Concepção e Análise de Algoritmos

Tema 1 – MeetupRider
Partilha de viagens em grupos

Turma 4 Grupo 2

Adriana Gonçalves – up201808911@fe.up.pt
Catarina Fernandes – up201806610@fe.up.pt
Luís Torres Silva – up201808912@fe.up.pt

25 de maio de 2020

Índice

Descrição do tema

Pretende-se implementar uma aplicação que permita aos seus utilizadores partilhar viagens – ridesharing – de forma a otimizar o percurso efetuado em função das origens e dos destinos do condutor e dos passageiros.

O objetivo é o condutor de um determinado veículo disponibilizá-lo na aplicação, indicando quantos passageiros pode transportar, qual é o seu destino, tal como outras restrições, por exemplo hora de partida e chegada. Tendo isto em conta, procuramos encontrar passageiros em função dos seus destinos, da proximidade das suas origens, e das restrições da viagem que pretendem realizar, de forma a otimizar o percurso feito pelo condutor.

A implementação pode ser dividida em várias fases:

Fase 0: Pré-processamento dos grafos

A descrição desta etapa encontra-se detalhada mais à frente neste relatório, mas visa evitar processamentos desnecessários por parte do algoritmo. Para isso tentou-se reduzir o número de vértices e arestas a ser processados.

Fase 1: Verificação da possibilidade de navegação entre a origem e o destino do condutor

Inicialmente calculamos a distância mais curta da originDriver ao destinationDriver, sem passar por outros utilizadores. Com base neste percurso e utilizando a maxDetourDistance descobrimos qual a área à volta do percurso mais curto em que o utilizador se pode deslocar.

Fase 2: Percurso com um veículo e vários passageiros

Na segunda fase, o objetivo é encontrar o caminho mais curto entre a origem (normalmente o ponto de partida do motorista) e o destino final (que também costuma ser o destino do motorista), mas desta vez dando boleia a outros utilizadores.

Para este efeito, consideramos a distância máxima que um condutor se pode deslocar como a distância obtida pela soma do percurso mais rápido com a maxDetourDistance, que é definida pelo utilizador.

Usando a distância máxima que o condutor se pode deslocar num certo percurso, podemos excluir os utilizadores que estão longe demais para fazerem parte do trajeto em questão. Assim, ficamos apenas com os utilizadores a quem podemos dar boleia.

Resta-nos encontrar o caminho ótimo , ou seja, o caminho que nos permita transportar maior número de utilizadores tendo em conta a lotação do carro e tentando sempre minimizar a distância percorrida.

Fase 3: Percurso com vários veículos e vários passageiros

Na última fase, assumimos o problema na íntegra tendo disponível vários veículos que deverão dar boleia ao maior número de utilizadores possível gerindo a lotação da viatura e os horários do condutor e dos passageiros de maneira a minimizar não só a distância percorrida mas também o número de viagens necessárias entre todos, ou seja, o número de veículos utilizados.

Assim, assumimos que pode haver rotatividade entre passageiros (caso um passageiro chegue ao seu destino antes do condutor concluir a sua viagem, passa a haver mais um lugar vago no carro, que pode voltar a ser preenchido).

Formalização do problema

Dados de entrada

originDriver – ponto de origem do condutor

originPassenger – ponto de origem do passageiro em questão

destinationDriver - destino do condutor

destinationPassenger - destino do passageiro em questão

earliestDepartureTime - hora de partida o mais cedo possível para um determinado utilizador

latestDepartureTime - hora de partida mais tardia possível para um determinado utilizador

maxDetourDistance - distância máxima total, definida pelo condutor, que este está disposto a percorrer para além do percurso mais rápido

Va - vetor com o conjunto de veículos existentes, sendo Va[i] o seu i-ésimo elemento, em que cada elemento é caracterizado por:

- vehicleCapacity: capacidade total do veículo (sem contar com o condutor)
- numberSeatsAvailable: número de lugares disponíveis no veículo (não inclui condutor)
    - numberSeatsAvailable inicial = vehicleCapacity
    - 1ª iteração: esta variável não é relevante, visto que só estamos a calcular o melhor percurso da origem ao destino
- numberPassengers – número de passageiros com lugar já atribuído no veículo (não inclui condutor)
- vehicleFull: numberSeatsAvailable = 0
- vehicleId: corresponde à matrícula do mesmo

G(V, E) - grafo dirigido pesado composto por:

