ROUTING ALGORITHM FOR OCEAN SHIPPING AND URBAN DELIVERIES(DA) PROGRAMMING PROJECT II

Group Members:

Gabriela Dias Salazar Neto Silva – up202004443

Diogo Alexandre da Silva Santos – up202009291

23/24

OBJETIVOS DO PROJETO

- Implementar uma abordagem exaustiva para resolver o Problema do Caixeiro Viajante (TSP)
- Desenvolver e analisar heurísticas para encontrar soluções aproximadas
- Trabalhar em grupos para aprimorar competências interpessoais e de gestão de projetos

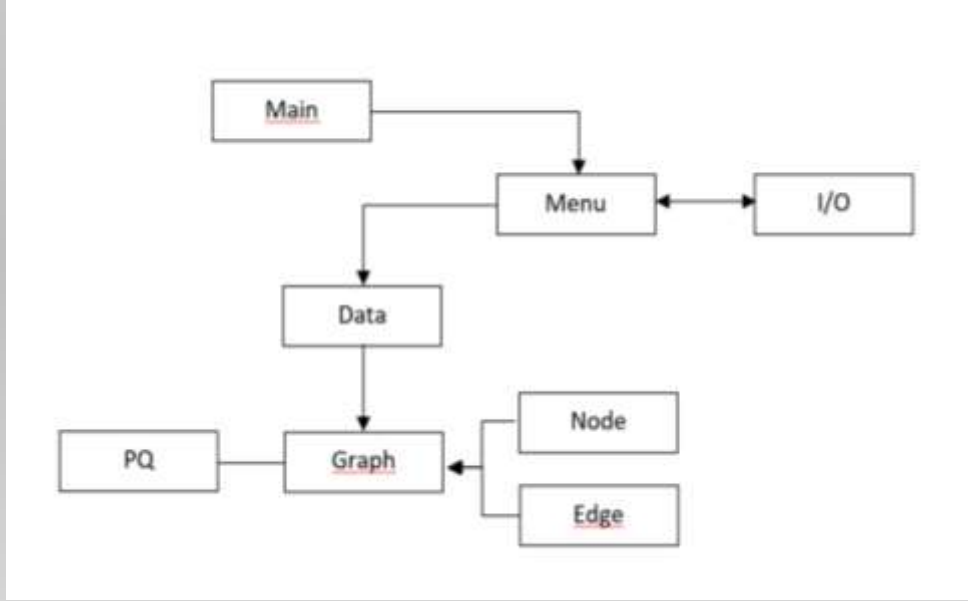
TECNOLOGIAS UTILIZADAS

- Linguagem de programação: C++
- Bibliotecas: Standard C++ Libraries
- Ferramentas: CMake, Make, Git

DESCRIÇÃO DO PROBLEMA

- **Problema**: Encontrar rotas ótimas para veículos em cenários de entrega urbana e transporte marítimo
- **TSP**: tenta determinar a menor rota para percorrer uma série de cidades, retornando à cidade de origem.
- **Métodos**:
  - Algoritmos de *backtracking* para grafos pequenos
  - Algoritmos de aproximação para grafos grandes

CLASS DIAGRAM



- Menu: Responsável pela interface do utilizador, lidando com entradas e saídas e conectando-se aos dados do projeto. Apresenta menus interativos para o usuário escolher ações.
- Data: Responsável por ler diferentes tipos de grafos e armazenar os dados, além de fornecer acesso às informações do grafo. Conecta o Menu à classe Graph.
- Graph: Configura os atributos do grafo e implementa funções complexas relacionadas a grafos, mantendo uma coleção de nós e arestas, e métodos para manipulá-los.
- Node: Representa um nó no grafo, com identificador único, coordenadas e lista de arestas de saída para nós adjacentes.
- Edge: Representa uma aresta no grafo, com nó de origem, nó de destino, distância entre os nós e uma representação em string da distância.
- MutablePriorityQueue: Implementa uma fila de prioridade mutável, usada no algoritmo de Prim para manter uma lista de nós com distâncias até a árvore geradora mínima atual.

