U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

ROUTING ALGORITHM FOR OCEAN SHIPPING AND URBAN DELIVERIES

2nd Project - KEY TOPICS FOR PRESENTATION

Design of Algorithms

- Hélder Costa (202108719)
- Luís Duarte (202108734)
- Luís Jesus (202108683)

Dataset's Reading Description

- There are basically two files' formats in this dataset: graphs given by an edges file and graphs given by a vertex file + edges file.

- Thus, we created two different functions to accomplish the dataset's reading as efficient as possible.

- Some files (for example, tourism.csv) have labels in the nodes, so, our class vertex has an option field "label".

- Also, Real-Word Graphs give us coordinates (latitude and longitude). Therefore, class vertex also has those two fields that may be filled in case they are provided.

	A	B	C	D	E
	origem	destino	distancia	label origem	label destino
0	0	1	1300	carmo	dLuis
1	0	2	1000	carmo	se
2	0	3	450	carmo	clerigos
3	0	4	750	carmo	bolsa
4	1	2	450	dLuis	se
5	1	3	950	dLuis	clerigos
6	1	4	450	dLuis	bolsa
7	2	3	500	se	clerigos
8	2	4	600	se	bolsa
9	3	4	750	clerigos	bolsa

	id	longitude	latitude
0	0	-47.84923	-15.6743
1	1	-47.654252	-15.601082
2	2	-47.812892	-15.649247
3	3	-47.828528	-15.675495

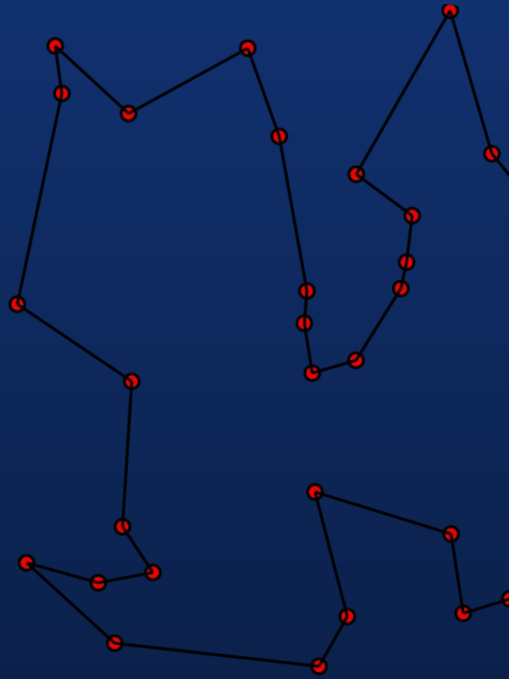
	origem	destino	distancia
0	0	1	25793.4
1	0	2	9851.6
2	0	3	7216.7
3	0	4	26900.9

Motivation of this project

- This project is essentially centered in a well-known and studied NP-hard problem called the “Travelling Salesman Problem”.

- The TSP poses a significant challenge due to its combinatorial nature and practical relevance in various domains such as logistics, transportation, and network routing.

- Through this project, we seek to improve problem-solving skills and ultimately enable cost savings and operational efficiency in real-world applications.



4.1. BACKTRACKING ALGORITHM

- The first approach for TSP was going into a backtracking algorithm approach.
- We systematically all possible tracks in the graph keeping track of the best path and its cost as it progresses.
- The algorithm exhaustively searches through the solution space to find the optimal solution to the TSP.
- Thus, it is a good solution for very small graphs (for example, the Toy Graphs). However, for few bigger inputs it just gets completely useless due to its complexity of…

O(n!)

```
void Graph::tsp_backtracking(std::vector<int> &path, std::vector<int> &bestPath,
                             double &minCost, double cumulatedCost) {
    int first = path.front();
    int last = path.back();

    if (path.size() == vertexSet.size()) {
        for (Edge* e : findVertex( id: last)->getEdges()) {
            Vertex* v = e->getDest();
            if (v->getId() == first) {
                double cycleCost = cumulatedCost + e->getWeight();
                if (cycleCost < minCost) {
                    minCost = cycleCost;
                    bestPath = path;
                }
                break;
            }
        }
        return;
    }

    for (Edge* e : findVertex( id: last)->getEdges()) {
        Vertex* v = e->getDest();
        if (!v->isVisited()) {
            double new_cost = cumulatedCost + e->getWeight();
            path.push_back(v->getId());
            v->setVisited(true);
            tsp_backtracking( &: path,  &: bestPath,  &: minCost,  cumulatedCost: new_cost);
            path.pop_back();
            v->setVisited(false);
        }
    }
}
```

