src/com/interview/graph/TravelingSalesmanHeldKarp.java, retr. March 17, 2016

[6] Held-Karp algorithm, Wikipedia. Web: https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm, retr. March 10, 2016