- V - Vértices (representam os pontos do mapa que poderão coincidir com a origem/destino de um utilizador) com os seguintes atributos:
    - id - identificador de cada vértice
    - latitude - parte das coordenadas de cada vértice
    - longitude - parte das coordenadas de cada vértice
    - isOrig - Se é origem do condutor (vértice inicial)
    - isDest - Se é destino do condutor (vértice final)
    - Número de pessoas que precisam de boleia

    - adj - contido em E, representa o conjunto de arestas (estradas) que saem de um determinado vértice
- E - Arestas (representam caminhos entre os utilizadores/estradas e podem ter apenas um sentido ou ambos os sentidos)
    - distance - peso da aresta que neste caso traduz a distância entre dois vértices
    - id - identificador da aresta
    - orig - pertencente a V, representa o vértice de origem da aresta
    - dest - pertencente a V, representa o vértice de destino da aresta

Dados de saída

Va - conjunto de veículos existentes

Um elemento do tipo Viagem, composto por:

- id do veículo que vai realizar a viagem
- um vetor com todas as arestas a percorrer de forma ordenada - representa o percurso ótimo
- a distância total prevista
- hora inicial da viagem
- hora final prevista

Restrições

addressDriver ≠ destinationDriver e, para o mesmo passageiro, originPassenger ≠ destinationPassenger, pois não é necessário uma deslocação para chegar ao mesmo sítio

addressDriver ≠ destinationPassenger pois um condutor não irá voltar atrás na sua rota (para isso é que servem os taxistas)

maxDetourDistance > 0

latestDepartureTime > earliestDepartureTime

$\forall va \in Va$, vehicleCapacity > 0, pois não faz sentido um utilizador disponibilizar um veículo se não há lugar para mais ninguém

$\forall e \in E$, distance > 0, pois todas as ruas têm um determinado comprimento, o que significa que o valor da distância não pode ser negativo nem zero (os vértices no extremo dessa aresta seriam coincidentes)

$\forall v \in V$, adj $\in E$, todas as arestas adjacentes a um vértice fazem parte do conjunto de arestas E do grafo

$\forall e \in E$, orig $\in V$, todos os vértices que se qualificam como origem de uma aresta são pertencentes ao conjunto de vértices V do grafo

$\forall e \in E$, dest $\in V$, todos os vértices que se qualificam como destino de uma aresta são pertencentes ao conjunto de vértices V do grafo

O originDriver deve corresponder ao vértice inicial da primeira aresta da lista de arestas devolvida

O destinationDriver deve corresponder ao vértice final da última aresta da lista de arestas devolvida

Função objetivo

O objetivo do problema passa por encontrar uma sequência de arestas a percorrer de forma a minimizar o peso total das mesmas e passando por todos os vértices de passageiros, indo desde o ponto inicial (addressDriver) até ao ponto final (destinationDriver). Passa também por minimizar a quantidade de veículos utilizados para transportar o mesmo número de passageiros. Assim, o nosso objetivo será minimizar as seguintes funções:

f = $\sum$ distance(e), e $\in$ Conjunto de arestas percorridas pelo condutor

g = | veículos usados |

Perspetiva de solução

Pré-processamento do grafo

Antes de implementarmos qualquer uma das fases, devemos pré-processar o grafo G de maneira a reduzir o seu número de vértices e arestas. Isto ajuda a aumentar a eficiência temporal dos algoritmos que aplicarmos. Assim, devemos ignorar as arestas do grafo que se encontram inacessíveis (por exemplo devido a algum bloqueio na estrada).

Para este efeito, temos duas opções: podemos realizar uma breadth-first search (busca em largura) ou uma depth-first search (busca em profundidade):

Breadth-First Search

Dado um vértice de origem v, explora-se sistematicamente o grafo de maneira a descobrir todos os vértices a que se pode chegar a partir de v, ou seja, os seus vértices adjacentes. Só depois de descobrir todos os vértices adjacentes de v é que se passa para outro vértice.

Para aplicarmos este algoritmo, cada vértice deverá registar este campo adicional:

- discovered: variável booleana

Este campo é inicializado com discovered = false e será útil para ajudar a determinar se um vértice já foi ou não visitado. Caso o discovered esteja a falso, sabemos que o vértice em questão ainda não foi visitado e procedemos ao seu processamento, caso contrário, o vértice não é processado. Assim, asseguramos que cada vértice é processado apenas uma vez.

É também utilizada uma fila para manter um registo dos vértices a serem processados.

Assim, a sua **complexidade temporal** é de $O(|E|+|V|)$ em que $|E|$ representa o número de arestas e $|V|$ o número de vértices. Como a complexidade temporal de inserção e remoção da fila são de $O(1)$, estas operações não aumentam a complexidade temporal do algoritmo.

