FACULDADE DE ENGENHARIA DA
UNIVERSIDADE DO PORTO

CONCEPÇÃO E ANÁLISE DE ALGORITMOS

À procura de estacionamento

Beatriz Baldaia up201505633@fe.up.pt
Daniel Machado up201506365@fe.up.pt
Nelson Costa up201403128@fe.up.pt
Sofia Alves up201504570@fe.up.pt

Abril 7, 2017

À procura de estacionamento

Beatriz Baldaia, Daniel Machado, Nelson Costa, Sofia Alves
Faculdade de Engenharia da Universidade do Porto
email: {up201505633, up201506365, up201403128, up201504570}@fe.up.pt

Abstração- O principal objetivo deste projeto é resolver o problema de planear o trajeto mais curto ou o mais barato de um destino a um parque de estacionamento e deste a um ponto de interesse, intersetando-se ou não uma bomba de gasolina para abastecer. Assim, o mapa/rede de estradas pode ser abstraído por um grafo e a solução do problema será a procura de um caminho ótimo entre os dois nós, origem e destino. Para tal recorremos á pesquisa em profundidade (para avaliar a conetividade) e ao conhecido algoritmo Dijkstra, no entanto, como o utilizador impõe condições ao trajeto, este algoritmo teve de ser adaptado as características do percurso.

I. INTRODUÇÃO

De uma forma mais detalhada, este projeto primeiro calcula o caminho ótimo a carro de uma origem a um parque de estacionamento (parquímetro ou garagem de preços variados), sendo que o parque segue dois critérios de escolha: ser o parque mais próximo do destino ou ser o parque mais barato e a uma distância não superior á especificado pelo utilizador. Segundo, o programa calcula o melhor caminho (o mais curto) do parque ao destino. Como este segundo caminho é realizado a pé, o sentido da rua não interessa, visto que não é de um carro que se trata. O nosso trabalho também aborda a opção de antes do estacionamento se intersetar uma bomba de gasolina a fim de se abastecer o veiculo. Tal implicará um processo mais robusto que o anterior de modo a obter-se sempre o melhor itinerário.

Um trajeto consiste num conjunto de pontos, também chamados por vértices/nós, ligados por aresta, representados no problema por ruas. Esta estrutura é, portanto, normalmente abstraída pelo conceito matemático de grafo. Todos os nós representam um local acessível ao publico e têm associado uma posição no plano xOy. Todos as arestas representam uma rua entre dois vértices, podem ser uni ou bidirecionais e tem um peso, isto é, distância em metros, importante para o calculo do itinerário. Este relatório encontra-se organizado por secções sendo que a secção II engloba os aspetos e definições das signas que serão referidas ao longo do relatório, a III e IV a explicação e estudo da pesquisa em profundidade e do algoritmo de Dijkstra, a V a performance de ambos os algoritmos e, finalmente, a VI a conclusão.

II. LEGENDA

Antes de prosseguirmos, é importante definir os elementos centrais da discussão:

- G representa o grafo de ligações entre os diversos pontos.
- V representa o conjunto de nós (ou vértices), ou seja, pontos (de interesse ou não) no mapa.
- E representa o conjunto de arcos (ou arestas), ou seja, as ruas do mapa em questão.
- v_i representa um nó (ou vértice).
- (x_i, y_i) representa as coordenadas x,y do nó v_i.

- $e_{i,j}$ representa o arco entre o nó v_i e o nó v_j.
- $dist_{i,j}$ representa a distância de um nó v_i a um nó v_j podendo estes não estar diretamente ligados por uma só aresta (distância do caminho ótimo calculado).
- p_i representa o preço (em euros por hora) do nó v_i que corresponde a um parque de estacionamento.
- d_{max} representa a distância máxima, dada pelo utilizador, do parque de estacionamento ao destino.
- $d_{i,j}$ - Representa a distancia do arco $e_{i,j}$ e é calculada a partir da expressão $\sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}, \forall ei, j \in G$.
- $A_{i,j}$ representa o conjunto de nós de um determinado caminho que liga os nós v_i e v_j.
- $B_{i,j}$ representa o conjunto de arestas de um determinado caminho que liga os nós v_i e v_j.
- $path_{vi}$ representa a sequência de nós que constituem o trajeto ótimo.

