Sistema de Evacuação

Tema 3

Rui Guedes
up201603854@fe.up.pt

César Pinho
up201604039@fe.up.pt

Bruno Carvalho
up201606517@fe.up.pt

11 de Abril de 2018

Conteúdo

1 Introdução

1.1 Definições preliminares

Ao longo do relatório usamos as seguintes definições e conceitos:

- G refere-se à abstração de grafo (ou à classe Graph);
- M refere-se ao mapa *real* extraído de www.openstreetmap.org e representado por G.
- V é o conjunto dos vértices de G, ou a sua cardinalidade num contexto numérico;
- E é o conjunto das arestas de G, ou a sua cardinalidade num contexto numérico;
- R é o conjunto das ruas, com nome ou não, de M;
- $v \in V$, $v_i \in V$ são vértices de V.
- $e \in E$, $e_i \in E$ são arestas de E. A aresta e_{ij} é aquela que liga v_i a v_j.
- $r \in R$, $r_i \in R$ são ruas de R.
- *meta*, *nodes*, *roads*, *subroads* são ficheiros de dados que servem como *input* do programa. Mais informação na subsecção 4.1.
- $\Theta(\mathsf{expression}(n))$ refere-se à complexidade temporal ou espcial de uma função ou algoritmo (Big O Notation).
  Mais precisamente, Θ_t é a complexidade temporal, e Θ_e a complexidade espacial de um algoritmo ou função.

1.2 Tema 3

1.2.1 Especificação do Tema

A especificação e descrição do Tema 3 – Sistema de Evacuação é a seguinte:

> Um sistema de evacuação permite retirar em segurança os utentes presentes numa área acidentada. Considere a rede de auto-estradas de Portugal e suponha a ocorrência de um acidente grave (incêndio, por ex) que obriga ao corte de um troço da auto-estrada e desvio dos automóveis em circulação. Neste trabalho, pretende-se implementar um sistema que determine os percursos alternativos que cada automóvel deve realizar. Considere a existência de múltiplos automóveis nos vários troços das auto-estradas, cada automóvel possui o seu destino. Cada troço da auto-estrada possui uma capacidade limitada, isto é, número máximo de automóveis que suporta (este valor não

deve ser ultrapassado, correndo o risco de parar o trânsito automóvel nesse troço). O sistema deve indicar, para cada automóvel, o melhor percurso alternativo que deve realizar para evitar o troço acidentado e chegar ao seu destino em segurança, no menor tempo possível.

Avalie a conectividade do grafo, a fim de evitar que locais de destino se encontrem em zonas inacessíveis a partir do ponto onde se encontra o automobilista quando da ocorrência do acidente.

1.2.2 Descrição do Tema

O tema do presente trabalho consiste num sistema de evacuação que permite não só retirar em segurança os utentes presentes numa área acidentada como também definir trajetos para os quais serão evitadas zonas acidentadas. Para tal serão consideradas diferentes redes de estradas nas quais ocorrerão acidentes graves que por sua vez implicam o corte das mesmas.

Cada estrada possui uma capacidade limitada, isto é, possui um número máximo de automóveis que suporta, existindo, portanto, o risco de congestionamento do trânsito nos diversos troços existentes.

Assim torna-se evidente a necessidade de determinar percursos alternativos para os utentes de modo a evitarem estas zonas acidentadas, tendo este trabalho como principal objetivo, encontrar o melhor percurso alternativo que cada utente deve realizar para evitar o troço acidentado e chegar ao seu destino em segurança, no menor tempo possível.

2 Interpretação do Tema e do Problema

2.1 Identificação do Problema

Após uma a análise detalhada do tema anteriormente descrito, tornou-se evidente o problema em questão, problema este coincidente com o objetivo principal do trabalho, isto é, obter o melhor percurso alternativo, tendo em conta a segurança e o tempo, para cada utente. Deste modo procedeu-se então à formalização do problema decompondo este em quatro subproblemas:

1. **Cálculo do melhor percurso para cada utente tem em conta apenas a distância.**

   Numa primeira fase do problema o objetivo passa por simplesmente determinar o percurso mais curto entre dois pontos definidos pelo utente, isto tendo em conta o grafo considerado para tal efeito. Nesta fase ignora-se a maior parte dos fatores

intervenientes no que é considerado o objetivo final deste trabalho, e, portanto, considera-se apenas a distância entre dois pontos.

