U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

27 de Abril de 2016

EASY PILOT: SISTEMA DE NAVEGAÇÃO

Conceção e Análise de Algoritmos

Turma 6, Grupo C
José Aleixo Peralta da Cruz
up201403526@fe.up.pt

José Carlos Alves Vieira
up201404446@fe.up.pt

Renato Sampaio Abreu
up201403377@fe.up.pt

Índice

1. Descrição do tema 3
2. Formalização do problema 3
   2.1. Dados de entrada 3
   2.2. Limites da aplicação 3
   2.3. Situações de contorno 3
   2.4. Resultados esperados 4
3. Descrição da Solução 5
   3.1. Algoritmo aplicado em termos de consulta 5
   3.2. Aplicação do peso às arestas 5
   3.3. Algoritmo para obter o caminho mais curto 5
   3.4. Algoritmo para encontrar o ponto de interesse mais próximo 6
   3.5. Existência de zonas inacessíveis 6
   3.6. Existência de portagens 7
   3.7. Pesquisa exata de *strings*: algoritmo KMP 7
   3.8. Pesquisa aproximada de *strings*: algoritmo Wagner-Fischer 7
4. Diagrama de classes 8
5. Contribuição 9
6. Conclusão 10
7. Referências 11

1. Descrição do tema

No âmbito da unidade curricular Conceção e Análise de Algoritmos, foi atribuída ao nosso grupo a tarefa de elaborarmos um programa em C++ que, através da implementação de grafos, funcionasse de forma semelhante a uma interface de GPS.

Neste sistema de navegação, apelidado "*Easy Pilot*", o utilizador tem à sua disposição um mapa, no qual pode selecionar os pontos de início e de final de uma viagem. O programa encarrega-se de encontrar o caminho mais curto possível entre esses dois pontos, que deve evitar zonas inacessíveis. Fica ainda ao alcance do utilizador adicionar pontos de interesse que queira visitar durante essa viagem, sendo que o programa adapta o trajeto para incluir esses locais.

2. Formalização do problema

2.1. Dados de entrada

Como dados de entrada no programa são utilizados mapas que representam localizações reais, obtidos no OpenStreetMaps (OSM - www.openstreetmaps.org). Os ficheiros exportados a partir do OSM estão escritos em XML, pelo que se usa um *parser* para transformar essa informação em texto, de forma a facilitar a leitura dos dados.

O *parser* origina três ficheiros de texto, cada um contendo informações necessárias para a elaboração de um grafo que representa a área geográfica extraída, segundo a seguinte tabela.

Ficheiro	*Informação do grafo (estrutura)*	*Equivalência real*
FicheiroA.txt	Nós (id, latitude, longitude)	Locais
FicheiroB.txt	Arestas (id, nome, sentido)	Estradas
FicheiroC.txt	Conexões (idAresta, idNó1, idNó2)	Conexões entre estradas

O grafo G obtido pode ser declarado da seguinte forma: $G = (N, A)$. N é o conjunto dos pontos geográficos relevantes ao mapa e A é o conjunto das secções de estrada que interligam esses pontos.

2.2. Limites da aplicação

No programa são usados mapas que representam uma área geográfica relativamente pequena, pois o grafo torna-se exponencialmente mais complexo quanto maior for o mapa usado.

Assim, uma limitação desta aplicação é que nem sempre será encontrado um caminho que siga todas as condições estabelecidas. Isto acontece porque a rota que cumpriria essas restrições está fora da área coberta pelo programa.

2.3. Situações de contorno

Com o objetivo de adicionar funcionalidades e permitir alguma liberdade no percurso que o utilizador seleciona, a aplicação possibilita a adição de de pontos de interesse ao trajeto, desde que estes sejam possíveis de alcançar a partir do ponto inicial atual. É importante referir que neste caso o utilizador tem a opção de escolher se pretende fazer

o percurso selecionando visitando os pontos de interesse de acordo com a ordem que foram introduzidos ou se prefere que o percurso resultante seja o mais curto possível, isto é que a soma total dos pesos do percurso seja minimizada.

