ROUTING ALGORITHM FOR OCEAN SHIPPING AND URBAN DELIVERIES

Done By:

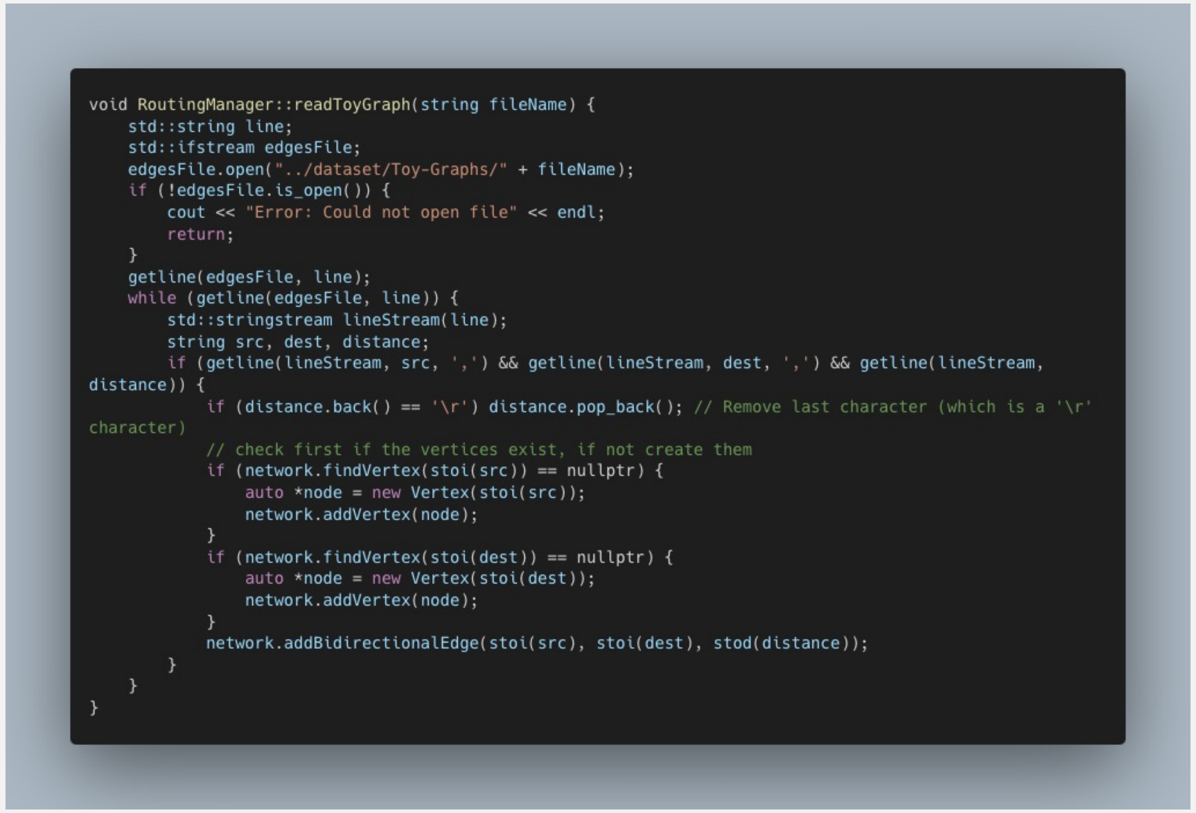
Carlos Oliveira

Yves Bonneau

Sara Azevedo

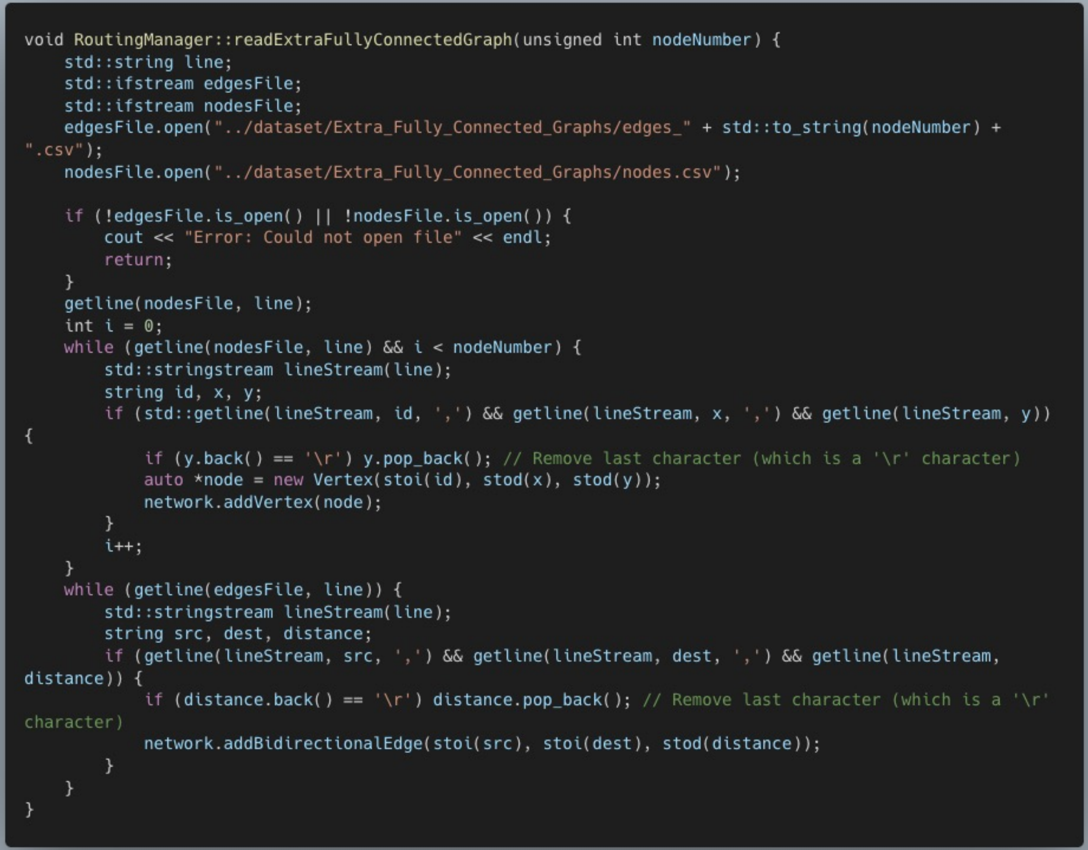
READ THE DATASET

To read the datasets provided in this project, we implemented three separate functions, one for each set of files. In each function, we open the corresponding CSV files and check for successful opening, printing an error message if a file cannot be opened. We skip the headers, then read and convert the node and edge data, creating Vertex objects. Finally, we add bidirectional edges to the network. Below, I will include pictures of the functions for clarity and reference.

