Routing Algorithm for Ocean Shipping and Urban Deliveries - TSP

2nd Project - Design of Algorithms

Made by:

Diogo Miguel Farias Pinto - up201906067
Matilde Pinho Borges Sequeira - up202007623

Problem Description

In this programming project, we were asked to design efficient algorithms to find optimal routes for vehicles in generic shipping and delivery scenarios, from urban deliveries to ocean shipping.

This problem corresponds to the TSP :

- The Travelling Salesman is a problem that tries to determine the shortest route to visit a set of cities, only once, returning to the starting city.

Backtracking Algorithm - Implementation

The Backtracking algorithm works by recursively exploring all possible solutions to a problem.

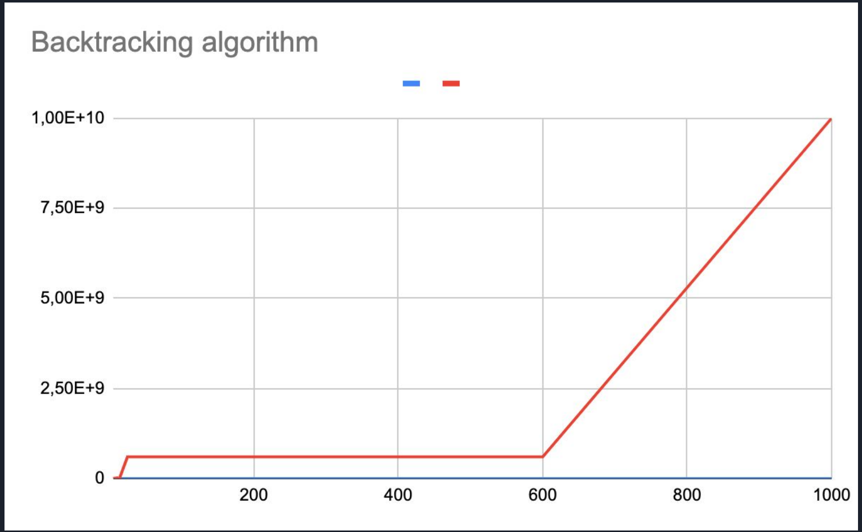
- We start at the provided starting point;
- Explore all possible extensions of the current solution;
- If a city is still unvisited, set it as visited and update the total cost;
- If a city is already visited, we backtrack to the previous one, to find another solution;
- Repeat the last 3 steps until all possible solutions have been explored.
- Finally we return the optimal solution.

```cpp
template <class T>
int Graph<T>::back_tracking_algorithm() {
    int n = vertexSet.size();
    distMatrix = (double**) malloc(n*sizeof(double*));
    for(int i = 0; i < n; ++i)
        distMatrix[i] = (double*) malloc(n*sizeof(double));

    for(auto vert : vertexSet)
        vert->setVisited(false);

    int ans = INF;
    Vertex<T>* startNode = findVertex(0);

    back_tracking_algorithm_rec(startNode, n, 1, 0, &ans);

    //deleteMatrix(distMatrix, n);
    for(auto vert : vertexSet)
        vert->setVisited(false);

    return ans;
}
```

```cpp
template <class T>
void Graph<T>::back_tracking_algorithm_rec(Vertex<T>* curr, int n, int count, int cost, int* ans) {
    auto startNode = findVertex(0);
    Edge<T>* edgeCurrStart = nullptr;
    for (Edge<T>* e: curr->getAdj()){
        if (e->getDest()->getInfo() == startNode->getInfo()) {
            edgeCurrStart = e;
            break;
        }
    }

    // If reached last node and edge connects to start
    if(count == n && edgeCurrStart != nullptr) {
        if (*ans >= cost + edgeCurrStart->getWeight())
            *ans = cost + edgeCurrStart->getWeight();
        return;
    }


    for(Edge<T>* e : curr->getAdj()){
        if(!e->getDest()->isVisited()){
            e->getDest()->setVisited(true);
            back_tracking_algorithm_rec(e->getDest(), n, count + 1, cost + e->getWeight(), ans);
            e->getDest()->setVisited(false);
        }
    }
}
```