Assim numa primeira fase, o problema resume-se a um problema trivial de caminho mais curto tendo em conta a distância, sendo este resolúvel, recorrendo a determinados algoritmos que se encontram descritos na secção referente à descrição da solução implementada.

2. **Cálculo do melhor percurso para cada utente tem em conta a distância mais os locais acidentados.**

   Numa segunda fase, na qual já existe um ou vários algoritmos para resolver o problema base do tema deste trabalho, é introduzido um novo fator no mesmo, fator este referente à introdução de locais acidentados na rede de estradas considerada.

   Deste modo passam, portanto, a existir locais a ser evitados e por sua vez inacessíveis devido ao seu estado, no entanto, a introdução deste novo fator, não altera o funcionamento dos algoritmos isto porque, os locais acidentados passam a ser tidos em conta como locais inexistentes e consequentemente passam a ser ignorados pelo algoritmo no cálculo do melhor percurso para cada utente.

3. **Cálculo do melhor percurso para cada utente tem em conta a distância, os locais acidentados, e o tempo.**

   Numa terceira fase, é introduzido o fator tempo que passa a ser considerado pelos algoritmos como o peso da aresta no grafo. Este tempo é obtido tendo em conta a distância de cada estrada, já utilizada nos subproblemas anteriores, e ainda a sua velocidade máxima.

   Este cálculo consiste numa operação matemática simples, que traduz na divisão da distância pela velocidade, obtendo-se assim o novo fator, isto é, o tempo que se demora a percorrer uma determinada estrada.

4. **Cálculo do melhor percurso para cada utente tem em conta a distância, os locais acidentados, o tempo e a ocupação da rede de estradas considerada.**

   Numa quarta e última fase introduz-se por fim o último fator que é a ocupação da estrada, isto é, o número de carros presentes num troço tendo em conta a capacidade máxima para esse mesmo troço.

   Este novo fator é introduzido no trabalho na medida em que a velocidade com que uma determinada estrada é percorrida, tem em conta a sua ocupação.

   Para tal é tido em conta a relação linear velocidade-densidade na qual um aumento de densidade provoca uma diminuição da velocidade na estrada. Deste modo, quando o utente pretende verificar qual o melhor percurso para ir de um ponto A a um ponto B, o algoritmo fornece os resultados tendo em conta o estado do grafo, isto é, a ocupação da rede de estradas nesse momento, pois esta varia dinamicamente. Quando densidade de carros é máxima, ou seja, quando a ocupação

atinge o limite máximo a velocidade é praticamente nula pois a estrada encontra-se congestionada.

2.2 Formalização do problema

2.2.1 Dados de entrada

Os dados de entrada neste trabalho correspondem à inicialização do grafo que posteriormente será utilizado para visualização dos dados de saída e sobre o qual irão operar os algoritmos de caminho mais curto descritos na secção referente à descrição da solução implementada. Para tal é necessário fazer um pré-processamento dos dados de entrada para que estes algoritmos possam ser devidamente executados.

Este pré-processamento encontra-se definido e explicado na secção 4.

2.2.2 Dados de saída

Os dados de saída neste trabalho correspondem ao melhor percurso possível a ser percorrido pelo utente no instante em que este pretende se deslocar entre dois pontos, evitando todos os locais acidentados pelo caminho.

Estes dados podem ser obtidos visualmente através da GUI e à qual se encontram associadas um conjunto de cores que descrevem o funcionamento do mesmo.

