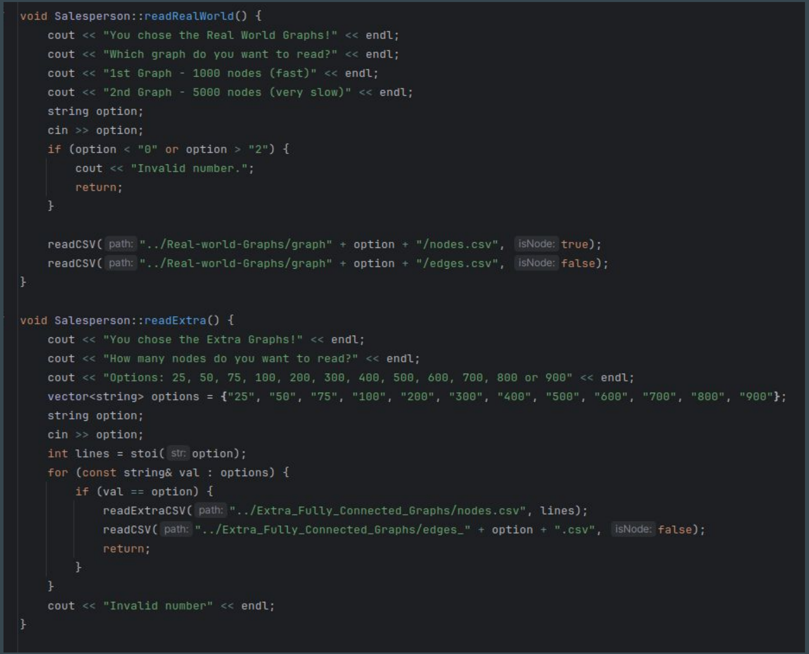
Travelling Salesperson Problem

●●●

Eduardo Baltazar
Gonçalo Remelhe
Joana Noites

Parsing Data

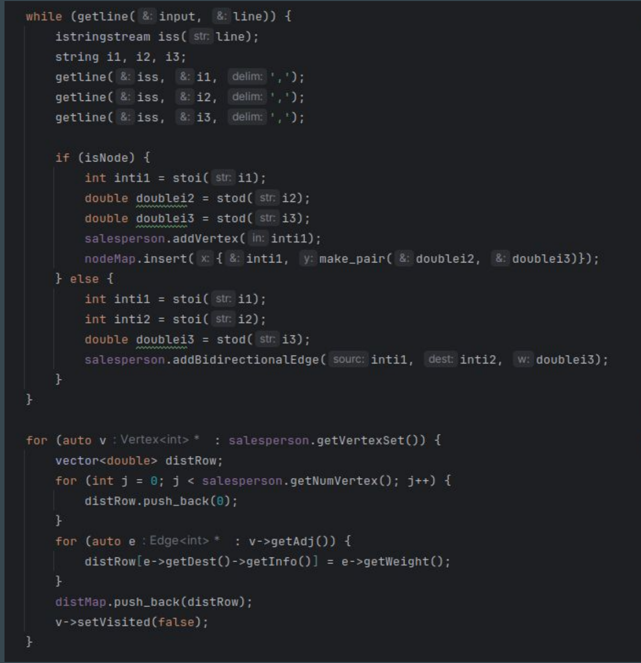


```
void Salesperson::readRealWorld() {
    cout << "You chose the Real World Graphs!" << endl;
    cout << "Which graph do you want to read?" << endl;
    cout << "1st Graph - 1000 nodes (fast)" << endl;
    cout << "2nd Graph - 5000 nodes (very slow)" << endl;
    string option;
    cin >> option;
    if (option < "0" or option > "2") {
        cout << "Invalid number.";
        return;
    }

    readCSV( path: "../Real-world-Graphs/graph" + option + "/nodes.csv",  isNode: true);
    readCSV( path: "../Real-world-Graphs/graph" + option + "/edges.csv",  isNode: false);
}

void Salesperson::readExtra() {
    cout << "You chose the Extra Graphs!" << endl;
    cout << "How many nodes do you want to read?" << endl;
    cout << "Options: 25, 50, 75, 100, 200, 300, 400, 500, 600, 700, 800 or 900" << endl;
    vector<string> options = {"25", "50", "75", "100", "200", "300", "400", "500", "600", "700", "800", "900"};
    string option;
    cin >> option;
    int lines = stoi( str: option);
    for (const string& val : options) {
        if (val == option) {
            readExtraCSV( path: "../Extra_Fully_Connected_Graphs/nodes.csv", lines);
            readCSV( path: "../Extra_Fully_Connected_Graphs/edges_" + option + ".csv",  isNode: false);
            return;
        }
    }
    cout << "Invalid number" << endl;
}
```