4.1. BACKTRACKING ALGORITHM...

1) The algorithm is called and path has the starting vertex;
2) If the path contains all the vertices, a cycle has been formed. If there's an edge from the last vertex to the first, it calculates the cycle cost and if it's lower than the current minimum, minimum cost and best path are updated.
3) If not, we'll be exploring the neighboring vertices of the last vertex in the path.
4) If each vertex has not been visited yet, we calculate the new cost.
5) Update the path, and recursively call the function.
6) After the recursive call, the code backtracks by removing the last vertex from the path and marks it unvisited to explore other paths.

4.2. TRIANGULAR APPROXIMATION HEURISTIC

- This heuristic is based on the following assumption: “The least distant path to reach a vertex j from i is always to reach j directly from i, rather than through some other vertex”.
- Basically, we organized the algorithm in the following way:

1) Vertex (id= 0) is the starting and ending point for TSP;
2) Construct the MST from 1 using Prim’s Algorithm;
3) Execute a pre-order traversal in the MST, creating a path;
4) Iterate through the path and calculate the path cost, taking in consideration the Triangular Inequality proposal.

```
double Graph::triangularInequalityHeuristic() {
    std::vector<Vertex*> path;
    std::vector<Edge*> edges;
    resetVisited();
    MSTPrim(edges);

    resetVisited();
    Vertex* first = vertexSet.find(0)->second;
    preOrderTraversal(edges, first, path);

    double distance = computePathDistance(path);
    return distance;
}
```

4.2. MST using Prim's Algorithm with MutablePriorityQueue

```
void Graph::MSTPrim(std::vector<Edge*> &edges) {

    for (auto v : vertexSet) {
        v.second->setDistance(std::numeric_limits<double>::max());
        v.second->setVisited(false);
        v.second->setPath(nullptr);
    }

    Vertex* first = vertexSet.find(0)->second;
    first->setDistance(0);

    MutablePriorityQueue<Vertex> q;
    q.insert(first);

    while (!q.empty()) {
        Vertex *v = q.extractMin();
        v->setVisited(true);
        for (Edge *e: v->getEdges()) {
            Vertex *w = e->getDest();
            if (!w->isVisited()) {
                auto oldDist = w->getDistance();
                if (e->getWeight() < oldDist) {
                    w->setDistance(e->getWeight());
                    w->setPath(e);
                    if (oldDist == std::numeric_limits<double>::max()) {
                        q.insert(w);
                    } else {
                        q.decreaseKey(w);
                    }
                }
            }
        }
    }

    for (auto v : vertexSet) {
        if (v.second->getPath() != nullptr) {
            edges.push_back(v.second->getPath());
        }
    }

    // Sort edges by weight
    std::sort(edges.begin(), edges.end(), [](Edge* e1, Edge* e2) {
        return e1->getWeight() < e2->getWeight();
    });
}
```

4.2. Pre-order Traversal

- The code sets the current vertex as visited and adds it to the path vector.
- It then iterates through each edge in the minimum spanning tree and checks if the origin vertex of the edge is the same as the current vertex and if the destination vertex has not been visited yet.
- If both conditions are met, the function recursively calls itself with the destination vertex as the new starting point to continue the traversal.

Overall, the code performs a pre-order traversal of a graph, visiting vertices in the order of the traversal and storing them in the path vector.

```
void Graph::preOrderTraversal(const std::vector<Edge *>& mst, Vertex *v, std::vector<Vertex *> &path) {
    v->setVisited(true);

    path.push_back(v);

    for (Edge* e : mst) {
        if (e->getOrigin()->getId() == v->getId() && !e->getDest()->isVisited()) {
            preOrderTraversal(mst, e->getDest(), path);
        }
    }
}
```

