





OCEAN SHIPPING AND URBAN DELIVERIES

Presented By

- Ana Sofia Baptista - up202207334
 - Eduardo Santos - up202207521
 - Pedro Pedro - up202206961
- 
- 

CLASSES

- MutablePriorityQueue - classe apresentada em aula e usada no algoritmo de Prim para MST's
- Graph - (class Vertex - lista de Edges, set de Edges, lista de incomings, lista de children, double longitude, double latitude, id, booleano visited, Vertex path, double dist), (class Edge - destino, origem, double distance, Edge reverse)
- DataManip - vetor bestPath, double bestCost
- Menu - com um Data, onde são criados vários menus para utilização do user

LEITURA DATASET

```
void DataManip::readTourism(string filename) {

    ifstream in(s: filename);
    int destino, origem;
    string labelo, labeld;
    double distancia;
    string line;

    getline(& in, & line);

    if (in.is_open()) {

        while(getline(& in, & line)){

            istringstream iss(str: line);
            iss >> origem;

            iss.ignore();
            iss >> destino;

            iss.ignore();
            iss >> distancia;

            iss.ignore();
            iss >> labelo;
            iss.ignore();
            iss>>labeld;

            graph_.addVertex(id: origem, longitude: 0, latitude: 0);
            graph_.addVertex(id: destino, longitude: 0, latitude: 0);
            graph_.addBidirectionalEdge(& origem, & destino, w: distancia);

        }

    } else
        cout << "Could not open the file\n";

}
```

```
void DataManip::readEdges(string filename) {

    ifstream in(s: filename);
    int destino, origem;
    double distancia;
    string line;

    // Verifica se a primeira linha é "origem,destino,distancia"

    if (in.is_open()) {

        while(getline(& in, & line)){

            if (line == "origem,destino,distancia" || line == "origem,destino,haversine_distance") {
                std::getline(& in, & line);
            }

            istringstream iss(str: line);
            iss >> origem;

            iss.ignore();
            iss >> destino;

            iss.ignore();
            iss >> distancia;

            graph_.addVertex(id: origem, longitude: 0, latitude: 0);
            graph_.addVertex(id: destino, longitude: 0, latitude: 0);
            graph_.addBidirectionalEdge(& origem, & destino, w: distancia);

        }

    } else
        cout << "Could not open the file\n";

}
```

LEITURA DATASET

```
void DataManip::readEdgesLarge(string filename) {

    ifstream in(s: filename);
    int destino, origem;
    double distancia;
    string line;

    // Verifica se a primeira linha é "origem,destino,distancia"

    if (in.is_open()) {
        std::getline(& in, & line);
        while(getline(& in, & line)){

            istringstream iss(str: line);
            iss >> origem;

            iss.ignore();
            iss >> destino;

            iss.ignore();
            iss >> distancia;
            graph_.addBidirectionalEdge(& origem, & destino, w: distancia);

        }
    } else
        cout << "Could not open the file\n";
}
```

```
void DataManip::readNodes(string filename) {

    ifstream in(s: filename);
    int id;
    double longitude, latitude;
    string line;

    getline(& in, & line);

    if (in.is_open()) {

        while(getline(& in, & line)){

            istringstream iss(str: line);

            iss >> id;
            iss.ignore();
            iss >> fixed >> setprecision(n: 15) >> longitude;
            iss.ignore();
            iss >> fixed >> setprecision(n: 15) >> latitude;

            graph_.addVertex(id, longitude, latitude);

        }
    } else
        cout << "Could not open the file\n";
}
```

GRAFO

Vertex:

- Unorder Map de Edges
- Set de Edges
- Unorder Map de Edges que entram
- Vetor children
- Double longitude
- Double latitude
- Int id
- Booleana visited
- Vertex path
- Double dist

```
class Vertex{  
  
    unordered_map<int ,Edge*> adj;  
    set<Edge*,CompareEdge> adjSet;  
    unordered_map<int ,Edge*> incoming;  
    vector<Vertex*> children;  
    double longitude;  
    double latitude;  
    int id;  
    bool visited = false;  
    Vertex *path = nullptr;  
    double dist = 0;  
}
```

GRAFO

Edge:

- Vértice Destino
- Vértice Origem
- Distância
- Edge reverse

```
class Edge{  
    Vertex* dest;  
    Vertex* orig;  
    double distance;  
    Edge *reverse = nullptr;  
};
```