A. Input

O input deste modelo é o grafo que representa a rede de estradas que será analisada como descrito anteriormente. Este grafo será submetido a processamento prévio de forma a poderem ser aplicados os algoritmos, descritos na subsecção *D*.

B. Output

O output do modelo é a sequencia de nós que correspondem ao caminho ótimo. Enfrentamos diferentes critérios de seleção do caminho ótimo, nomeadamente a escolha ou do parque mais próximo do destino ou do parque mais barato, não se excedendo, no entanto, uma distância máxima dada entre o parque e o destino. Também temos a opção de passar ou não por uma bomba de gasolina antes de estacionar, o que também vai influenciar o calculo do trajeto. Assim estamos perante quatro possíveis definições para a função objetivo.

C. Função objetivo

O objetivo é encontrar o caminho ótimo que conecta os nós v_i e v_j. Como mencionado anteriormente, este relatório considera quatro possíveis definições de função objetivo:

- Minimizar $\Sigma_{e_{k,k+1} \in B_{i,j}} \mathrm{d}_{\mathrm{k,k+1}}$, isto é, minimizar a distância total entre os pontos v_i e v_j, parque de estacionamento e destino respetivamente.
- Minimizar $p_i \wedge \left(\Sigma_{e_{k,k+1} \in B_{i,j}} \mathrm{d}_{\mathrm{k,k+1}}\right) \leq d_{max}$, isto é, minimizar o preço a gastar no parque de estacionamento (v_i) e ao mesmo tempo minimizar a distância deste ao destino (v_j), não podendo nunca esta distância ultrapassar a máxima dada (d_{max}).
- Minimizar $\Sigma_{e_{k,k+1} \in B_{i,j}} \mathrm{d}_{\mathrm{k,k+1}} \wedge \Sigma_{e_{k,k+1} \in B_{a,b}} \mathrm{d}_{\mathrm{k,k+1}}$, isto é, minimizar a distância total entre os pontos v_i e v_j, parque de estacionamento e destino respetivamente, e minimizar a distância total entre os pontos v_a e v_b, bomba de gasolina e parque de estacionamento respetivamente.
- Minimizar $\left(\mathrm{pi} \wedge \left(\Sigma_{e_{k,k+1} \in B_{i,j}} d_{k,k+1}\right) \leq d_{max}\right) \wedge \Sigma_{e_{k,k+1} \in B_{a,b}} \mathrm{d}_{\mathrm{k,k+1}}$, isto é, minimizar o preço a gastar no parque de estacionamento (v_i) ao mesmo tempo que se minimiza a distância deste ao destino (v_j), não podendo nunca esta distância ultrapassar a máxima dada (d_{max}), e também minimizar a distância total entre os pontos v_a e v_b, bomba de gasolina e parque de estacionamento respetivamente.

Obvio que também faz sentido minimizar sempre a distância entre a origem e o parque de estacionamento calculado (aplicando-se a condição do primeiro ponto).

