2° Projeto DA

Routing Algorithm for Ocean Shipping and Urban Deliveries

Faculdade Engenharia Universidade Porto

Grupo G12_5

Projeto - Gestão de um sistema de ligações de entregas via marinha e terrestre

Elementos do grupo:

- Rafael Teixeira (up202108831@up.pt)
- João Costa (up202108714@up.pt)
- João Sobral (up202108736@up.pt)

Classes

Menu
(Creates the interface for the user)

FileManager
(Reads the documents and creates the structure of the graph)

Algorithms
(Contains the algorithms used to solve the problems)

Leitura do dataset

Foram nos fornecidas tabelas que continham informações pertinentes:

- Identificação das localidades e distâncias entre elas

Para tratarmos de fazer a gestão destas ligações fizemos o seguinte:

- Lemos as tabelas linha a linha ignorando o título e fomos inserindo os dados respetivos como vértices (id de partida e id de destino) e arestas do grafo (distância entre local de partida e destino)

Criação dos grafos

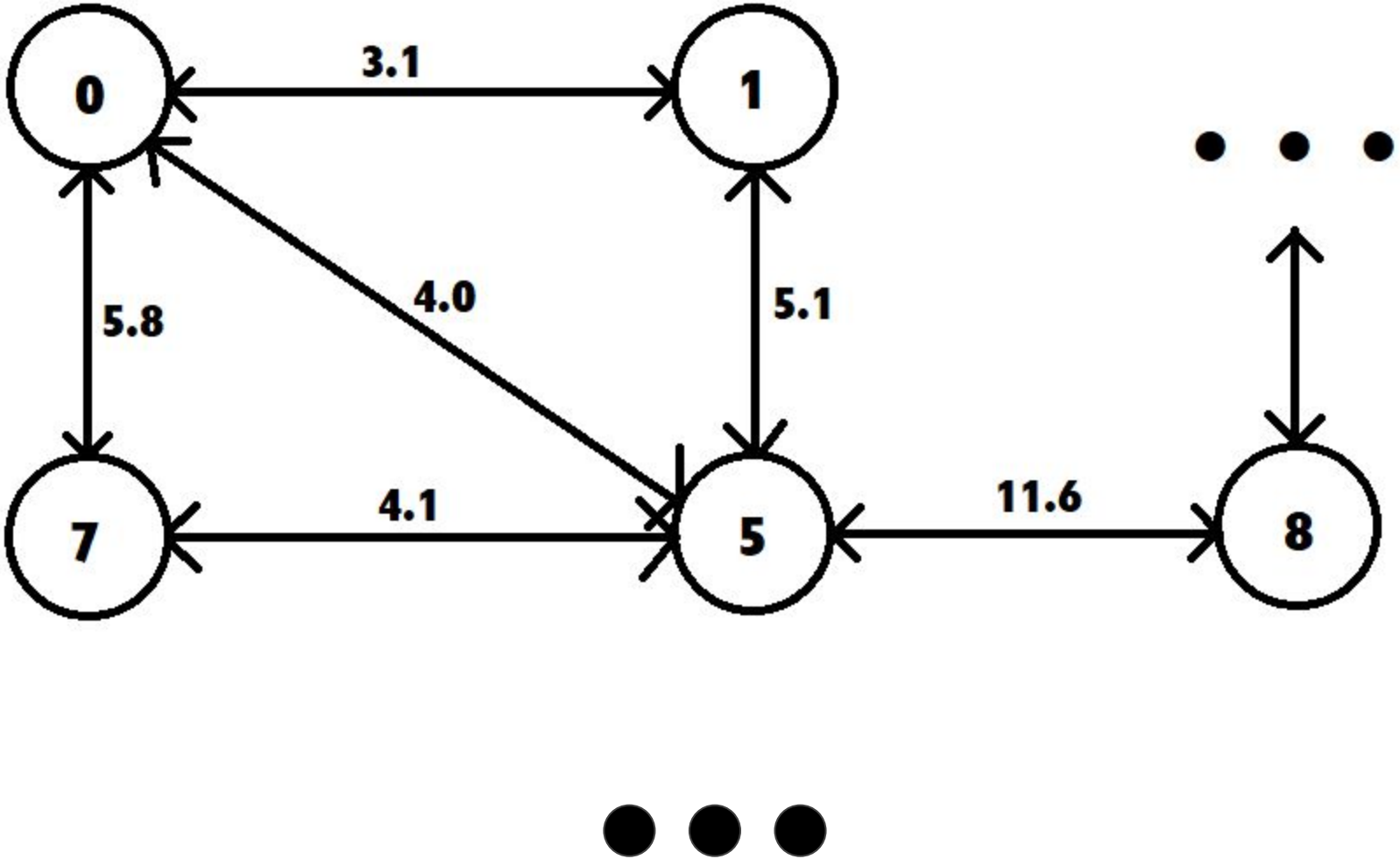
```
Graph FileManager::readExtraFullyConnectedGraph(string file, Graph g) {
    string source, dest, weight, skip;
    int idSource, idDest;
    ifstream filename;
    filename.open( s: "../Project2Graphs/Extra_Fully_Connected_Graphs/" + file);
    if (!filename.is_open()) cout << "File creation failed" << endl;
    else {
        if (!filename.eof()) { getline( &: filename, &: skip, delim: ',');}
        if (!filename.eof()) { getline( &: filename, &: skip, delim: ',');}
        if (!filename.eof()) { getline( &: filename, &: skip,  delim: '\n');}

        while (!filename.eof()) {
            getline( &: filename, &: source, delim: ',');
            getline( &: filename, &: dest, delim: ',');
            getline( &: filename, &: weight, delim: '\n');

            idSource = stoi( str: source);
            idDest = stoi( str: dest);

            if (g.findVertex( id: idSource) == nullptr){
                g.addVertex( id: idSource);
            }
            if (g.findVertex( id: idDest) == nullptr){
                g.addVertex( id: idDest);
            }
            g.addBidirectionalEdge( sourc: idSource, dest: idDest, w: stod( str: weight));
        }
    }
    return g;
}
```