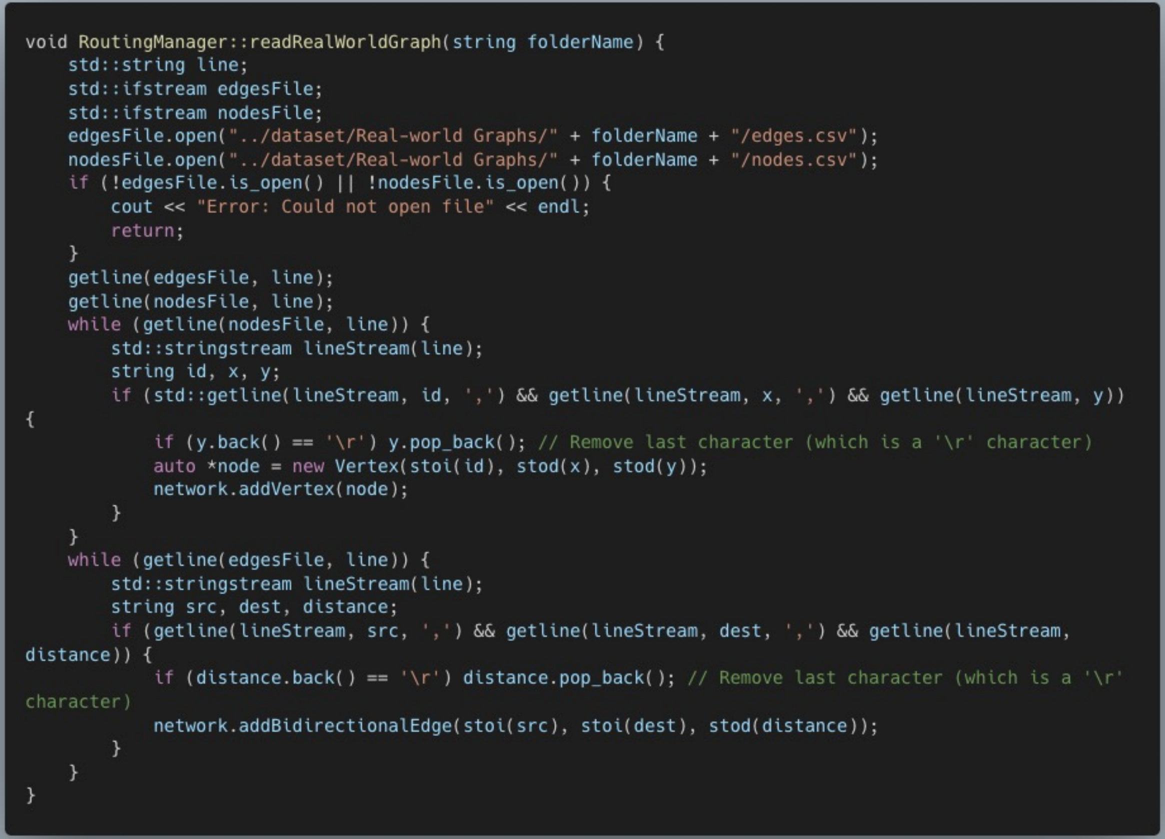


```
void RoutingManager::readToyGraph(string fileName) {
    std::string line;
    std::ifstream edgesFile;
    edgesFile.open("../dataset/Toy-Graphs/" + fileName);
    if (!edgesFile.is_open()) {
        cout << "Error: Could not open file" << endl;
        return;
    }
    getline(edgesFile, line);
    while (getline(edgesFile, line)) {
        std::stringstream lineStream(line);
        string src, dest, distance;
        if (getline(lineStream, src, ',') && getline(lineStream, dest, ',') && getline(lineStream,
distance)) {
            if (distance.back() == '\r') distance.pop_back(); // Remove last character (which is a '\r'
character)
            // check first if the vertices exist, if not create them
            if (network.findVertex(stoi(src)) == nullptr) {
                auto *node = new Vertex(stoi(src));
                network.addVertex(node);
            }
            if (network.findVertex(stoi(dest)) == nullptr) {
                auto *node = new Vertex(stoi(dest));
                network.addVertex(node);
            }
            network.addBidirectionalEdge(stoi(src), stoi(dest), stod(distance));
        }
    }
}
```

READ THE DATASET(CODE)



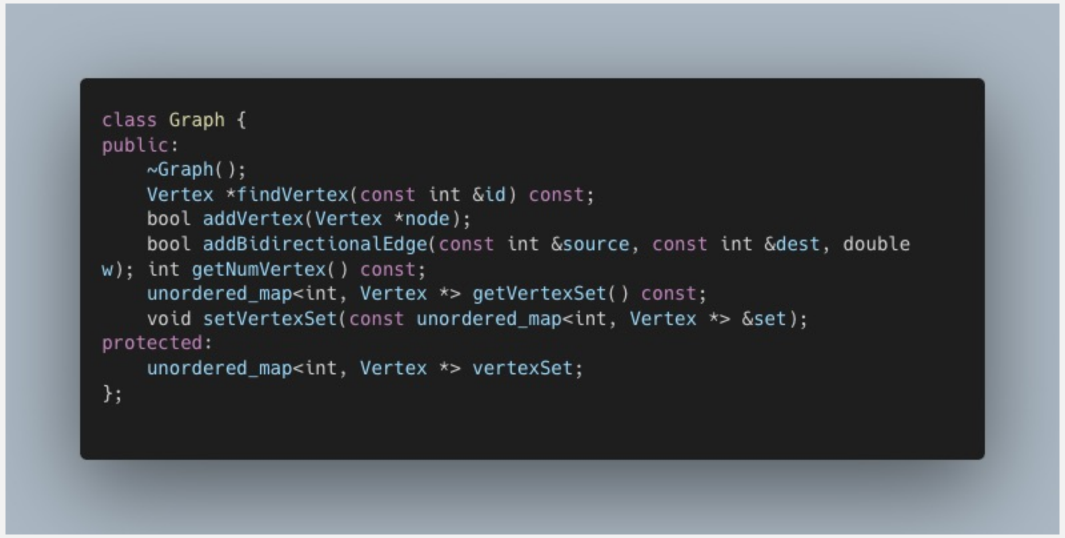
```cpp
void RoutingManager::readExtraFullyConnectedGraph(unsigned int nodeNumber) {
    std::string line;
    std::ifstream edgesFile;
    std::ifstream nodesFile;
    edgesFile.open("../dataset/Extra_Fully_Connected_Graphs/edges_" + std::to_string(nodeNumber) +
".csv");
    nodesFile.open("../dataset/Extra_Fully_Connected_Graphs/nodes.csv");

    if (!edgesFile.is_open() || !nodesFile.is_open()) {
        cout << "Error: Could not open file" << endl;
        return;
    }
    getline(nodesFile, line);
    int i = 0;
    while (getline(nodesFile, line) && i < nodeNumber) {
        std::stringstream lineStream(line);
        string id, x, y;
        if (std::getline(lineStream, id, ',') && getline(lineStream, x, ',') && getline(lineStream, y))
{
            if (y.back() == '\r') y.pop_back(); // Remove last character (which is a '\r' character)
            auto *node = new Vertex(stoi(id), stod(x), stod(y));
            network.addVertex(node);
        }
        i++;
    }
    while (getline(edgesFile, line)) {
        std::stringstream lineStream(line);
        string src, dest, distance;
        if (getline(lineStream, src, ',') && getline(lineStream, dest, ',') && getline(lineStream,
distance)) {
            if (distance.back() == '\r') distance.pop_back(); // Remove last character (which is a '\r'
character)
            network.addBidirectionalEdge(stoi(src), stoi(dest), stod(distance));
        }
    }
}
```