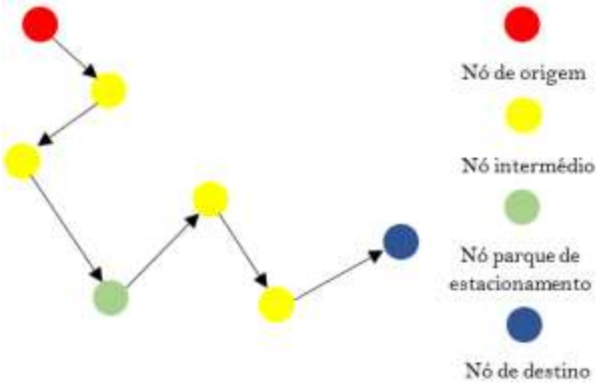


Figura 1: Exemplo da rede de estradas

D. Adaptação do grafo ao problema enfrentado

Nos quatro pontos anteriores é sempre necessário calcular o percurso do utilizador do parque ao destino que por ser realizado a pé torna-se irrelevante o sentido das ruas. Quando lemos os ficheiros com as informações das arestas/ligações entre nós (recolhido a partir de *OSM2TXT Parser*), cada está associada a uma rua, cuja informação é também filtrada de um ficheiro. Cada rua tem um id, nome e um booleano que indica se a rua é de um ou dois sentidos. Se uma rua é de dois sentidos, ambos os vértices que constituem uma aresta associada a essa rua vão guardar essa mesma aresta num vetor de arestas (atributo *adj* da classe *Vertex*). Deste modo, seria de prever que se a rua fosse de um só sentido apenas o nó inicial da aresta guardaria a mesma e esta guardaria o nó de destino. No entanto, neste caso, o vértice de destino não teria acesso aos nós precedentes que se ligam a ele o que nos impossibilita considerar o livre movimento bidirecional do utilizador quando se desloca a pé. Assim, o que fazemos é que mesmo que a rua seja de um só sentido todos os nós que são a base das arestas que constituem estas ruas guardam no seu vetor *adj* as arestas, mesmo que não sejam o nó origem das mesmas. Mas se tal acontece como é que sabemos quando é que um carro está a andar em contramão? Bem, o que fazemos é acrescentar mais um atributo ás arestas denominado *real* (um booleano) que se for true significa que o nó que a guarda pode ser nó origem da aresta e se for false é exclusivamente nó destino da aresta.

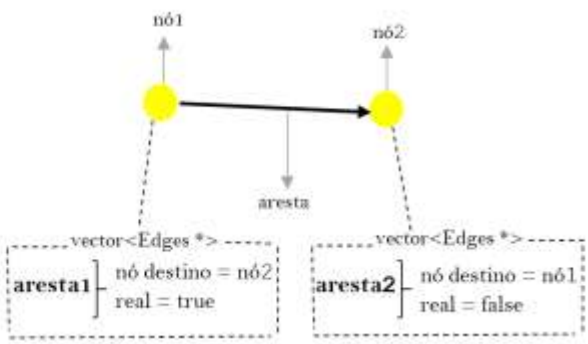


Figura 2: Exemplo do contorno do problema de arestas unidirecionais

III. ALGORITMO DE DIJKSTRA

Algoritmo 1 Algoritmo de Dijkstra

```
void Dijkstra (Vertex s) {
for (Vertex v : vertexSet) {
v.path = null; v.dist = INFINITY;}
s.dist = 0;
PriorityQueue<Vertex> q =
new PriorityQueue<Vertex>();
q.insert(s);
while ( !q.isEmpty() ) {
  Vertex v = q.extractMin();
    for(Edge e : v.adj) {
      Vertex w = e.dest;
       if (v.dist + e.weight < w.dist) {
         w.dist = v.dist + e.weight;
         w.path = v;
         if (w.queueIndex == -1)
           q.insert(w);
         else
           q.decreaseKey(w);
       }}}}
```

A. Complexidade temporal e espacial

O algoritmo de Dijkstra guarda numa fila de prioridade os nós que serão processados, por isso o tamanho desta fila nunca ultrapassará o número de vértices do grafo, assim, a complexidade espacial é $O(|V|)$. No inicio desta fila encontram-se os vértices com menor valor *dist* (distância desde a origem até a esse vértice).

O algoritmo é normalmente implementado usando-se uma *binary heap* para ordenar os nós que serão processados pela sua menor distância (*dist*). A inserção de um novo nó na binary heap apresenta complexidade temporal de $O(\log(N))$, sendo N o número de elementos na pilha. A função **decreaseKey** diminui o valor da chave de um nó após a atualização da pilha. Esta operação assume que o valor da nova chave é menor ou igual ao valor atual e é também de complexidade $O(\log(N))$.