Grafo para gestão das ligações das localidades

Para criação dos grafos tivemos o seguinte raciocínio:

- Cada node será identificado por um id, correspondente a cada localidade
- O que ligará os nodes serão as distâncias entre localidades
- Para construção deste grafo, consideramos a utilização de edges bidirecionais

No slide posterior é possível visualizar um esboço desta implementação do grafo


0
1
3.1
5.8
4.0
5.1
7
4.1
5
11.6
8

Funcionalidades implementadas

Funcionalidades implementadas:

1. Problema TSP usando a estratégia de Backtracking
2. Problema TSP tendo em conta a Heurística de Aproximação Triangular
3. Problema TSP com a nossa própria Heurística

Funcionalidades implementadas (Cont.)

1. Para resolução deste problema, tivemos que recorrer à estratégia de Backtracking. Para atingir esse objetivo, implementamos as seguintes funções: solve(), backtrack() e calculateEdgeCost(), explicadas a seguir:

solve(): Esta função é o ponto de entrada para resolver o TSP. Aqui, é criado um vetor que guarda os caminhos, marcado o vértice inicial (vértice 0) como visitado e chamada a função backtrack() para procurar o melhor caminho.

backtrack(): Esta função recursiva tem como objetivo encontrar o melhor caminho e recebe dois parâmetros: o caminho atual e o custo atual do caminho. A função explora todas as combinações possíveis de vértices, tentando cada vértice não visitado como o próximo vértice no caminho. Esta função implementa também a estratégia de bounding ao fazer a comparação do custo atual do caminho que estamos a percorrer com o melhor guardado até ao momento, portanto se o nosso caminho atual já tiver um maior custo que o melhor guardado, este irá ser ignorado.

Caso Base: Se o tamanho do caminho atual for igual ao número total de vértices no grafo, significa que todos os vértices foram visitados. A função então calcula o custo do caminho atual adicionando o custo da aresta que conecta o último vértice ao vértice inicial. Se o custo atual for menor do que o melhor custo encontrado até agora, as variáveis bestPath e bestCost são atualizadas.

Passo Recursivo: Para cada vértice não visitado, a função marca-o como visitado, adiciona-o ao caminho e chama recursivamente a função backtrack com o estado atualizado. Após a chamada recursiva, o vértice é removido do caminho e marcado como não visitado para retroceder e explorar outras possibilidades.

calculateEdgeCost(): Esta função calcula o custo de uma aresta entre dois vértices. Recebendo os ID's do vértice de origem e destino como parâmetros, esta função itera pela lista de vértices adjacentes ao vértice de origem para encontrar a aresta correspondente. Se encontrada, o peso da aresta é retornado, caso contrário, é retornado um valor infinito.

Funcionalidades implementadas (Cont.)

1. Como este algoritmo tem um custo temporal demasiado pesado, apesar de ainda recorrer à estratégia de bounding, torna-se impossível corrê-lo para os grafos maiores como os do Real-World.

   O algoritmo explora todas as combinações possíveis dos vértices, começando pelo vértice 0, para encontrar o melhor caminho com o custo mínimo -> O ((N + 1)!)

Funcionalidades implementadas (Cont.)

2. Para resolução deste problema, tivemos que recorrer à estratégia de Aproximação Triangular. Para atingir esse objetivo, implementamos as seguintes funções: TriangularApproximation() e PathDfs(), explicadas a seguir:

TriangularApproximation(): Esta função é o ponto de entrada para resolver o TSP. Aqui, é criada uma priority queue que é usada para construir a minimum spanning tree que é guardada no graph original selecionando as edges que a constituem através da função setSelected(), após obter a mst é chamada a função PathDfs(). O vetor path é imprimido elemento por elemento, e o custo é calculado através do path.

PathDfs(): Esta função executa uma busca em profundidade (DFS) a partir do vértice de origem fornecido. Ele explora recursivamente todos os vértices adjacentes do vértice de origem, seguindo arestas selecionadas (arestas que são marcadas como "selecionadas"). Os vértices visitados são adicionados ao vetor caminho, que acompanha a ordem de passagem.

Caso Base: A recursão continua até que não haja mais vértices adjacentes selecionados para explorar.

Passo Recursivo: Se o vértice adjacente for selecionado (ou seja, a aresta que o conecta ao vértice de origem é marcada como selecionada), ele adiciona o ID do vértice de destino ao vetor de caminho e chama recursivamente PathDfs() com o vértice adjacente como a nova origem.

O algoritmo explora uma combinação, começando pelo vértice que o utilizador insere, para encontrar o melhor caminho com o custo mínimo -> O (V + E)

Funcionalidades implementadas (Cont.)