FUNÇÕES IMPLEMENTADAS - 2.1

BACKTRACKING ALGORITHM

- O algoritmo pesquisa exaustivamente no espaço de soluções para encontrar a solução ótima para o TSP
- Boa solução para grafos pequenos
- Time Complexity: $O(n!)$

```
void DataManip::RecursiveBackTracking(vector<int>& path, double currCost, int currPos) {  
  
    for (auto pair : pair<const int, Edge*> : graph_.findVertex( id: currPos)->getAdj()) {  
        Edge *edge = pair.second;  
        int nextVertex = edge->getDest()->getId();  
        if (edge && !edge->getDest()->isVisited()) {  
  
            if (Bound( currCost: currCost + edge->getDistance())) {  
  
                path.push_back(nextVertex);  
  
                edge->getDest()->setVisited(true);  
  
                RecursiveBackTracking( &: path, currCost: currCost + edge->getDistance(), currPos: nextVertex);  
  
                edge->getDest()->setVisited(false);  
                path.pop_back();  
  
            }  
  
        }  
  
    }  
  
    if (Solution(path)) {  
        auto vertex1 : Vertex * = graph_.findVertex( id: path.back());  
        auto vertex2 : Vertex * = graph_.findVertex( id: path[0]);  
        double newCost = graph_.getDistance( v: vertex1, w: vertex2);  
        if (currCost + newCost < bestCost) {  
            bestCost = currCost + newCost;  
            bestPath = path;  
  
        }  
    }  
    return;  
}
```

FUNÇÕES IMPLEMENTADAS - 2.2

TRIANGULAR APPROXIMATION ALGORITHM

- Utiliza uma aproximação heurística baseada em triângulos para encontrar uma solução próxima da ótima para o TSP.
- Boa solução para grafos grandes
- Time Complexity: $O((V + E) \log V)$

```
double DataManip::TriangularApprox(vector<int> &path) {  
    //1. Executar o Prims para calcular o MST  
    graph_.MSTprims();  
  
    //2. Executar uma Visita pré ordem  
    vector<int> minPath;  
    graph_.preOrderVisit( id: 0, & minPath);  
  
    // 3. Construir o tour H a partir do caminho pré-ordem  
    vector<int> tour;  
    unordered_set<int> visited;  
    visited.insert( x: minPath[0]);  
  
    tour.push_back(minPath[0]);  
  
    for (int i = 1; i < minPath.size(); ++i) {  
        if (visited.find( x: minPath[i]) == visited.end()) {  
            tour.push_back(minPath[i]);  
            visited.insert( x: minPath[i]);  
        }  
    }  
  
    path = tour;  
  
    // 4. Calcular o custo do tour  
    double cost = CalculateTourCost( & path);  
  
    return cost;  
}
```


FUNÇÕES IMPLEMENTADAS - 2.3

NEAREST NEIGHBOUR APPROXIMATION

- Seleciona iterativamente o próximo vértice que é o mais próximo do vértice atual até que todos os vértices sejam visitados.
- Bom para todo o tipo de grafos onde uma solução rápida e simples é desejada, mesmo que não seja a ótima.
- Time Complexity: $O(V + E)$

```
double DataManip::NearestNeighborApprox(vector<int> &path) {
    unordered_set<int> not_visited;
    double cost = 0;

    // reset graph
    for (auto v : pair<const int, Vertex*> : graph_.getVertexSet()) {
        v.second->setVisited(false);
        not_visited.insert(x: v.second->getId());
    }

    int current_stop = 0;
    path.push_back(current_stop);
    graph_.findVertex(id: current_stop)->setVisited(true);
    not_visited.erase(x: current_stop);

    while (!not_visited.empty()) {
        bool found = false;
        int next_stop = -1;
        double min_distance = std::numeric_limits<double>::max();
        Vertex *currentVertex = graph_.findVertex(id: current_stop);

        for (auto pair : pair<const int, Edge*> : currentVertex->getAdj()) {
            Edge* edge = pair.second;
            Vertex *vertex = edge->getDest();
            double distance = edge->getDistance();

            if (distance == 0) {
                distance = calculateDistance(lat1: currentVertex->getLatitude(), lon1: currentVertex->getLongitude(),
            }

            if (!vertex->isVisited() && distance < min_distance) {
                next_stop = vertex->getId();
                min_distance = distance;
                found = true;
            }
        }
    }
}
```