Tabela 1: Cores da Interface

- Azul – corresponde a um vértice no seu estado normal.
- Vermelho – corresponde a vértices ou arestas acidentados, na medida em que estes não podem ser utilizados por qualquer algoritmo na pesquisa do melhor percurso possível entre dois pontos.
- Verde – corresponde ao melhor percurso possível entre dois pontos, tendo em conta a situação atual do grafo.
- Ciano – corresponde ao melhor percurso possível previsto entre dois pontos. Esta cor encontra-se associada aos algoritmos que envolvem simulação pois implicam dinamismo no percorrer deste percurso que pode ou não ser variável dependendo dos acontecimentos a ele associados.
- Magenta – corresponde aos vértices selecionados pelo utente, tanto o vértice de origem como o de chegada.
- Amarelo – corresponde à cor dos vértices não alcançáveis. A cada execução de um algoritmo para encontrar o melhor percurso possível é realizada uma pesquisa prévia que determina os vértices não alcançáveis.

3 Classes e Estruturas de Dados

3.1 Filosofia da Implementação

A base do programa é a abstração Graph, baseada naquela que é usada nos exercícios da aula prática 5. É extendida consideravelmente, suportando os algoritmos que listamos na Tabela 4 e a acidentação de vértices e de arestas, representando acidentes nas vias e impossibilitando movimento.

No projeto usamos a versão 1.2 da API GraphViewer fornecida pelos professores. Esta API é parcialmente encapsulada pela classe Graph.

O ponto de partida normal do programa é a leitura de um conjunto de ficheiros de texto – já listados na subsecção 1.1 – que descrevem um mapa extraído de www.openstreetmap.org e processado pelo OpenStreetMapsParser (fornecido pelos professores). Não existe outro ponto de partida para o programa.

O grafo G procura representar com precisão e proporção um mapa real. Como tal, G é estático depois de inicializado, sendo que certas funcionalidades do género *create* e *remove* não estão disponíveis.

3.2 Classes

Tabela 2: Classes Principais

Graph	A abstração de G e a base do programa. Corresponde na realidade ao sistema de ruas e interseções do mapa. É estritamente orientado, (normalmente) planar e portanto esparso em termos de arestas (o que significa $V \sim E$, i.e. $\Theta(V) = \Theta(E)$).
Vertex	A abstração de um vértice v. Corresponde na realidade a uma interseção de ruas, ou a uma parte de uma rua para delinear o seu trajeto curvilíneo. Cada entrada do ficheiro *nodes* é um vértice v.
Road	A abstração de uma rua r do mapa real. Corresponde a uma rua no mapa real, e é constituído por uma sequência (caminho) de arestas e_j, $j = 1, \ldots, k$. Cada entrada do ficheiro *roads* é uma rua r.
Edge	A abstração de uma aresta e. Corresponde a uma porção maioritariamente retilínea de um rua no mapa real, sem interseções ou bifurcações. Cada entrada do ficheiro *subroads* é uma aresta e.
Subroad	A abstração de uma porção de uma estrada s. À semelhança da abstração Edge, corresponde a uma porção maioritariamente retilínea de uma rua no mapa real, sem interseções ou bifurcações, embora esta seja destinada a gerir e armazenar informação usada pela Edge. Cada entrada do ficheiro subroads é uma porção s.

3.2.1 Graph

Graph encapsula V, a API de visualização do grafo e ainda todos os algoritmos implementados, além das estruturas de dados auxiliares à programação dinâmica destes . A classe Graph contém os atributos:

- width, height: dimensões do mapa do grafo.
- vertexSet: vetor de apontador para os vértices que constituem o grafo.
- accidentedVertexSet: vetor de apontadores para os vértices que constituem o grafo e que se encontram intransitáveis.
- gv: apontador para o objeto da classe GraphViewer responsável pela representação visual do grafo.
- scale: fator de escala do mapa visualizado no ecrã.

3.2.2 Vertex

Cada Vertex detém uma posição no mapa do grafo (coordenadas cartesianas (x_i, y_i) inteiras) projetada pela sua posição em M. Pode ou não estar *acidentado*. Cada Vertex é responsável por gerir as arestas que dele saem – o que efetivamente significa que V encapsula E.

