Faculdade de Engenharia da Universidade do Porto
Mestrado Integrado em Engenharia Informática e Computação

Conceção e Análise de Algoritmos

RELATÓRIO

Tema 10 - À Procura de Estacionamento Primeira Parte

TURMA 8, GRUPO 3
Henrique Sousa, up201906681@fe.up.pt
Mateus Silva, up201906232@fe.up.pt
Melissa Silva, up201905076@fe.up.pt

Índice

Introdução

O presente relatório constitui a primeira parte de um trabalho de grupo para a unidade curricular de *Conceção e Análise de Algoritmos*, no ano letivo de 2020/2021. Nele, e no âmbito dos algoritmos e estratégias lecionadas na UC, os alunos tiveram oportunidade de escolher um tema para o qual deveriam concecionar uma boa abordagem através de algoritmos em grafos.

O nosso grupo ficou com o Tema 10, designado *À procura de Estacionamento*.

Descrição do Tema

A contextualização do tema começa partindo dos serviços municipais do Porto, que desejam implementar um sistema que auxilie os condutores na pesquisa eficiente de lugares em parques de estacionamento da cidade, devido ao grande impacto que a utilização destes tem na fluência do tráfego automóvel.

A situação base é a seguinte: um condutor desloca-se de um ponto de partida P até um ponto de destino D, intencionando estacionar o seu carro num dos parques existentes nos arredores de D, pelo que devemos considerar que o condutor poderá ter a necessidade de executar determinadas tarefas durante tal percurso — isto obriga-o a passar por outros pontos na cidade, desde bombas de gasolina a cafés. O condutor tem, então, uma previsão do tempo de estacionamento que necessita.

Os preços de estacionamento variam de parque para parque, mas têm em comum dependenderem do tempo de utilização. Podem ainda assumir uma política de preços fixa ou dinâmica (preço varia de acordo com o número de vagas do parque).

O sistema deverá ser capaz de auxiliar o condutor no cálculo do melhor percurso a realizar até ao parque de estacionamento, quando este pretende otimizar:

- a distância percorrida até ao parque de estacionamento;
- o preço a pagar pelo estacionamento;
- a distância a percorrer a pé até ao ponto de destino D.

Deverá ser capaz de identificar zonas com pouca accessibilidade através do grafo obtido para os parques de estacionamento.

Problematização

Já com o contexto em mente, torna-se mais fácil dividir o problema em fases. Para o nosso problema, acreditamos ser ideal o seguinte plano de três fases, aumentando a complexidade do problema, tendo os três objetivos de otimização em mente.

1ª Iteração: Caminho Mais Curto Passando por Pontos Intermédios

Numa primeira fase, consideramos o percurso mais direto possível para o condutor: procuramos pelo caminho mais curto priorizando apenas minimizar a distância percorrida desde o ponto de partida P ao ponto de destino D.

Denominamos este primeiro percurso *direto* devido a não considerarmos quaisquer paragens intermédias, nem a distância feita desde o parque até ao destino D, tratando-o como se fizesse parte da distância percorrida em automóvel até ao parque de estacionamento— isto retira um grau de complexidade ao

cálculo, já que se trata de um dos três pontos que a nossa resolução pretende otimizar. Consideramos também as diferentes políticas de preços que podem existir nos diversos parques. Analisamos os preços de cada parque e optamos por aquele com o melhor compromisso entre distância e preço.

A análise dos preços passa pela consulta da política de preços efetiva num parque, podendo ser fixa ou variável - na última, o preço varia com o número de vagas no parque. Como já mencionado, o preço de utilização do parque depende sempre da duração do estacionamento. Com isto, temos em conta o número de vagas existentes em cada parque.

2ª Iteração: Caminho Mais Curto Passando por Pontos Intermédios Tendo em Conta Preços

Nesta fase, passamos a considerar que o condutor pode desejar passar por paragens intermédias. Caso queira, verificamos se este deseja estacionar num parque das suas imediações ou apenas passar por elas.