A sua **complexidade espacial** é de, no pior caso (quando todos os vértices são processados), $O(|V|)$ onde $|V|$ representa o número total de vértices no grafo.

```
for each v ∈ V do discovered(v) ← false
Q ← ∅

ENQUEUE(Q, s)
discovered(s) ← true

while Q ≠ ∅ do
        v ← DEQUEUE(Q)
        for each w ∈ Adj(v) do
                if not discovered(w) then
                        ENQUEUE(Q, w)
                        discovered(w) ← true
```

Figura 1 - Pseudocódigo do algoritmo Breadth-First Search

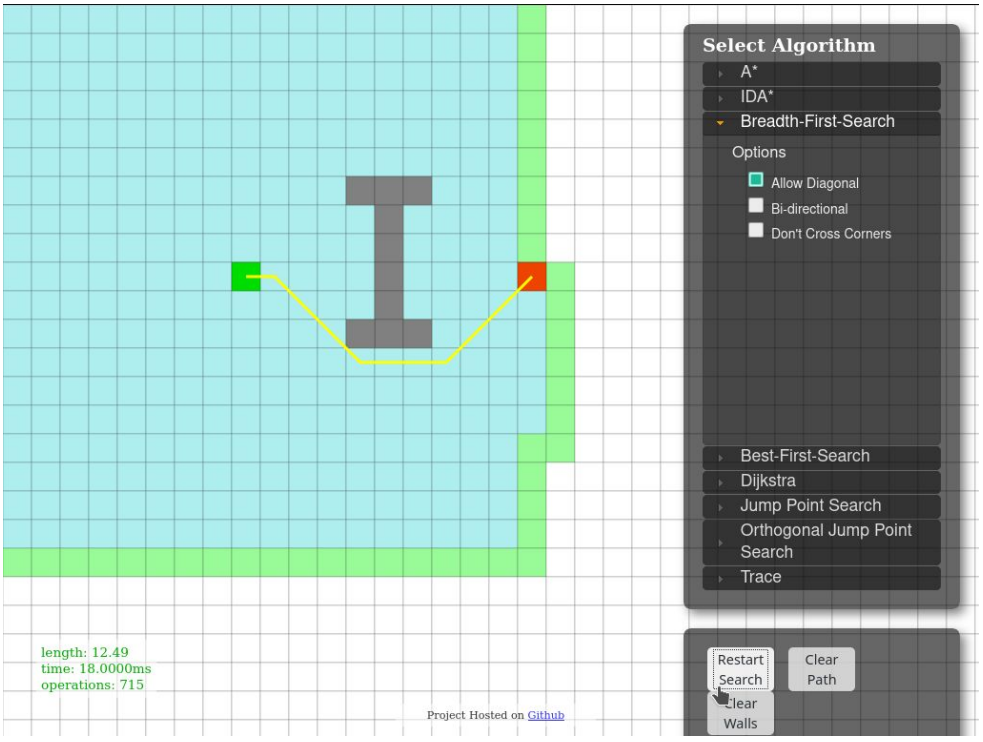


Figura 2 - Visualização do algoritmo Breadth-First Search

Depth-First Search

O algoritmo começa num vértice v e explora um ramo do grafo até não conseguir avançar mais em profundidade. Quando todos os ramos que partem desse vértice forem explorados, são exploradas as arestas que saíram do vértice a partir do qual v foi descoberto. Este processo é repetido até todos os vértices serem processados.

Tal como no algoritmo BFS, cada vértice deverá registar este campo adicional, que irá servir impedir o processamento de um vértice mais do que uma vez.

- discovered: variável booleana

Para auxiliar nesta busca, pensámos em recorrer a uma pilha e assim aproveitamos a sua estrutura last-in/first-out.

