Projeto de Concepção e Análise de Algoritmos
Saúde Emérita: Transporte de saúde para idosos (tema 7)

Turma 4, Grupo 5

Elgner Eristido Gomes Ramos	ei12173@fe.up.pt
Luís Pedro Rodrigues de Morais	ei08082@fe.up.pt

1. Descrição do problema

A cadeia de luxo de lares de idosos Emeritus procura desenvolver um sistema para gestão de uma frota de viaturas e profissionais subcontratados do ramo da saúde (como por exemplo, ambulâncias) que no dia-a-dia deslocarão os idosos entre os lares e os centros de saúde ou hospitais. As necessidades por detrás destas deslocações podem ter um cariz de urgência (por exemplo no caso de risco de vida) ou não (consultas, realização de exames, etc).

Neste trabalho pretende-se implementar o sistema para a identificação do percurso ótimo a ser realizado pelo transporte de idosos, entre os lares que tenham idosos a necessitar transporte e os centros de saúdes (se não for uma urgências) e hospitais (se for uma urgência). Inicialmente, considere apenas uma viatura a executar os transportes. A cada requisição, será necessário avaliar se o transporte poderá esperar por mais doentes no mesmo lar, ou precisa ser realizado imediatamente, em função da sua urgência. Posteriormente, considere a presença de mais do que uma viatura, e eventualmente viaturas dedicadas a transportes específicos, como por exemplo, transportes urgentes, não urgentes e de recolha dos idosos até aos respectivos lares (não considerar para o casos inicial).

Algumas vezes, obras nas vias públicas podem fazer com que certas zonas tornem-se inacessíveis, inviabilizando o acesso ao destino de alguns doentes. Avalie a conectividade do grafo, a fim de identificar lares, centros de saúde e hospitais com pouca acessibilidade.

Considere a possibilidade de utilizar mapas reais, extraídos do OpenStreetMaps (www.openstreetmap.org) e coordenadas geográficas dos centros de saúde e hospitais, na implementação do sistema.

2. Formalização do problema

2.1. Dados de entrada

type - tipo de urgência (urgente/não urgente)
N_i - sequência de ambulâncias disponíveis para a realização do serviço

- lotação - lotação da viatura de transporte

$G_i = (V_i, E_i)$ - grafo dirigido pesado que representa o local onde vamos implementar a solução:

- V - Vértices (representam ligações entre as ruas e pontos de interesse(Hospital/Centro de saúde)
  - id - identificador do vértice
  - type - NULL, Lar, Hospital, CentroSaude, Garagem
  - number - número de idosos no lar (null se o vértice não for um lar)
  - Adj $\subseteq$ E - Arestas ligadas ao vértice

- E - Arestas (representam o caminho entre dois pontos):
  - id - identificador da aresta
  - weight - peso da aresta (distância em km entre dois vértices)
  - dest $\in V_i$ - vértice de destino
- I - Vértice Inicial (Garagem)
- F - Vértice FInal (Hospital ou Centro de Saúde)

2.2. Dados de Saída

$G_f = (V_f, E_f)$ - grafo dirigido pesado

- weight - Somatório do peso de todas as arestas percorridas no caminho.
- lotação - lotação utilizada pela viatura de transporte
- P - Sequência ordenada de todas as arestas do caminho

2.3 Restrições

$\forall$ i $\in$ [1 ; $|N_f|$], lotação(N_f[i]) > 0, lotação representa o número de passageiros que uma viatura pode transportar.

type = “Urgente” v “NotUrgent”
type(V_i) = “null”, “lar”, “hospital”, “centroSaude”, “garagem” (pode ser null, se o vértice não corresponder a um lar, hospital, centro de saúde ou garagem)
$\forall$ v $\in$ V_i, v >= 0 (v pode ser 0, se o vértice não corresponder a um lar ou não existirem transportes pendentes)

$\forall$ e $\in$ E_i, weight > 0, pois weight represente uma distância em km entre dois pontos no mapa.
$\forall$ e $\in$ E e tem de estar disponível para ser utilizado.

Se N_f < N_i Não se usaram todas as ambulâncias, logo todos os pedidos foram efetuados com sucesso.

$|N_f|$ <= $|N_i|$, pois o número de viaturas existentes não pode ser superar o número de viatura utilizadas.

Seja e_1 o primeiro elemento de P. e_1 $\in$ Adj(I), pois as viaturas partem todas da garagem.

Seja e_P o último elemento de P. e_P $\in$ F, pois as viaturas terminam no Hospital ou Centro de Saúde.

