Ferramenta de Planeamento de Rotas Individuais

Desenho de Algoritmos 2024/2025

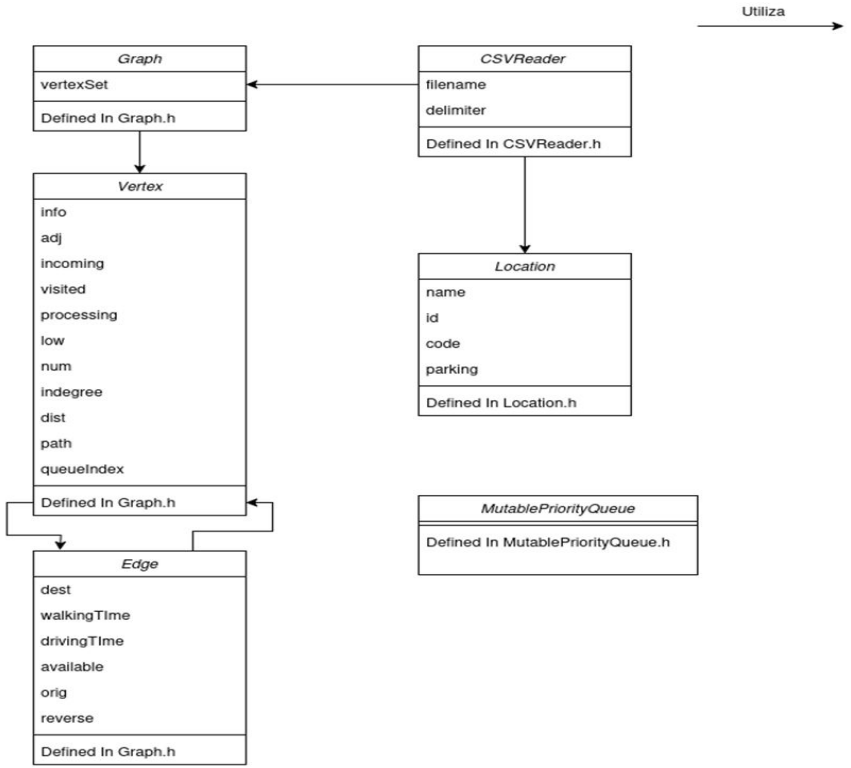
Turma 4, Grupo 4

- Diogo Soares Rocha – up201606166
- Francisco João Gonçalves Calado Araújo - up201806326

Diagrama de Classes

Para além das classes fornecidas, criámos também:

- CSVReader
- Location



Leitura dos Dados

O processo de leitura dos dados fornecidos está centralizado na classe ***CSVReader***, localizada nos ficheiros ***CSVReader.h*** e ***CSVReader.cpp***.

```
public:
    explicit CSVReader(std::string filename, char delimiter = ',');
    void readLocationData(Graph<Location> *cityGraph);
    void readDistanceData(Graph<Location> *cityGraph);
};
```

- O construtor recebe o caminho do ficheiro e o delimitador.
- O método ***readLocationData()***
  - Divide os campos: nome, ID, código, e valor de estacionamento.
  - Remove vírgulas e converte ID e parking para inteiros.
  - Adiciona um vértice ao grafo.
- O método ***readDistanceData()***
  - Lê os códigos das localizações e ambos os tempos de deslocamento.
  - Converte os tempos para inteiros e marca com “X” os tempos de condução inválidos.
  - Procura os vértice correspondentes no gráfico.
  - Cria arestas nas duas direcções.

- Ambos os métodos estão equipados com a capacidade de capturar qualquer tipo de erro de conversão, marcando as arestas como inválidas e mostrando a mensagem de erro adequada.

Grafo Utilizado Para Representar Os Dados

- O **grafo** é **dirigido** e **ponderado**.
- Cada **vértice** representa uma localização da cidade:
  - Contém um objecto de tipo **Location**, que guarda da respectiva cidade, o nome, ID, código e se é estacionamento.
- Cada **aresta** representa um caminho direto entre duas localizações:
  - Contém dois pesos, o tempo a conduzir e a caminhar.