Este algoritmo partilha as mesmas características de complexidade temporal e complexidade espacial do algoritmos BFS, sendo que o facto de recorrermos a uma pilha em vez de uma fila não altera a complexidade temporal, dado que as operações de inserção e remoção são de *O(1).*

Assim, a complexidade temporal é de $O(|E| + |V|)$ em que $|E|$ representa o número de arestas e $|V|$ o número de vértices e a **complexidade espacial** é de, no pior caso, $O(|V|)$ onde $|V|$ representa o número total de vértices no grafo.

```
let S be a stack
S.push(v)
while S is not empty do
        v = S.pop()
        if discovered(v) ← false
                discovered(v) ← true
                for all edges from v to w in G.adjacentEdges(v) do
                        S.push(w)
```

Figura 3 - Pseudocódigo do algoritmo Depth-First Search

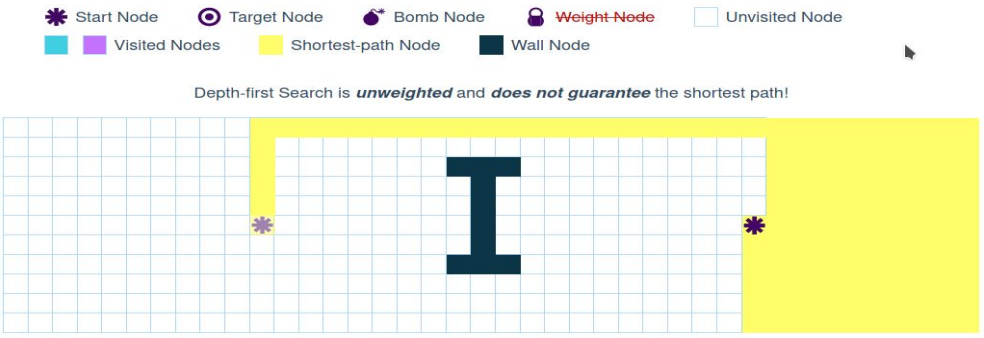


Figura 4 - Visualização do algoritmo Depth-First Search

Durante a implementação iremos testar qual dos algoritmos se adequa melhor ao nosso problema em concreto e a partir desses dados iremos escolher o algoritmo mais adequado.

Conectividade do grafo

Não será necessário implementar a restrição relativamente à conectividade do grafo uma vez que o percurso inverso seria considerado uma nova viagem e

nenhum dos utilizadores tem necessariamente de realizar a viagem inversa. Desta forma, considera-se cada percurso como independente não tendo de ser garantido o mesmo percurso no sentido inverso.

Estratégias de Implementação

Após o pré-processamento, é reduzido o volume de dados a trabalhar de forma a que o cálculo da distância mínima entre 2 vértices, efetuado na fase 1, seja feito com maior eficácia.

Na **fase 1** vamos iterar sobre o grafo com o objetivo de determinar o conjunto de arestas a percorrer num determinado percurso, de forma a determinar o percurso ótimo, que será aquele que tiver uma distância total menor.

Para isto temos duas opções:

Opção 1

Utilizar o **algoritmo Floyd-Warshall**, que calcula a distância mínima entre qualquer par de vértices de um grafo dirigido cujas arestas tenham um peso não negativo. Pensamos também em adaptar este algoritmo para incluir a reconstrução de caminhos entre os vários vértices, coisa que seria útil não só para a fase 1 como para as outras.

O primeiro passo é inicializar a matriz solução M[i][j] com os pesos 0 se i = j ou ∞ caso i ≠ j. No final de todas as iterações, o resultado esperado será obter a matriz M preenchida com as distâncias mínimas de cada par de vértices devidamente organizadas através dos índices da própria matriz. Assim, deduzimos que a complexidade espacial é de $O(|V|^2)$, pois é necessário guardar os valores numa matriz de $|V| \times |V|$.

É um bom algoritmo para usar em grafos densos, pois a sua complexidade temporal mantém-se ($O(|V|^3)$) ao contrário de outros algoritmos como o Dijkstra, cuja complexidade temporal, no pior caso, é $O(|V|^3 * log(|V|))$.

No entanto, devido à quantidade de cálculos requeridos, seria do nosso interesse otimizar este algoritmo. Uma ideia seria reduzir ainda mais o número de vértices processados, adaptando a BFS ou DFS usada anteriormente de maneira a “ignorar” também vértices cuja distância à origem fosse maior que a maxDetourDistance. Para isto teríamos apenas que adicionar um campo

adicional dist ao BFS ou DFS que contabilizasse a distância total percorrida. Só após este processamento adicional é que aplicaríamos, então, o algoritmo.

Opção 2

Utilizar um dos seguintes quatro algoritmos:

O **algoritmo de Dijkstra** é um algoritmo ganancioso, que funciona apenas com arestas cujo peso seja positivo. A sua aplicação resulta numa árvore de caminhos mais curtos desde a origem (originDriver) até aos outros pontos do grafo.

Este algoritmo apresenta uma complexidade temporal de $O((|V|+|E|)*log(|V|))$ e uma complexidade espacial de $O(|V|)$.