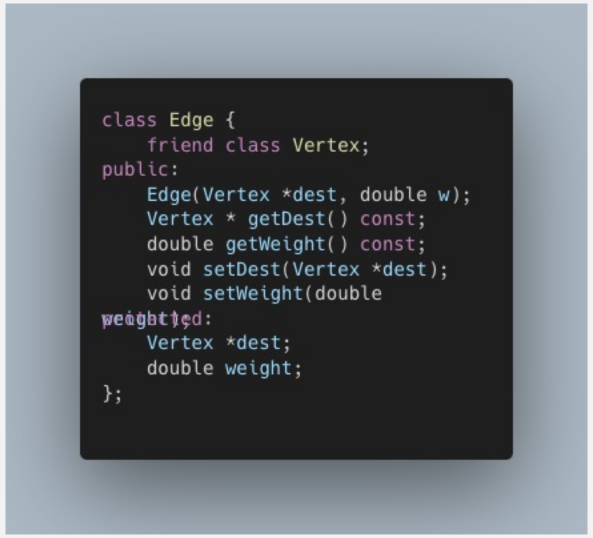
```cpp
void RoutingManager::readRealWorldGraph(string folderName) {
    std::string line;
    std::ifstream edgesFile;
    std::ifstream nodesFile;
    edgesFile.open("../dataset/Real-world Graphs/" + folderName + "/edges.csv");
    nodesFile.open("../dataset/Real-world Graphs/" + folderName + "/nodes.csv");
    if (!edgesFile.is_open() || !nodesFile.is_open()) {
        cout << "Error: Could not open file" << endl;
        return;
    }
    getline(edgesFile, line);
    getline(nodesFile, line);
    while (getline(nodesFile, line)) {
        std::stringstream lineStream(line);
        string id, x, y;
        if (std::getline(lineStream, id, ',') && getline(lineStream, x, ',') && getline(lineStream, y))
{
            if (y.back() == '\r') y.pop_back(); // Remove last character (which is a '\r' character)
            auto *node = new Vertex(stoi(id), stod(x), stod(y));
            network.addVertex(node);
        }
    }
    while (getline(edgesFile, line)) {
        std::stringstream lineStream(line);
        string src, dest, distance;
        if (getline(lineStream, src, ',') && getline(lineStream, dest, ',') && getline(lineStream,
distance)) {
            if (distance.back() == '\r') distance.pop_back(); // Remove last character (which is a '\r'
character)
            network.addBidirectionalEdge(stoi(src), stoi(dest), stod(distance));
        }
    }
}
```

GRAPHS USED FOR THE DATASET

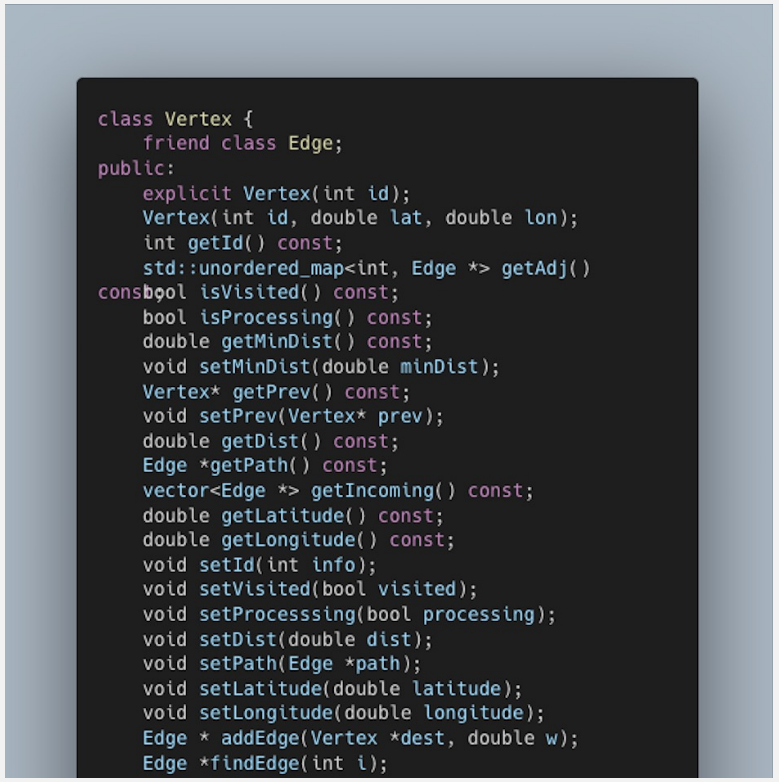
To represent the dataset in this project, we utilize a custom graph structure consisting of three classes: Graph, Vertex, and Edge. The Graph class manages a collection of Vertex objects and their connections. The Vertex class represents individual nodes in the graph, each with an identifier, coordinates, and adjacency list of edges. The Edge class defines the connections between vertices, specifying the destination vertex and the weight of the edge. These classes work together to efficiently store and manipulate graph data.



```
class Graph {
public:
    ~Graph();
    Vertex *findVertex(const int &id) const;
    bool addVertex(Vertex *node);
    bool addBidirectionalEdge(const int &source, const int &dest, double
w); int getNumVertex() const;
    unordered_map<int, Vertex *> getVertexSet() const;
    void setVertexSet(const unordered_map<int, Vertex *> &set);
protected:
    unordered_map<int, Vertex *> vertexSet;
};
```



```
class Edge {
    friend class Vertex;
public:
    Edge(Vertex *dest, double w);
    Vertex * getDest() const;
    double getWeight() const;
    void setDest(Vertex *dest);
    void setWeight(double
weight); protected:
    Vertex *dest;
    double weight;
};
```