Por outro lado com o objetivo de simular um cenário real, em que uma zona da estrada pode-se encontrar em obras ou, por qualquer outra razão inacessível, inclui-se um conjunto de zonas inacessíveis ao grafo, definidas inicialmente no ficheiro de cada cidade. Contudo, o utilizador tem a opção de adicionar novas zonas inacessíveis ou até remover as existentes. O programa necessita então de verificar os nodes que não podem acedidos, consoante estas zonas, e por consequência delinear novas rotas.

Cada mapa tem locais já definidos que representam pórticos. Nestes, as respetivas conexões adjacentes implicam um peso acrescido ao já existente. Desta forma tentamos representar a forma de funcionamento real dos pórticos.

2.4. Resultados esperados

Esta aplicação tem como funcionalidade auxiliar o utilizador na escolha do melhor percurso numa determinada localização.

Assim, o utilizador pode selecionar um dos mapas disponíveis, acedendo depois a um menu de navegação. Com isto, o utilizador pode escolher as diferentes opções de navegação. Em relação a estas opções, o utilizador pode selecionar tanto o ponto de origem como de destino, adicionar ou remover pontos de interesse (pontos que serão visitados no percurso). Além disso, é também dada a opção de remover zonas inacessíveis ou adicioná-las, o que obviamente pode implicar mudanças na conetividade do grafo e alterações drásticas no percurso final.

Posto isto, em relação ao percurso, o utilizador ainda pode escolher se pretender efetuar um percurso mais econômico, isto é, minimizar o custo da viagem (evitando pórticos) mas como consequência aumentar o tempo de viagem, ou o percurso mais rápido, ou seja, minimizar o tempo de viagem mas aumentar o custo. A opção “Path Visualization” permite então visualizar o percurso de acordo com a informação dada pelo utilizador.

3. Descrição da Solução

3.1. Algoritmo aplicado em termos de consulta

A estrutura do grafo que representa o mapa do OpenStreetMaps tende a ter maior profundidade que largura. Isto porque há um vértice para cada oscilação significativa da longitude ou da latitude ao longo de um determinada rua, mesmo que nela não hajam cruzamentos nem entroncamentos. Assim uma grande parte dos vértices tem apenas um vértice-filho, já que representam a continuação da mesma estrada e não a interseção com outras.

Na aplicação, o número de nós do grafo não ultrapassa os 700, pelo que a solução nunca estará muito afastada do ponto de partida. A distância entre esses dois pontos não será fixa, nem será necessário procurar toda a árvore.

Com base nestas predições, o programa usa como algoritmo de consulta da informação de cada nó a busca em largura, em detrimento da busca em profundidade. A complexidade da busca em largura no pior cenário possível, num determinado grafo $G = (V, A)$ é a seguinte:

- Complexidade temporal: $O(|A|)$
- Complexidade espacial: $O(|V|)$

A busca em profundidade tem uma complexidade espacial média menor que a busca em largura, mas nem sempre obtém o caminho mais curto entre a origem e a solução.

3.2. Aplicação do peso às arestas

A principal função do nosso programa é encontrar o menor percurso entre dois locais. No grafo importado a partir do OpenStreetMaps, as arestas não tinham peso, pelo que a distância geográfica entre dois pontos não tinham influência no cálculo da trajetória.

Seja a aresta A a que liga o nó $N1$ ao nó $N2$, o peso atribuído a essa aresta é:

$$W_A = floor(\sqrt{(longitude_{N1} - longitude_{N2})^2 + (latitude_{N1} - latitude_{N2})^2} * 10000)$$

Ou seja, a distância entre os dois pontos, em termos de latitude e longitude, multiplicada por um fator 10^4 e arredondada ao menor inteiro mais próximo. A multiplicação pelo fator visa aumentar a grandeza do peso, já que a diferença entre latitudes e longitudes encontra-se na casa das centésimas do grau.