2.4 Função Objectivo

A função objectivo devolve como solução o caminho ótimo para cada uma das deslocações conforme o tipo de urgência do transporte, de um destino inicial (Local onde a viatura de transporte se encontra) até ao destino final (Hospital ou Centro de Saúde), de forma a que todos os idosos sejam transportados.
A solução otimizada passa pela minimização das funções:

$f = |N|$

$P = \sum w(e), e \in E$

Privilegiando sempre a função f, de modo a que seja necessário o número mínimo possível de viaturas para os transportes.

3. Perspectiva de Solução

3.1 Etapas de Concepção

A perspectiva de solução passará por um conjunto de etapas até atingirmos os resultados finais desejados, etapas essa que serão mencionadas abaixo.

Etapa 1 :Recolha de dados

Nessa etapa consiste em selecionar uma área da mapa não muito grande devido ao tempo de processamento que esse pode causar nos algoritmos a serem implementadas.
Esses dados da área da mapa recolhida (ruas,estradas,..) serão guardados em ficheiros txt que depois servirão para criação do Grafo.

Etapa 2 :Pré-processamento do Grafo

De forma simplificar e otimizar o tempo de processamento do algoritmos nos Grafos, nessa etapa o objectivo é eliminar o tamanho do Grafo reduzindo os números de Nós e Arestas que o constituem, reduzindo o tempo e o espaço de pesquisa para encontrar a solução desejada..

Etapa 3 :Verificar se há caminho possível entre dois pontos ou localidades

Depois de saber os pontos de partida e de destino o objetivo é fazer uma pesquisa no grafo de forma verificar se existe pelo menos um caminho entre eles, nessa fase não é importante encontrar o caminho ótimo mas sim verificar a acessibilidades entre esses pontos .

Etapa 4 :Encontar o melhor caminho entre dois pontos ou localidades

No final o objectivo é encontrar o melhor caminho de um ponto de partida até um destino de acordo com um dado critério especificado pelo usuário, podendo ou não passar por várias ponto de interesse (lares,centro saúde,hospital...) previamente especificado pelo usuário, tentando sempre alcançar o máximo desses pontos de interesse .

3.2 Algoritmos

No projecto iremos usar alguns algoritmos sobre o grafo tanto no seu pré-processamento como a obtenção do melhor caminho desejado de acordo com os parâmetros de entrada que consiste no objetivo final do projecto,algoritmos esses que serão descritas a seguir.

Pesquisa em Largura

A pesquisa em largura e um dos algoritmos mais simples para exploração de um Grafo, começa por um vértice, digamos **S**, especificado pelo usuário. O algoritmo visita **S**, depois visita todos os vizinhos de **S**, depois todos os vizinhos dos vizinhos, e assim por diante.O algoritmo numero os vértices, em sequência, na ordem em que eles são descobertos (ou seja, visitados pela primeira vez). Para fazer isso, o algoritmo usa uma fila (= *queue*) de vértices. No começo de cada iteração, a fila contém vértices que já foram numerados mas têm vizinhos ainda não numerados.

```
BUSCA-EM-LARGURA (n, Adj, r)
1   para u ← 1 até n faça
2       cor[u] ← branco
3   cor[r] ← cinza
4   F ← CRIA-FILA(r)
5   enquanto F não está vazia faça
6       u ← SAI-DA-FILA(F)
7       para cada v em Adj[u] faça
8           se cor[v] = branco
9               então cor[v] ← cinza
10                        ENTRA-NA-FILA(v, F)
11      cor[u] ← preto
12  devolva cor[1..n]
```

Pesquisa em Profundidade

O algoritmo consiste em explorar todas as arestas a partir do último vértice encontrado, sendo implementado de forma recursiva e utilizando o método de *backtracking*.

Quando todas as arestas de um vértice forem exploradas, retorna e explora as restantes arestas do vértice que o antecedia. O algoritmo aprofunda a pesquisa até que encontre o vértice pretendido

```
BUSCA-EM-PROFUNDIDADE (n, Adj, r)
 1  para u ← 1 até n faça
 2      cor[u] ← branco
 3  cor[r] ← cinza
 4  P ← CRIA-PILHA (r)
 5  enquanto P não estiver vazia faça
 6      u ← COPIA-TOPO-DA-PILHA (P)
 7      v ← PRÓXIMO (Adj[u])
 8      se v ≠ NIL
 9          então se cor[v] = branco
10                      então cor[v] ← cinza
11                            COLOCA-NA-PILHA (v, P)
12          senão cor[u] ← preto
13                TIRA-DA-PILHA (P)
14  devolva cor[1..n]
```

Dijkstra

O Algoritmo de Dijkstra (E.W. Dijkstra) é um dos algoritmos que calcula o caminho de custo mínimo entre vértices de um grafo. Escolhido um vértice como raiz da pesquisa, este algoritmo calcula o custo mínimo deste vértice para todos os demais vértices do grafo. Ele é bastante simples e com um bom nível de performance. Ele não garante, contudo, a exatidão da solução caso haja a presença de arcos com valores negativos.