The csv files are read as seen in this example and then inserted in the graph

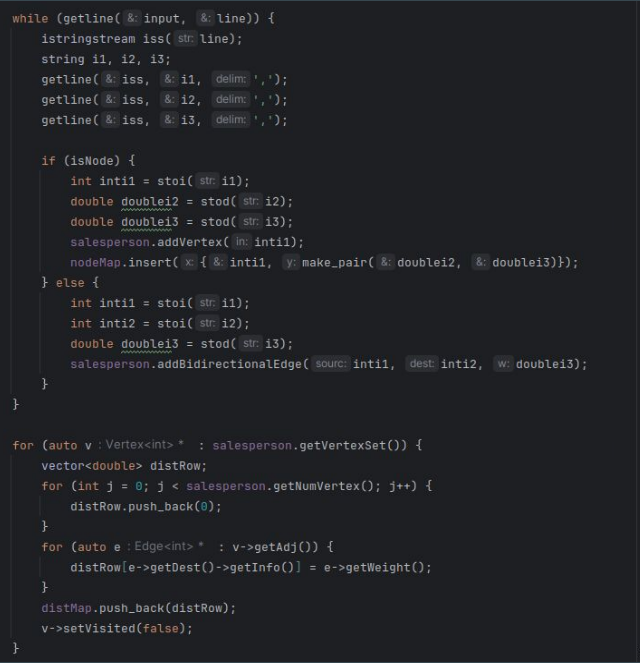
Parsing Data

This is how we store data in the graph



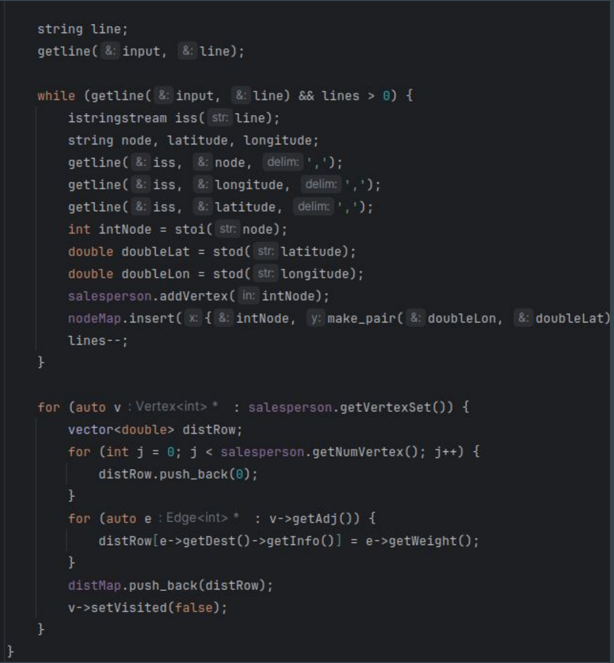
```
while (getline( &: input,  &: line)) {
    istringstream iss( str: line);
    string i1, i2, i3;
    getline( &: iss,  &: i1,  delim: ',');
    getline( &: iss,  &: i2,  delim: ',');
    getline( &: iss,  &: i3,  delim: ',');

    if (isNode) {
        int inti1 = stoi( str: i1);
        double doublei2 = stod( str: i2);
        double doublei3 = stod( str: i3);
        salesperson.addVertex( in: inti1);
        nodeMap.insert( x: { &: inti1,  y: make_pair( &: doublei2,  &: doublei3)});
    } else {
        int inti1 = stoi( str: i1);
        int inti2 = stoi( str: i2);
        double doublei3 = stod( str: i3);
        salesperson.addBidirectionalEdge( sourc: inti1,  dest: inti2,  w: doublei3);
    }
}

for (auto v : Vertex<int> *  : salesperson.getVertexSet()) {
    vector<double> distRow;
    for (int j = 0; j < salesperson.getNumVertex(); j++) {
        distRow.push_back(0);
    }
    for (auto e : Edge<int> *  : v->getAdj()) {
        distRow[e->getDest()->getInfo()] = e->getWeight();
    }
    distMap.push_back(distRow);
    v->setVisited(false);
}
```