3.3. Algoritmo para obter o caminho mais curto

Ao se aplicar peso a cada aresta do grafo, torna-se necessário aplicar um algoritmo que obtenha o caminho mais curto entre dois vértices, tendo em conta esse peso.

O algoritmo de Floyd-Warshall foi o que escolhemos para calcular esse caminho, umas vez que incorpora no seu cálculo o peso das arestas entre os diferentes vértices.

Calcula-se o caminho mais curto do nó da origem para todos os outros nós do grafo, e finalmente seleciona-se, desde o nó origem, o nó de destino, retornando o caminho mais curto entre esses dois pontos.

Este é um algoritmo baseado em programação dinâmica, com complexidade $O(|V^3|)$.

3.4. Algoritmo para encontrar o ponto de interesse mais próximo

O vetor que contém os pontos de interesse está ordenado segundo a ordem pela qual o utilizador os inseriu. Caso o utilizador deseje que os pontos de interesse sejam visitados de modo a percorrer o menor caminho, é necessário ordenar esse vector para que esses pontos de interesse fossem visitados do menos distanciado para o mais distanciado. Para isso, criou-se um vetor auxiliar que é retornado por uma função.

Pseudo-código:

```
1: repetir enquanto (PointsOfInterest.size() for diferente de vetorAuxiliar.size())
2:
3:    for (todos os pontos de interesse)
4:       verificar qual o ponto de interesse mais próximo do último ponto processado;
5:       inserir no vetorAuxiliar;
6:
7:    return vetorAuxiliar;
```

De seguida pode-se aplicar novamente o algoritmo de Floyd-Warshall para obtenção do caminho mais curto, entre o ponto de partida e o ponto de chegado, mas desta vez passando por todos os pontos de interesse.

3.5. Existência de zonas inacessíveis

As zonas inacessíveis correspondem a conexões (edges), definidas entre dois vértices, que estão bloqueadas, ou seja, a escolha do percurso ideal não poderá utilizar essas conexões.

Posto isto, foi necessário alterar a forma de funcionamento “normal” dos algoritmos implementados ao longo das aulas práticas, visto que a outra opção seria eliminar do grafo as conexões bloqueadas e adicioná-las novamente quando o utilizar assim o quisesse.

Desta forma, para algoritmos como Depth-first search ou Breadth-first search foi necessário verificar que apenas era feita a pesquisa em novos vértices se estes ainda não tivessem sido visitados e a a conexão para esse vértice não esteja bloqueada. Caso o último ponto se verificasse, a conexão e o respetivo vértice de destino desta seriam ignorados. No caso de algoritmos de “shortest path finding”, tal como Floyd-Warshall, basta considerar que se a aresta estiver bloqueada o seu peso total é INT_INFINITY, o que implica que essa aresta não poderá ser considerada para constituir o “path” do percurso mais rápido.

Através das alterações implementas anteriormente, e com recurso ao Breadthfirst search conseguimos facilmente verificar a conetividade do grafo e com isso restringir tanto a escolha de pontos de destino e pontos de interesse específicos, pois podiam estar em zonas ao qual, a partir do ponto inicial, não fossem possível atingir. Desta forma, o percurso existirá sempre devido à execução das verificações

anteriormente referidas, as quais obrigam a que todos os pontos escolhidos sejam válidos, isto é, alcançáveis.

3.6. Existência de portagens

Aos mapas importados foram adicionadas portagens em certos locais, para que o utilizador tivesse a oportunidade de escolher entre dois tipos de caminho: o mais rápido ou o mais barato. Estas portagens comportam-se como os pórticos que existem atualmente nas SCUT portuguesas, ou seja, só quando o carro passa no local do pórtico é que é cobrada a quantia monetária.

No grafo, esta funcionalidade traduz-se num aumento do peso das arestas que saem do vértice que foi declarado como sendo uma portagem. Esse aumento é igual ao preço da portagens em cêntimos.