FUNÇÕES IMPLEMENTADAS - 2.3

SIMULATED ANNEALING

- O algoritmo é submetido a perturbações, calculadas de forma probabilística, na busca de uma solução mais eficaz. Baseia-se na equação $P(x) = e^{\Delta/T}$. Em que delta é a distância entre os dois caminhos, e T é a temperatura atual. A temperatura decresce em cada iteração.
- Bom para grafos médios.
- Time Complexity: $O(V * V)$

```
double DataManip::simulatedAnnealing(std::vector<int>& path, double initialTemperature, double coolingRate) {
    int numVertices = graph_.getVertexSet().size();
    int maxIterations = 100;

    std::vector<int> currentSolution = path;
    bestPath = currentSolution;
    double currentCost = CalculateTourCost(&currentSolution);
    bestCost = currentCost;

    const double minTemperature = 1e-6;
    double temperature = initialTemperature;

    std::mt19937 randomGenerator(sd, std::chrono::system_clock::now().time_since_epoch().count());
    std::uniform_int_distribution<int> vertexDist(a: 1, b: numVertices - 1);
    std::uniform_real_distribution<double> probabilityDist(a: 0.0, b: 1.0);

    while (temperature > minTemperature) {
        for (int iteration = 0; iteration < maxIterations / (1 + log(x: 1 + temperature)); ++iteration) {
            std::vector<int> neighborSolution = currentSolution;
            int swapIndex1 = vertexDist(&randomGenerator);
            int swapIndex2 = vertexDist(&randomGenerator);
            while (swapIndex1 == swapIndex2) {
                swapIndex2 = vertexDist(&randomGenerator);
            }

            std::swap(&neighborSolution[swapIndex1], &neighborSolution[swapIndex2]);

            double neighborCost = CalculateTourCost(&neighborSolution);

            double acceptanceProbability = exp(x: (currentCost - neighborCost) / temperature);

            if (neighborCost < currentCost || probabilityDist(&randomGenerator) < acceptanceProbability) {
                currentSolution = neighborSolution;
                currentCost = neighborCost;
            }

            if (currentCost < bestCost) {
                bestPath = currentSolution;
                bestCost = currentCost;
            }
        }
    }
}
```

COMPARAÇÃO ENTRE ALGORITMOS

REAL WORLD GRAPH 1

```
Triangular Approximation for RW graph 1:  
The minimum cost to travel between all points is 1.14179e+06  
Execution time: 0.157945 seconds.
```

```
Nearest Neighbour for RW graph 1:  
The minimum cost to travel between all points is 1.00541e+06  
Execution time: 0.116897 seconds.
```

REAL WORLD GRAPH 3

```
Triangular Approximation for RW graph 3:  
The minimum cost to travel between all points is 3.19855e+06  
Execution time: 8.81438 seconds.
```

```
Nearest Neighbour for RW graph 3:  
The minimum cost to travel between all points is 7.27158e+06  
Execution time: 7.89213 seconds.
```

REAL WORLD GRAPH 2

```
Triangular Approximation for RW graph 2:  
The minimum cost to travel between all points is 2.05824e+06  
Execution time: 3.43534 seconds.
```

```
Nearest Neighbour for RW graph 2:  
The minimum cost to travel between all points is 4.00049e+06  
Execution time: 1.47235 seconds.
```

COMPARAÇÃO ENTRE ALGORITMOS

TOY GRAPH - TOURISM

```
Backtracking algorithm for Tourism:  
The minimum cost to travel between all points is 2600  
Execution time: 4.0578e-05 seconds.
```

```
Triangular Approximation for Tourism  
The minimum cost to travel between all points is 2600  
Execution time: 1.8787e-05 seconds.
```

TOY GRAPH - STADIUM

```
Backtracking algorithm for Stadium:  
The minimum cost to travel between all points is 341  
Execution time: 2.05189 seconds.
```

```
Triangular Approximation for Stadium:  
The minimum cost to travel between all points is 398.1  
Execution time: 3.4362e-05 seconds.
```

HEURISTICAS IMPLEMENTADAS

- Para as nossas heurísticas, embora tenhamos testado diversos algoritmos, decidimos apresentar apenas:
 - Nearest Neighbour: Uma solução mais eficiente em termos de tempo de execução, com custos que se aproximam do ótimo.
 - Simulated Annealing: Embora se ja uma solução menos eficiente em termos de tempo de execução, resulta em custos menores.