Locations reports:

Location ID	Location Name	Code	Parking
1	Trindade	TRI	No
2	Campo Alegre	CAL	Yes
3	Bolhão	BOL	Yes
4	Aliados	ALI	No
5	Sé	SE	No
6	Ribeira	RIB	Yes
7	Foz	FOZ	Yes
8	Clerigos	CLE	No

Edges reports:

From	To	Walking Time	Driving Time
Trindade	Campo Alegre	20	10
Trindade	Bolhão	15	X
Campo Alegre	Trindade	20	10
Campo Alegre	Aliados	25	8
Campo Alegre	Bolhão	8	5
Bolhão	Trindade	15	X
Bolhão	Campo Alegre	8	5
Bolhão	Sé	10	12

Descrição das Funcionalidades Implementadas e Algoritmos Associados

O sistema foi concebido para permitir o planeamento de rotas eficientes numa cidade representada por um grafo, tendo em consideração diferentes modos de deslocação e restrições do utilizador.

Neste capítulo serão apresentadas:

- Todas as funcionalidades implementadas no âmbito do projeto.
- O algoritmo de **Dijkstra** associado à execução destas mesmas.

Algoritmo de Dijkstra

O algoritmo de Dijkstra é a base de todas as rotas calculadas neste projeto, servindo tanto para planeamento a pé como de carro. A implementação encontra-se no ficheiro ***Dijkstra.cpp*** e foi adaptada para funcionar sobre um ***Graph<Location>***, com suporte para arestas com dois tipos de peso: ***walking*** ou ***driving.***

1. **Validação inicial**
   - **1.1.** Se o vértice de origem ou de destino for inválido, ou se forem iguais, o algoritmo termina imediatamente.

2. **Inicialização do grafo**
   - **2.1.** É criada priority queue.
   - **2.2.** Todos os vértices são marcados como não visitados.
   - **2.3.** As distâncias iniciais marcadas como infinito.
   - **2.4.** O caminho é limpo.
   - **2.5.** A distância da origem é marcada como 0 e esta é inserida na fila.

3. **Enquanto a fila não estiver vazia**
   - **3.1.** É extraído o vértice com menor distância.
   - **3.2.** Se for destino, o algoritmo termina.
   - **3.3.** Se a distância for superior a um eventual ***currentBest***, o vértice é ignorado.

```
std::vector<Vertex<Location>*> path;
if (!src || !dest) {
    std::cout << "Invalid source or destination!" << std::endl;
    return path;
}

if (src == dest) {
    std::cout << "Source and destination are the same!" << std::endl;
    path.push_back(src);
    return path;
}
```

```
MutablePriorityQueue<Vertex<Location>> pq;

// **Initialize distances and paths**
for (Vertex<Location>* location : city->getVertexSet()) {
    location->setVisited(false);
    location->setPath(nullptr);
    location->setDist(std::numeric_limits<int>::max());
    location->setQueueIndex( value: 0);
}

src->setDist(0);
pq.insert(src);
```

```
while (!pq.empty()) {
    Vertex<Location>* current = pq.extractMin();
    current->setVisited(true);

    if (current == dest) {
        break;  // Stop early if destination is reached
    }

    if (current->getDist() > currentBest) {
        continue;
    }
```

Algoritmo de Dijkstra

4. **Relaxamento das arestas**
   4.1. Para cada aresta:
      4.1.1. Ignora arestas indisponíveis ou com tempo inválido.
      4.1.2. Se o destino estiver visitado ou indisponível, também é ignorado
      4.1.3. Caso contrário, calcula-se a nova distância e, se for melhor, atualiza-se os atributos ***dist***, ***path*** e **posição na fila**.