No caso de existirem paragens intermédias, estas serão ordenadas, com o objetivo de visitarmos primeiro os locais mais próximos de **P** antes dos que estão mais próximos de **D**. Isto resulta num caminho que geograficamente fará mais sentido, tendo uma sequência lógica que reduz a distância percorrida, contribuindo assim para o nosso objetivo.

Se o condutor pretender estacionar, procuramos um parque nas proximidades da paragem intermédia, não considerando a distância feita desde o parque até ao ponto intermédio a pé, tratando-a como se fizesse parte da distância percorrida em automóvel até ao parque (à semelhança do que se faz com o parque de estacionamento nas proximidades de **P** e **D** na 1ª Iteração).

De outra forma, apenas consideramos a distância do condutor do último ponto onde esteve até ao ponto intermédio e daí calculamos a próxima paragem.

Temos em conta os preços de cada parque e tentamos chegar a um bom compromisso, seguindo a mesma lógica descrita na 1ª Iteração.

3ª Iteração: Caminho Mais Curto Passando por Pontos Intermédios Tendo em Conta Preços e Caminho Mais Curto Feito a Pé ao Destino D

Nesta última iteração, deixamos de considerar o parque de estacionamento nas proximidades de **D** e o destino **D** como iguais, passando a analisar qual o parque de estacionamento que, para além de ter o preço mais atrativo ao condutor, constitui uma menor distância a pé até **D**.

Aplicamos a mesma lógica a todos os outros parques de estacionamento e ao ponto de interesse mais próximo deste.

Além disso, podemos, eventualmente, analisar se um parque de estacionamento à mesma distância de dois pontos de interesse ou se a distância entre dois pontos de interesse não será curta o suficiente para considerar fazer o caminho a pé entre eles.

Formalização do Problema

Como solicitado, passaremos agora à listagem e definição formal das diferentes componentes do problema, desde os dados de entrada e saída às nossas funções objetivo.

Dados de Entrada

Os dados de entrada compreendem-se como o conjunto de dados que, quando analisados, nos permitem obter os dados de saída através do seu tratamento e aplicação da nossa estratégia.

- **`T:`** estrutura de dados linear contendo o tempo que um condutor irá ficar em cada parque de estacionamento (paragens intermédias, caso opte por estacionar e o parque próximo de **`D`**), necessário para cálculos na segunda e terceira iterações;
- **`G(V, E):`** grafo dirigido pesado, onde:
  - **`V`** é um Vértice, representando um ponto na cidade, contendo:
    - Um conjunto de coordenadas que definem a posição do vértice:
      - **`Lat`**, a latitude do ponto;
      - **`Lon`**, a longitude do ponto;
      - **`Alt`**, a altitude do ponto;
    - **`Adj`**, um conjunto de arestas E que partem do vértice;
    - **`ID`**, um identificador do vértice que permite distingui-lo de outros parques;
    - **`NV`**, o número de vagas disponível no parque;
    - **`CP`**, possível variação de preço quando o parque tiver em vigor uma política de preços dinâmica — caso contrário, será nula;
    - **`C`**, o preço por hora de estacionamento.
  - **`E`** é uma aresta, representando um caminho de união entre dois pontos (vértices) da cidade, contendo:
    - **`W`**, o peso da aresta, representando a distância entre os dois pontos;
    - **`ID`**, o identificador da aresta;
    - **`dest`**, um apontador para o vértice **`V`**$_{\mathbf{d}}$ de destino da aresta.
- **`POI:`** estrutura de dados linear (por exemplo, um vetor) contendo todos os possíveis pontos de interesse de um condutor — bombas de gasolina, cafés, restaurantes, museus... qualquer lugar que possa ser uma paragém intermédia;

- **stops:** estrutura de dados linear contendo os pontos de interesse do condutor;
- **P**, o identificador do vértice/ponto da cidade de partida;
- **D**, o identificador do vértice/ponto da cidade de destino;
- **DT**, estrutura de dados linear contendo valores inteiros (1 ou 0) que determinam se o utilizador pretende, ou não, estacionar perto de uma paragem intermédia (respetivamente);
- W_1, o peso do critério de otimização da distância percorrida até ao parque de estacionamento;
- W_2, o peso do critério de otimização do preço ao pagar pelo estacionamento;
- W_3, o peso do critério de otimização da distância a percorrer a pé até ao ponto de destino **D**.