Backtracking Algorithm - Efficiency

Dataset	Execution time (µs)
Tourism (5 nodes)	137 ≈ 0,000137s
Stadiums (11 nodes)	17115619 ≈ 17 s
Shipping (14 nodes)	284912 ≈ 0,28 s
edges_25 (25 nodes connected)	+ 10 min
edges_600 (600 nodes connected)	+ 10 min
Graph 1 (1000 nodes)	+ 10 min



Triangular Approximation - Implementation

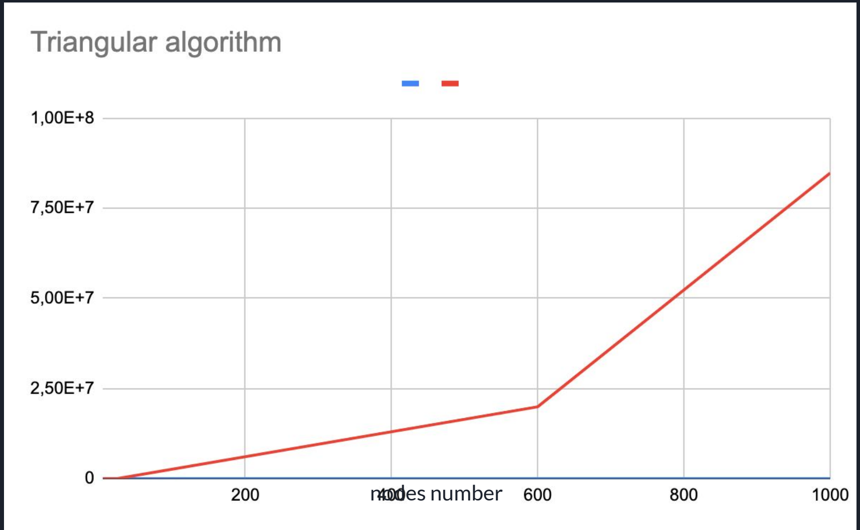
For this approach, we followed these steps:

- Construct a MST using Prim's algorithm;
- Perform a DFS using the MST created;
- During the DFS, we visit all the nodes in a triangular pattern;
- Return to the starting point once all the nodes have been visited.

```
template <class T>
std::vector<Vertex<T>*> Graph<T>::tspTriangular(int* cost) {
    for(auto v : vertexSet) {
        v->setVisited(false);
        v->setPath(nullptr);
        v->setDist(DOUBLE_INF);
        for(auto e : v->getAdj())
            e->setSelected(false);
    }
    prim_algorithm();

    for(auto v : vertexSet)
        v->setVisited(false);

    auto startNode = findVertex( in: 0);
    std::vector<Vertex<T>*> path = dfsTriangular( v: startNode);
    //path.push_back(startNode);

    *cost = 0;
    for (int i = 0; i < path.size() - 1; i++)
        *cost += getDistFromTo( src: path[i], dst: path[i + 1]);

    return path;
}
```

Triangular Approximation - Efficiency

Dataset	Execution time (µs)
Tourism (5 nodes)	108
Stadiums (11 nodes)	407
Shipping (14 nodes)	480
edges_25 (25 nodes connected)	1408
edges_600 (600 nodes connected)	19907820 ≈ 19,9 s
Graph 1 (1000 nodes)	84782000 ≈ 84,7s



Nearest Neighbor - Implementation

For our own heuristic we chose the Nearest Neighbor, and these are the steps we followed:

