TRAVELLING SALESPERSON PROBLEM

Group 12_9

RUI CRUZ UP202208011

MIGUEL GUERRINHA UP20225038

TIAGO TEIXEIRA UP202208511

PROBLEM PRESENTED

- For this project, we were given a selection of 18 datasets of different sizes to analyze the Travelling Salesperson Problem, in the context of urban deliveries and ocean shipping, and find ways to solve it with heuristics designed by us.
- We were then challenged to implement a basic exhaustive algorithm for the routing problem in small graphs, using TSP abstraction, and afterwards, to develop solutions for the TSP in more complex examples.

INDEX:

1. STRUCTURE
2. DATASET PARSING
3. HEURISTICS IN DEPTH
4. INTERFACE
5. CONCLUSIONS

1.1 STRUCTURE – MAIN FILES

main.cpp	• Responsible for initializing the program
Coordinate	• Coordinate representation for a node
Haversine	• Haversine distance calculator for two nodes
Interface	• System runner and menu displays
TSPManager	• Heuristics Functions and Data Parsing

1.2 STRUCTURE – GRAPH

(In relation to Graph.cpp/.h, Edge.cpp/.h and Vertex.cpp/.h)

- To represent the graphs used in the algorithms, a vector of vertices (defined in Graph.h) was used. Vertices are connected by edges or in particular cases, “realistically” by distances in between nodes (haversine calculation is used).
- To achieve this last point, each vertex was designed to hold information about its coordinates and distance related in order to compute its proximity to some other vertex.

2.1 DATASET PARSING

To parse each one of the datasets, the following three functions, with Time Complexity of **O(n)**, take as a parameter a file that can be read and have its content placed in a graph:

- void load_graph(const string &file) :: adds vertices and edges to the graph along with relative distances;
- void load_coordinates_medium(const string &file, int num_nodes) :: using a previously populated graph, latitude and longitude are inserted in the respective nodes from the graph. For this function, only a limited number of nodes is read (num_nodes);
- void load_coordinates_real(const string &file) :: like the function above, coordinates are asserted to the respective vertices, yet this time, every line from the .csv is loaded.

2.2 DATASET PARSING

- For **Toy Graph Datasets**, a simple call to the first function (load_graph) is done, taking as a parameter one of the .csv provided (shipping, stadiums or tourism).
- For **Extra Fully Connected Graph Datasets**, both load_graph and load_coordinates_medium are called, to populate the graph with vertices and edges, and afterwards, with info regarding the coordinates.
- For **Real-World Graph Datasets**, same method as above is followed, with load_graph and load_coordinates_real, the graph is populated with positional vertices and edges.

3.1 HEURISTICS IN DEPTH

Backtracking Algorithm

This algorithm is achieved with the following functions:

- void tsp_backtracking() – **O(N! + N)** :: exhaustive approach to test all possible solutions.
- void tspBTUtil(const Graph& graph, vector<Vertex*> &path, unsigned int pos, double cost, double &min_cost, vector<Vertex*> &min_path) – **O(N!)** :: calculates every possible path to find an optimal solution.
- bool is_not_in_path(const Vertex* node, const vector<Vertex*> &path, unsigned int pos) – **O(N)** :: returns true if the node passed as a parameter isn't included in the path vector.

The first function makes use of the second to find the optimal minimal solution for a path and prints out the nodes included. The second function, recursively looks for solutions and asserts the most optimal path in the min_path variable. The last function simply works as an auxiliar for the second.

3.2 HEURISTICS IN DEPTH

Triangular Approximation Heuristic

The Triangular Approximation is achieved with the first function, that calls the other three for auxiliar calculations:

- void tsp_triangular_approx() – **O((E * log(V)) + 2V + E + 2N)** :: Triangular Approximation Algorithm. Path found is printed out, along with its cost.
- void preorderTraversal(Vertex* root, vector<Vector*> &tour) – **O(V + E)** :: This function applies a DFS algorithm to "rebuild" a path.
- double getTourCost(const vector<Vector*> &tour) – **O(N)** :: taking a path as a parameter, the function calculates the cost associated to such path (**Haversine Distance** calculation is used when two vertices are connected by an edge that is not explicitly described in the dataset).
- vector<Vertex*> prim(Graph* g) – **O(E * log(V))** :: Prim's algorithm is used on the graph passed as a parameter (**MutablePriorityQueue.h** is utilized to manage included nodes on a priority queue).