Dados de Saída

Os dados de saída compreendem-se como o conjunto de dados obtido através da análise e/ou processamento dos dados de entrada.

- C_f, a sequência ordenada de todas as arestas que fazem parte do caminho mais curto obtido até um parque nas imediações de **D**.
- P_f, a sequência ordenada de todas as arestas que fazem parte do percurso feito a pé entre o parque de estacionamento no final de C_f e **D**.
- **price**, preço final do estacionamento.

Restrições

Com o intuito de evitar erros e problemas desnecessários na implementação da nossa estratégia e algoritmos, decidimos elaborar um conjunto de restrições aos nossos dados.

Nos dados de entrada

- Não aceitamos tempos nulos ou negativos em **T**;
- Não aceitamos valores diferentes de 0 e 1 em **DT**;
- $\forall e \in E, W > 0$, dado que o peso de uma aresta representa a distância entre dois pontos no mapa;
- Não aceitamos valores fora de $[0, 1]$ em W_1, W_2 e W_3.

Nos dados de saída

- Dado que, de outra forma, o caminho obtido não seria válido:
  - O primeiro elemento de C_f tem de pertencer a **Adj(P)**;
  - O último elemento de P_f tem de pertencer a **Adj(D)**;

- $\forall e \in E, W > 0$, dado que o peso de uma aresta representa a distância entre dois pontos no mapa;

Funções Objetivo

O objetivo do programa consiste em minimizar as seguintes três componentes, representáveis pelas funções objetivo abaixo explícitas.

❖ **Distância Percorrida desde P ao Parque de Estacionamento nas proximidades de D**

$$f = \sum_{e \in P} W(e)$$

❖ **Preço do Parque de Estacionamento**

$$p = T \times (C + (NV \times CP))$$

❖ **Distância Percorrrida a Pé do Parque de Estacionamento a D**

$$pd = \sum_{e \in PD} W(e)$$

❖ **Otimização de Múltiplos Critérios**

$$F = w_1 * c_1 + w_2 * c_2 + w_3 * c_3$$

Sendo c_1 a distância percorrida até ao parque de estacionamento; c_2 o preço ao pagar pelo estacionamento e c_3 a distância a percorrer a pé até ao ponto de destino D.

Assim, a solução ideal do problema estará no resultado da minimização das funções mencionadas em cima.

Solução Proposta

Partindo dos dados definidos e descritos na secção anterior, estamos então mais aptos a apresentar a nossa proposta de solução, passado ainda por algumas das nossas decisões a respeito dos algoritmos e estratégia adotados.

Obtenção do Parque Ideal para Cada Paragem

Sejam...

- P o ponto de partida;
- D o ponto de destino;
- a o peso da distância percorrida de carro x;
- b o peso do preço do parque y;
- c o peso a distância percorrida a pé z;
- u a distância em linha reta a um parque de carro;
- w a distância em linha reta desde o parque de estacionamento a D;
- P_i os parques disponíveis;
- C_n o último ponto intermédio do percurso feito de carro desde P até ao último parque.

Denominem-se então P_1, P_2 e P_3 os parques com menor distância percorrida a pé, de carro, e de menor preço, respetivamente. Uma vez que tomaremos como mais importante a minimização da distância percorrida a pé, P_1 pode ser considerada uma solução ótima ou, pelo menos, boa.