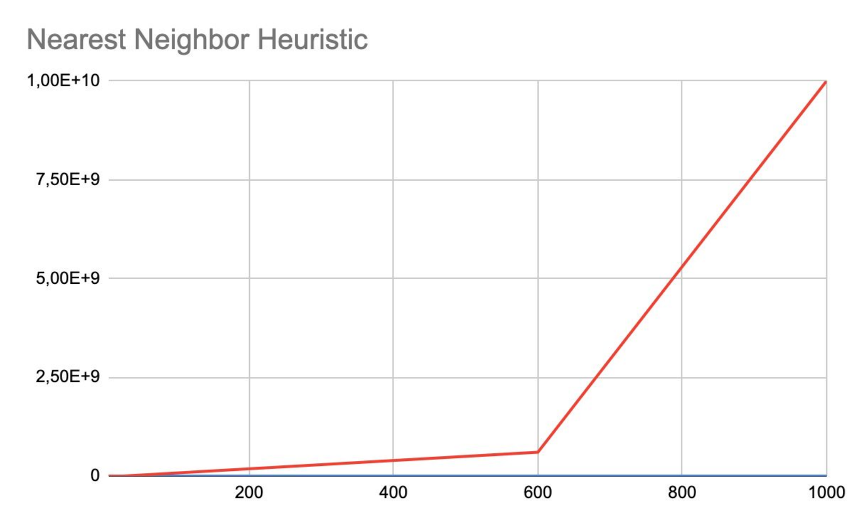
- Start on the provided starting point and check for all the neighbor nodes;
- Choose the lightest edge possible and move to that neighbor node;
- Repeat this process until all the nodes are visited;
- Return to the starting node.

```cpp
void algorithms::nearestNeighborHeuristic(Graph<int> graph) {


    std::chrono::steady_clock::time_point begin = std::chrono::steady_clock::now();

    double cost;
    std::vector<Vertex<int>*> path = graph.nearestNeighborHeuristic(cost);

    std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();

    std::cout << "Path is: ";
    for(auto it = path.begin(); it < path.end(); ++it){
        std::cout << (*it)->getInfo();
        if(it+1 != path.end()) {
            std::cout << " -> " << std::endl;
        }
    }
    std::cout << "The cost is: " << cost << std::endl;
    std::cout << "Running time: " << std::chrono::duration_cast<std::chrono::microseconds>(end - begin).count() << " ms" << std::endl;



}
```

```cpp
<T>
std::vector<Vertex<T> *> Graph<T>::nearestNeighborHeuristic(double &cost) const {

    std::vector<Vertex<T>*> path;

    auto startNode = findVertex(0);

    for(auto v : vertexSet)
        v->setVisited(false);

    path.push_back(startNode);

    startNode->setVisited(true);
    int numNodes = vertexSet.size();

    Vertex<T>* current = startNode;
    for (int i = 0; i < numNodes - 1; i++) {
        int minDistance = std::numeric_limits<int>::max();
        Vertex<T>* nextNode = nullptr;
        for (auto v : vertexSet) {
            if (!v->isVisited() && getDistFromTo(current, v) < minDistance) {
                minDistance = getDistFromTo(current, v);
                nextNode = v;
            }
        }
        if (nextNode != nullptr) {
            nextNode->setVisited(true);
            path.push_back(nextNode);
            cost = cost + minDistance;
            current = nextNode;
        }

    }
    cost += getDistFromTo(current, startNode);
    path.push_back(startNode); // Return to the start node
    return path;

}
```

Nearest Neighbor - Efficiency

Dataset	Execution time (µs)
Tourism (5 nodes)	219
Stadiums (11 nodes)	1300
Shipping (14 nodes)	2151
edges_25 (25 nodes connected)	17452
edges_600 (600 nodes connected)	*+ 10 min*
Graph 1 (1000 nodes)	*+ 15 min*



TSP in the Real World

Assuming that not all the graphs are fully connected, we developed the following algorithm:

- Ask the user to select the origin node;
- Perform a BFS on the graph, starting on the origin to, to check if the graph is connected (if there is a possible path);
- If so, we calculate the path starting on that node, using the nearest neighbor heuristic (the one with better cost results).

Comparing Results

Dataset	Backtracking cost	Triangular Approximation cost	Nearest Neighbor cost
Tourism (5 nodes)	2600	2600	2600
Stadiums (11 nodes)	341	398.1	405.1
Shipping (14 nodes)	86.7	123.7	79.9
edges_25 (25 nodes connected)	-	349573	300939

Final Results

Our conclusions are that:

- For our implementation, graphs with a big number of edges and nodes, especially the connected ones, take a lot of time to finish, for all the algorithms;

- The Triangular Approximation is much faster and has better results, for the most cases, than our Backtracking algorithm, only usable for small graphs;

- The approach we took on doing the Nearest Neighbor Heuristic has much better results than the Triangular Approximation, for the most cases, although it has a bigger running time.

Contribution

In terms of contribution, we were able to distribute the work between the two of us, in the best way possible.

Diogo Pinto - 50%

Matilde Sequeira - 50%