4.2. Path Cost Computation

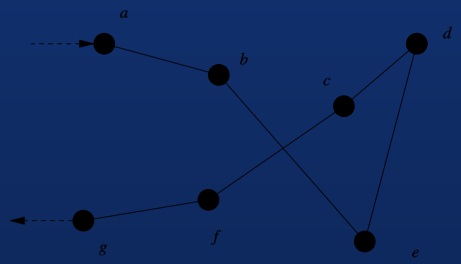
- Iterates through each pair of consecutive vertices in the path and checks if they are connected directly;
- If they are not connected, it calculates the distance using the haversine formula, else, it retrieves the edge weight between these nodes.
- After checking all the pairs, it checks if first and last vertex are connected (to close/complete the tour), and applies the logic described in the step before.

```
double Graph::computePathDistance(std::vector<Vertex *> &path) {
    double distance = 0.0;

    for (int i = 0; i < path.size() - 1; ++i) {
        Vertex* v1 = path[i];
        Vertex* v2 = path[i + 1];

        if (!connectedNodes(v1->getId(), v2->getId())) {
            distance += haversineDistance(v1, v2);
            continue;
        }

        for (Edge* e : v1->getEdges()) {
            if (e->getDest()->getId() == v2->getId()) {
                distance += e->getWeight();
                break;
            }
        }
    }

    int last = path.back()->getId();
    int first = path.front()->getId();

    if (!connectedNodes(last, first)) {
        distance += haversineDistance(findVertex(last), findVertex(first));
    }
    else {
        for (Edge* e : path.back()->getEdges()) {
            if (e->getDest()->getId() == first) {
                distance += e->getWeight();
                break;
            }
        }
    }

    return distance;
}
```

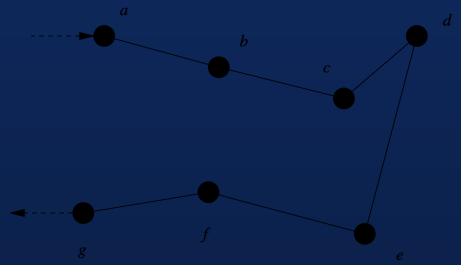
4.3. OUR OWN HEURISTIC

In total we did 3 heuristics, almost all with the 2-approximation algorithm as a base and with a focus on speed:

- Local search using 2-opt only (starts with a ascending vertex id route);
- 2-approximation with 2-opt optimization at the end;
- Simulated Annealing using 2-opt.



The 2-approximation with 2-opt optimization is the most consistent one, since simulated annealing is non-deterministic and local search using 2-opt only might reach an local optima way before the global optima.
Depending on the circumstances, eg: size/limited runtime, one algorithm might have an edge over another.



4.3. OUR OWN HEURISTIC - Simulated Annealing

Simulated annealing tries to, in a non-deterministic way, overcome the barriers to local search.

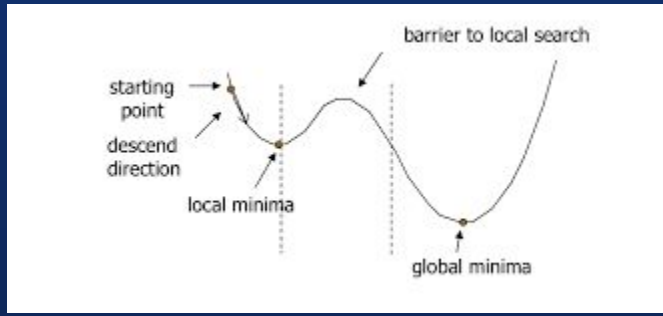
On larger graphs, it's unlikely that the found minima will be the global minima, so we need some way to perturb the algorithm to, in a probabilistic way, choose a worse cycle that might overcome the barrier and find a better tour.

This is based on this equation:

$$P(x) = e^{\Delta/T}$$

Where Δ is the distance change between two tours, and T is the current temperature.

The current temperature decreases by a ratio each iteration to allow for higher perturbation at the start and then settle on a solution.



Median improvement: 10%

GREATEST DIFFICULTIES

The greatest difficulties/challenges during the development of this project were:

- Understanding the logic of some algorithms for the TSP;
- Getting the correct parameters to use simulated annealing correctly;
- The project requirements are too abstract and therefore we can't make many constraints that simplify the problem and are therefore bound to the theoretical generic algorithms;
- Balance the work flow with other course units' projects as well as managing the fatigue due to previous work load;
- Develop a balanced solution for our own heuristic without "imitating" too much already existing algorithms.

MEMBERS' CONTRIBUTIONS

- We all worked as much as we could during the development of this project. The disponibility of each member was not always aligned because of different ways of managing time, but we could still make it.
- Members were assigned tasks that they felt more comfortable doing: some did the more aesthetic part, others more algorithmic parts, etc.

Hélder Costa: 33.3%

Luís Duarte: 33.3%

Luís Jesus: 33.3%