Com o intuito de verificar a existência de uma melhor solução, faz-se o seguinte:

1. Itera-se os parques tal que P_j seja o parque com menor distância em linha reta a D , ainda não processado.
    a. Se $a \times x_2 + b \times y_3 + c \times w_j$ for maior do que o mínimo obtido até então, conclui-se que a solução já foi encontrada, restando apenas cessar o algoritmo;
    b. Se $a \times u_j + b \times y_j + c \times w_j$ for maior do que o mínimo então podemos saltar esta iteração;
    c. Calcula-se z_j. Se $a \times u_j + b \times y_j + c \times z_j$ for maior do que o mínimo obtido até então, podemos saltar a iteração corrente;
        i. Caso contrário, estamos perante uma possível melhoria da solução: calcula-se então x_j e analisa-se novamente a soma, agora através de $a \times x_j + b \times y_j + c \times z_j$ — se esta for menor do que o mínimo, esta torna-se a nova solução.

A utilização de w_j no algoritmo anterior permite encontrar a condição de paragem de forma mais eficiente, isto devido a ser evitado o cálculo da distância efetiva e o caminho numa das suas iterações. Utilizando os mínimos x_2 e y_3 e

iterando em ordem ascendente de w_j garantimos sempre que se a soma for maior do que a solução, todos os cálculos posteriores serão maiores.

Através das somas em **b.** e **i.**, diminui-se o número de cálculos: se qualquer uma delas for maior do que o mínimo, poderemos cessar essa iteração. Uma vez que o caminho será mais longo do que a distância em linha reta, se qualquer uma das somas for maior do que o mínimo, então poderemos terminar essa iteração.

Se nenhum destes passos intermédios provocar um efeito positivo, resta apenas calcular o valor do parque em questão efetivamente e verificar se este melhora a solução. Por fim, iterando em ordem crescente e em linha reta a D, permite terminar o algoritmo o mais rápido possível com o mínimo de distâncias e respetivos caminhos calculados.

Obtenção do Melhor Percurso Geral

Para pontos muito próximos uns dos outros, podem ocorrer os casos de o parque mais vantajoso ser o mesmo do ponto intermédio anterior ou não.

No primeiro caso, é trivial concluir que o utilizador deverá sempre ir primeiro aos dois pontos a pé e só depois regressar ao carro. Já no outro caso, ir a pé a ambos poderá também ser vantajoso, pelo que discutiremos agora como decidir o melhor parque para, efetivamente, estacionar.

Iterando os parques P_i, estando cada um destes associado ao ponto intermédio C_i e P_{n+1} ao destino D. Para cada par, sejam A a última paragem do utilizador — ou seja, para P_1, A é P, o local de partida e para P_i (com i $\neq 1$), A é C_{i-1}.

O peso do percurso para se estacionar em P_i é dado pela soma:

$$c \times d(C_{i-1}, P_{i-1}) + a \times d(P_{i-1}, P_i) + 2 \times c \times d(P_i, C_i) + b \times p(P_i) + a \times d(P_i, P_{i+1})$$

Já o peso do percurso para não se estacionar em P_i é dado por:

$$c \times d(C_{i-1}, C_i) + c \times d(C_i, P_{i-1}) + b \times p(P_{i-1}) + a \times d(P_{i-1}, P_{i+1})$$

Onde $d(x, y)$ expressa a distância entre os pontos x e y e $d(P_i)$ é o preço de estacionamento no parque P_i.

A distância entre os pontos na segunda soma terá de ser calculada; contudo, da mesma forma que acontece ao analisar os parques: se o minorante da soma (dado pelas distâncias em linha reta) for maior do que a primeira soma, chegamos à conclusão que será necessário mudar de parque. Para a segunda soma, apenas se adiciona o preço se o tempo adicional que se pretende ficar em C_i implicar um acréscimo de preço.

Na eventualidade de não se usar o parque P_i, deveremos associar ao ponto C_io parque P_{i-1} de forma a que mais do que dois pontos possam ser percorridos a pé entre mudanças de parque de estacionamento.