A classe Vertex contém os seguintes atributos:

- id: inteiro positivo que identifica, de forma única, um vértice.
- x, y: coordenadas cartesianas.
- accidented: booleano que indica se o vértice se encontra acidentado ou não. Os vértices acidentados estão cortados ao transito impossibilitando a passagem de veículos.
- adj: vetor de arestas com origem neste vértice e que o ligam a outros distintos.
- accidentedAdj: vetor de arestas com origem neste vértice que se estão atualmente intransitáveis.
- graph: apontador para o objeto da classe Graph que contém o vértice e que o representa.

3.2.3 Edge

Cada Edge encapsula, essencialmente, a informação da subroad que é representada por uma aresta e os vértices que a delimitam. A classe Edge contém os seguintes atributos:

- id: inteiro positivo que identifica uma aresta de forma única.
- source: vértice de início da aresta.
- dest: vértice de destino de aresta.
- accidented: booleano que indica se a aresta se encontra intransitável ou não.
- graph: apontador para o objeto da classe Graph que contém a aresta e que a representa.
- subroad: apontador para o objeto da classe Subroad que esta aresta representa.

3.2.4 Road

Cada Road encapsula a informação que descreve a rua real em M: o seu nome e o comprimento. A classe Road contém os atributos seguintes:

- id: inteiro positivo que identifica uma estrada de forma única.
- name: nome da estrada.
- bothways: booleano que indica se a estrada é de dois sentidos ou não.
- totalDistance: distância total da estrada.
- maxSpeed: limite de velocidade da estrada calculado a partir da distância.

3.2.5 Subroad

Cada SubRoad contém a informação do mundo real sobre uma Edge. A classe Subroad contem os seguintes atributos:

- distance: comprimento da sub estrada.
- road: apontador para a rua à qual pertence.
- actualCapacity: quantidade de carros presentes na sub estrada.
- maxCapacity: quantidade máxima de carros que podem existir na sub estrada. Para além dos atributos referidos, foram também usados outros adicionais, nas classes Vertex e Edge, específicos para certos atributos, que serão abordados mais á frente.

4 Inicialização de um Mapa

4.1 Rotina de criação de ficheiros de dados

Vamos agora descrever, muito sucintamente, o procedimento para se extrair um novo mapa de `https://www.openstreetmap.org/` e incluí-lo no nosso projeto. O processo não deverá demorar mais do que dois minutos.

No site www.openstreetmap.org, depois de delimitada a região, extraímos o ficheiro-mapa *map.osm* usando (preferencialmente) a Overpass API. Depois corremos o ficheiro na aplicação OpenStreetMapParser, com o ficheiro exportado.

O Parser gera três ficheiros: um contendo a informação relativa a V, o ficheiro *nodes*; outro contendo informação relativa a R, o ficheiro *roads* – que descreve a topologia de G; e outro contendo a informação de E, descrevendo a decomposição das ruas – o ficheiro *subroads*.

Além destes três ficheiros, um quarto ficheiro de configuração *meta* deve ser manualmente criado com a informação geral do mapa, e algumas opções-extra para customizar a inicialização de G.

Este ficheiro de configuração suporta os campos presentes na tabela 3, em estilo key=val;

Tabela 3: Opções do ficheiro *meta*

	Obrigatórios:
min_longitude	A longitude mínima de G.
max_longitude	A longitude máxima de G.
min_latitude	A latitude mínima de G.
max_latitude	A latitude máxima de G.
nodes	O número de nodes (de M e de G).
edges	O número de edges (de M e de G).
	Opcionais:
density	(Opcional) A densidade (visual) dos vértices.
width	(Opcional) A largura preferida de G.
height	(Opcional) A altura preferida de G.
boundaries	(Opcional) Delimitar a área do grafo com vértices-fantasma.
oneway	(Opcional) Sobrepõr o parâmetro direcional no ficheiro *roads* e definir todas as ruas como unidirecionais.
bothways	(Opcional) Sobrepõr o parâmetro direcional no ficheiro *roads* e definir todas as ruas como bidirecionais.
straight_edges	(Opcional) Desenhar todas as arestas retílineas.