Este algoritmo parte de uma estimativa inicial para o custo mínimo e vai sucessivamente ajustando esta estimativa. Ele considera que um vértice estará fechado quando já tiver sido obtido um caminho de custo mínimo do vértice tomado como raiz da pesquisa até ele. Caso contrário ele dito estar aberto.

```
DIJKSTRA (n, Adj, f, r)      comentário: f ≥ 0
1   para u ← 1 até n faça
2       dist[u] ← ∞
3   dist[r] ← 0
4   Q ← CRIA-FILA-VAZIA ( )
5   para v crescendo de 1 até n faça
6       INSERE-NA-FILA (v, Q)
7   enquanto Q não está vazia faça
8       u ← EXTRAI-MIN (Q)
9       para cada v em Adj[u] faça
10          se dist[u]+f(uv) < dist[v]
11              então DIMINUI-CHAVE (v, Q, dist[u]+f(uv))
12  devolva dist[1..n]
```

Floyd-Warshall

O algoritmo de Floyd-Warshall recebe como entrada uma matriz de adjacência que representa um grafo (V , E) orientado e pesado. O valor de um caminho entre dois vértices é a soma dos valores de todas as arestas ao longo desse caminho. As arestas E do grafo podem ter valores negativos, mas o grafo não pode conter nenhum ciclo de valor negativo. O algoritmo calcula, para cada par de vértices, o menor de todos os caminhos entre os vértices. Por exemplo, o caminho de menor custo. A ordem de complexidade é Θ (| V | 3).

```
ROTINA fw(Inteiro[1..n,1..n] grafo)
    # Inicialização
    VAR Inteiro[1..n,1..n] dist := grafo
    VAR Inteiro[1..n,1..n] pred
    PARA i DE 1 A n
        PARA j DE 1 A n
            SE dist[i,j] < Infinito ENTÃO
                pred[i,j] := i
    # Laço principal do algoritmo
    PARA k DE 1 A n
        PARA i DE 1 A n
            PARA j DE 1 A n
                SE dist[i,j] > dist[i,k] + dist[k,j] ENTÃO
                    dist[i,j] = dist[i,k] + dist[k,j]
                    pred[i,j] = pred[k,j]
    RETORNE dist
```

4. Casos de Utilização e Funcionalidades

O projecto a ser implementado terá uma interface fácil de usar num terminal onde o utilizador pode começar por escolher a área do mapa que esse quer fazer a pesquisa, após essa fase será feito uma listagens de todos os hospitais, centro de saúde e lares existentes nesta região.
O objetivo será mostrar um mapa com os percursos que cada uma das viaturas têm de fazer de acordo com as especificações introduzidas pelo utilizador e as prioridades previamente definidas na aplicação (casos urgentes e não urgentes).
No cálculo dos caminhos a fazer pelas viaturas, será mostrado se existe ou não um caminho possível entre os dois pontos e, caso exista, é mostrado o caminha mais curto/eficiente entre os mesmos.

5. Iterações

5.1 Iteração 1 - Um Veículo Um Caminho

"Um veículo um caminho" é a formalização mais simples do problema.
O problema consiste em selecionar um "lar de idosos" para o qual é necessário o transporte, um veículo(ambulância ou normal) e um Hospital ou Centro de Saúde como destino final.
É gerada uma lista de veículos que possam chegar ao lar, bem como uma lista dos Centros de Saúde ou Hospitais atingíveis pelo lar.
Como o grafo foi pré-processado e passado pelo algoritmo de Floyd-Warshall, o caminho ótimo para este cenário é o caminho mais curto do **veículo ao lar** e d**o lar ao Hospital ou Centro de Saúde**.

```
vector<Vertex<MapPoint> * >  GraphWorkout::oneVehicleOneItineration(Vehicle * v , HealthStation * healthCare , NursingHome * nr, int type){
    //type 0 = both
    //type 1 = vehicle to nursehome
    //type 2 = nursehome to healthcare

    MapPoint healthstation = healthCare->getMapPoint();
    MapPoint vehicle = v->getMapPoint();
    MapPoint nursinghome = nr->getMapPoint();
    vector<Vertex<MapPoint>*> result, temp;

    if(type == 0 || type == 1) {
        result = originalGraph->getfloydWarshallPath(vehicle, nursinghome);
    }

    if(type == 0 || type == 2) {

        temp = originalGraph->getfloydWarshallPath(nursinghome, healthstation);
    }

    if(type == 1) return result;
    if(type == 2){
        if(temp.size() != 0)
            return temp;
    }
    if(type == 0) {
        vector<Vertex<MapPoint> * > error;
        if(temp.size()==0 || result.size()==0){
            return error;
        }
        result.insert(result.end(),  first: temp.begin() + 1, temp.end());
    }

    return result;
```