O percurso final implicará estacionar em todos os parques associados aos pontos C_i e ao destino. Caso C_i's adjacentes apresentem o mesmo parque, sabemos

que será mais eficiente circular a pé entre esses, retornando, depois, ao parque a que estão associados.

Algoritmos Considerados

Algoritmo de Dijkstra Unidirecional

Começamos por considerar o algoritmo de Dijkstra. Concebido por Edsger Dijkstra em 1956 e publicado em 1959, é capaz de solucionar o problema do caminho mais curto num grafo dirigido ou num grafo não dirigido com arestas de peso não-negativo.

Dado um vértice origem, este explora as arestas do grafo, encontrando todos os vértices alcançáveis a partir de s (vértices adjacentes), passando à exploração do vértice seguinte – por outras palavras, encontram-se todos os vértices que distam k do vértice origem antes de descobrir qualquer vértice que diste $k + 1$.

Para tal efeito, usa-se uma fila de prioridade que guarda a ordem dos próximos vértices a serem pesquisados, priorizando vértices cuja soma total do peso das arestas do caminho seja mínima — procura-se maximizar o ganho imediato, minimizando a distância entre a origem e o destino, sendo portanto designado como um algoritmo ganancioso.

Em cada vértice tratado é guardada a informação relativa ao vértice que o antecede no caminho e, após se encontrar o vértice final na fila de prioridade, percorre-se o caminho encontrado no sentido contrário, guardando-o para que este seja retornado, até se regressar ao vértice inicial.

O algoritmo possui uma complexidade temporal de $\mathbf{O}\big((|\mathrm{V}| + |\mathrm{E}|) \times \log(|\mathrm{V}|)\big)$, onde V é o número de vértices e E o número de arestas.

```
DIJKSTRA(G, s):   // G=(V,E), s ∈ V
1.    for each v ∈ V do
2.       dist(v) ← ∞
3.       path(v) ← nil
4.    dist(s) ← 0
5.    Q ← ∅   // min-priority queue
6.    INSERT(Q, (s, 0)) // inserts s with key 0
7.    while Q ≠ ∅ do
8.       v ← EXTRACT-MIN(Q) // greedy
9.       for each w ∈ Adj(v) do
10.         if dist(w) > dist(v) + weight(v,w) then
11.            dist(w) ← dist(v)+ weight(v,w)
12.            path(w) ← v
13.            if w ∉ Q then  // old dist(w) was ∞
14.               INSERT(Q, (w, dist(w)))
15.            else
16.               DECREASE-KEY(Q, (w, dist(w)))
```

Figura 1 - Pseudocódigo para algoritmo de Djikstra.

Dijkstra Bidirecional

Este algoritmo executa o algoritmo de Dijkstra no sentido de s para t e em sentido inverso de t para s num grafo invertido, alternando entre um e outro.

Em Dijkstra unidirecional é processada uma área de raio r, enquanto neste algoritmo são processadas duas áreas de raio $\frac{r}{2}$, o que permite um *speed-up* de aproximadamente duas vezes, melhorando o tempo de execução.

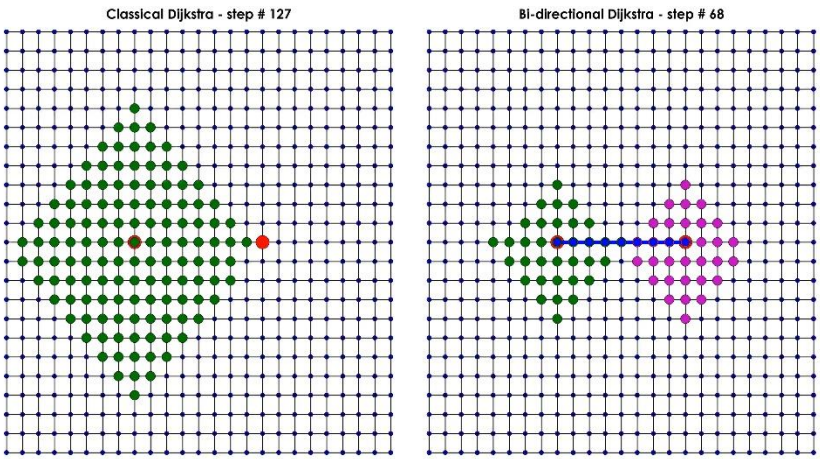

Figura 2 - Comparação de áreas das duas versões do algoritmo de Dijkstra.