Os limites geográficos, o número de vértices e o número de arestas são todos fornecidos pelo Parser e definem a geometria de M. O ficheiro *meta* é lido com expressões regulares para cada parâmetro, assumindo a sintaxe key=val; portanto as opções podem estar distribuídas de qualquer forma.

Os quatro ficheiros deverem ser colocados na pasta *resources*. A convenção de nomenclatura dos ficheiros é simples: escolhido um nome representativo qualquer, por exemplo *city*, os quatro ficheiros deverão ser chamados *city_meta.txt*, *city_nodes.txt*, *city_roads.txt* e *city_subroads.txt*.

4.2 Cálculo das dimensões do grafo

Os vértices listados no ficheiro *nodes* têm a sua localização apresentada em coordenadas geográficas. Para representar os vértices v em GraphViewer estas coordenadas têm de ser convertidas em coordenadas cartesianas (x, y), de modo que se mantenha a proporcionalidade do mapa.

As dimensões de M são calculadas diretamente através das fórmulas $\mathsf{height\ (km)} = 110.574 \cdot \delta\mathsf{latitude}$ e $\mathsf{width\ (km)} = 111.320 \cdot \delta\mathsf{longitude} \cdot |\cos(\overline{\mathsf{latitude}})|$.

A proporção height:width de M é também a proporção visual G. Especificada uma largura (*width*), altura (*height*) ou densidade de vértices (*density*) preferida no ficheiro *meta*, determina-se as dimensões de G.

Para determinar as dimensões de G a partir de uma densidade d, note-se que $\mathsf{width} \times \mathsf{height} \times d = V$, e portanto

$$\mathsf{width} = \sqrt{\frac{V}{d \times \mathsf{height{:}width}}}$$

Por omissão usamos densidade 0.0001.

4.3 Ponto de partida e leitura dos ficheiros de dados

Todos os ficheiros de dados são lidos usando expressões regulares std::regex da STL. Isto permite que erros nos ficheiros de dados sejam detetados de forma simples e direta, mas também significa que qualquer mudança manual nestes deva ser feita com cuidado. A ter em consideração as seguintes situações, que são automaticamente detetadas:

- (Erro) Os parâmetros obrigatórios do ficheiro *meta* têm de estar todos presentes.
- (Warning) O número de vértices indicado em *meta* deve ser igual ao número de nodes (linhas) lidos em *nodes*.
- (Warning) O número de arestas indicado em *meta* deve ser igual ao número de subroads (linhas) lidas em *subroads*.
- (Erro) As latitudes e as longitudes lidas têm de estar dentro dos limites geográficos especificados.

Em suma, para carregar um mapa seguem-se os passos:

1. Lê-se o ficheiro de configuração *meta*.
2. Inicializa-se G (e GraphViewer) com as dimensões calculadas.
3. Lê-se o ficheiro *nodes* e cria-se V.
4. Lê-se o ficheiro *roads* e cria-se R.
5. Lê-se o ficheiro *subroads* e cria-se E.

5 Algoritmos

Necessitamos de algoritmos que assegurem e realizem três tarefas distintas:

- **Deteção de conetividade rápida (em $\Theta_t(V)$)**
  Determinar a conetividade do grafo em tempo linear.
- **Determinação do caminho mais curto**
  Encontrar o caminho mais curto entre quaisquer dois vértices.
- **Determinação do caminho mais rápido (simulação)**
  Usando o algoritmo de *Dijkstra*, uma função heurística de tempo, e uma dinâmica de grafo que gere – automaticamente e de forma aproximada – o trânsito do grafo, é possível visualizar a simulação de uma viagem de um utente através do grafo.

Para este fim implementámos as quatro classes de algoritmos na tabela 4.

Tabela 4: Algoritmos Implementados

Greedy BFS Distância	Para encontrar uma aproximação do (possivelmente o) caminho mais curto entre dois vértices no grafo.
Dijkstra Distância	Para encontrar o caminho mais curto entre dois vértices do grafo.
A* Distância	Para encontrar o menor caminho entre dois vértices, usando a heurística de distância ao destino para melhorar a prioritização dos vértices.