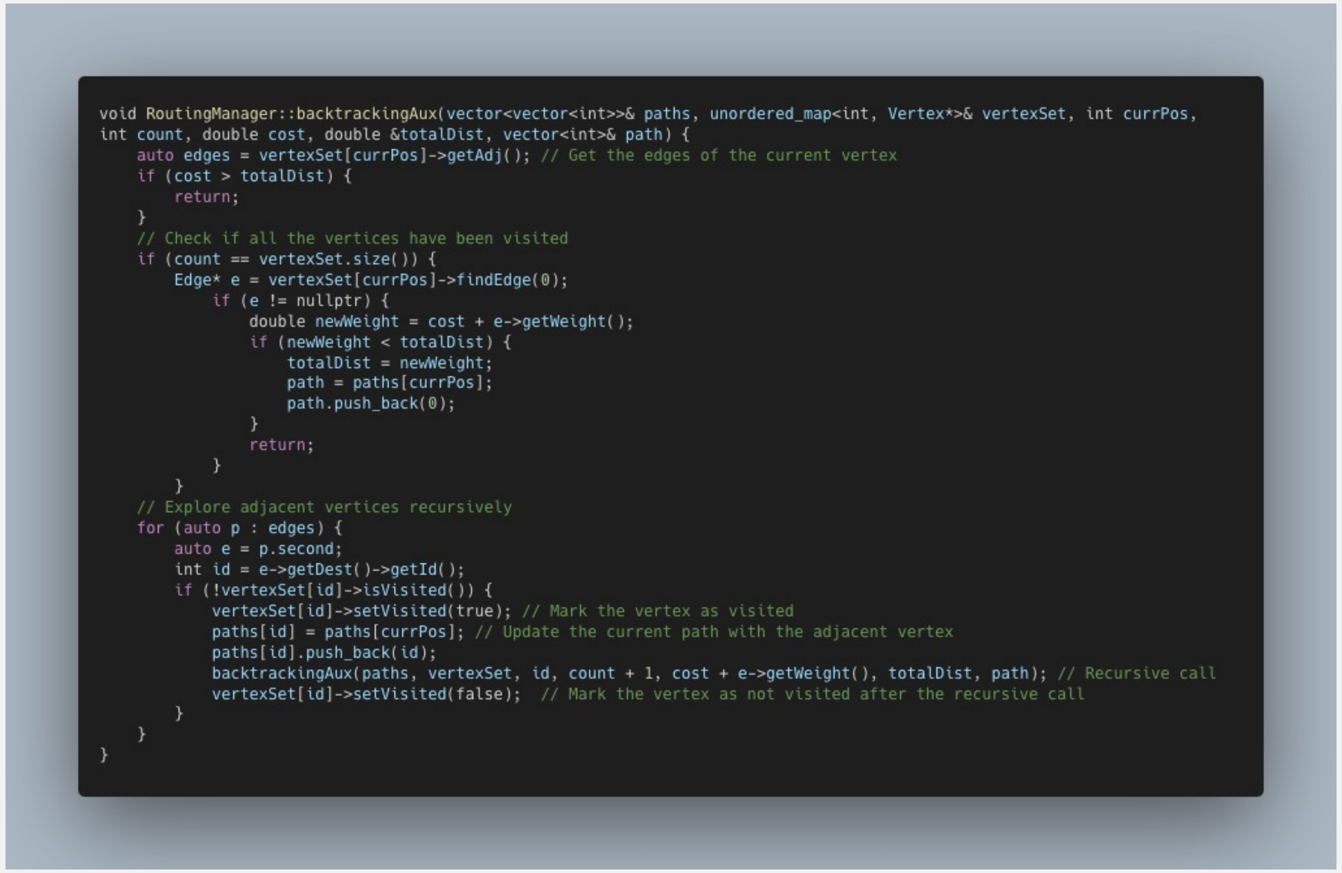
```
class Vertex {
    friend class Edge;
public:
    explicit Vertex(int id);
    Vertex(int id, double lat, double lon);
    int getId() const;
    std::unordered_map<int, Edge *> getAdj()
const; bool isVisited() const;
    bool isProcessing() const;
    double getMinDist() const;
    void setMinDist(double minDist);
    Vertex* getPrev() const;
    void setPrev(Vertex* prev);
    double getDist() const;
    Edge *getPath() const;
    vector<Edge *> getIncoming() const;
    double getLatitude() const;
    double getLongitude() const;
    void setId(int info);
    void setVisited(bool visited);
    void setProcesssing(bool processing);
    void setDist(double dist);
    void setPath(Edge *path);
    void setLatitude(double latitude);
    void setLongitude(double longitude);
    Edge * addEdge(Vertex *dest, double w);
    Edge *findEdge(int i);
```

BACKTRACKING ALGORITHM

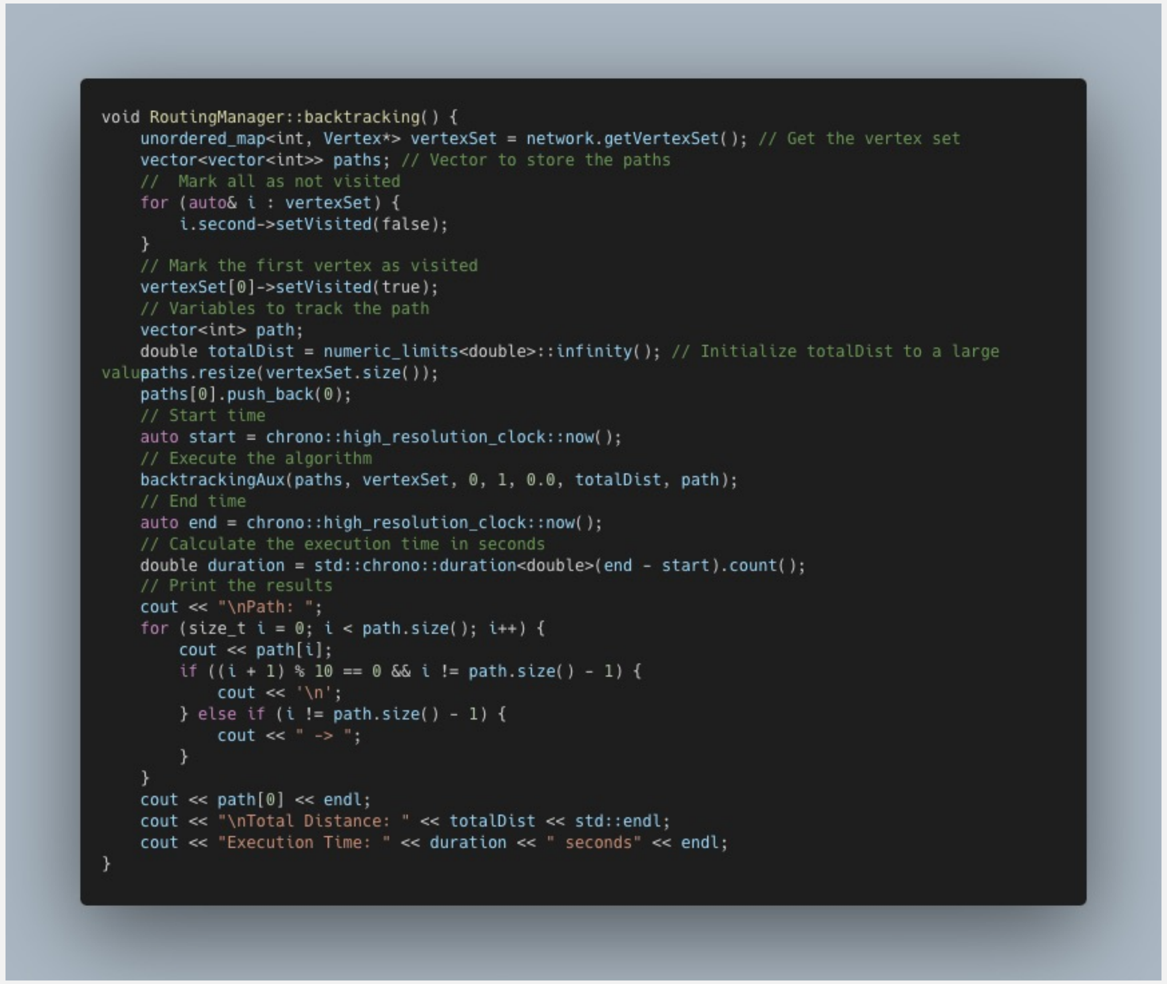
To solve the Traveling Salesman problem (TSP) we implemented the backtracking algorithm to discover the optimal path in a weighted graph.