3.3 HEURISTICS IN DEPTH

Other Heuristics - Christofides

- void tsp_christofides_algorithn() – **O(VE + (E * log(V)) + 3V + N^2 + (N^2 + 2E) + NE)** :: Christofides Algorithm function.
- Vector<Edge*> computeGreedyMWPM(vector<vector<double>> matrix, const vector<Vector*> &oddVertices) – **O(N^2 + 2E)** :: This function is responsible for finding a minimum-weight perfect match, using a greedy approach. Edges between odd degree vertices are found using a distance matrix. After sorted, every edge that grants minimum weight matching is stored in a result vector and then returned.

The first function starts by constructing a MST, making use of the prim algorithm shown previously (prim(Graph* g)). Then, it collects vertices that have an odd degree in a vector and builds a distance matrix for these. Afterwards, these last two arguments are sent to the second function to be processed. By resetting the visited status for all vertices, constructing a circuit starting at the first vertex and finally converting it into a path, the total cost is calculated. The path found and its cost are printed out.

3.4 HEURISTICS IN DEPTH

TSP in the Real World

The Nearest Neighbour Heuristic was chosen to reach solutions to the TSP Problem in **any** graph provided (whether it's fully connected or not). For this particular case, an arbitrary starting point is passed as a parameter to the following function:

- void tsp_nearest_neighbour(int idx) – **O(V + N * E + N);**

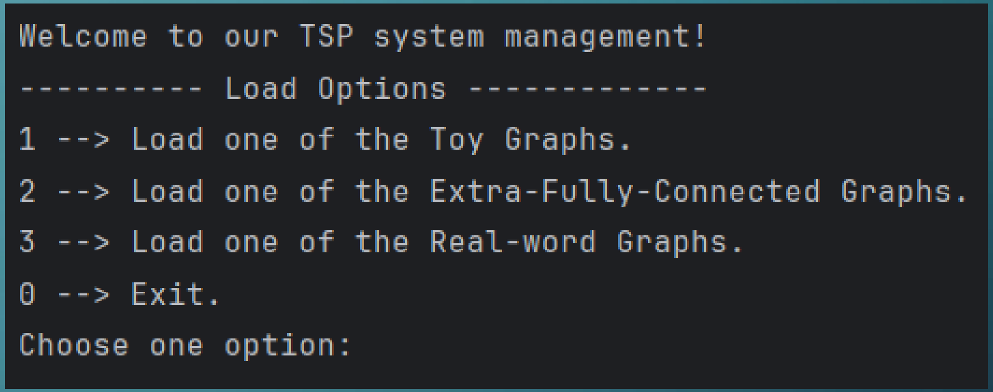
Whenever a solution is unfeasible, which means that there isn't a path that visits all nodes, with the last node being equal to the first, the function prints out the message "No path found" and returns.

If a path is found, then the function prints out the order in which the nodes are visited, along with the total cost associated to such solution.

4.1 INTERFACE

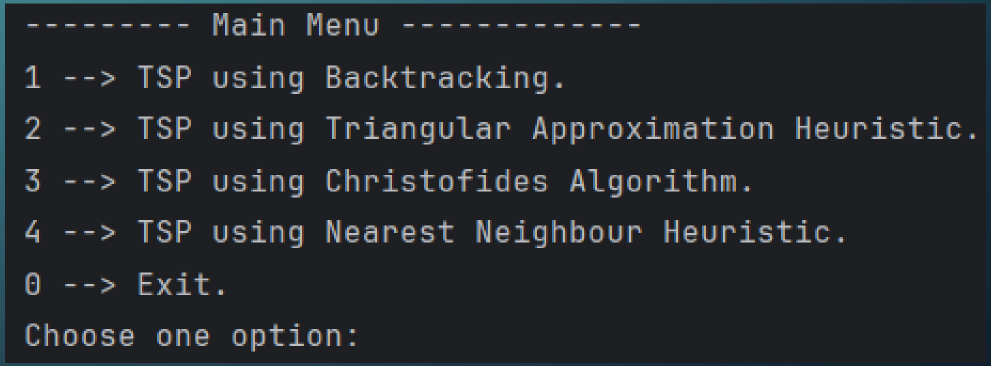
By running the program, the **First Menu** is displayed. The user may choose one of four options, forth being used to close the program and the first three to load a specific dataset from the 18 available.