5.1 Conetividade

Para determinar a conetividade do grafo G a partir de um vértice v, ou seja, determinar a *closure* G'_v de G que inclui v, usamos o algoritmo *Breadth First Search*, sem destino. Ver Algoritmo 1. Assumamos que G'_v tem o conjunto de vértices V'_v e de arestas E'_v.

Complexidade Temporal A complexidade temporal de *Breadth First Search* (Pesquisa em largura) é $\Theta(E+V)$. Prova. Todas as instruções do algoritmo têm complexidade temporal constante ($\Theta(1)$). O *loop while* exterior executa V'_v vezes e o *loop for* interior executa E'_v vezes no total. Logo $\Theta_t = \Theta(E'_v + V'_v) = \Theta(E'_v) + \Theta(V'_v) = \Theta(E) + \Theta(V) = \Theta(E + V)$.

Complexidade Espacial A complexidade espacial de *Breadth First Search* é $\Theta_e(V)$. Todas as variáveis exceto queue têm espaço de memória constante, e queue armazena no máximo V'_v vértices. Logo $\Theta_e = \Theta(V'_v) = \Theta(V)$.

Correção A ideia básica subjacente a este algoritmo

As considerações acima também se aplicariam ao caso em que o algoritmo terminasse cedo (*early exit*) O algoritmo *BFS* é utilizado como algoritmo auxiliar dos restantes, para determinar quais são os vértices alcançáveis. Como alternativa poderia-se usar o algoritmo *Depth First Search*, com complexidade temporal e espacial equivalentes.

Algorithm 1 Breadth First Search(v *origin*)

```
queue = new Queue()                                  ▷ Simple Queue
queue.push(origin)

while not queue.empty() do
    current = queue.front()
    queue.pop()
    for next in adj(current) do
        if accident in next then
            continue
        end if
        if not next.visited then
            next.visited = true
            queue.push(next)
        end if
    end for
end while
```

5.2 Determinação de caminho mais curto

Para determinar o caminho mais curto (em termos de distância euclidiana) disponibilizamos três algoritmos distintos: *Greedy Best-First Search* – pesquisa gananciosa muito rápida e não necessariamente correta – *Dijkstra* e *A**. Ver Algoritmos 2, 3, 4, 5.

Em todos os algoritmos, é assumido que uma limpeza do campo v.cost foi realizada pelo menos para todos os vértices de V'_v, designando o valor inicial $v.\text{cost} = \infty, \forall v \in V'_v$.

Complexidade Temporal A complexidade temporal dos algoritmos é $\Theta_t((V+E)\ln V)$ se PriorityQueue for implementada como uma Árvore Binária. Prova. O *loop while* exterior executa no máximo $V'_v \leq V$ vezes e o *loop for* interior executa no máximo $E'_v \leq E$ vezes no total. As operações de push e pop executam em tempo $\Theta(\ln \text{sizeof}(\text{pqueue}))$, e como o tamanho de queue é no máximo $V'_v \leq V$ a qualquer momento então $\Theta(\ln \text{sizeof}(\text{pqueue})) = \Theta(\ln V)$. Logo $\Theta_t = \Theta(E+V) \cdot \Theta(\ln V) = \Theta((E+V)\ln V)$.

Complexidade Espacial A complexidade espacial dos algoritmos é $\Theta_e(V)$. A lógica é idêntica ao algoritmo *BFS* de conetividade (ver 5.1).