Real World Graphs



```
while (getline( &: input,  &: line)) {
    istringstream iss( str: line);
    string i1, i2, i3;
    getline( &: iss,  &: i1,  delim: ',');
    getline( &: iss,  &: i2,  delim: ',');
    getline( &: iss,  &: i3,  delim: ',');

    if (isNode) {
        int inti1 = stoi( str: i1);
        double doublei2 = stod( str: i2);
        double doublei3 = stod( str: i3);
        salesperson.addVertex( in: inti1);
        nodeMap.insert( x: { &: inti1,  y: make_pair( &: doublei2,  &: doublei3)});
    } else {
        int inti1 = stoi( str: i1);
        int inti2 = stoi( str: i2);
        double doublei3 = stod( str: i3);
        salesperson.addBidirectionalEdge( sourc: inti1,  dest: inti2,  w: doublei3);
    }
}

for (auto v : Vertex<int> *  : salesperson.getVertexSet()) {
    vector<double> distRow;
    for (int j = 0; j < salesperson.getNumVertex(); j++) {
        distRow.push_back(0);
    }
    for (auto e : Edge<int> *  : v->getAdj()) {
        distRow[e->getDest()->getInfo()] = e->getWeight();
    }
    distMap.push_back(distRow);
    v->setVisited(false);
}
```

Toy Graphs

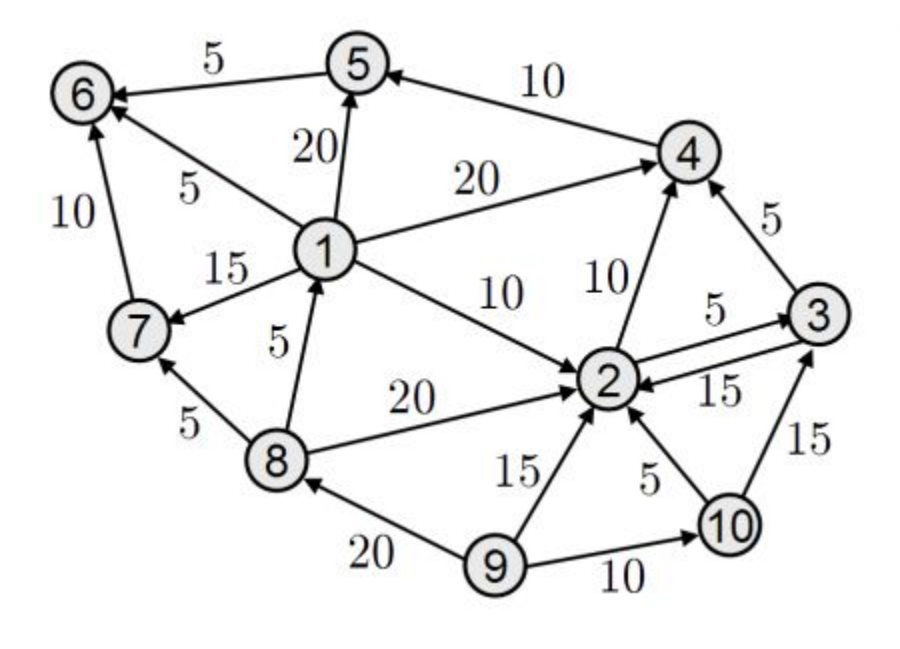


```
string line;
getline( &: input,  &: line);

while (getline( &: input,  &: line) && lines > 0) {
    istringstream iss( str: line);
    string node, latitude, longitude;
    getline( &: iss,  &: node,  delim: ',');
    getline( &: iss,  &: longitude,  delim: ',');
    getline( &: iss,  &: latitude,  delim: ',');
    int intNode = stoi( str: node);
    double doubleLat = stod( str: latitude);
    double doubleLon = stod( str: longitude);
    salesperson.addVertex( in: intNode);
    nodeMap.insert( x: { &: intNode,  y: make_pair( &: doubleLon,  &: doubleLat)
    lines--;
}

for (auto v : Vertex<int> *  : salesperson.getVertexSet()) {
    vector<double> distRow;
    for (int j = 0; j < salesperson.getNumVertex(); j++) {
        distRow.push_back(0);
    }
    for (auto e : Edge<int> *  : v->getAdj()) {
        distRow[e->getDest()->getInfo()] = e->getWeight();
    }
    distMap.push_back(distRow);
    v->setVisited(false);
}
}
```