Na nossa implementação cada vértice irá ter um campo “path”, que indicará o vértice anterior no caminho mais curto. Seguindo uma abordagem gananciosa, podemos utilizar uma fila de prioridade mutável para maximizar o ganho.

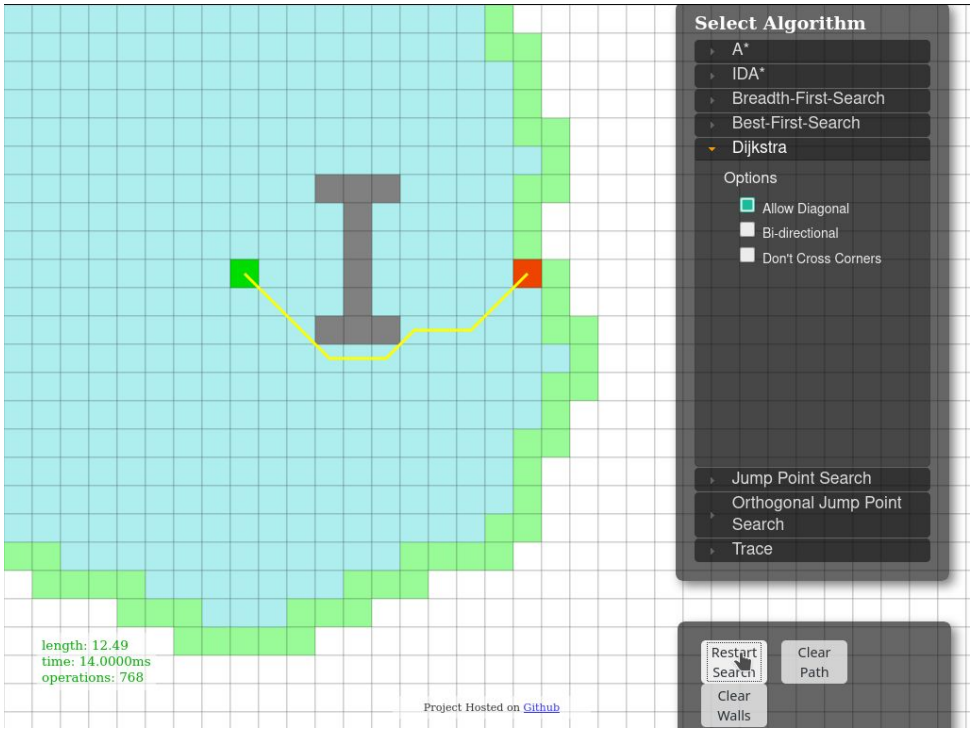


Figura 5 - Visualização do algoritmo Dijkstra

O **algoritmo A***, que pode até ser visto como uma extensão do algoritmo Dijkstra, apresenta algumas vantagens em relação a ele, pelo que será considerada a sua implementação.

Este algoritmo utiliza heurística para guiar a sua pesquisa, o que faz com que ele processe menos vértices e seja bastante mais rápido, pois usa a distância euclidiana entre o vértice de origem e destino (valor que se obtém facilmente num mapa) como critério de descoberta de vértices “promissores”.

Assim, obtém melhores resultados em grafos mais densos, o que faz com que seja adequado para implementar em redes de estradas, como é o caso deste projeto.

A sua complexidade temporal e espacial é igual à do algoritmo Dijkstra, sendo $O((|V|+|E|)*log(|V|))$ e $O(|V|)$ respetivamente.

No entanto, apesar de ser mais rápido, já que analisa menos vértices, o seu resultado não é garantidamente a solução ótima. Terá de ser avaliada a diferença em termos de tempo de execução tendo em conta a possibilidade de se obter uma solução não-ótima.