Correção Dijkstra Inicialmente todos os vértices v têm cost infinito. Logo a primeira vez que um vértice next $\neq$ *origin* é encontrado no *loop for* interior, next.cost é infinito e a desigualdade newcost $< v$.next.cost é automaticamente verdadeira, designando o valor newcost a next.cost. Sabendo que *origin.cost* é 0 pode-se provar por indução sobre a iteração j do *loop while* que v.cost é o caminho mais curto de *origin* a v usando apenas os vértices já iterados $\{v_1 = \textit{origin}, v_2, v_3, \ldots, v_j\}$. Isto demonstra a correção do algoritmo 3 (*late exit*). Suponhamos que o algoritmo 4 é inválido, e aquando do retorno na instrução break o melhor caminho não é o determinado. Pela indução deduzida acima sabemos que o melhor caminho determinado, digamos com comprimento l, usa apenas os vértices $\{v_1 = \textit{origin}, v_2, \ldots, v_k = \textit{destination}\}$. Portanto o melhor caminho usa um vértice v' que não está neste conjunto, e como tal ainda não foi iterado no *loop while*. Logo v'.cost $>$ v_k.cost = destination.cost. Se d é o comprimento do melhor caminho, então v'.cost $< d$ pois as distâncias são não negativas. Mas então $d >$ destination.cost $= l$. Contradição.

Algorithm 2 Greedy Best-First Search(v *origin*, v *destination*)

```
pqueue = PriorityQueue()
pqueue.push(origin)
while not pqueue.empty() do
    current = pqueue.front()
    pqueue.pop()
    for next in adj(current) do
        if accident in next then
            continue                                  ▷ Clear vertices only
        end if
        if not next.visited then
            next.visited = true
            next.priority = distance(next, destination)   ▷ Greedy
            pqueue.push(next)
        end if
    end for
end while
```

Algorithm 3 Dijkstra(v *origin*) ▷ Late Exit

```
pqueue = PriorityQueue()                                  ▷ Supporting decrease key
pqueue.push(origin)
while not pqueue.empty() do
    current = pqueue.front()
    pqueue.pop()
    for next in adj(current) do
        if accident in next then
            continue                                      ▷ Clear vertices only
        end if
        newcost = current.cost + distance(current, next)
        if newcost < next.cost then                       ▷ Found a shorter path
            next.cost = newcost
            next.priority = newcost
            next.previous = current
            pqueue.push(next)                             ▷ Decrease key if already present
        end if
    end for
end while
```

Algorithm 4 Dijkstra(v *origin*, v *destination*) ▷ Early Exit

```
pqueue = PriorityQueue()                                  ▷ Supporting decrease key
pqueue.push(origin)
while not pqueue.empty() do
    current = pqueue.front()
    pqueue.pop()
    if current == destination then
        break
    end if
    for next in adj(current) do
        if accident in next then
            continue                                      ▷ Clear vertices only
        end if
        newcost = current.cost + distance(current, next)
        if newcost < next.cost then                       ▷ Found a shorter path
            next.cost = newcost
            next.priority = newcost
            next.previous = current
            pqueue.push(next)                             ▷ Decrease key if already present
        end if
    end for
end while
```

Algorithm 5 A*(v *origin*, v *destination*)

```
pqueue = PriorityQueue()                                  ▷ Supporting decrease key
pqueue.push(origin)
while not pqueue.empty() do
    current = pqueue.front()
    pqueue.pop()
    if current == destination then
        break
    end if
    for next in adj(current) do
        if accident in next then
            continue                                      ▷ Clear vertices only
        end if
        newcost = current.cost + distance(current, next)
        if newcost < next.cost then                       ▷ Found a shorter path
            next.cost = newcost
            next.priority = newcost + distance(next, destination)   ▷ A*
            next.previous = current
            pqueue.push(next)                             ▷ Decrease key if already present
        end if
    end for
end while
```

5.3 Determinação do caminho mais rápido entre vértices

Para determinar o caminho mais rápido usamos uma função heurística simples, chamemos-lhe time, capaz de avaliar o tempo que um veículo demora a atravessar uma aresta, baseado no comprimento da rua subjacente e no seu congestionamento (número de carros presentes naquele momento, face ao máximo).

Aquando da execução deste algoritmo é tirada uma *fotografia* do estado do grafo – do género de um satélite – o que significa que a validade do caminho descoberto está limitada à conformidade de G com esta *fotografia*.

Para simular a complexa dinâmica do trânsito no grafo G, este algoritmo é executado múltiplas vezes seguidas, de forma interativa: o condutor percorre apenas uma rua, ou até apenas uma aresta, e então o caminho óptimo é recalculado tendo em consideração um novo estado do grafo G – uma nova *fotografia* – idem até chegar ao seu destino. Ver Algoritmo 6.