5. **Reconstrução do caminho**
   5.1. Após atingir o destino, o caminho é reconstruído percorrendo o atributo **path** de cada vértice, do fim para o início.

Complexidade:

A implementação, com ***MutablePriorityQueue***, assegura uma complexidade de

$$O((V + E) * \log V)$$

- Cada vértice é inserido e extraído uma única vez da fila de prioridade: **O(V * log V)**.
- Cada aresta pode provocar uma operação de ***decreaseKey***, que tem custo logarítmico: **O(E * log V)**.
- No total, o algoritmo realiza **V** extrações e até **E** relaxamentos, ambos com custo **log V** associado.

```
for (Edge<Location>* street : current->getAdj()) {
    if (street->getTime(isWalking) == -1 || !street->isAvailable()) {
        continue;
    }
    Vertex<Location>* next = street->getDest();

    if (next->isVisited() || !next->getInfo().isAvailable()) {
        continue;  // Skip visited nodes
    }

    double newDist = current->getDist() + street->getTime(isWalking);

    if (newDist < next->getDist() && newDist <= currentBest) {
        next->setDist(newDist);
        next->setPath(street);  // Store the edge, not the vertex
        if (next->getQueueIndex() == 0) {  // If not in the queue, insert it
            pq.insert(next);
        } else {  // If already in the queue, update it
            pq.decreaseKey(next);
        }
    }
}
```

```
// **If no path was found**
if (dest->getDist() == std::numeric_limits<int>::max()) {
    return path;
}

// **Reconstruct path using edges**

Vertex<Location>* v = dest;

while (v != src) {
    Edge<Location>* edge = v->getPath();
    if (!edge) {
        std::cerr << "Error: Path reconstruction failed!" << std::endl;
        return path;
    }
    path.push_back(v);
    v = edge->getOrig();  // Move to previous vertex
}
path.push_back(src);
std::reverse( first: path.begin(),  last: path.end());

return path;
```

Independent Route Planning

Esta funcionalidade permite encontrar a melhor rota de condução entre dois pontos da cidade, considerando os tempos de deslocação de carro definidos no grafo.

A lógica principal encontra-se no módulo ***IndependentRoutePlanning***, mais concretamente na função ***planDrivingRoute()***, que realiza os seguintes passos:

1. **Seleção de vértices**
   1.1. O utilizador escolhe as localizações de origem e destino com ***Utils::chooseStartAndEndingCities()***. Caso as localizações sejam inválidas, o processo termina com uma mensagem de erro.
2. **Cálculo da rota mais eficiente**
   2.1. A função ***findBestDrivingRoute()*** é chamada com os vértices selecionados. Esta utiliza o algoritmo de Dijkstra para obter o caminho com menor tempo de condução, recorrendo ao tempo de carro como peso das arestas. O resultado é impresso com ***Utils::printRoute()***.
3. **Cálculo de rota alternativa**
   3.1. O sistema tenta encontrar um segundo caminho viável, utilizando novamente ***Dijkstra***, mas ignorando arestas já utilizadas (a lógica dessa parte está encapsulada). Caso exista, é apresentada ao utilizador como rota alternativa.

```
Choose the starting city (Id number): 3
Choose the destination (Id number): 8
Best Driving Route: Bolhão(3) -> Campo Alegre(2) -> Aliados(4) -> Clerigos(8)    Best Time: 19
Best Alternative Driving Route: Bolhão(3) -> Foz(7) -> Clerigos(8)   Best Time: 34
```

Restricted Route Planning

Esta funcionalidade permite calcular uma rota de condução personalizada, respeitando restrições impostas pelo utilizador, como localizações a evitar ou segmentos a não utilizar, e incluindo pontos de paragem obrigatória.

A lógica principal encontra-se no módulo ***RestrictedRoutePlanning***, mais concretamente na função ***planRestrictedRoute()***, que realiza os seguintes passos:

1. **Seleção da origem e destino**
   1.1. O utilizador escolhe os vértices inicial e final com ***chooseStartAndEndingCities()***. Caso algum seja inválido, o processo é interrompido.