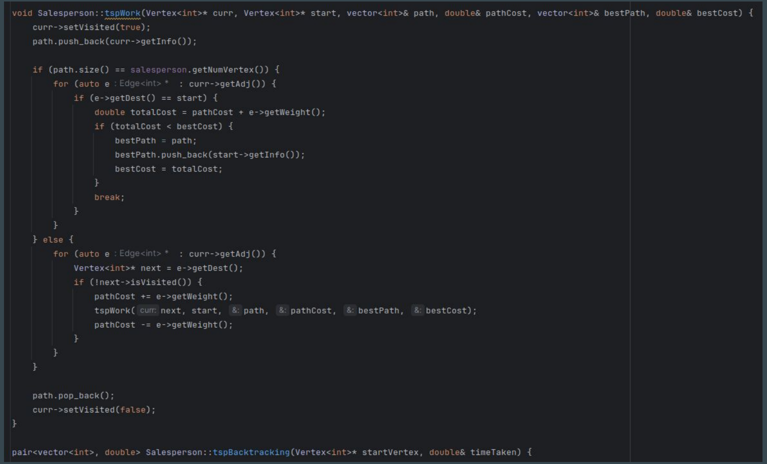
Extra Graphs

Graph

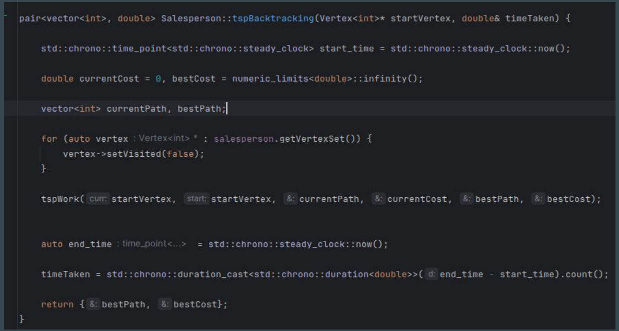
- The Graph used in the project is the same as the one used in TP classes;



Algorithms - Backtracking



```
void Salesperson::tspWork(Vertex<int>* curr, Vertex<int>* start, vector<int>& path, double& pathCost, vector<int>& bestPath, double& bestCost) {
    curr->setVisited(true);
    path.push_back(curr->getInfo());

    if (path.size() == salesperson.getNumVertex()) {
        for (auto e : Edge<int> *  : curr->getAdj()) {
            if (e->getDest() == start) {
                double totalCost = pathCost + e->getWeight();
                if (totalCost < bestCost) {
                    bestPath = path;
                    bestPath.push_back(start->getInfo());
                    bestCost = totalCost;
                }
                break;
            }
        }
    } else {
        for (auto e : Edge<int> *  : curr->getAdj()) {
            Vertex<int>* next = e->getDest();
            if (!next->isVisited()) {
                pathCost += e->getWeight();
                tspWork( curr: next, start,  &: path,  &: pathCost,  &: bestPath,  &: bestCost);
                pathCost -= e->getWeight();
            }
        }
    }

    path.pop_back();
    curr->setVisited(false);
}

pair<vector<int>, double> Salesperson::tspBacktracking(Vertex<int>* startVertex, double& timeTaken) {
```