LEITURA DO DATASET

- **Responsabilidades da classe Data**:
  - Leitura de Dados: Implementa métodos para ler os dados dos grafos a partir de arquivos CSV.
  - Armazenamento de Grafos: Utiliza a classe Graph para armazenar os grafos lidos a partir dos arquivos.
  - Validação de Dados: Garante que os dados lidos sejam válidos e estejam corretamente formatados antes de armazená-los.
  - Fornecimento de Dados: Fornece métodos para aceder aos grafos armazenados pela classe, permitindo que outras partes do sistema os utilizem para planejar roteiros turísticos.
- **Métodos principais:**
  - readRealGraphs(): Leitura dos dados de grafos reais
  - readToyGraphs(): Leitura dos dados de grafos de brinquedo
  - readExtraGraphs(): Leitura dos dados de grafos extras

INTERFACE COM UTILIZADOR

- **Responsabilidades da classe Menu**:
  - Interface de Utilizador: Oferece métodos para exibir mensagens, solicitar entrada do utilizador e apresentar opções de menu.
  - Interação com a Classe Data: Utiliza os métodos da classe Data para aceder aos dados dos grafos e fornecer esses dados ao utilizador conforme necessário.
  - Validação de Entrada: Garante que as entradas do utilizador sejam válidas e respondidas adequadamente pelo sistema.
  - Gerenciamento de Fluxo: Controla o fluxo da interação do utilizador com o sistema, garantindo uma experiência suave e intuitiva.
- **Métodos auxiliares:**
  - printTitle(): Impressão do título do projeto
  - getUserInput(): Validação da entrada do usuário
  - clearScreen(): Limpeza da tela

CLASSE MUTABLEPRIORITYQUEUE

- **Responsabilidades da classe MutablePriorityQueue**:
  - Inserção e Extração de Elementos: Fornece métodos para inserir novos elementos na fila de prioridade e extrair o elemento mínimo.
  - Diminuição de Chave: Permite que a chave de um elemento na fila de prioridade seja diminuída, mantendo a integridade da estrutura.
  - Gerenciamento de Índices: Mantém um índice associado a cada elemento na fila de prioridade para facilitar a diminuição de chave e outras operações eficientes.
- **Métodos principais:**
  - insert(): Inserção de elementos na fila de prioridade
  - extractMin(): Extração do elemento mínimo da fila de prioridade
  - decreaseKey(): Diminuição da chave de um elemento na fila de prioridade

FUNCIONALIDADES PRINCIPAIS

Graph Information

- Exibe e manipula os dados dos grafos, incluindo nós e arestas.
- Tempo de Complexidade: O(V + E), onde V é o número total de nós e E é o número total de arestas no grafo. Isso ocorre principalmente durante a leitura dos arquivos de entrada.

Backtracking Algorithm - T2.1

- Algoritmo de volta para encontrar o caminho mais curto entre todos os nós, percorrendo todas as possíveis combinações.
- Tempo de Complexidade: Depende do tamanho do grafo e da complexidade do algoritmo exato, mas geralmente é exponencial.

Minimum Spanning Tree (better approximation) - T2.2

- Árvore geradora mínima usando uma aproximação mais rápida, mas menos precisa, adequada para grafos maiores.
- Tempo de Complexidade: O(E log V), semelhante ao algoritmo de Prim.

IMPLEMENTED FUNCTIONALITIES AND ASSOCIATED ALGORITHMS

Minimum Spanning Tree (worse approximation but faster for larger graphs) - T2.2

- Árvore geradora mínima usando uma aproximação mais rápida, mas menos precisa, adequada para grafos maiores.
- Tempo de Complexidade: O(E log V), semelhante ao algoritmo de Prim.