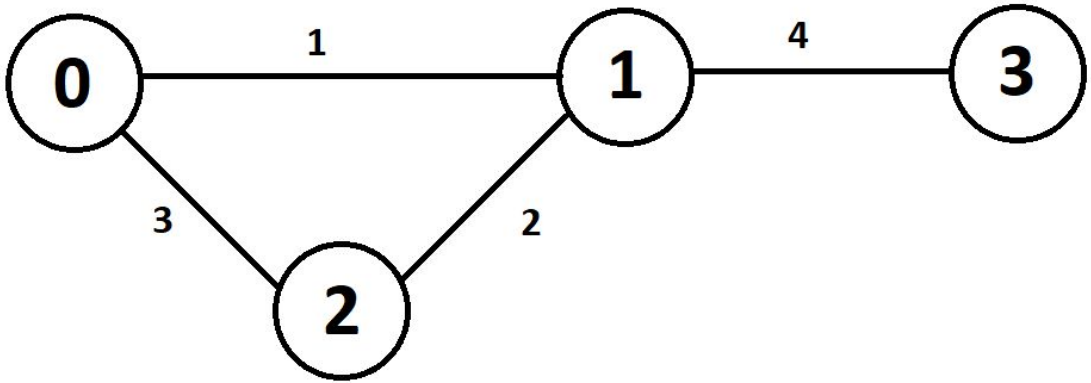
3. Para resolução deste problema, tivemos que recorrer à estratégia de Nearest Neighbor. Para atingir esse objetivo, implementamos as seguintes funções: NearestNeighbor(), explicadas a seguir:

NearestNeighbor(): Esta função é o ponto de entrada para resolver o TSP. Aqui, é criada uma priority queue que é usada para saber qual é o vértice mais próximo do atual que ainda não foi visitado adicionando-o ao vector path. Faz-se isto sucessivamente até que todos os vértices estejam visitados e após obter a MST é chamada a função PathDfs(). O vetor path é imprimido elemento por elemento e o custo é calculado através do path.

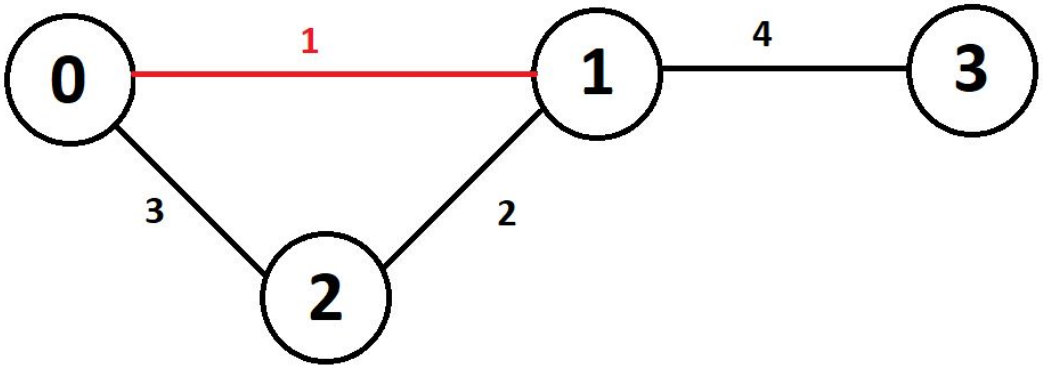
Este algoritmos só funciona se o graph a explorar for completo, ou seja, todos os vértices têm arestas para todos.

O algoritmo explora uma combinação, começando pelo vértice que o utilizador insere, para encontrar o melhor caminho com o custo mínimo -> O (V * E)