A execução deve ser terminada aquando do processamento de um vértice x já tratado na outra direção, podendo o caminho mais curto passar por x ou não, o que significa que Dijkstra Bidirecional retorna um caminho *bom* mas não

o *melhor* caminho possível, não sendo, portanto, o algoritmo mais indicado para usar na nossa solução.

```
s_pr ← g(s)
t_pr ← g(t)
Open_F ← {s}
Open_B ← {t}
for all n ∈ V − {s, t} do
    n_pr ← ∞
end for
p ← ∞
while Open_F ≠ ∅ AND Open_B ≠ ∅ do
    prmin_F = get-min(Open_F)
    prmin_B = get-min(Open_B)
    if prmin_F + prmin_B + ε ≥ p then
        return path for p
    end if
    if Forward frontier is expanded then
        n = delete-min(Open_F)
        Closed_F = Closed_F ∪ n
        for all succ ∈ n_successors do
            if succ ∈ Closed_F then
                continue
            else
                priority ← pr(succ)
                if succ ∈ Open_F then
                    if succ_pr > priority then
                        succ_pr = priority
                    end if
                else
                    succ_pr = priority
                    Open_F ← Open_F ∪ {succ}
                end if
            end if
            if succ ∈ Open_B AND g_F(succ) + g_B(succ) < p then
                p ← g_F(succ) + g_B(succ)
            end if
        end for
    else
        //Expand backward frontier analogously
    end if
end while
return failure
```

Figura 3 - *Pseudocódigo do algoritmo Dijkstra bidirecional.*

Algoritmo A*

Tendo consciência que necessitamos de um tempo de execução melhor do que os obtidos para Dijkstra unidirecional ou Dijkstra bidirecional, percebemos que o algoritmo A^* seria o melhor algoritmo para um cálculo mais eficiente das distâncias.

Chegamos a esta conclusão após uma pequena análise dos tempos execução de Dijkstra unidirecional e A^*, usando como recurso o vídeo *Compare A* with Dijkstra algorithm* de Kevin Wang, encontrado no YouTube:

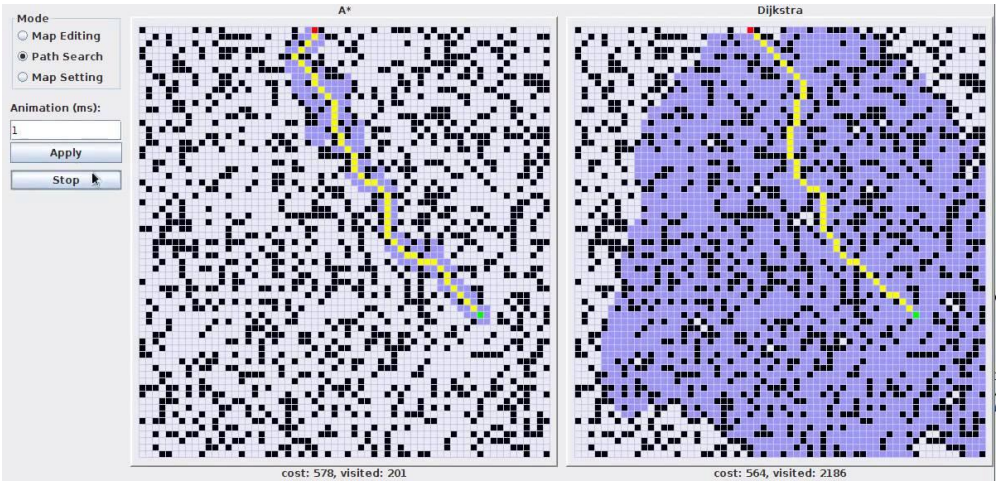

Figura 4 - Captura de ecrã do vídeo mencionado.