5.2 Iteração 2 - Um Veículo Urgente

Nesta segunda iteração, abordamos a possibilidade de haver um pedido urgente de um veículo para um lar e o futuro transporte do lar para um Hospital.
Introduzindo o lar onde existe um paciente em estado urgente, é encontrado o veículo mais próximo do lar e, neste caso, o veículo tem de ser uma ambulância.(usando a função nearVehicle). A seguir é procurado o Hospital mais próximo(usando a função nearHealthStation). Com estes 3 pontos é calculado o caminho ótimo para o transporte.
O caminho ótimo é o caminho mais curto pois, como dito no ponto anterior, o algoritmo Floyd-Warshall garante isso.

```
vector<vector<Vertex<MapPoint>*>> GraphWorkout:: multipleVehicleMultipleItineration() {
    vector<vector<Vertex<MapPoint>*>> result;
    vector<Vertex<MapPoint>*> temp;

    for(unsigned int i = 0; i < vehicles.size(); i++) {
        while (nursingHome.size() > 0 ) {
            if(vehicles[i]->getVehicleCapacity() < nursingHome[0]->getElderlyNumber())
                break;
            temp = oneVehicleMultipleItineration( v: vehicles[i]);
            result.push_back(temp);
        }
    }


    return result;
}
```

5.3 Iteração 3 - Um Veículo Vário Caminhos

Nesta iteração, abordamos a possibilidade de um veículo poder passar por vários lares e recolher os idosos em espera para um consulta normal até a sua capacidade estar cheia. Selecionamos um Centro de Saúde para as consultas como destino final e de seguida um veículo, é escolhido o lar mais próximo do mesmo usando a função nearNursingHome sucessivamente (de lar para lar) até o veículo não ter capacidade para mais idosos. Posteriormente é calculado o caminho do último lar visitado pelo veículo até ao centro de saúde.
O caminho ótimo é o caminho mais curto devido ao algoritmo de Floyd-Warshall, como já referido anteriormente.

```
vector<Vertex<MapPoint> * > GraphWorkout::oneVehicleMultipleItineration(Vehicle * v){
    vector<Vertex<MapPoint> * > result, temp;

    Vertex<MapPoint> * garage = originalGraph->getVertex(v->getMapPoint());
    int currentVertex = garage->posAtVec;
    vector<Vertex<MapPoint> * > error;
    distBetHealthLocation(currentVertex, isSort: true);

    Vertex<MapPoint> * current = (*originalGraph)(currentVertex), * next = (*originalGraph)(nursingHome[0]->posAtVec);

    result = originalGraph->getfloydWarshallPath(current->getInfo(), next->getInfo());//shortest from vehicle to nursingHome

    while (nursingHome.size() > 0 && v->getVehicleCapacity() >= nursingHome[0]->getElderlyNumber()) {

        nursingHome[0]->getMapPoint().print();


        if (nursingHome.size() == 0)
            break;
        distBetHealthLocation(current->posAtVec, isSort: true);//recalculates the distances and sorts again
        current = next;
        next = (*originalGraph)(nursingHome[0]->posAtVec);
        temp = originalGraph->getfloydWarshallPath(current->getInfo(), next->getInfo());//shortest from nursingHome to nursingHome
        for (size_t i = 0; i < nursingHome.size(); i++) {//iterate the vertexes in the new path and check for nursingHome passed
            if (temp.size() > 0
                && find(temp.begin(), temp.end(), val: (*originalGraph)(nursingHome[i]->posAtVec)) != temp.end()
                && v->getVehicleCapacity() >= nursingHome[i]->getElderlyNumber()) {//if the nursingHome is in the new path
                v->addPass( value: -nursingHome[i]->getElderlyNumber());
                nursingHome[i]->getMapPoint().print();
                nursingHome.erase( position: nursingHome.begin()+i);
                i--;
            }

        }
        if(temp.size()==0 || result.size()==0){
            result = error;
        }
    result.insert(result.end(), first: temp.begin() + 1, temp.end());

    return result;
}
```