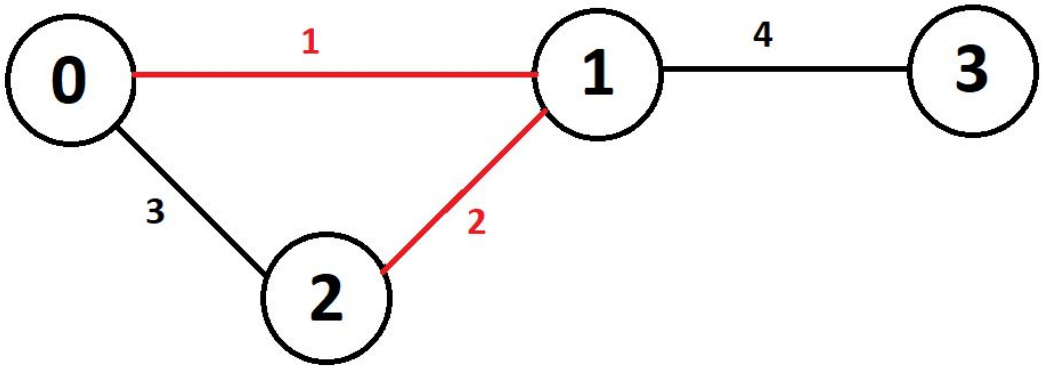
2. NearestNeighbor



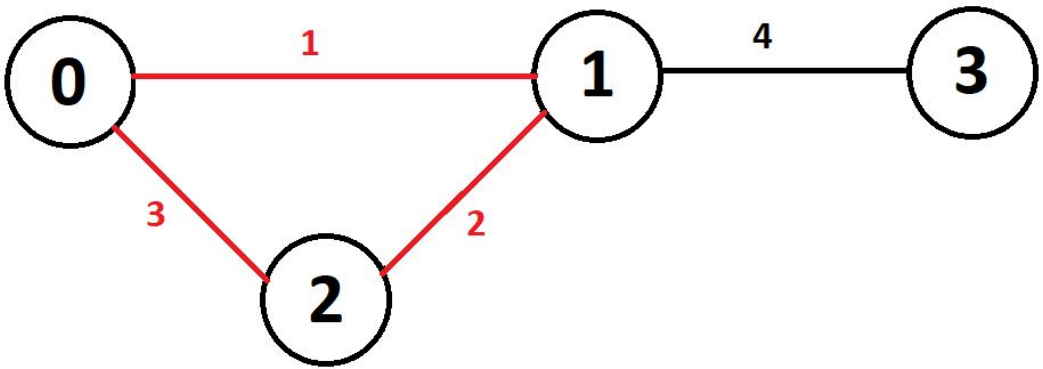
2. NearestNeighbor



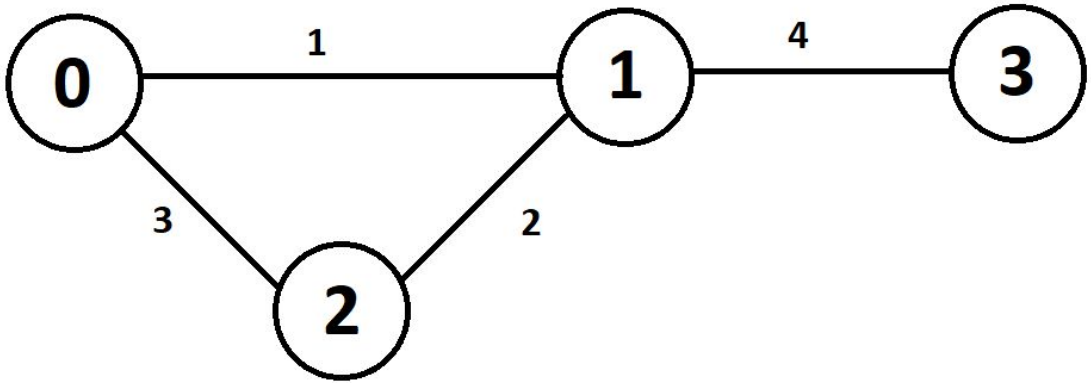
2. NearestNeighbor



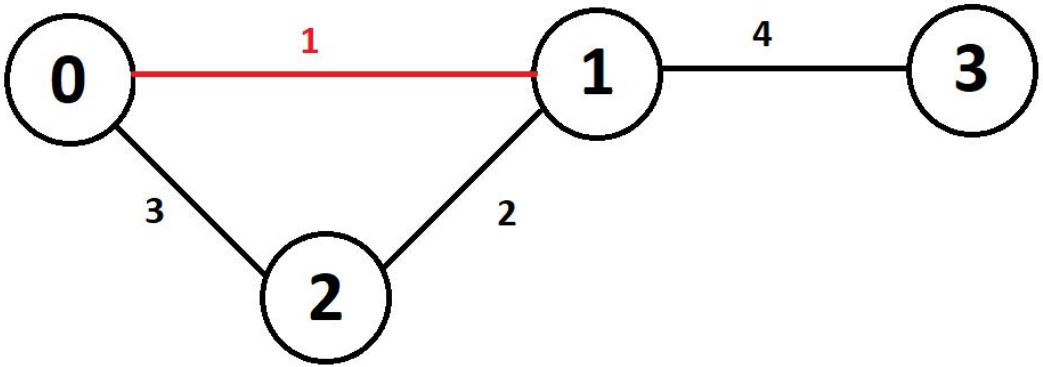
2. NearestNeighbor



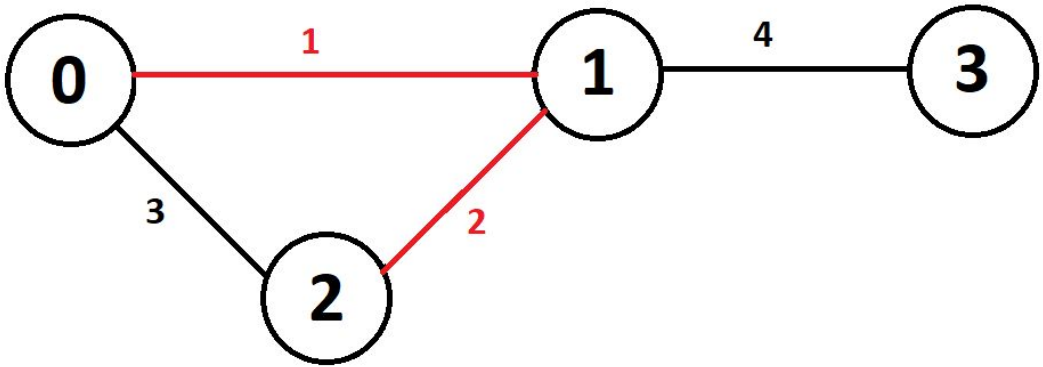
3. TriangularApproximation - Exemplo 1



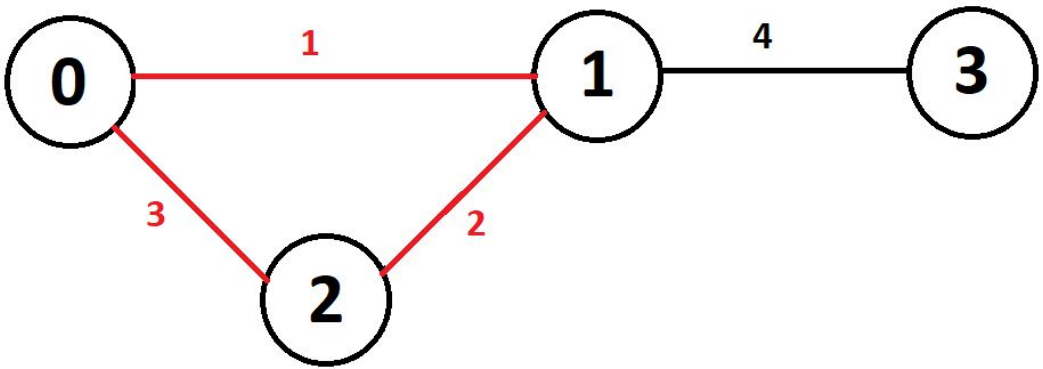
3. TriangularApproximation - Exemplo 1



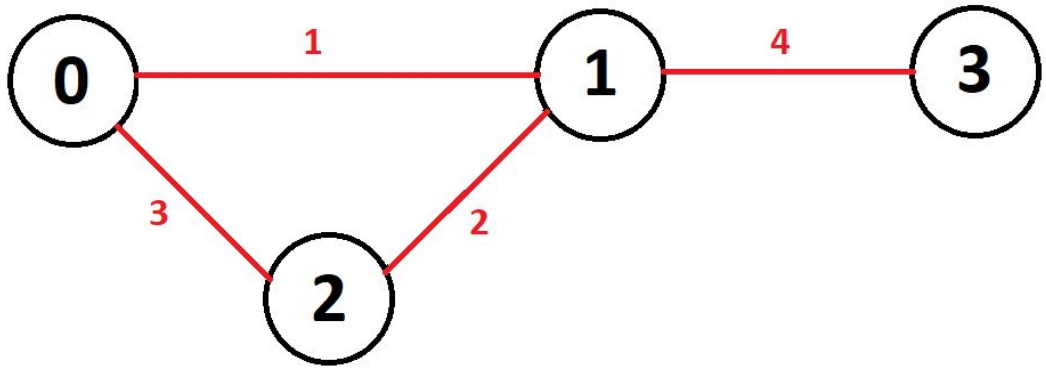
3. TriangularApproximation - Exemplo 1



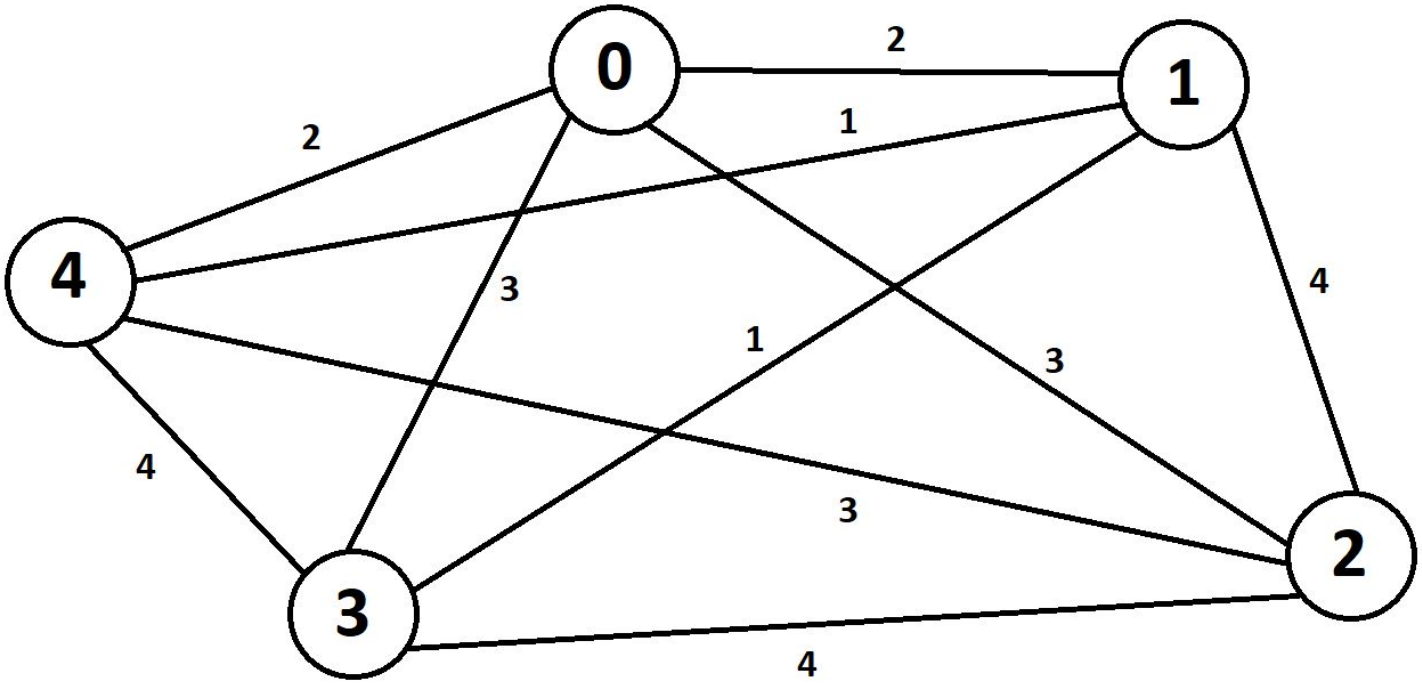
3. TriangularApproximation - Exemplo 1



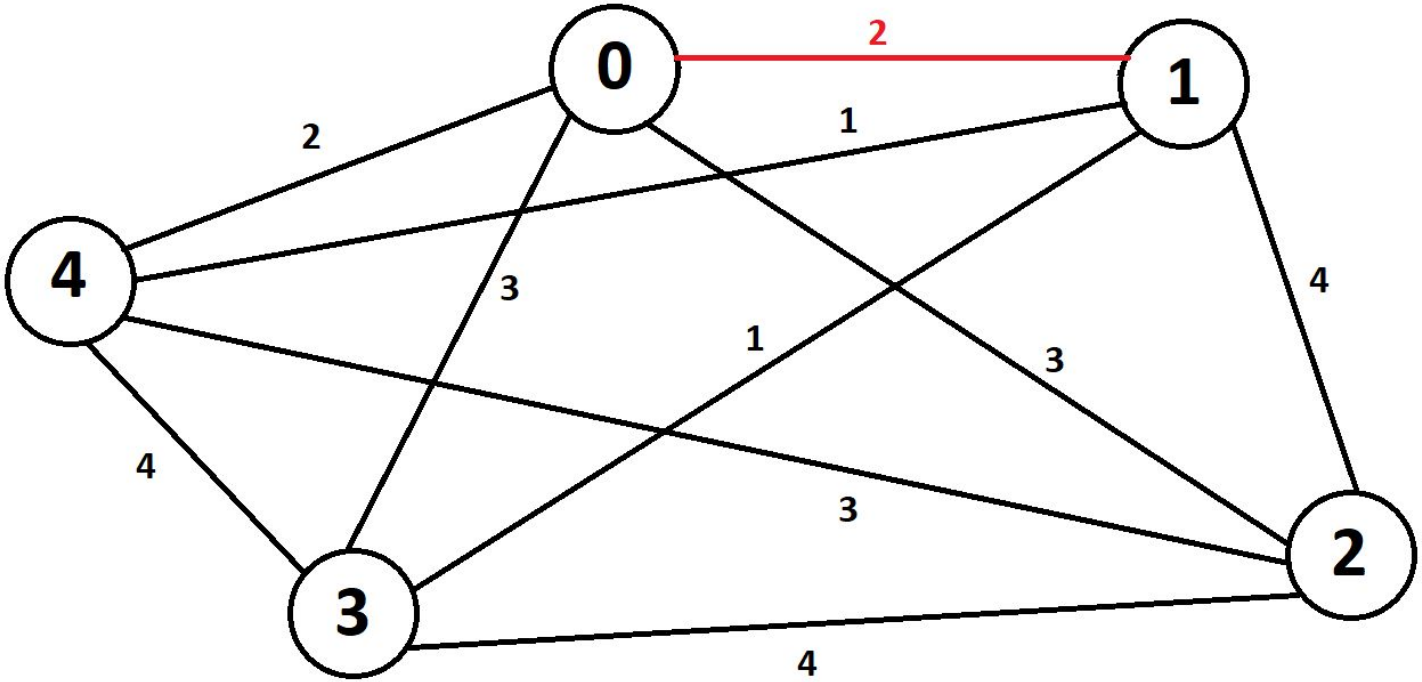
3. TriangularApproximation - Exemplo 1



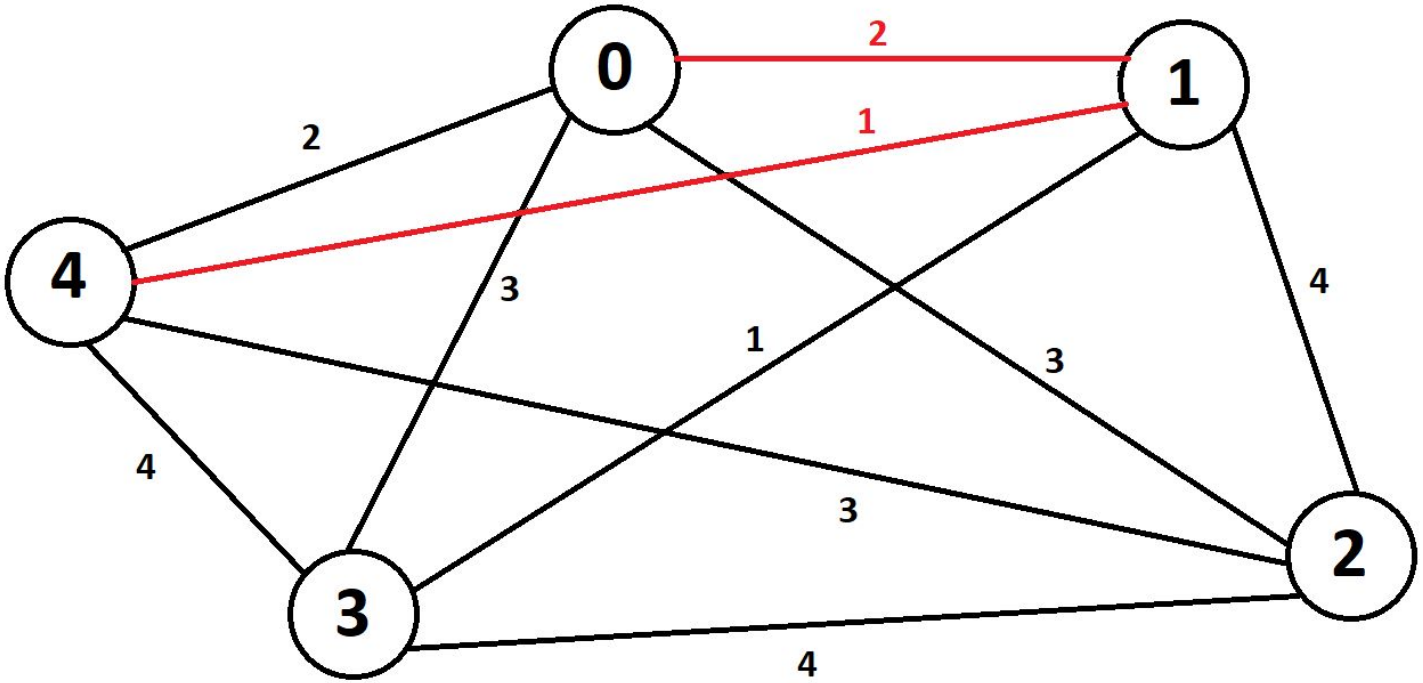
3. TriangularApproximation - Exemplo 2



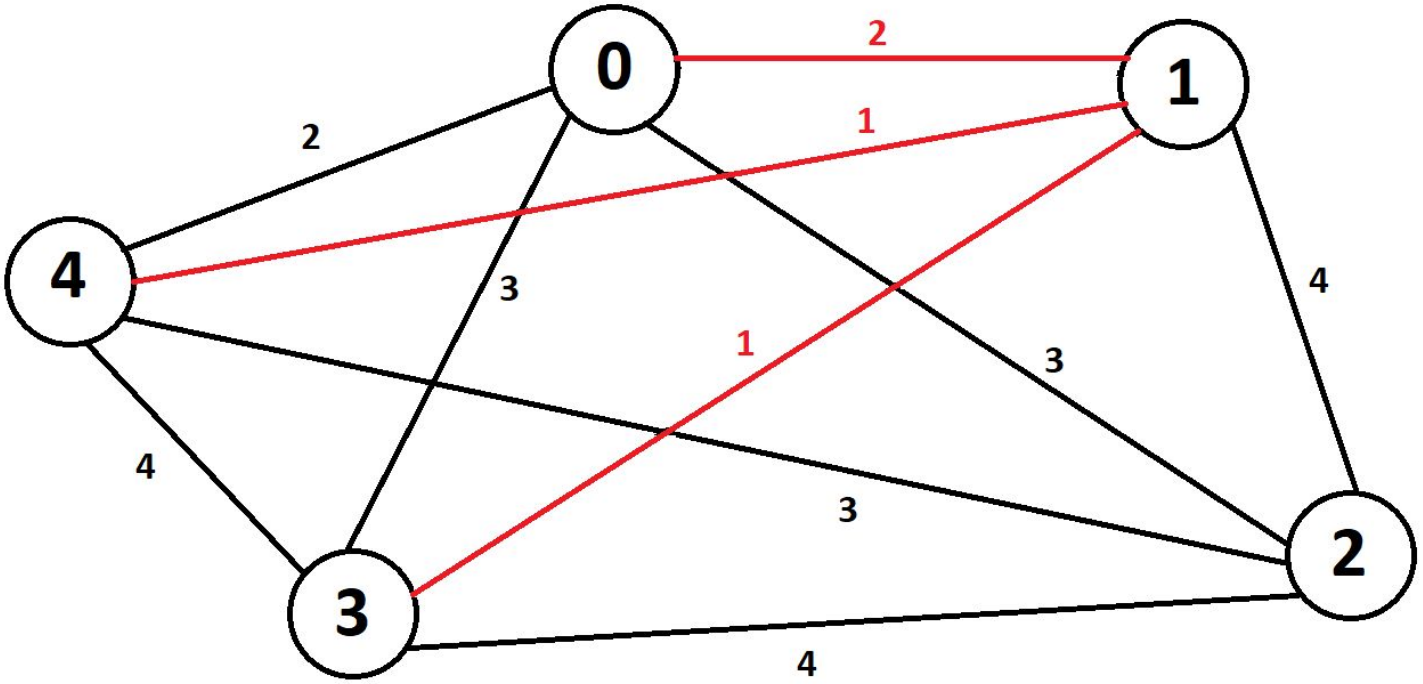