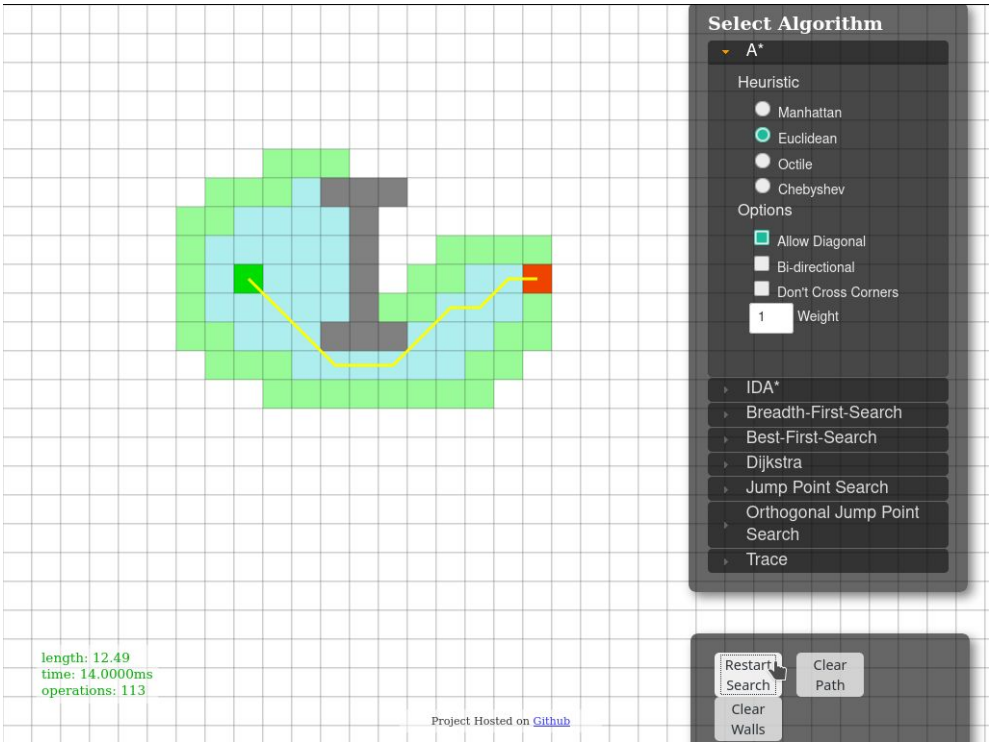


Figura 6 - Visualização do algoritmo A*

O **Algoritmo Dijkstra bidirecional** aplica o Dijkstra duas vezes, partindo da origem e partindo do destino, terminando quando os caminhos se “encontram” no mesmo vértice.

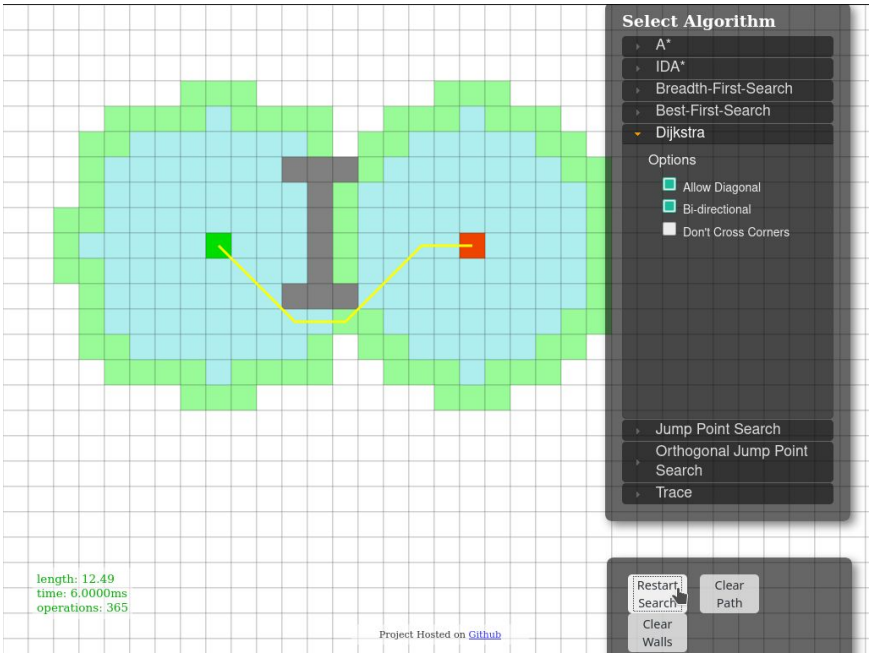


Figura 7 - Visualização do algoritmo Dijkstra bidirecional

O **algoritmo A* bidirecional** parte dos mesmos pressupostos do algoritmo A*, sendo o seu funcionamento semelhante mas partindo do ponto inicial e do ponto final simultâneamente até encontrar um ponto intermédio comum.