5.4 Iteração 4 - Vários Veículos Vários Caminhos

Nesta iteração, abordamos a situação mais complexa do problema que passar por fazer o transporte de todos os idosos dos lares para os Centros de Saúde mais próximos.
Nesta situação não é escolhido qualquer parâmetro.
Para cada veículo a calculado o caminho para o lar mais próximo com idosos à espera de uma consulta e posteriormente passar por vários lares mais próximos do anterior até ter capacidade cheia. Quando estiver ao máximo de capacidade é calculado o caminho para o Centro de Saúde mais próximo do último lar visitado.
O caminho ótimo é o caminho mais curto devido ao algoritmo de Floyd-Warshall, como já referido anteriormente.

```
vector<vector<Vertex<MapPoint>*>> GraphWorkout:: multipleVehicleMultipleItineration() {
    vector<vector<Vertex<MapPoint>*>> result;
    vector<Vertex<MapPoint>*> temp;

    for(unsigned int i = 0; i < vehicles.size(); i++) {
        while (nursingHome.size() > 0 ) {
            if(vehicles[i]->getVehicleCapacity() < nursingHome[0]->getElderlyNumber())
                break;
            temp = oneVehicleMultipleItineration( v: vehicles[i]);
            result.push_back(temp);
        }
    }


    return result;
}
```

6. Funções Auxiliares

6.1 nearHealthStation()

Função usada no cálculo do Centro de Saúde ou Hospital mais próximo de um ponto através do algoritmo de Dijkstra.

```
HealthStation* GraphWorkout::nearHealthStation(MapPoint node,int type) {

    int distMinim = INT_MAX;
    int nodePosition = -1;


    for (unsigned int i = 0; i < healthCareLocation.size(); i++) {
        if( healthCareLocation[i]->getType()==type){
            originalGraph->dijkstraShortestPath(node);
            int currentDist = originalGraph->getVertex( v: healthCareLocation[i]->getMapPoint())->getDist();
            if (currentDist <= distMinim) {
                nodePosition = i;
                distMinim = currentDist;
            }

        }

    }

    return healthCareLocation[nodePosition];

}
```

6.2 nearVehicle()

Função usada no cálculo do veículo mais próximo de um ponto através do algoritmo de Dijkstra.

```
Vehicle* GraphWorkout::nearVehicle(MapPoint node,int type) {

    int distMinim = INT_MAX;
    int nodePosition = -1;


    for (unsigned int i = 0; i < vehicles.size(); i++) {
        if( vehicles[i]->getVehicleType()==type){

            originalGraph->dijkstraShortestPath( s: vehicles[i]->getMapPoint());

            int currentDist = originalGraph->getVertex(node)->getDist();

            if (currentDist <= distMinim) {
                nodePosition = i;
                distMinim = currentDist;
            }

        }

    }

    return vehicles[nodePosition];

}
```

6.3 nearNursingHome()

Função usada no cálculo do veículo mais próximo de um ponto através do algoritmo de Dijkstra.

```
NursingHome* GraphWorkout::nearNursingHome(MapPoint node) {
    int distMinim = INT_MAX;
    int nodePosition = -1;

    originalGraph->dijkstraShortestPath(node);

    for (unsigned int i = 0; i < nursingHome.size(); i++) {
        int currentDist=originalGraph->getVertex( v: nursingHome[i]->getMapPoint())->getDist();
        if (currentDist < distMinim) {
            nodePosition  = i;
            distMinim = currentDist;
        }
    }

    if (nodePosition == -1) {
        MapPoint* node= new MapPoint( id: -1, x: -1, y: -1);
        NursingHome* nullNode= new NursingHome(*node, number: -1);
        return nullNode;
    } else
        return nursingHome[nodePosition];


}
```

6. Conclusão

Neste relatório, discutimos as soluções possíveis para o problema apresentado, utilizando as técnicas e conhecimentos abordados tanto nas aulas teóricas como práticas.
Foram implementados os algoritmos de Pesquisa em Largura e Pesquisa em Profundidade para testar a conectividade entre os vértices do grafo.
Os algoritmos de Dijskstra e Floyd-Warshall para o cálculo de percursos ótimos (caminhos mais curtos entre os pontos).
Foi implementada a visualização dos grafos e dos caminhos obtidos pelo algoritmo, utilizando o GraphViewer fornecido.

A elaboração deste relatório e a perspectiva de abordagem ao problema foram desenvolvidos pelos dois membros do grupo.
O esforço dedicado por cada elemento do grupo é de 50%.