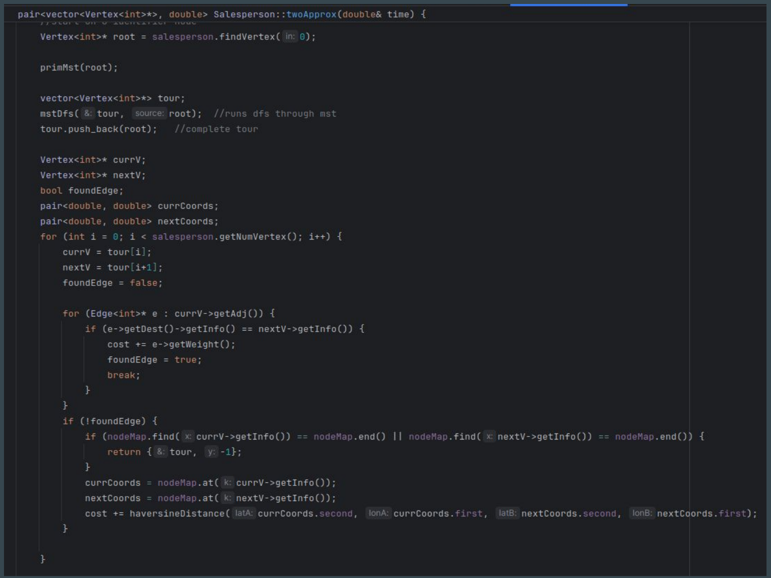


```
pair<vector<int>, double> Salesperson::tspBacktracking(Vertex<int>* startVertex, double& timeTaken) {

    std::chrono::time_point<std::chrono::steady_clock> start_time = std::chrono::steady_clock::now();

    double currentCost = 0, bestCost = numeric_limits<double>::infinity();

    vector<int> currentPath, bestPath;

    for (auto vertex : Vertex<int> * : salesperson.getVertexSet()) {
        vertex->setVisited(false);
    }

    tspWork( curr: startVertex,  start: startVertex,  &: currentPath,  &: currentCost,  &: bestPath,  &: bestCost);


    auto end_time : time_point<...>  = std::chrono::steady_clock::now();

    timeTaken = std::chrono::duration_cast<std::chrono::duration<double>>( d: end_time - start_time).count();

    return { &: bestPath,  &: bestCost};
}
```

The tspWork method uses a recursive approach to explore all possible paths from the current vertex to each neighbouring unvisited vertex. It maintains a record of the current path and its associated cost. Upon reaching a complete path, it compares the total cost to the best-known cost, updating the best path and cost if a more optimal solution is found.

The tspBacktracking method invokes the tspWork method to start the search process from the specified starting vertex. Then, the method returns the best path found along with its associated cost, providing a comprehensive solution to the TSP with insights into computational efficiency.