- We did an auxiliary function called "backtrackingAux" which executes a recursive depth-first search with backtracking, visiting the graph and marking each vertex as visited.
- It verifies if all vertices have been visited, updating the minimum total weight and optimal path if a better solution is identified.
- Unvisited adjacent vertices are explored by marking them as visited, updating the current path and recursively continuing the search.
- The main backtracking function, called "backtracking" initializes essential data structures, marks the starting vertex as visited, and then initiates the recursive search.
- In the end, the function prints the path with the minimum weight, the total weight, and the execution time.

BACKTRACKING ALGORITHM (CODE)



```cpp
void RoutingManager::backtrackingAux(vector<vector<int>>& paths, unordered_map<int, Vertex*>& vertexSet, int currPos,
int count, double cost, double &totalDist, vector<int>& path) {
    auto edges = vertexSet[currPos]->getAdj(); // Get the edges of the current vertex
    if (cost > totalDist) {
        return;
    }
    // Check if all the vertices have been visited
    if (count == vertexSet.size()) {
        Edge* e = vertexSet[currPos]->findEdge(0);
            if (e != nullptr) {
                double newWeight = cost + e->getWeight();
                if (newWeight < totalDist) {
                    totalDist = newWeight;
                    path = paths[currPos];
                    path.push_back(0);
                }
                return;
            }
        }
    // Explore adjacent vertices recursively
    for (auto p : edges) {
        auto e = p.second;
        int id = e->getDest()->getId();
        if (!vertexSet[id]->isVisited()) {
            vertexSet[id]->setVisited(true); // Mark the vertex as visited
            paths[id] = paths[currPos]; // Update the current path with the adjacent vertex
            paths[id].push_back(id);
            backtrackingAux(paths, vertexSet, id, count + 1, cost + e->getWeight(), totalDist, path); // Recursive call
            vertexSet[id]->setVisited(false);  // Mark the vertex as not visited after the recursive call
        }
    }
}
```



```cpp
void RoutingManager::backtracking() {
    unordered_map<int, Vertex*> vertexSet = network.getVertexSet(); // Get the vertex set
    vector<vector<int>> paths; // Vector to store the paths
    //  Mark all as not visited
    for (auto& i : vertexSet) {
        i.second->setVisited(false);
    }
    // Mark the first vertex as visited
    vertexSet[0]->setVisited(true);
    // Variables to track the path
    vector<int> path;
    double totalDist = numeric_limits<double>::infinity(); // Initialize totalDist to a large
valupaths.resize(vertexSet.size());
    paths[0].push_back(0);
    // Start time
    auto start = chrono::high_resolution_clock::now();
    // Execute the algorithm
    backtrackingAux(paths, vertexSet, 0, 1, 0.0, totalDist, path);
    // End time
    auto end = chrono::high_resolution_clock::now();
    // Calculate the execution time in seconds
    double duration = std::chrono::duration<double>(end - start).count();
    // Print the results
    cout << "\nPath: ";
    for (size_t i = 0; i < path.size(); i++) {
        cout << path[i];
        if ((i + 1) % 10 == 0 && i != path.size() - 1) {
            cout << '\n';
        } else if (i != path.size() - 1) {
            cout << " -> ";
        }
    }
    cout << path[0] << endl;
    cout << "\nTotal Distance: " << totalDist << std::endl;
    cout << "Execution Time: " << duration << " seconds" << endl;
}
```

TRIANGULAR APPROXIMATION HEURISTIC

To facilitate geographical calculations essential for solving the Traveling Salesman Problem (TSP), we implemented auxiliary functions:

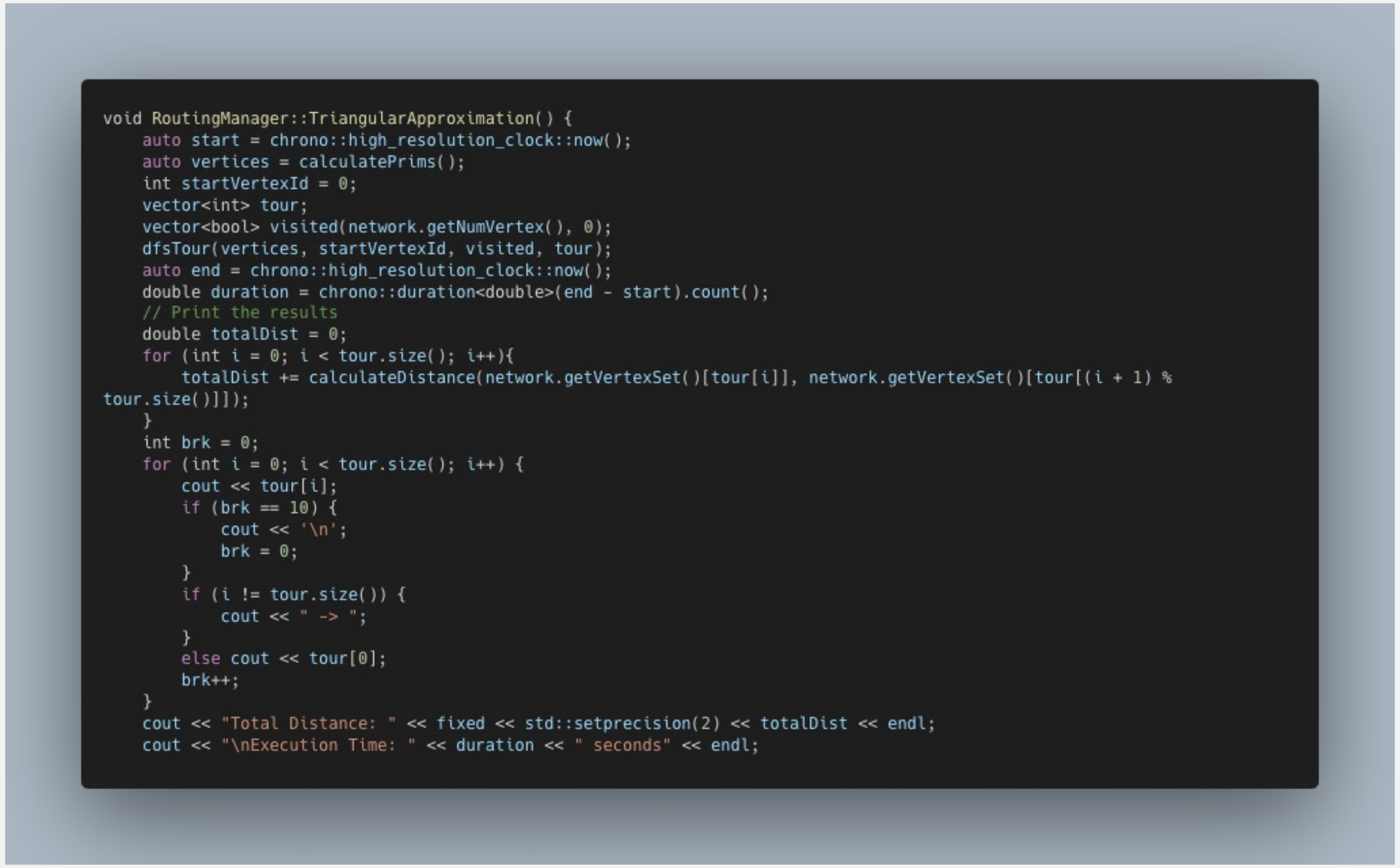
- The “degreesToRadians” function converts degrees to radians, a step for subsequent geographical computations.
- The “haversineDistance” function computes the distance between two points on the Earth's surface, considering their latitude and longitude coordinates.
- The “calculateDistance” function that determines the distance between two vertices. It accesses a direct edge if available or computes the Haversine distance if no direct edge exists, ensuring accurate distance evaluation for the TSP algorithm.

To connect all vertices in the graph while minimizing total edge weight, we employ the calculatePrims function:

- Using Prim's algorithm, “calculatePrims” constructs a minimum spanning tree (MST) for the graph. This guarantees connectivity among all vertices while optimizing total edge weight.
- The “dfsTour” function conducts a depth-first traversal on the MST, generating a tour that visits each vertex exactly once

TRIANGULAR APPROXIMATION HEURISTIC (CODE)

- In the end, we used the “TriangularApproximation” function as the main TSP algorithm. It initiates the generation of a tour based on a heuristic derived from the MST, aiming to approximate the optimal TSP solution as you can see below.



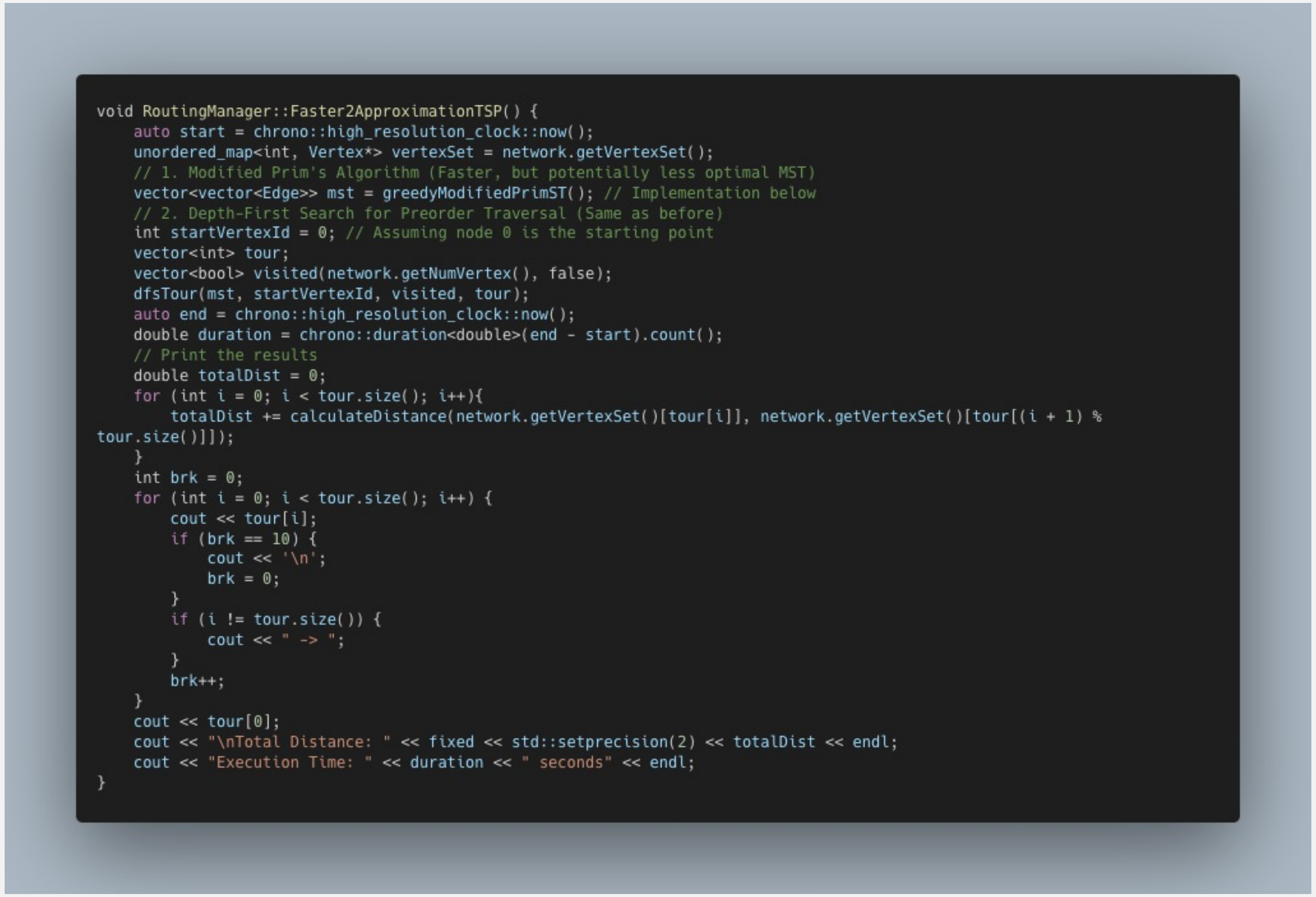
```
void RoutingManager::TriangularApproximation() {
    auto start = chrono::high_resolution_clock::now();
    auto vertices = calculatePrims();
    int startVertexId = 0;
    vector<int> tour;
    vector<bool> visited(network.getNumVertex(), 0);
    dfsTour(vertices, startVertexId, visited, tour);
    auto end = chrono::high_resolution_clock::now();
    double duration = chrono::duration<double>(end - start).count();
    // Print the results
    double totalDist = 0;
    for (int i = 0; i < tour.size(); i++){
        totalDist += calculateDistance(network.getVertexSet()[tour[i]], network.getVertexSet()[tour[(i + 1) %
tour.size()]]);
    }
    int brk = 0;
    for (int i = 0; i < tour.size(); i++) {
        cout << tour[i];
        if (brk == 10) {
            cout << '\n';
            brk = 0;
        }
        if (i != tour.size()) {
            cout << " -> ";
        }
        else cout << tour[0];
        brk++;
    }
    cout << "Total Distance: " << fixed << std::setprecision(2) << totalDist << endl;
    cout << "\nExecution Time: " << duration << " seconds" << endl;
```