O que torna este algoritmo diferente e mais eficiente para várias pesquisas é que, para cada nó, é usada uma função $f(n)$ que calcula uma estimativa do custo total de um caminho usar o nó em questão.

A^* expande caminhos que são mais curtos usando $f(n) = g(n) + h(n)$, sendo $g(n)$, o custo até ao momento para chegar ao nó n e $h(n)$, uma função heurística e o custo estimado de n até ao destino — devemos ter cuidado de modo a nunca fazer uma estimação por excesso deste valor.

O algoritmo Dijkstra é uma variação de A^* onde o valor de $h(n) = 0$, o que faz com que a pesquisa seja feita com expansão igual em todas as direções ao passo que A^* analisa apenas a área na direção do destino.

Utilizando uma boa função heurística é possível optimizar o cálculo do custo de cada vértice e melhorar a performance de A^*. Esta função deverá retornar um valor inferior ao custo real que atingir o nosso destino realmente implica, de modo a que consigamos uma melhor performance do algoritmo: caso o valor seja superior, o tempo de execução seria mais rápido. No entanto, os valores finais seriam pouco precisos, o que não nos interessa muito.

Após uma análise de possíveis funções heurísticas — como, por exemplo, *Manhattan Distance* — chegamos à conclusão que a função que melhor se adequa ao nosso problema seria a **Distância Euclidiana ao Destino**. Esta permite o cálculo do custo de cada vértice de uma forma mais precisa e ao longo de uma grande área, tendo em conta se o vértice se aproxima (ou não) do destino.

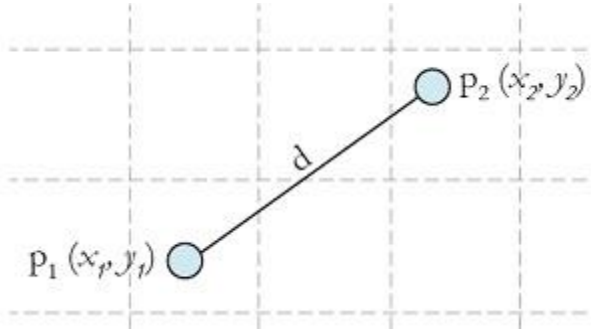

Figura 5 – Fórmula para a Distância Euclidiana definida num referencial de coordenadas (x, y).

A maior desvantagem de A* é o facto da complexidade espacial ser exponencial, no entanto pensamos que o *payoff* de o escolher em relação a Dijkstra Unidirecional compensa.

```
G_SCORE = 0 // used to index g-score
F_SCORE = 1 // used to index f-score
PREVIOUS = 2 // used to index previous

function a_star(graph, start_node, target_node):

    visited = {} // visited list as dictionary
    unvisited = {} // unvisited list as dictionary

    // add every node to the unvisited list
    for each key in graph:
        unvisited[key] = [∞ , ∞ , NULL]
    next key

    // update values for start node in unvisited list
    h_score = heuristic(start_node)
    unvisited[start_node] = [0, h_score, NULL]

    // repeat until there are no nodes in the unvisited list
    finished = False
    while finished == False
        if unvisited.length == 0 then // no nodes left to evaluate
            finished = True
        else:
            // get node with lowest f-score from open list
            current_node = get_minimum(unvisited)
            if current_node == target_node:
                finished = True
                // copy data to visisted list
                visited[current_node] = unvisited[current_node]
            else:
                // examine neighbours
                for neighbour in graph[current_node]:
                    // only check unvisited neighbours
                    if neighbour not in visited then
                        // calculate new g-score
                        new_g_score = unvisited[current_node][G_SCORE] + graph[current_node][neighbour]
                        // check if new g-score is less
                        if new_g_score < unvisited[neighbour][G_SCORE] then
                            unvisited[neighbour][G_SCORE] = new_g_score
                            unvisited[neighbour][F_SCORE] = new_g_score + heuristic(neighbour)
                            unvisited[neighbour][PREVIOUS] = current_node
                        endif
                    endif
                next neighbour

                // add current node to visited list
                visited[current_node] = unvisited[current_node]
                // remove from unvisited list
                del unvisited[current_node]
            endif
        endif
    endwhile

    //return final visited list
    return visited
endfunction
```