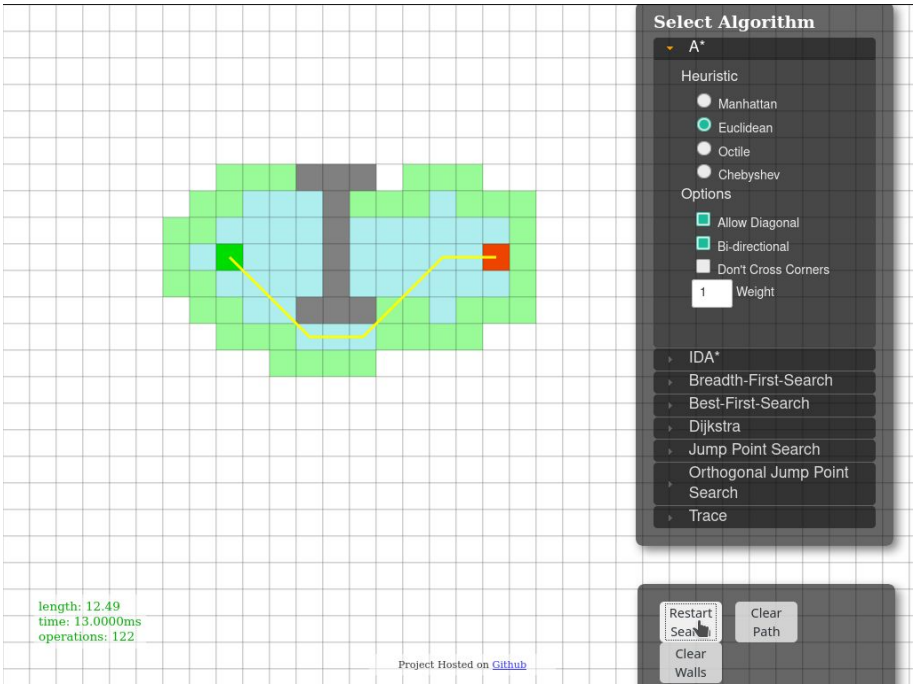


Figura 8 - Visualização do algoritmo A* bidirecional

Na fase de implementação do projeto, iremos avaliar as vantagens e desvantagens de cada opção e a partir desses dados iremos escolher a opção e algoritmo mais favorável.

A **segunda fase** deste trabalho, é uma variação dos populares problemas "Travelling salesman" e "Vehicle Routing". A principal diferença do problema proposto e do problema "Travelling Salesman" é a não rotatividade do percurso, ou seja, o ponto final do percurso é diferente do ponto inicial, o que não será um impedimento à aplicação de algoritmos semelhantes com pequenas modificações.

A maneira mais óbvia de resolver este problema seria recorrer a **Brute-force search combinado com recursividade**, contudo este tipo de implementação apresenta uma complexidade temporal e espacial $O(|V|!)$, o que não seria viável visto que a dimensão de um mapa poderá ser enorme.

Uma alternativa seria o **algoritmo Nearest Neighbour,** que é um algoritmo ganancioso que escolhe a próxima paragem usando o critério de qual é que está mais perto do ponto atual, normalmente o algoritmo retorna um caminho curto mas, nem sempre é o caminho ótimo, razão pela qual se terá de avaliar o seu desempenho comparativamente com o **algoritmo 2-opt.**

2-opt heuristic é um algoritmo que compara todas as combinações de troca válidas possíveis. Embora este algoritmo seja por vezes aplicado em problemas do estilo "travelling salesman" em que se pressupõe um regresso à origem, neste caso em que o ponto final é fixo poderá também ser aplicável, e é igualmente adequado ao problema.

A **fase 3** do problema é a gestão dos vários passageiros através de vários veículos, que é um problema bastante complexo que não pode ser resolvido em tempo polinomial (NP-difícil). Assim, optamos por uma abordagem de atribuição de lugares num veículo consoante a compatibilidade do horário dos condutores e dos passageiros, estando os passageiros dentro da área abrangida pelo maxDetourDistance do condutor.

Casos de utilização

De forma a implementar todas as funcionalidades descritas abaixo e a tornar a nossa aplicação útil e intuitiva, estruturamos a mesma da seguinte forma:

Um menu inicial no qual o utilizador seleciona se está à procura de boleia ou se está disponível para dar boleia. Dependendo desta escolha, o utilizador será classificado como:

- condutor
  - será levado para um novo menu no qual irá registar o veículo, definir a sua morada de origem e de destino, escolher o intervalo de tempo para a partida e por fim decidir o valor máximo para a distância que o condutor está disposto a percorrer a mais para recolher passageiros.
- passageiro
  - será levado também para um novo menu no qual terá de definir a sua morada de origem e destino e também o seu intervalo de partida.

Após cada utilizador disponibilizar as informações necessárias, poderão selecionar a opção de encontrar uma viagem que corresponda às suas especificações. Será também implementado um sistema de visualização da rota obtida utilizando a ferramenta GraphViewer.

