Routing Algorithm for Ocean Shipping and Urban Deliveries

Diogo Venade (up*202207805*)
Tiago Monteiro (up*202108391*)
Vasco Costa (up*202109923*)

Class Diagram

Menu

- Gere as várias opções do menu, chamando corretamente Application.

Application

- Classe principal para manipulação de dados.
- Contém os dados necessários para todas as operações.
- Leitura dos ficheiros CSV.

Graph

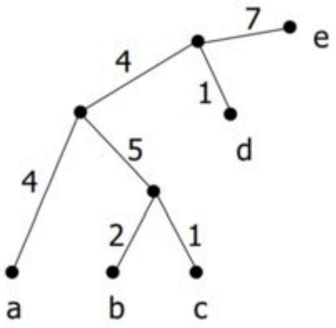
- Classe fornecida nas aulas modificada.
- Representa todos os dados fornecidos.

Leitura de Dados e Graph

Graph

Todos os dados são guardados no grafo, o qual tem uma matriz de distâncias, que é também preenchida durante a leitura dos ficheiros.

M	a	b	c	d	e
a	0	11	10	9	15
b	11	0	3	12	18
c	10	3	0	11	17
d	9	12	11	0	8
e	15	18	17	8	0




```
// distance matrix:
int n = graph.getVertexSet().size();
visited.resize(n, x:std::vector<bool>(n, value:false));
distanceMatrix.assign(n, val:std::vector<float>(n, value:std::numeric_limits<float>::infinity()));

for(auto vertex: graph.getVertexSet()) {
    distanceMatrix[vertex->getCode()][vertex->getCode()] = 0;
}
for(auto vertex: graph.getVertexSet()) {
    for (auto edge: vertex->getAdj()) {
        distanceMatrix[edge->getOrig()->getCode()][edge->getDest()->getCode()] = edge->getWeight();
        distanceMatrix[edge->getDest()->getCode()][edge->getOrig()->getCode()] = edge->getWeight();// reverse

    }
}
graph.setDistanceMatrix(distanceMatrix);
```

Complexidade: *O(n)*

Menu Principal

```
===============TSP Choose Graph to be processed================
1. Small Graph - tourism
2. Small Graph - stadiums
3. Small Graph - shipping
4. Large Graph 1
5. Large Graph 2
6. Large Graph 3
7. Medium Graph 25
8. Medium Graph  500
0. Exit
```

```
===============Choose Heuristic or Algorithm===============
1. Backtracking
2. 2-approximation
3. Nearest Neighbor
4. Christofides-Serdyukov
5. Real World
6. Exit
>
```

```
5. Real World
6. Exit
>5
 Please enter the source for the real world TSP:0
 TSP Tour Sequence: 0 -> 3 -> 2 -> 1 -> 4 -> 0
Total Distance: 2600
Execution time: 3 milliseconds
```

Funcionalidades Implementadas

T2.1 - Algoritmo de Backtracking

Este algoritmo encontra garantidamente a solução ótima para o problema do Travelling Salesman. No entanto, é extremamente ineficiente, sendo apenas útil para grafos muito pequenos - *os toy graphs*.

```cpp
if (count == n && distanceMatrix[currPos][0] > 0) {
    if (cost + distanceMatrix[currPos][0] < ans) {
        ans = cost + distanceMatrix[currPos][0];
        path.push_back(x: 0); // add start vertex to complete the tour
        path.pop_back(); // remove the start vertex after printing
    }
    return;
}
```

Caso base

```cpp
for (auto edge : graph.findVertex(currPos)->getAdj()) {
    int nextVertex = edge->getDest()->getCode();
    if (!edge->getDest()->isVisited() && distanceMatrix[currPos][nextVertex] > 0) {
        edge->getDest()->setVisited(true);
        path.push_back(nextVertex); // add next vertex to the current path
        tspBacktrackingAux(currPos: nextVertex, n, count + 1, cost + distanceMatrix[currPos][nextVertex], [&] ans, [&] pat
        path.pop_back(); // backtrack
        edge->getDest()->setVisited(false); // backtrack
    }
}
```

Backtracking

Complexidade: *O(n!)*

Funcionalidades Implementadas

T2.2 - Aproximação Triangular

Este algoritmo faz uso da desigualdade triangular: o caminho mais curto de um vértice i para um vértice j é diretamente de i para j, ao invés de recorrer a vértices intermédios. O custo resultante deste algoritmo nunca será superior ao dobro do custo ótimo.

```
primMST(); // Construct the MST using Prim's algorithm
std::vector<bool> visited( n: distanceMatrix.size(),  value: false);
preorderTraversal( root: 0,  [&] visited); // Perform preorder traversal of MST
```

Complexidade: *O(V+E)*

Funcionalidades Implementadas

T2.2/T2.3 - Algoritmo de Prim

Este algoritmo foi utilizado para descobrir a MST dos grafos, que é útil para calcular algoritmos de aproximação. Sabendo que a MST tem o custo mínimo para ligar todos os vértices do grafo, pode-se concluir que esse valor funciona como um "lowest bound" e, portanto, o valor da shortest trip será sempre igual ou superior a esta.

```
primMST(); // Construct the MST using Prim's algorithm
std::vector<bool> visited( n:distanceMatrix.size(), value:false);
preorderTraversal( root:0, [&]visited); // Perform preorder traversal of MST
```