Algorithms - Triangular Approximation

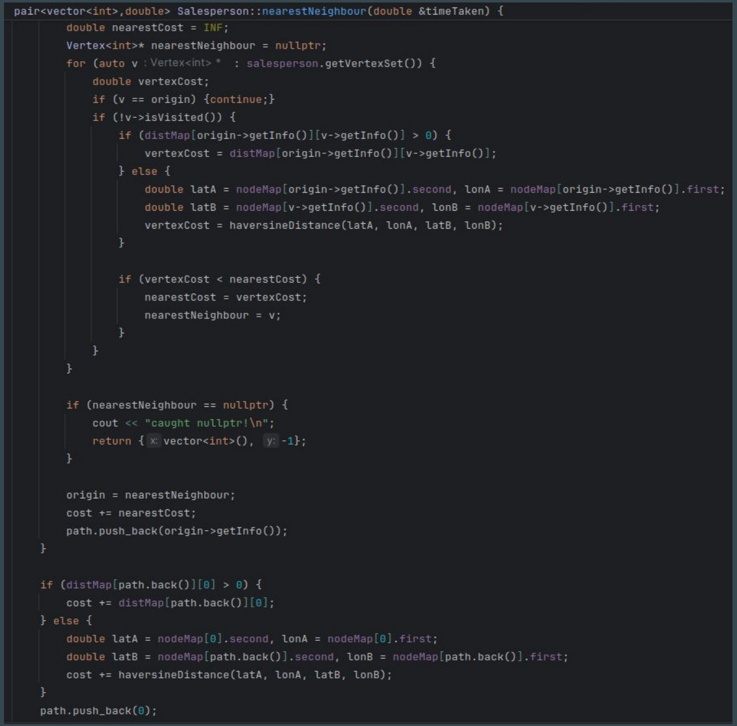


```
pair<vector<Vertex<int>*>, double> Salesperson::twoApprox(double& time) {
    Vertex<int>* root = salesperson.findVertex( in: 0);

    primMst(root);

    vector<Vertex<int>*> tour;
    mstDfs( &: tour,  source: root);  //runs dfs through mst
    tour.push_back(root);   //complete tour

    Vertex<int>* currV;
    Vertex<int>* nextV;
    bool foundEdge;
    pair<double, double> currCoords;
    pair<double, double> nextCoords;
    for (int i = 0; i < salesperson.getNumVertex(); i++) {
        currV = tour[i];
        nextV = tour[i+1];
        foundEdge = false;

        for (Edge<int>* e : currV->getAdj()) {
            if (e->getDest()->getInfo() == nextV->getInfo()) {
                cost += e->getWeight();
                foundEdge = true;
                break;
            }
        }
        if (!foundEdge) {
            if (nodeMap.find( x: currV->getInfo()) == nodeMap.end() || nodeMap.find( x: nextV->getInfo()) == nodeMap.end()) {
                return { &: tour,  y: -1};
            }
            currCoords = nodeMap.at( k: currV->getInfo());
            nextCoords = nodeMap.at( k: nextV->getInfo());
            cost += haversineDistance( latA: currCoords.second,  lonA: currCoords.first,  latB: nextCoords.second,  lonB: nextCoords.first);
        }

    }
```

This algorithm constructs a Minimum Spanning Tree (MST) using Prim's algorithm by calling the primMst method. Subsequently, it performs a Depth-First Search (DFS) traversal on the MST to generate a tour, appending each visited vertex to the tour vector. Finally, the method closes the tour by connecting the last vertex back to the starting vertex, ensuring a complete tour.

During the tour construction, the method calculates the total cost by summing the weights of the edges. For edges present in the MST, their weights are directly added to the cost. If an edge is not found in the MST, indicating a missing connection, the method computes the Haversine distance between the corresponding geographic coordinates and adds it to the cost.

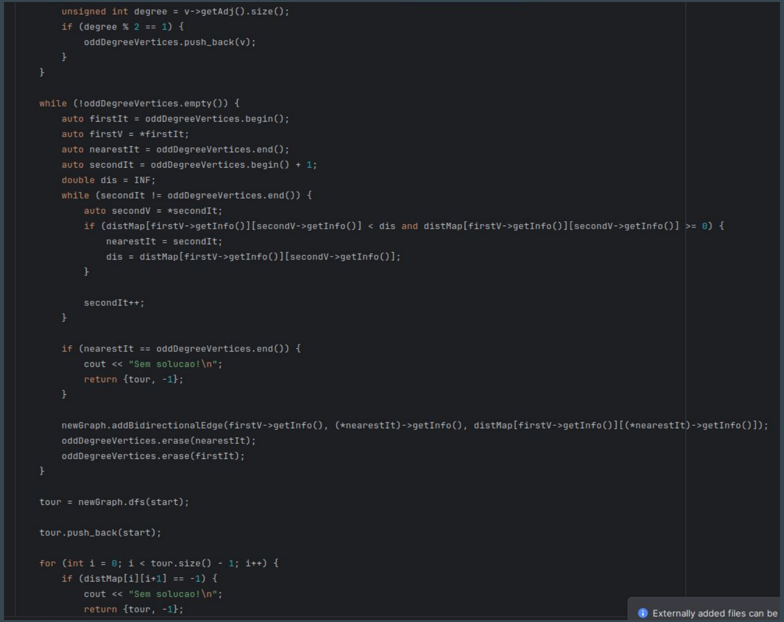
Algorithms - The other heuristic - Nearest Neighbors



```
pair<vector<int>,double> Salesperson::nearestNeighbour(double &timeTaken) {
        double nearestCost = INF;
        Vertex<int>* nearestNeighbour = nullptr;
        for (auto v : Vertex<int> *  : salesperson.getVertexSet()) {
            double vertexCost;
            if (v == origin) {continue;}
            if (!v->isVisited()) {
                if (distMap[origin->getInfo()][v->getInfo()] > 0) {
                    vertexCost = distMap[origin->getInfo()][v->getInfo()];
                } else {
                    double latA = nodeMap[origin->getInfo()].second, lonA = nodeMap[origin->getInfo()].first;
                    double latB = nodeMap[v->getInfo()].second, lonB = nodeMap[v->getInfo()].first;
                    vertexCost = haversineDistance(latA, lonA, latB, lonB);
                }

                if (vertexCost < nearestCost) {
                    nearestCost = vertexCost;
                    nearestNeighbour = v;
                }
            }
        }

        if (nearestNeighbour == nullptr) {
            cout << "caught nullptr!\n";
            return { x: vector<int>(),  y: -1};
        }

        origin = nearestNeighbour;
        cost += nearestCost;
        path.push_back(origin->getInfo());
    }

    if (distMap[path.back()][0] > 0) {
        cost += distMap[path.back()][0];
    } else {
        double latA = nodeMap[0].second, lonA = nodeMap[0].first;
        double latB = nodeMap[path.back()].second, lonB = nodeMap[path.back()].first;
        cost += haversineDistance(latA, lonA, latB, lonB);
    }
    path.push_back(0);
```