Minimum Distance Recursive Traversal - T2.3

- Trabalha recursivamente para encontrar o caminho mais curto entre todos os nós.
- Tempo de Complexidade: O(E), onde E é o número de arestas totais do grafo

Nearest Neighbor Algorithm - T2.4

- Algoritmo do vizinho mais próximo para encontrar um caminho aproximado mais curto a partir de um determinado nó.
- Tempo de Complexidade: O(V^2), onde V é o número de nós no grafo. Isso ocorre principalmente devido ao processo de escolha do próximo nó mais próximo.

IMPLEMENTED FUNCTIONALITIES AND ASSOCIATED ALGORITHMS

```cpp
void Graph::backtrackingApproach(double &shortestDistance, int *shortestCycle, bool distanceType) {
    for (auto& pair : nodes) {
        pair.second->visited = false;
    }
    int cycle[nodes.size()];
    cycle[0] = 0;
    nodes.find(0)->second->visited = true;
    backtrackingApproachRec(0, shortestDistance, 1, (int) nodes.size(), cycle, shortestCycle, distanceType);
}

void Graph::backtrackingApproachRec(double distance, double &shortestDistance, int currentIndex, int n, int cycle[], int shortestCycle[], bool distanceType) {
    if (currentIndex == n) {
        distance += distanceBetweenNodes(cycle[currentIndex - 1], cycle[0], distanceType);
        if (distance < shortestDistance) {
            shortestDistance = distance;
            for (unsigned i = 0; i < n; i++) {
                shortestCycle[i] = cycle[i];
            }
        }
    } else {
        for (int i = 0; i < n; i++) {
            if (!(nodes.find(i)->second->visited)) {
                if (distance + distanceBetweenNodes(cycle[currentIndex - 1], i, distanceType) < shortestDistance) {
                    cycle[currentIndex] = i;
                    nodes.find(i)->second->visited = true;
                    backtrackingApproachRec(distance + distanceBetweenNodes(cycle[currentIndex - 1], i, distanceType), shortestDistance, currentIndex + 1, n, cy
                    nodes.find(i)->second->visited = false;
                }
            }
        }
    }
}
```

backTracking approach

```cpp
vector<int> Graph::primMST(bool distanceType, double &totalDistance) {
    MutablePriorityQueue<Node> queue;
    vector<int> prim;
    vector<int> preOrder;
    for (auto& pair : nodes) {
        pair.second->visited = false;
        pair.second->distance = numeric_limits<double>::infinity();
        queue.insert(pair.second);
    }
    nodes.find(0)->second->distance = 0;
    nodes.find(0)->second->path = nullptr;
    nodes.find(0)->second->visited = true;
    queue.decreaseKey(nodes.find(0)->second);
    while (!queue.empty()) {
        Node* origin = queue.extractMin();
        prim.push_back(origin->id);
        for (auto edge : origin->edgesOut) {
            if (!(nodes.find(edge->dest)->second->visited) && edge->distance < nodes.find(edge->dest)->second->
                nodes.find(edge->dest)->second->distance = edge->distance;
                nodes.find(edge->dest)->second->path = edge;
                queue.decreaseKey(nodes.find(edge->dest)->second);
            }
        }
        nodes.find(origin->id)->second->visited = true;
    }
    for (auto& pair : nodes){
        pair.second->visited = false;
    }
    totalDistance = orderMST(0, prim, preOrder, distanceType);
    return preOrder;
}
```

primMST

PRINCIPAIS DIFICULDADES

Maintainability and Extensibility:

Uma das principais dificuldades enfrentadas durante o desenvolvimento do projeto foi a manutenção e extensibilidade do código. À medida que o projeto cresce, torna-se desafiador manter e estender a base de código. Garantir que o código esteja bem organizado, modular e siga as melhores práticas pode facilitar os esforços futuros de desenvolvimento e manutenção. Além disso, acomodar novos recursos ou requisitos enquanto mantém a compatibilidade com versões anteriores requer um planeamento e design cuidadosos. A estruturação do código de forma a permitir fácil manutenção e adição de novos recursos foi uma preocupação constante ao longo do desenvolvimento do projeto.

Participation of each group member

Gabriela Dias Salazar Neto Silva – 40%
Diogo Alexandre da Silva Santos – 60%