INTERFACE

Algorithm Menu

- 1 - Backtracking
- 2 - TriangularApprox
- 3 - Heuristics

- e - Exit

Choose the data set

- 1 - Toy-Graphs
- 2 - Extra_Fully_Connected_Graphs
- 3 - Real-world Graphs

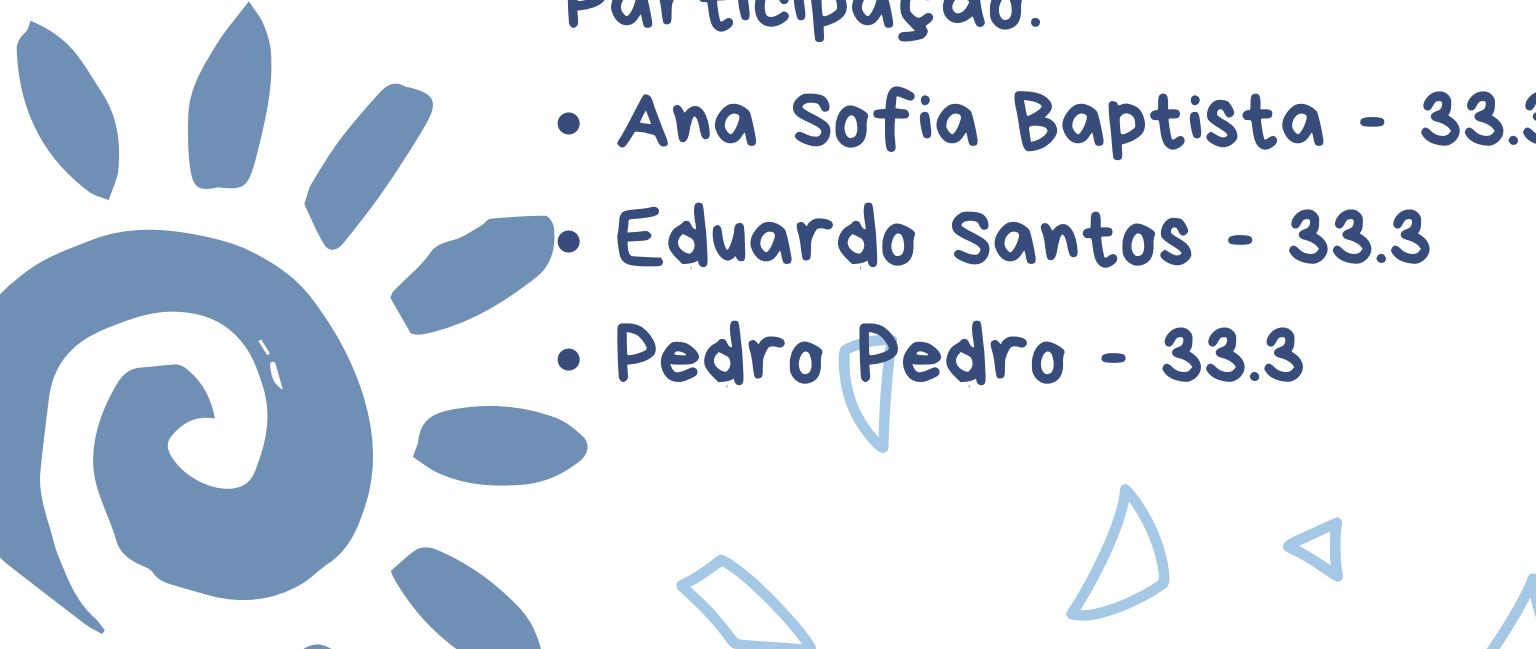

- b - back
- e - Exit



DIFICULDADES

Este projeto, foi no geral, um grande desafio, uma vez que exigiu um enorme esforço da nossa parte e forçou-nos a procurar as melhores soluções para o TSP. Concluímos que este problema é notoriamente difícil, e para encontrar a solução mais eficiente possível, tivemos que explorar uma ampla variedade de algoritmos, sendo que apenas selecionamos os mais eficientes e que deram melhores resultados.

Participação:

- Ana Sofia Baptista - 33.3
 - Eduardo Santos - 33.3
 - Pedro Pedro - 33.3
- 
- 



Obrigado