The nearestNeighbour algorithm starts from an arbitrary vertex and iteratively selects the nearest unvisited neighbor until all vertices are visited, forming a tour. Then identifies the origin vertex and adds it to the path. Next, it iterates through the remaining vertices, calculating the cost to each unvisited neighbor. If a direct distance between vertices is available in the distance map, it is used; otherwise, the Haversine distance formula is employed to compute the distance based on geographic coordinates. When all the vertices are finally visited, the method calculates the distance from the last vertex back to the origin, closes the tour, and returns the resulting path along with its total cost

Algorithms - TSP in the real world - Christofides Algorithm



```
            unsigned int degree = v->getAdj().size();
            if (degree % 2 == 1) {
                oddDegreeVertices.push_back(v);
            }
        }

        while (!oddDegreeVertices.empty()) {
            auto firstIt = oddDegreeVertices.begin();
            auto firstV = *firstIt;
            auto nearestIt = oddDegreeVertices.end();
            auto secondIt = oddDegreeVertices.begin() + 1;
            double dis = INF;
            while (secondIt != oddDegreeVertices.end()) {
                auto secondV = *secondIt;
                if (distMap[firstV->getInfo()][secondV->getInfo()] < dis and distMap[firstV->getInfo()][secondV->getInfo()] >= 0) {
                    nearestIt = secondIt;
                    dis = distMap[firstV->getInfo()][secondV->getInfo()];
                }

                secondIt++;
            }

            if (nearestIt == oddDegreeVertices.end()) {
                cout << "Sem solucao!\n";
                return {tour, -1};
            }

            newGraph.addBidirectionalEdge(firstV->getInfo(), (*nearestIt)->getInfo(), distMap[firstV->getInfo()][(*nearestIt)->getInfo()]);
            oddDegreeVertices.erase(nearestIt);
            oddDegreeVertices.erase(firstIt);
        }

        tour = newGraph.dfs(start);

        tour.push_back(start);

        for (int i = 0; i < tour.size() - 1; i++) {
            if (distMap[i][i+1] == -1) {
                cout << "Sem solucao!\n";
                return {tour, -1};
```

The Christofides' Algorithm is a heuristic algorithm that guarantees a result no bigger that 1.5 times the optimal solution, even though it is not the best algorithm for incomplete graphs. This algorithm starts with the construction of a MST (we used Prim's algorithm). Next, from the MST we pick the edges with odd degree and then we find the perfect matching for all of the odd vertices. On our approach, we used the Nearest Neighbour greedy algorithm due to the enormous structures of the real world graphs. Finally, this algorithm finds an Eulerian circuit from our MST with the perfect match and removes the repeated visited vertices.

Final considerations

- All work was evenly distributed among the group members;
- All the proposed functions were implemented;