Complexidade: *O(E log V)*

Funcionalidades Implementadas

T2.3 - Outras Heurísticas

Implementamos 2 algoritmos heurísticos neste ponto: o *Nearest Neighbor* e o algoritmo de Christofides (com o *Blossom Algorithm)*. O primeiro é mais rápido, mas menos exato. O segundo, vice-versa.

```
 Total weight of MST: 231.2
TSP Tour Sequence: 0 -> 1 -> 2 -> 10 -> 5 -> 4 -> 8 -> 7 -> 6 -> 9 -> 3 -> 0
Total Cost: 398.1
Execution time: 5 milliseconds
```

Aproximação Triangular - stadiums.csv

```
 Total distance of TSP tour (Nearest Neighbor): 407.4 units
Execution time: 1 milliseconds
```

Nearest Neighbor - **O(V²)**

```
 Total weight of MST: 231.2
Total Cost: 391.4
Execution time: 3 milliseconds
```

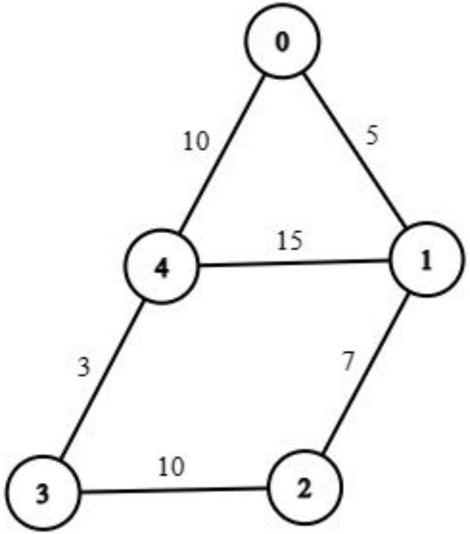
Christofides - **O(EV²)**

Funcionalidades Implementadas

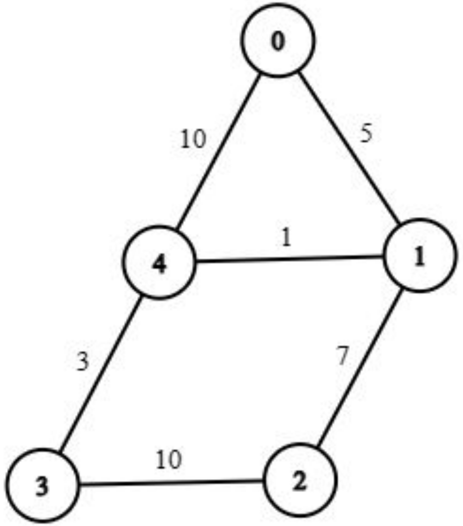
T2.4 - TSP *Real World*

Neste ponto, foi implementada uma versão modificada do *Nearest Neighbors*, que não calcula a distância de *Haversine* entre vértices sem ligação direta. A principal desvantagem desta estratégia reside no facto de, se chegar a um *dead-end*, o que pode acontecer em grafos não completos, não irá encontrar um ciclo hamiltoniano, quando este pode na verdade existir.

Complexidade: $O(V^2)$



Algorithm works



Reaches dead-end

Reflexões

T2.4 - TSP *Real World*

Qualquer estratégia que envolvesse *backtracking* não seria exequível, dada a sua complexidade. Por exemplo, chegamos a pensar em usar *backtracking* para confirmar que existe pelo menos um ciclo hamiltoniano, mas concluímos que demoraria demasiado tempo.

Existem teoremas que fornecem condições suficientes mas não necessárias para a existência de um ciclo hamiltoniano, como o teorema de Ore. De pouco serviram, no entanto, pois um grafo podia não obedecer ao teorema e ainda assim ser hamiltoniano - é o caso do *shipping*. Num grafo não completo, detetar estes ciclos é particularmente difícil, devido à possível existência de *dead ends*.

Determinar se um grafo é hamiltoniano é um problema NP-completo, pelo que não é conhecida uma solução polinomial para o mesmo - crucial para os *large graphs*.

Reflexões

T2.4 - TSP *Real World*

Outra estratégia que tentamos implementar foi:

- calcular os *shortest paths* entre todos os vértices usando o algoritmo de Floyd-*Warshall*;
- usar o algoritmo dos *nearest neighbors* num novo grafo modificado com as distâncias calculadas no ponto anterior, colmatando a falta de *edges* em grafos não completos;
- adicionar os *paths* reconstruídos do *Floyd-Warshall* ao caminho encontrado.

Esta estratégia resultaria, em teoria, num caminho curto que visitaria todas as cidades, começando e terminando no mesmo ponto. No entanto, iria repetir cidades, desrespeitando uma das premissas do problema, e a nossa implementação sofria de *segmentation faults*.

Resultados - Large Graph 2

```
Total weight of MST: 2.42943e+06
Total Cost: 3.49567e+06
Execution time: 1379156 milliseconds
================Choose Heuristic or Algorithm================
1. Backtracking
2. 2-approximation
3. Nearest Neighbor
4. Christofides-Serdyukov
```

Christofides

```
Total distance of TSP tour (Nearest Neighbor): 5.64938e+06 units
Execution time: 486 milliseconds
================Choose Heuristic or Algorithm================
1. Backtracking
2. 2-approximation
3. Nearest Neighbor
4. Christofides-Serdyukov
5. Real World
```

Nearest Neighbor

```
> 2
Total weight of MST: 2.42943e+06
Total Cost: 3.51934e+06
Execution time: 1618 milliseconds
```

Triangular Approximation

Dificuldades Encontradas

Eficiência na leitura de dados - Estrutura do grafo talvez não tenha sido a mais indicada
Windows vs Linux - resultados diferentes
Solução para o T2.4

Participação

Diogo Venade: 33%
Tiago Monteiro: 33%
Vasco Costa: 33%