OTHER HEURISTICS

For this problem we created a faster but potentially less optimal approach to solve the Traveling Salesman Problem. Here is what we've done:

- The created a "greedyModifiedPrimST" function that initializes vertices for Minimum Spanning Tree (MST) construction.
- Then, using a modified version of Prim's algorithm, the function constructs the MST by iteratively selecting the closest unvisited neighbour for each vertex.
- In the main function, the MST is generated using the "greedyModifiedPrimST" function, prioritizing speed over optimality by selecting the nearest unvisited neighbour for each vertex.
- After MST construction, a Depth-First Search (DFS) traversal is performed on the MST to generate a tour, ensuring each vertex is visited exactly once.
- The function prints the tour along with the total distance travelled and displays the execution time for the algorithm.

OTHER HEURISTICS (CODE)

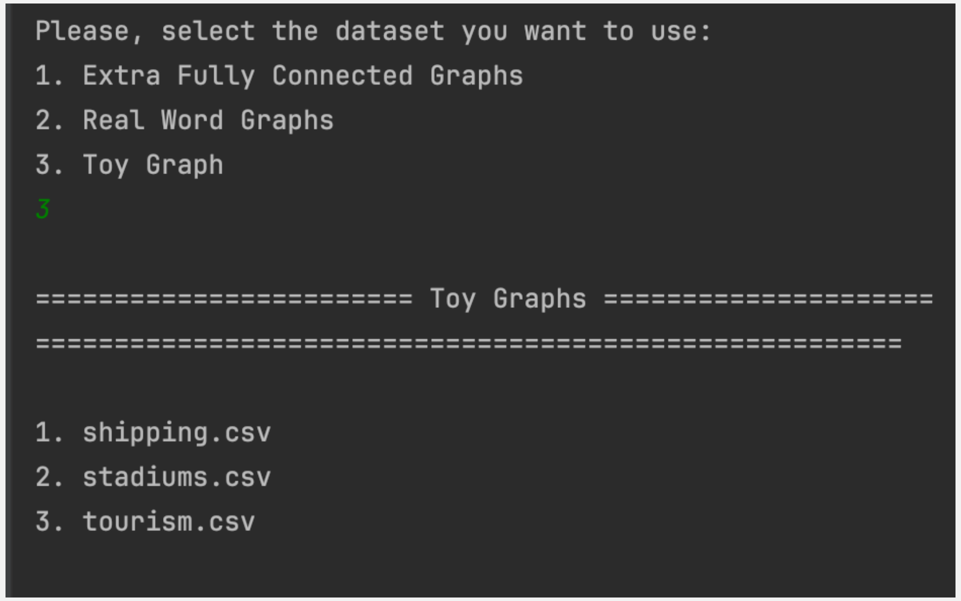


```cpp
void RoutingManager::Faster2ApproximationTSP() {
    auto start = chrono::high_resolution_clock::now();
    unordered_map<int, Vertex*> vertexSet = network.getVertexSet();
    // 1. Modified Prim's Algorithm (Faster, but potentially less optimal MST)
    vector<vector<Edge>> mst = greedyModifiedPrimST(); // Implementation below
    // 2. Depth-First Search for Preorder Traversal (Same as before)
    int startVertexId = 0; // Assuming node 0 is the starting point
    vector<int> tour;
    vector<bool> visited(network.getNumVertex(), false);
    dfsTour(mst, startVertexId, visited, tour);
    auto end = chrono::high_resolution_clock::now();
    double duration = chrono::duration<double>(end - start).count();
    // Print the results
    double totalDist = 0;
    for (int i = 0; i < tour.size(); i++){
        totalDist += calculateDistance(network.getVertexSet()[tour[i]], network.getVertexSet()[tour[(i + 1) %
tour.size()]]);
    }
    int brk = 0;
    for (int i = 0; i < tour.size(); i++) {
        cout << tour[i];
        if (brk == 10) {
            cout << '\n';
            brk = 0;
        }
        if (i != tour.size()) {
            cout << " -> ";
        }
        brk++;
    }
    cout << tour[0];
    cout << "\nTotal Distance: " << fixed << std::setprecision(2) << totalDist << endl;
    cout << "Execution Time: " << duration << " seconds" << endl;
}
```