After the user has loaded one of the datasets provided, the **Main Menu** is displayed. From here, the user can choose four different algorithms to resolve the TSP problem in the context of the dataset formerly selected. Once again, by typing in 0, you can close the program and terminate the system.



```
Welcome to our TSP system management!
---------- Load Options ------------
1 --> Load one of the Toy Graphs.
2 --> Load one of the Extra-Fully-Connected Graphs.
3 --> Load one of the Real-word Graphs.
0 --> Exit.
Choose one option:
```

First Menu - Dataset Loader

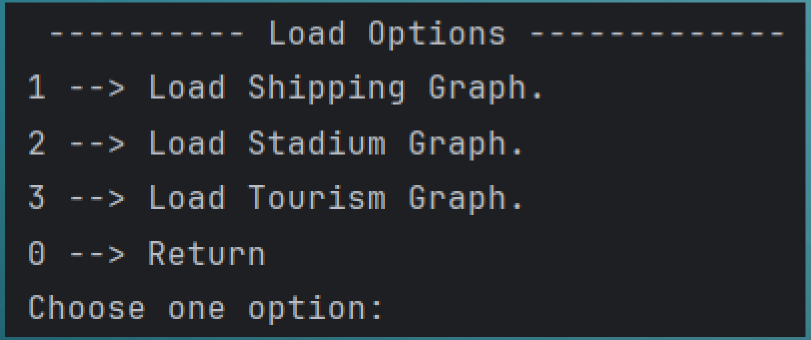


```
--------- Main Menu -------------
1 --> TSP using Backtracking.
2 --> TSP using Triangular Approximation Heuristic.
3 --> TSP using Christofides Algorithm.
4 --> TSP using Nearest Neighbour Heuristic.
0 --> Exit.
Choose one option:
```

Main Menu – Algorithm Selector

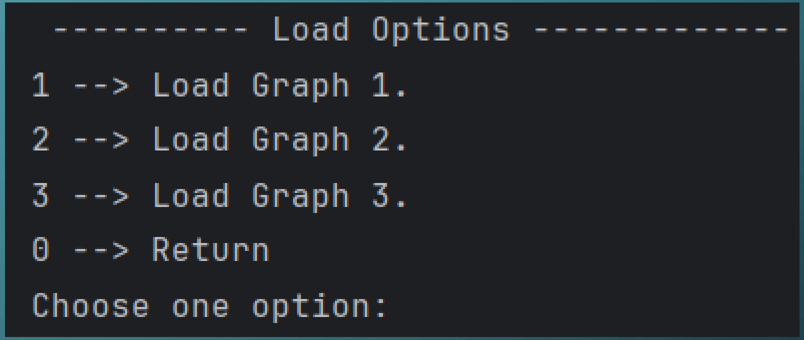
4.2 INTERFACE

Choosing option 1 in the **First Menu**, opens the **Toy Graph Menu**:



```
 ---------- Load Options -------------
1 --> Load Shipping Graph.
2 --> Load Stadium Graph.
3 --> Load Tourism Graph.
0 --> Return
Choose one option:
```

Choosing option 3 in the **First Menu**, opens the **Real-World Graph Menu**:

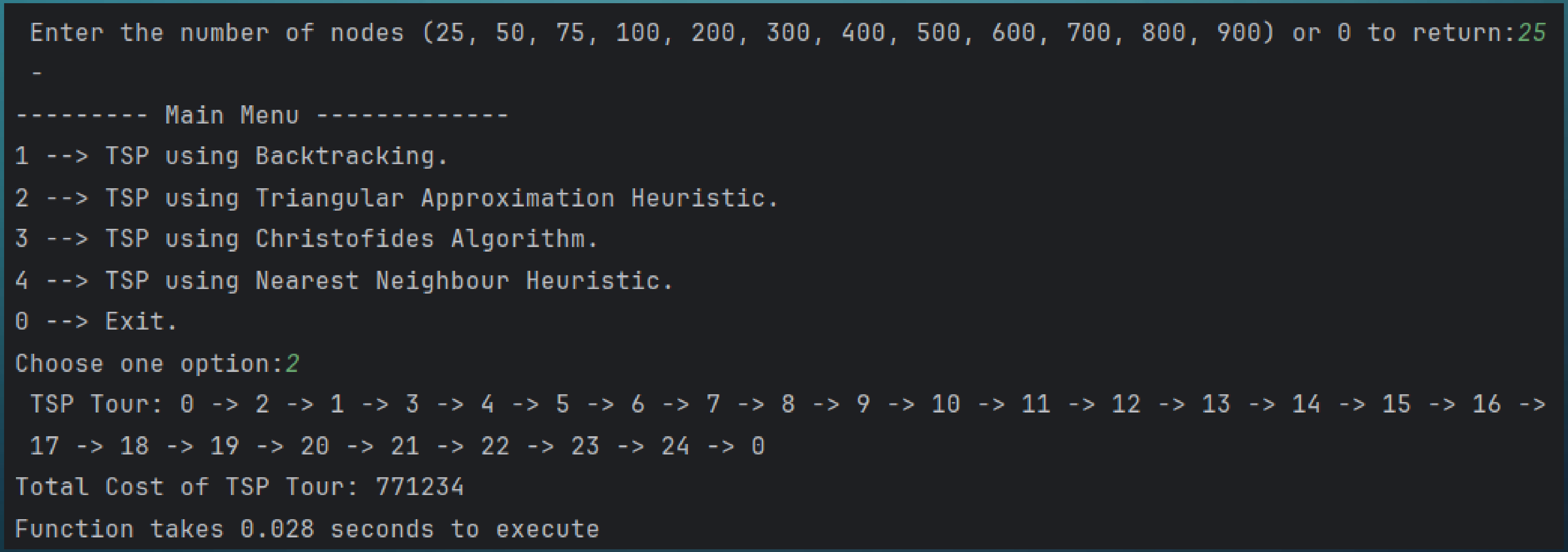


```
 ---------- Load Options -------------
1 --> Load Graph 1.
2 --> Load Graph 2.
3 --> Load Graph 3.
0 --> Return
Choose one option:
```

In both cases, after selecting one of the 3 dataset examples, you will be redirected to the **Main Menu**, where you will be able to choose of the algorithms provided.

4.3 INTERFACE

Selecting option 2 in the **First Menu** allows the user to input a number that will be used to limit the amount of vertices used in the graph. Below is an example of the application of this feature with 25 nodes, using the **Triangular Approximation** afterwards:



```
 Enter the number of nodes (25, 50, 75, 100, 200, 300, 400, 500, 600, 700, 800, 900) or 0 to return:25
 -
--------- Main Menu -------------
1 --> TSP using Backtracking.
2 --> TSP using Triangular Approximation Heuristic.
3 --> TSP using Christofides Algorithm.
4 --> TSP using Nearest Neighbour Heuristic.
0 --> Exit.
Choose one option:2
 TSP Tour: 0 -> 2 -> 1 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15 -> 16 ->
 17 -> 18 -> 19 -> 20 -> 21 -> 22 -> 23 -> 24 -> 0
Total Cost of TSP Tour: 771234
Function takes 0.028 seconds to execute
```

5. CONCLUSIONS

Heuristic	Complexity	Strengths	Weaknesses
Backtracking	O(N! + N)	Guarantees an optimal solution	Exponential time complexity
Triangular Approximation	O((E * log(V)) + 2V + E + 2N)	Fast to provide a reasonably good solution	Approximation quality can vary with datasets
Christofides	O(VE + (E * log(V)) + 3V + N^2 + (N^2 + 2E) + NE)	Closer to optimal, with a good approximation	More complex and higher computation time

Backtracking should be used when the exact optimal solution is needed, and the problem size is small.

Triangular Approximation is good for fast and straightforward solutions. This algorithm is particularly useful for large instances where the exact solution is impractical due to time and resource constraints.

Christofides Algorithm is ideal when the quality of the solution is crucial. Suitable for instances where a better approximation is necessary, and there are sufficient computational resources to handle its complexity.

PARTICIPATION

TASKS WERE DISTRIBUTED BY EACH OF THE MEMBERS OF THE GROUP AND WERE CONCLUDED IN THEIR RIGHT TIME.

OVERALL, GOALS PLANNED OUT WERE ACHIEVED.