Visto que a única diferença deste algoritmo de *Dijkstra* com o algoritmo de distância é o peso, e a função heurística tem complexidade temporal constante, a complexidade deste algoritmo é igual: $\Theta_t((E+V)\ln V)$ e $\Theta_e(V)$, tal como a sua correção.

Algorithm 6 Dijkstra(v *origin*, v *destination*) ▷ Simulation

```
pqueue = PriorityQueue()                                  ▷ Supporting decrease key
pqueue.push(origin)
while not pqueue.empty() do
    current = pqueue.front()
    pqueue.pop()
    if current == destination then
        break
    end if
    for next in adj(current) do
        if accident in next then
            continue                                      ▷ Clear vertices only
        end if
        newcost = current.cost + time(current, next)      ▷ Different weight
        if newcost < next.cost then                       ▷ Found a faster path
            next.cost = newcost
            next.priority = newcost
            next.previous = current
            pqueue.push(next)                             ▷ Decrease key if already present
        end if
    end for
end while
```

6 Funcionalidades Implementadas

Sobre o mapa é possível escolher um vértice ou uma aresta e causar ou reparar um acidente grave. Para além disso, é possível gerir o trânsito das arestas. Existe ainda uma opção que permite escolher o tipo de algoritmo a usar, de entre os disponíveis, ou se pretende usar uma simulação que dinamiza andeterminação do caminho mais rápido entre um vértice de origem e outro de destino, escolhidos pelo utente. Todas as funcionalidades implementadas solicitam ao utente a introdução de dados passo a passo, especificando o que é pretendido que seja introduzido e reagindo em conformidade com o input do utente. Por conseguinte, é fornecido o resultado das suas ações de forma gráfica e de fácil deteção e compreensão.

Tabela 5: Funcionalidades Implementadas

- Escolher o mapa no qual o utente pretende navegar sendo necessário introduzir o nome do mesmo;
- Acidentar um vértice indicando o seu id, disponibilizado pela interface gráfica;
- Acidentar uma aresta indicando o seu id, disponibilizado pela interface gráfica;
- Repara um vértice indicando o seu id, disponibilizado pela interface gráfica;
- Reparar uma aresta indicando o seu id, disponibilizado pela interface gráfica;
- Editar a informação de uma aresta, sendo necessário indicar o seu id, e escolhendo uma das seguintes opções:
  - Adicionar carros à aresta, indicando a quantidade sendo esta somada à atual;
  - Remover carros à aresta, indicando a quantidade a ser retirada à atual;
- Encontrar o caminho mais rápido, entre dois vértices indicados pelo utente, usando uma das seguintes opções:
  - Algoritmo *Greedy Best-First Search*;
  - Algoritmo *Dijkstra* aplicado a todos os vértices do grafo;
  - Algoritmo *Dijkstra* aplicado até ao vértice de destino;
  - Algoritmo *A**;
  - Simulação, aplicando variações de trânsito a cada aresta;
  - Simulação, aplicando variações de trânsito a cada estrada;
- Avaliação da performance dos algoritmos anteriores usando Benchmarking e sendo indicado o número de iterações a serem testar;
- Verificar a informação relativa à interface gráfica.

7 Conclusão

7.1 Principais dificuldades

Ao longo da resolução deste trabalho foram encontradas pelo grupo algumas dificuldades das quais se destacam o uso de mapas provenientes de OpenStreetMaps de forma escalada à GUI utilizada no trabalho, e ainda a decisão da melhor solução para resolver o problema enunciado.

Relativamente à primeira dificuldade, esta conseguiu ser superada recorrendo à criação de um novo ficheiro, associado a um determinado mapa, sobre o qual se guardou a informação necessária para inicializar a GUI de forma escalada, tornando a visualização do mapa realista.

No que diz respeito à segunda dificuldade, esta conseguiu ser superada através de uma prévia análise dos algoritmos a serem usados no trabalho aliada ao teste sucessivo desses algoritmos, e identificação e análise da sua performance.