2. **Definição de restrições**
   2.1. Com ***chooseNodesAndSegmentsToAvoid()***, o utilizador pode marcar localizações e segmentos do grafo que deverão ser ignorados na rota.

3. **Escolha de pontos intermédios**
   3.1. É possível indicar múltiplas localizações por onde a rota deverá obrigatoriamente passar, usando ***chooseMiddlePoint()***.

Restricted Route Planning

4. **Planeamento da rota segmentada**
   - **4.1.** A função ***findMultiStopRoute()*** liga os vértices sucessivamente:

     origem → paragem 1 → paragem 2 → … → destino.

     - **4.1.1.** Para cada sub-trecho, é chamado o algoritmo de ***Dijkstra***.
     - **4.1.2.** As distâncias parciais são acumuladas, e os caminhos são concatenados num vetor ***totalPath***.

5. **Apresentação da rota**
   - **5.1.** A rota completa é apresentada com ***Utils::printRoute()***:

```
Choose the starting city (Id number): 5
Choose the destination (Id number): 4
Avoid nodes (Id number):
2
Avoid Segments (Id number, Id number):
4,7
Choose the locations to make stops, in order, during path (Id number):

Best Driving Route: Sé(5) -> Bolhão(3) -> Foz(7) -> Clerigos(8) -> Aliados(4)    Best Time: 52
```

```
Choose the starting city (Id number): 5
Choose the destination (Id number): 4
Avoid nodes (Id number):

Avoid Segments (Id number, Id number):

Choose the locations to make stops, in order, during path (Id number):
7
Best Driving Route: Sé(5) -> Bolhão(3) -> Foz(7) -> Clerigos(8) -> Aliados(4)    Best Time: 52
```

Environmentally-Friendly Route Planning & Approximate Solution

Este módulo permite encontrar rotas entre dois pontos da cidade que combinam dois modos de deslocação: condução até um parque de estacionamento e caminhada até ao destino final. Também está preparado para encontrar soluções aproximadas quando não existe um caminho direto.

A lógica principal encontra-se no módulo ***HybridRoutes***, mais concretamente na função **planDrivingRoute()**, que realiza os seguintes passos:

1. **Escolha de localizações**
   - **1.1.** O utilizador seleciona a origem e o destino com ***chooseStartAndEndingCities()***.
   - **1.2.** São verificadas todas as restrições impostas pela natureza do problema, e são apresentadas mensagens de erro consoante a restrição detectada.
   - **1.3.** O tempo máximo a caminhar é pedido ao utilizador e de seguida validado.

Environmentally-Friendly Route Planning & Approximate Solution

2. Escolha da melhor rota

Após guardar todos os parques de estacionamento num vetor, a função ***findBestEnvFriendlyRoute()*** e esta encapsula a lógica da procura:

- **2.1.** Percorre todos os vértices com estacionamento (candidatos a ponto de transição).
- **2.2.** Para cada um:
  - **2.2.1.** Calcula rota de carro até esse ponto (***Dijkstra* com *isWalking* = *false***);
  - **2.2.2.** Calcula rota a pé do ponto até ao destino (***Dijkstra* com *isWalking* = *true***);
  - **2.2.3.** Se ambos os caminhos forem válidos, regista a solução.
- **2.3.** Seleciona a combinação com menor tempo total.
- **2.4.** Se não existir solução completa, procura rotas aproximadas: caminho de carro até um ponto qualquer que permita continuar a pé.