O ciclo **for** que abrange as linhas 2 e 3 executa $|V|$ vezes, visto que passa por todos os nós do grafo. O ciclo **while** ocorre no mínimo $|V|$ vezes, visto que cada nó é colocado pelo menos uma vez na fila de prioridade q. Para todos os nós que serão processados (que estão na fila q) todas as arestas que deles saem são analisadas no ciclo **for** da linha 10 a 19. Para cada aresta que está a ser analisada, no pior dos casos a função **decreaseKey** é sempre invocada e assim o ciclo **for** toma uma complexidade temporal de $O(\log(|V|))$. Este ciclo **for** é executado $|V|.|E|$ vezes porque todas as arestas de um vértice são visitadas e no pior dos casos isto é repetido para todos os vértices. Se o nó origem de uma aresta só é inserido uma vez na fila, este ciclo ocorre $|E|$ vezes. Assim, a complexidade do algoritmo é:

$$O(|V| \,.\, \log(|V|) + |V| \,.\, \log(|V|) + |E| \,.\, \log(|V|))$$

E se $|E|>|V|$, a complexidade é simplificada para:

$$O(|E| \,.\, \log(|V|))$$

IV. PESQUISA EM PROFUNDIDADE

Algoritmo 2 Pesquisa em Profundidade

```
1:  class Graph { …
2:  void dfs() {
3:     for (Vertex v : vertexSet)
4:        v.visited = false;
5:     for (Vertex v : vertexSet)
6:         if (! v.visited)
7:             dfs(v);
8:   }
9:   void dfs ( Vertex v ) {
10:    v.visited = true;
11:     for (Edge e : v.adj)
12:        if( !e.dest.visited )
13:           dfs ( e.dest );
14:  }
15:};
```

A. Complexidade temporal e espacial

Tal como referido nas aulas, nesta pesquisa as arestas são exploradas a partir do vértice v mais recentemente descoberto/visitado e que ainda tenha arestas a sair dele. Quando todas as arestas de v forem exploradas retorna-se para o vértice precedente a este de forma a explorar-se as suas restantes arestas. Se se mantiverem vértices por descobrir, um deles é selecionado como a nova fonte e o processo de pesquisa continua a partir daí. Todo o processo é repetido até todos os véticés serem descobertos (terem o atributo *visited* a true).

Como este algoritmo não aloca espaço extra, a complexidade espacial é constante ($O(1)$). O ciclo for que abrange as linhas 3 e 4 percorre todos os vértices do grafo, daí a sua complexidade ser $O(|V|)$, sendo V o número de vértices do grafo. Na função dfs() há mais um ciclo for que também percorre todos os vértices do grafo tem, por isso, também uma complexidade O(|V|). Como este algoritmo termina apenas quando todos os vértices forem visitados, a invocação da função dfs(Vertex v) realizada na linha 7 será feita para todos os vértices, ou seja, ocorrerá |V| vezes. O overload da função dfs que tem como parâmetro Vertex v tem no seu corpo de código um ciclo **for** que sofre |E| (E = número total de arestas do grafo) iterações, isto porque, como a pesquisa em profundidade só termina depois de todos os vértices serem visitados, todas as arestas do grafo serão visitadas uma e uma só vez. Assim, a complexidade temporal deste ciclo é $O(|E|)$. Deste modo, como neste algoritmo num ciclo de complexidade temporal $O(|V|)$ ocorre a chamada de uma função de complexidade temporal $O(|E|)$, a complexidade temporal deste algoritmo é:

$$O(|V| + |E|)$$

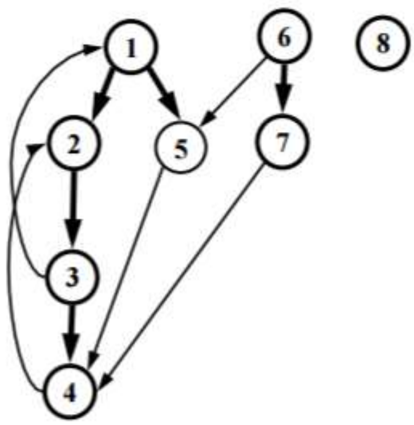


Figura 3: Exemplo em grafo dirigido (vértices numerados por ordem de visita e dispostos por profundidade de recursão)

V. PERFORMANCE DOS ALGORITMOS

Nesta parte vamos comparar a performance dos algoritmos utilizados, bem como complementar a evolução teórica da sua evolução temporal. Neste caso concreto, vamo-nos focar apenas no algoritmo de Dijkstra. Trata-se de um algoritmo cujo objetivo é encontrar o caminho mais curto entre os nós do grafo. Considera o conjunto dos S menores caminhos, iniciando com um vértice inicial I. Depois, a cada iteração, o algoritmo vai procurar nos vértices adjacentes o que possui uma menor distância relativa a I e adiciona-o a S.