O modo predefinido é o de encontrar o caminho mais curto, pelo que abrir o programa, o grafo terá sempre os pesos originais. Se o utilizador decidir escolher o caminho mais barato os pesos são alterados e os algoritmos escolhem o melhor caminho em função do novo peso. No entanto, se a alternativa a passar pela portagem for mais custosa em termos de distância (custo do combustível), do que pagar a própria portagem, o caminho mais barato poderá passar por uma portagem.

É possível ainda o utilizador escolher se quer que o caminho encontrado englobe portagens ou não. Caso não seja possível passar por pontos de portagem, o caminho é calculado evitando esses pontos.

3.7. Pesquisa exata de *strings*: algoritmo KMP

Para tornar possível a pesquisa de ruas e distritos na nossa aplicação, implementámos algoritmos de pesquisa de *strings*.

Na pesquisa exata de *strings*, recorremos ao algoritmo Knuth–Morris–Pratt (KMP). Para pesquisar ruas, aplicamos o algoritmo usando como argumentos a string introduzida pelo utilizador e todos os nomes de ruas disponíveis, até encontrar uma correspondência.

O algoritmo KMP tem complexidade temporal $O(n + m)$, em que n é igual, no nosso programa, ao tamanho do nome da rua a ser procurado e m ao tamanho do texto de procura. A soma destas complexidades provém do pré-processamento do padrão, que determina a função de deslocamento para equiparar o texto com o padrão, que é executado antes da função que efetua a comparação.

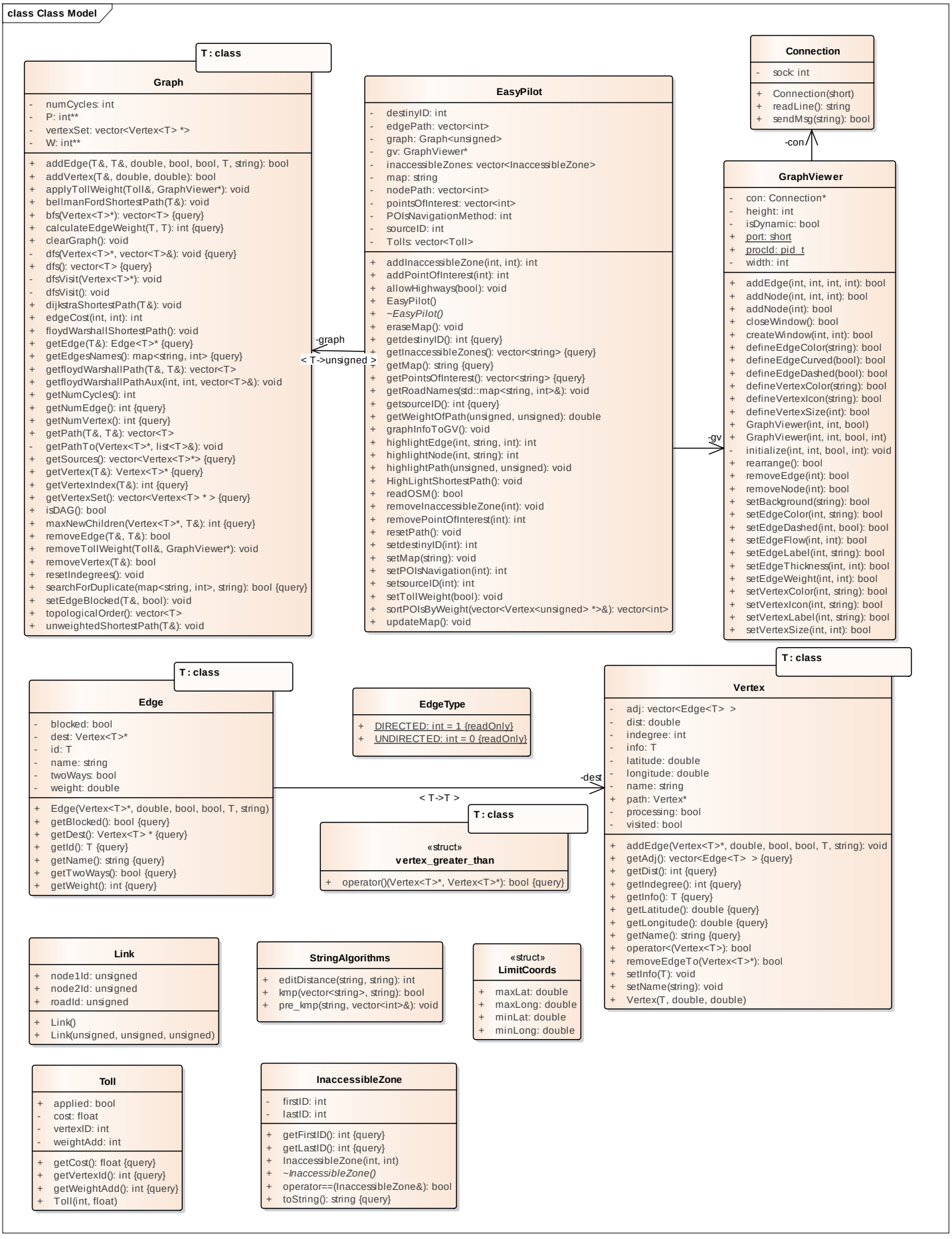
3.8. Pesquisa aproximada de *strings*: algoritmo Wagner-Fischer