3. Apresentação dos resultados

Dependendo do resultado, é atribuído a uma variável um dos seguintes valores:

A função ***printResults()*** formata e escreve os resultados baseada nessa variável.

```
#define PATH_FOUND 0
#define APPROXIMATED_PATH_FOUND 1
#define NO_PATH_FOUND_TO_PARKING 2
#define NO_PATH_FOUND_TO_DESTINATION 3
```

Environmentally-Friendly Route Planning & Approximate Solution

Complexidade:

Neste caso a função vai ter uma complexidade maior. O tempo de execução é proporcional ao **número de parques** testados, vezes duas execuções do ***Dijkstra*** (feitas em separado):

O(P * ((V + E) * log V)).

```
Choose the starting city (Id number): 8
Choose the destination (Id number): 5
Choose the maximum walking time: 5
Avoid nodes (Id number):

Avoid Segments (Id number, Id number):

Message: There are no possible paths to reach the destination walking within the max walking time of 5 min!
Suggested Approximated Solutions:
Suggestion 1:
DrivingRoute: Clerigos(8) -> Aliados(4) -> Campo Alegre(2) -> Bolhão(3)  Best Time: 19
ParkingNode: Bolhão(3)
WalkingRoute: Bolhão(3) -> Sé(5)     Best Time: 10
TotalTime: 29

Suggestion 2:
DrivingRoute: Clerigos(8) -> Aliados(4) -> Ribeira(6)    Best Time: 21
ParkingNode: Ribeira(6)
WalkingRoute: Ribeira(6) -> Sé(5)    Best Time: 10
TotalTime: 31
```

```
Choose the starting city (Id number): 8
Choose the destination (Id number): 5
Choose the maximum walking time: 18
Avoid nodes (Id number):

Avoid Segments (Id number, Id number):

DrivingRoute: Clerigos(8) -> Aliados(4) -> Campo Alegre(2) -> Bolhão(3)  Best Time: 19
ParkingNode: Bolhão(3)
WalkingRoute: Bolhão(3) -> Sé(5)     Best Time: 10
TotalTime: 29
```

Interface com o Utilizador

O sistema foi desenvolvido com uma interface de linha de comandos. O utilizador interage com o programa através de um menu textual, escolhendo opções numéricas para aceder às diferentes funcionalidades.

```
Choose the Dataset:
1 - Small (SmallLocations.csv / SmallDistances.csv)
2 - Large (Locations.csv / Distances.csv)
Enter your choice: 1
```

O primeiro *input* pedido ao utilizador é para escolher o *dataset* que este quer usar para experimentar os algoritmos desenvolvidos. Tem duas escolhas, o *dataset* pequeno e o completo, fornecidos no início do projeto.

```
Route Planning Analysis Tool Menu:
1. Print All Locations and Edges
2. Independent Route Planning
3. Restricted Route Planning
4. Environmentally-Friendly Route Planning
0. Exit
Enter your choice: 1
```

A função ***menu()*** controla o ciclo principal de interação. Apresenta ao utilizador um conjunto de opções:

Todas as entradas numéricas são validadas através da função ***getValidatedInt()*** definida em ***Utils.cpp***. Esta garante que o programa não avança com valores inválidos ou mal formatados, pedindo ao utilizador que volte a introduzir o valor corretamente.

```cpp
int getValidatedInt(const std::string& prompt) {
    int value;
    while (true) {
        std::cout << prompt;
        std::cin >> value;

        if (std::cin.fail()) {
            std::cin.clear();
            std::cin.ignore( n: std::numeric_limits<std::streamsize>::max(),  delim: '\n');
            std::cout << "Invalid input. Please enter a valid number." << std::endl;
        } else {
            std::cin.ignore( n: std::numeric_limits<std::streamsize>::max(),  delim: '\n');
            return value;
        }
    }
}
```

Funcionalidades em Destaque

As funcionalidades que nós escolhemos destacar foram a ***Environmentally-Friendly Route Planning*** e ***Approximate Solution.***

Eram o maior desafio do projeto, com bastantes restrições. Estas funcionalidades obrigaram-nos a pensar de forma criativa e a aplicar o ***Dijkstra*** de forma adaptada.

Participação no Trabalho

- Diogo Soares Rocha - **40%**
- Francisco João Gonçalves Calado Araújo - **60%**