3. TriangularApproximation - Exemplo 2



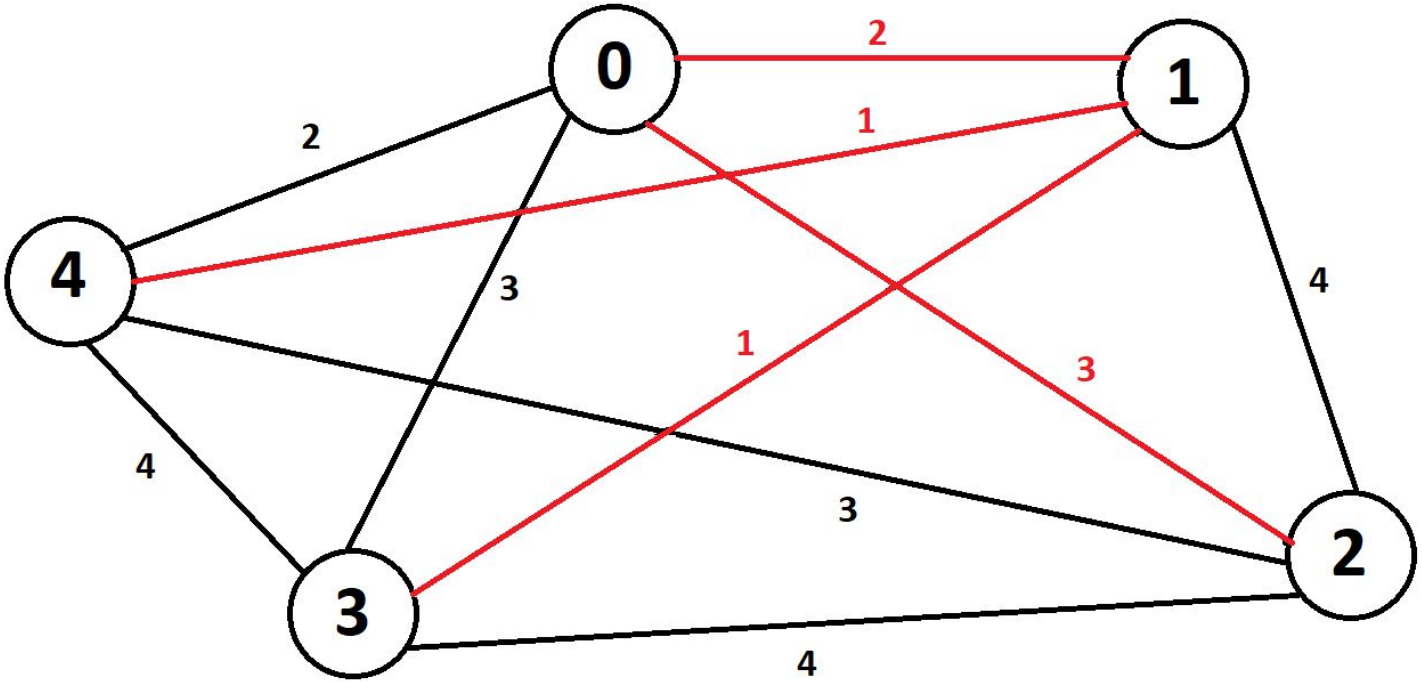
3. TriangularApproximation - Exemplo 2



3. TriangularApproximation - Exemplo 2



3. TriangularApproximation - Exemplo 2



Discussão de observações

1. Relativamente à feature 1 implementada, esta deve ser apenas utilizada em grafos de menor dimensão já que a sua complexidade temporal é fatorial. Com isto, mesmo com um processador muito rápido, demorariam muitos anos para que o algoritmo terminasse no caso de grafos como os fornecidos pela pasta “Real-world Graphs”. Porém, em destaque dos outros algoritmos, apesar de consumir muito tempo, este algoritmo acaba por fornecer sempre a melhor solução.

2. Relativamente à feature 2 implementada, de todos os algoritmos que construímos, esta recorre ao que menos tempo consome para