A complexidade do algoritmo é $O(|V| \,.\, \log(|V|) + |V| \,.\, \log(|V|) + |E| \,.\, \log(|V|))$. Sendo que, se $|E|>|V|$, a complexidade é simplificada para $O(|E| \,.\, \log(|V|))$. Assim, torna-se bastante óbvio o comportamento do gráfico relativo ao processamento do algoritmo em função do número de nós. Quando o número de nós ultrapassa 34, verifica-se um aumento no declive do gráfico, ou seja, passa-se de uma simples situação logarítmica, $O(|E| \,.\, \log(|V|))$, para uma soma de logaritmos. Isto verifica-se devido à mudança do equilíbrio arestas vs nós, passando a haver um número de nós superior ao número de arestas.

Figura 4: Performance temporal do algoritmo em função do número de nós do grafo

VI. CONCLUSÃO

A. Funções principais

- **void Graph::dijkstraShortestPathByFoot (Vertex * v)**
  Algoritmo de Dijkstra aplicado ao caminho pedestre. Tem como

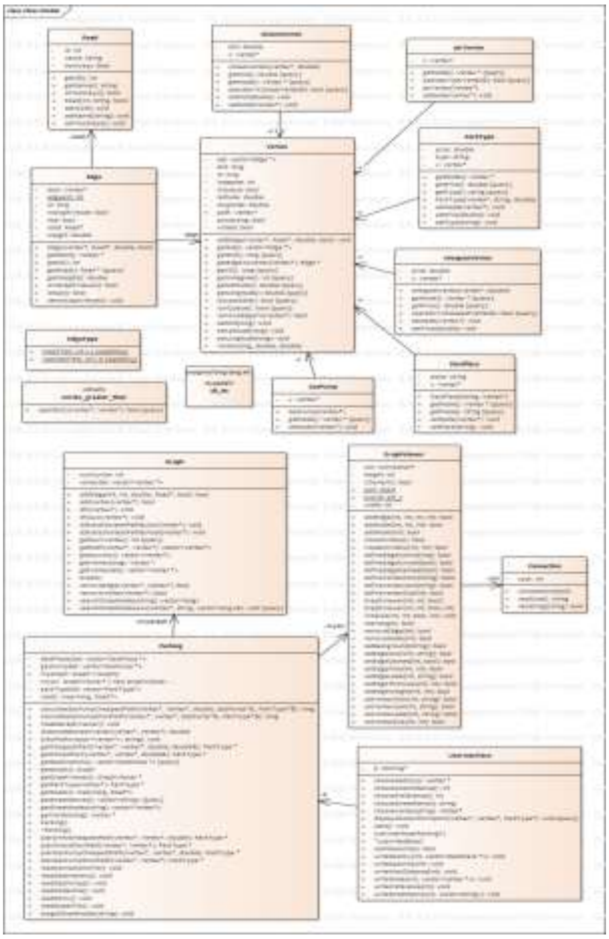
parâmetro o vértice de origem (a partir do qual começa a ser calculado o caminho mais curto).