Para a pesquisa aproximada de *strings* o algoritmo introduzido foi o de Wagner-Fischer. Este algoritmo recorre à programação dinâmica para calcular a distância de Levenshtein entre duas strings. A distância de Levenshtein é o número de alterações que se tem de fazer a uma string para que se transforme noutra.

Na nossa interface, definimos como valor máximo desta distância 3, para que as sugestões de string não divirjam muito da original.

Este algoritmo tem complexidade temporal e espacial $O(n * m)$, sendo n o tamanho do padrão a procurar e m o tamanho do texto onde se efetua a procura.

4. Diagrama de classes



5. Contribuição

Todos os membros do grupo contribuíram equitativamente para o desenvolvimento deste projeto.

O desenvolvimento do coração da aplicação foi trabalhado por todo o nosso grupo, incluindo isto a extração de dados dos ficheiros de texto, a criação do grafo a partir dos dados, a implementação de algoritmos relativos ao próprio grafo e a interface do programa.

Mais particularmente, o José Carlos tratou da adição da funcionalidade dos pontos de interesse, o Renato Abreu da circunscrição de zonas inacessíveis e o José Aleixo da influência das portagens.

Na segunda parte do projeto, trataram o José Carlos e o Renato Abreu da implementação dos algoritmos de correspondência de strings na interface e o José Aleixo da atualização do relatório.

6. Conclusão

O objetivo deste trabalho era desenvolver estratégias para determinar o menor caminho entre dois pontos, considerando diversos fatores externos. Esse problema foi resolvido recorrendo ao algoritmo de Floyd-Warshall.

Tivemos alguma dificuldade em lidar com as funcionalidades que deveriam ser implementadas na nossa aplicação, mas ao adaptarmos a estrutura do grafo às nossas necessidades, conseguimos ultrapassar esses obstáculos.

Ao nível da correspondência de strings, utilizando os algoritmos KMP e Wagner-Fischer, conseguimos implementar uma função de pesquisa na nossa interface.

7. Referências

Department of Information and Computer Science - University of California, Irvine. *ICS 161: Design and Analysis of Algorithms.* 15 de Fevereiro de 1996. https://www.ics.uci.edu/~eppstein/161/960215.html (acedido em 23 de Abril de 2016).

Rosetti, Rosaldo, e Ana Paula Rocha. *Material de Apoio para as aulas de Conceção e Análise de Algoritmos.* s.d.

Wikipedia. *Floyd–Warshall algorithm.* 1 de Abril de 2016. https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm (acedido em 24 de Abril de 2016).