Figura 6 - Pseudocódigo para Algoritmo A.*

Identificação de Casos de Uso e Funcionalidades a Implementar

O nosso programa deverá:

- possuir uma interface de texto para interagir com o utilizador, de modo a receber os Dados de Entrada a partir desta;
- permitir visualizar a área metropolitana do Porto (alvo do enunciado), através de uma ferramenta de visualização gráfica como o GraphViewer ou o *GraphViewerCpp* ou outra que se adeque melhor às nossas condições futuras;
- ser capaz de encontrar o caminho mais curto para o parque de estacionamento que esteja mais próximo do seu local de destino;
- ser capaz de dar ao utilizador a escolha de poder estacionar perto de uma paragem intermédia ou não, simplesmente passando por ela — como num *drive-through*;
- ser capaz de apresentar ao utilizador quanto dinheiro irá gastar em estacionamento no percurso pretendido;
- ser capaz de dar ao utilizador a possibilidade de definir quanto peso deseja dar a cada critério de otimização.

Conclusão

Neste relatório, foi exposta uma discussão teórica de uma possível solução para o problema exposto. Nesta, aplicamos diversos conceitos que aprendemos ao longo das aulas e analisamos com mais pormenor algoritmos discutidos nelas.

Confessamos que foi um pouco desafiante fazer um relatório antes da implementação do código, já que estamos habituados a realizá-lo durante ou até após a implementação deste. Pensamos, contudo, que conseguimos estruturar um bom plano e esperamos ser capazes de o pôr em prática de uma forma eficiente e sem grandes percalços.

As tarefas de pesquisa, compilação, escrita, edição e formatação do relatório foram igualmente divididas pelos três membros do grupo, que estiveram em constante comunicação e discussão. Desta forma, o esforço dedicado por cada elemento foi de 1/3.

Elencámos, contudo, algumas partes em que cada elemento se destacou:

- Henrique
    - Solução Proposta;
    - Dados de Saída;
    - Pesquisa de Algoritmos;
- Mateus
    - Formatação do Relatório;
    - Descrição do Tema;
    - Dados de Entrada;
    - Restrições;
- Melissa
    - Problematização;
    - Pesquisa de Algoritmos;
    - Funções Objetivo;
    - Identificação de Casos de Uso e Funcionalidades a Implementar.

Bibliografia

Websites

http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm, consultado a 7 de abril de 2021.

http://en.wikipedia.org/wiki/A*_search_algorithm, consultado a 7 de abril de 2021.

http://brilliant.org/wiki/a-star-search/, consultado a 7 de abril de 2021.

http://www.educative.io/edpresso/what-is-the-a-star-algorithm, consultado a 7 de abril de 2021.

http://isaaccomputerscience.org/concepts/dsa_search_a_star, consultado a 7 de abril de 2021.

Documentos

Abhishek Goyal et al, 2014, *Finding: A* or Djikstra*, Volume 2, Edição de Janeiro, consultado a 7 de abril de 2021 em http://www.hindex.org/2014/p520.pdf.

Sneha Sawlani, *Explaining the Performance of Bidirectional Dijkstra and A* on Road Networks*, 1 de janeiro de 2017, consultado a 7 de abril de 2021 em http://digitalcommons.du.edu/cgi/viewcontent.cgi?article=2303&context=etd.

Vídeos

http://www.youtube.com/watch?v=g024lzsknDo, analisado a 7 de abril de 2021.

http://www.youtube.com/watch?v=8Jjdp6f7oaE, consultado a 7 de abril de 2021.