alcançar um resultado. Embora não garanta a melhor solução para o problema do ciclo hamiltoniano, a função TriangularAproximation() fornece um ciclo de uma boa aproximação da solução ótima. Com isto, quando o objetivo é alcançar uma solução aceitável de forma rápida, este algoritmo é o preferível.

Discussão de observações (cont.)

3. Relativamente à feature 3 implementada, esta deve ser apenas utilizada em grafos completos, isto é, quando todos os vértices estão conectados entre si por arestas. A função NearestNeighbor() baseia-se no facto de que, num grafo completo, cada vértice tem vértices adjacentes para todos os outros vértices não visitados. Esta função recorre a uma priority queue para selecionar o destino vizinho mais próximo e assim sempre até que todos os vértices estejam visitados, porém, se estivermos a analisar um grafo incompleto, a função pode deparar-se com erros ou fornecer resultados incorretos. Apesar da complexidade deste algoritmo, muitas das vezes consegue encontrar um caminho mais rapidamente com uma solução suficientemente boa.

Interface

Ao dar início ao programa, o utilizador irá se deparar com uma interface que lhe irá pedir qual o tipo de grafos que quer analisar. Após isso, um menu com várias opções aparece, dando a chance ao utilizador de optar pela que mais lhe agrada. O número 0 termina o programa e a partir do número 1 são disponibilizadas todas as features implementadas

Neste menu, é possível fazer a opção exaustiva de todas as funcionalidades presentes, pois o programa só fecha quando o utilizador faz a inserção do número 0 e sempre que não seja introduzido um número como input, uma função auxiliar irá “obrigar” o utilizador a inserir um valor válido

```
================== Routing Algorithm for Ocean Shipping and Urban Deliveries Management ==================

1. Toy Graphs
2. Real World Graphs
3. Extra Fully Connected Graphs
1
1

Type of graph
1. Shipping
2. Stadiums
3. Tourism
3
3

================== Routing Algorithm for Ocean Shipping and Urban Deliveries Management ==================

1. Backtracking Algorithm
2. Triangular Approximation Heuristic
3. Other Heuristics
4. Choose Other Graph

0. Exit and Credits
```

Observações

Tivemos dificuldades:

- Em criar alguns métodos com complexidade temporal reduzida

Contribuição de cada elemento do grupo para o projeto:

- João Sobral: 1 / 3
- João Costa: 1 / 3
- Rafael Teixeira: 1 / 3