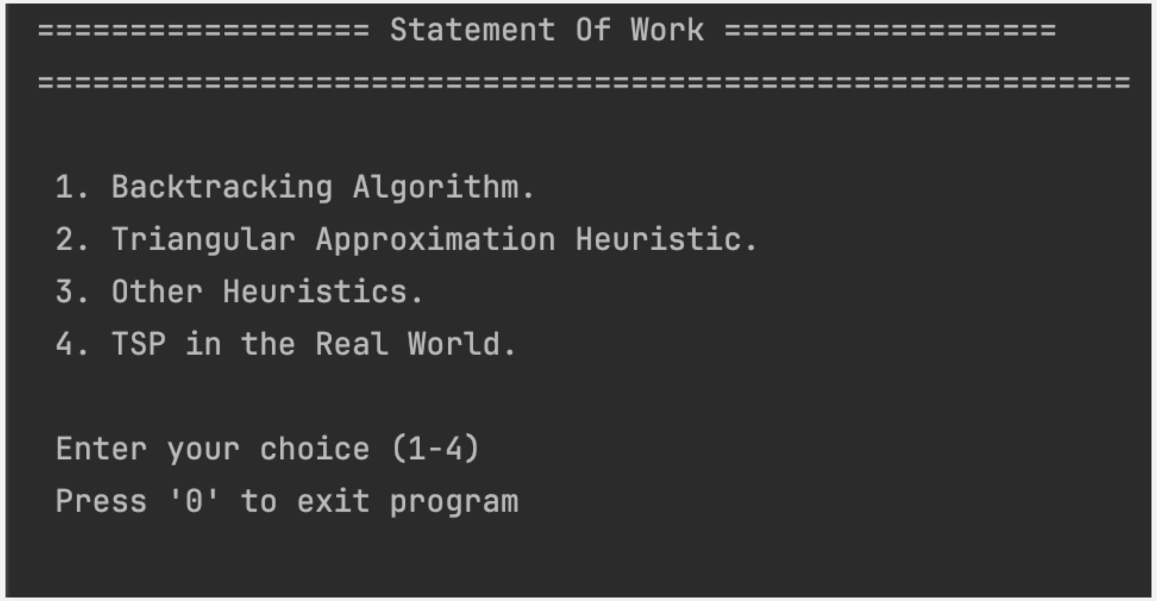
USER INTERFACE

As a user, in the menu you can do the following:

- Graph Menu: Displays a list of available graph types that can be read. After selecting a graph type, the menu then presents a list of available graphs of that type for the user choose from.
- Statement of Work Menu: This menu presents a list of all the algorithms that can be run on the selected graph.

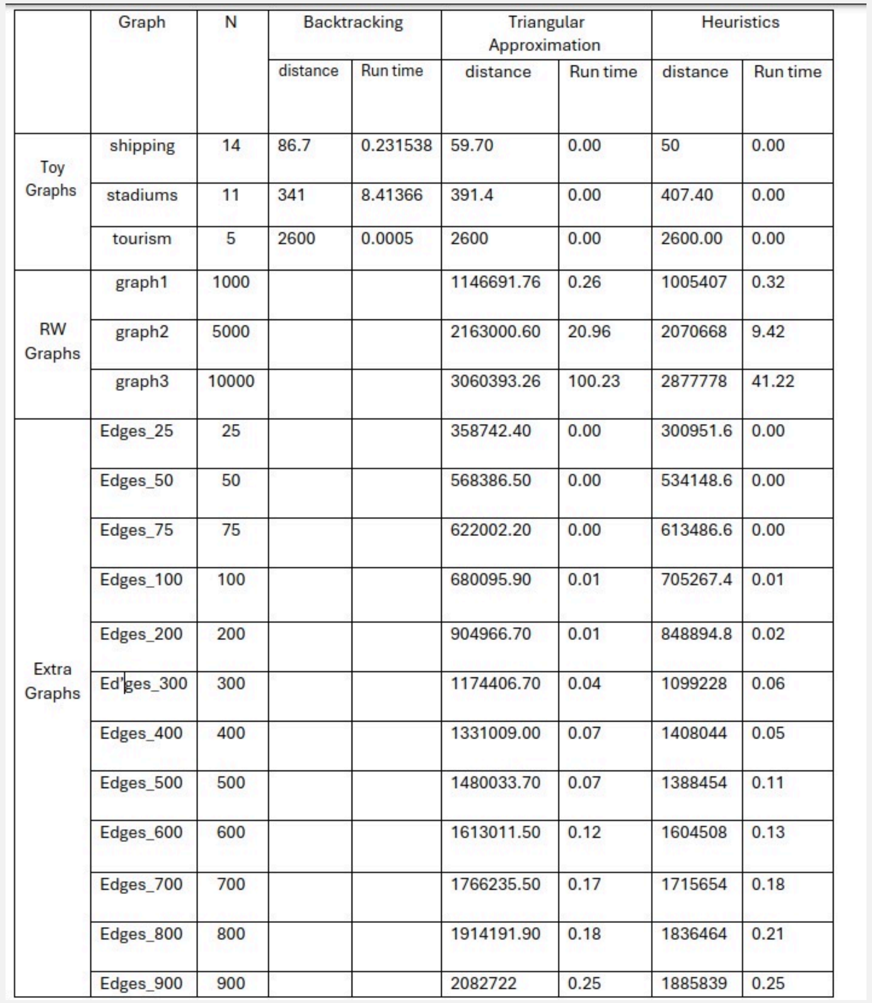


```
Please, select the dataset you want to use:
1. Extra Fully Connected Graphs
2. Real Word Graphs
3. Toy Graph
3

======================= Toy Graphs =====================
=======================================================

1. shipping.csv
2. stadiums.csv
3. tourism.csv
```



```
================== Statement Of Work ==================
======================================================


 1. Backtracking Algorithm.
 2. Triangular Approximation Heuristic.
 3. Other Heuristics.
 4. TSP in the Real World.

 Enter your choice (1-4)
 Press '0' to exit program
```

RUNTIME TABLE



	Graph	N	Backtracking		Triangular Approximation		Heuristics	
			distance	Run time	distance	Run time	distance	Run time
Toy Graphs	shipping	14	86.7	0.231538	59.70	0.00	50	0.00
	stadiums	11	341	8.41366	391.4	0.00	407.40	0.00
	tourism	5	2600	0.0005	2600	0.00	2600.00	0.00
RW Graphs	graph1	1000			1146691.76	0.26	1005407	0.32
	graph2	5000			2163000.60	20.96	2070668	9.42
	graph3	10000			3060393.26	100.23	2877778	41.22
Extra Graphs	Edges_25	25			358742.40	0.00	300951.6	0.00
	Edges_50	50			568386.50	0.00	534148.6	0.00
	Edges_75	75			622002.20	0.00	613486.6	0.00
	Edges_100	100			680095.90	0.01	705267.4	0.01
	Edges_200	200			904966.70	0.01	848894.8	0.02
	Edges_300	300			1174406.70	0.04	1099228	0.06
	Edges_400	400			1331009.00	0.07	1408044	0.05
	Edges_500	500			1480033.70	0.07	1388454	0.11
	Edges_600	600			1613011.50	0.12	1604508	0.13
	Edges_700	700			1766235.50	0.17	1715654	0.18
	Edges_800	800			1914191.90	0.18	1836464	0.21
	Edges_900	900			2082722	0.25	1885839	0.25