- **void Graph::dijkstraShortestPathByCar (Vertex * v)**
  Algoritmo de Dijkstra aplicado ao caminho realizado de carro. Tem como parâmetro o vértice de origem (a partir do qual começa a ser calculado o caminho mais curto).
- **vector<Vertex *> Graph::getPath(Vertex * origin, Vertex * dest)**
  Após a aplicação do algoritmo Dijkstra recorremos ao getPath para obtermos a sequência de nós, desde a origem (Vertex * origin) até ao destino (Vertex * dest), do caminho mais curto.
- **void Graph::dfs(Vertex * origin)**
  Algoritmo de pesquisa em profundidade para avaliar a conetividade do grafo. Tem como parâmetro o vértice a partir do qual se vai testar a conetividade, isto é, ver quais os pontos o utilizador consegue chegar de carro partindo de Vertex * origin.
- **ParkType * Parking::planDirectShortPath (Vertex * src, Vertex * dest)**
  Função que planeia o caminho de src até dest estacionando o carro no parque de estacionamento mais próximo de dest. Retorna o parque em que terá de estacionar o carro.
- **ParkType * Parking::planDirectCheapestPath (Vertex * src, Vertex * dest, double maxDist)**
  Função que planeia o caminho de src até dest estacionando o carro no parque mais barato e que não se encontra a uma distância superior a maxDist. Retorna o parque em que terá de estacionar o carro.
- **ParkType * Parking::planGasPumpShortPath (Vertex * src, Vertex * dest)**
  Função que planeia o caminho de src até dest estacionando o carro no parque de estacionamento mais próximo de dest e, antes de estacionar, parar numa bomba de gasolina. Retorna o parque em que terá de estacionar o carro.
- **ParkType * Parking::planGasPumpCheapestPath(Vertex * src, Vertex * dest, double maxDist)**
  Função que planeia o caminho de src até dest parando numa bomba de gasolina para abastecer e estacionando o carro no parque mais barato e que não se encontra a uma distância superior a maxDist. Retorna o parque em que terá de estacionar o carro.

B. Restrições

- Qualquer aresta tem um nó em cada extremo seu;
- Um carro tem que andar no sentido imposto pela rua em que se encontra caso esta seja unidirecional;
- Tanto o carro como o utilizador têm de andar sobre arestas do grafo;
- No caso de estarmos a planear o caminho mais barato, a distância entre o parque de estacionamento e o destino não pode ultrapassar a dada pelo utilizador;
- Se antes de estacionar o utilizador quiser abastecer numa bomba de gasolina terá ir a uma que lhe permita ir de carro até ao parque de estacionamento alvo;
- O ponto de origem, o ponto de destino (ponto de interesse), os parques de estacionamento e as bombas de gasolina têm de ser vértices do grafo.

C. Diagrama de classes



D. Principais dificuldades

O trabalho não foi nada de muito transcendente, mas inicialmente estávamos com problemas a ler os ficheiros porque os ids dos nós e ruas eram tão extensos que ultrapassavam a capacidade do type int, pormenor que nos estava a escapar. Para além de problemas recorrentes com o eclipse, o mais moroso provavelmente foi tratar dos menus.

E. Esforço dos elementos do grupo

Todos os elementos do grupo tiveram um papel importante na realização do projeto e todos trabalharam de forma igual, não havendo por isso nenhum que se destacasse pela positiva ou pela negativa.

F. Resultados obtidos

Os resultados são bastante satisfatórios porque como usamos o *OSM2TXT Parser* para converter ficheiros XML em ficheiros TXT, a fim de extrairmos as informações dos nós, ruas e ligações do mapa, conseguimos assegurar um plano mais realista, preciso e fiável. Orgulhamo-nos também da nossa interface pois é clara e de fácil compreensão, permitindo não só que o utilizador dê o seu input sem originar confusões como também obtenha uma resposta apresentável e legível. De referir ainda, é importante sublinhar o facto de que consideramos o movimento livre e bidirecional do utilizador quando caminha (a pé) do parque até ao destino, ou seja, considera-se qualquer aresta como bidirecional.

REFERÊNCIAS

[1] R. Sedgewick. Algorithms in C++ Part 5: Graph Algorithms, 3/E. New York, NY: Addison Wesley, 2002.

[2] M.A. Weiss. Data Structures and Algorithm Analysis in C++, 3/E. New York, NY: Addison Wesley, 2007.

[3] T. Cormen; C. Leiserson; R. Rivest; C. Stein. Introduction to Algorithms. Cambridge, MA: MIT Press, 2009