Naturalmente a interface poderá vir a ter mais menus e opções conforme o projeto for elaborado.

Na nossa abordagem à resolução do problema, estamos a ter em conta vários casos e funcionalidades correspondentes, de realçar as seguintes, listadas em nenhuma ordem em particular:

1. **caso:** o condutor quer ir para o seu destino sem de percorrer mais do que x quilómetros em relação ao percurso mais curto que liga a sua origem com o seu destino
   funcionalidade: possibilidade para o condutor definir o valor de x no início de cada viagem (x corresponde a maxDetourDistance)
2. **caso:** o condutor pode querer limitar o máximo de passageiros que transporta
   funcionalidade: para este efeito, o condutor pode definir uma capacidade diferente para cada viagem (vehicleCapacity), transportando assim apenas o número de pessoas que quer transportar
3. **caso:** o condutor deixa os passageiros nos seus destinos antes de chegar ao seu e está disponível para ir recolhendo novos passageiros conforme os anteriores forem saindo
   funcionalidade: para isto vamos implementar rotatividade de

passageiros, ou seja, desde que a capacidade máxima do veículo não tenha sido atingida é possível recolher mais passageiros

4. **caso:** sendo a aplicação de Ridesharing uma medida utilizada também para proteger o ambiente, reduzir o número de viagens totais é ideal (menos emissões de gases poluentes)
   funcionalidade: para este efeito vamos tentar otimizar o matching de viagens entre passageiros de forma a ser preciso um menor número de veículos para transportar o mesmo número de passageiros, o que é também tido em conta na criação do ponto anterior da rotatividade de passageiros
5. **caso:** é possível que certas estradas estejam cortadas ou impedidas de circulação por algum motivo, logo não as queremos ter em conta quando calculamos o nosso percurso ótimo
   funcionalidade: estas estradas bloqueadas serão eliminadas durante o pré-processamento do grafo

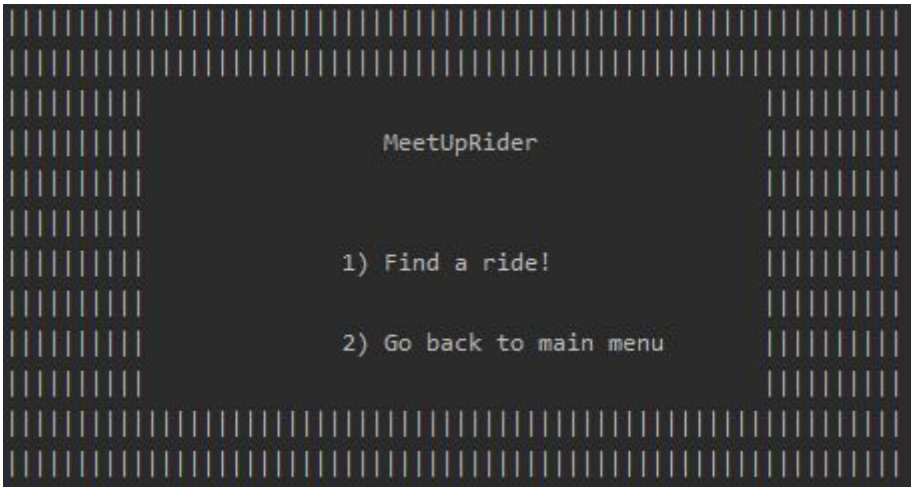
Principais casos de uso implementados

Foi implementado um sistema de menus simples e interativo. Inicialmente os utilizadores são separados de acordo com se são passageiros ou condutores.

De seguida, para permitir ao utilizador definir os seus pontos de origem e destino é apresentado um menu onde onde o utilizador escolhe qual a cidade na qual a viagem vai ocorrer.

Após a escolha da cidade os vetores que guardam os passageiros e condutores duma determinada localização são carregados com os registos que estão nos ficheiros texto. De seguida aparece um mapa da mesma com os ids dos vértices para que o utilizador consiga escolher quais os pontos que irá

selecionar. É então perguntado ao utilizador quais são esses pontos (origem e destino) e também todas as informações necessárias para termos os dados do passageiro/condutor. Após preencher todos os seus dados, é mostrado ao utilizador um menu que lhe permite procurar boleia.



Por fim, ao selecionar “Find a ride”, será apresentado o percurso a percorrer para passar nos pontos necessários.

É possível também mostrar a conectividade de um grafo a partir de um certo ponto inicial (parte não conexa a azul e o restante a vermelhor) bem como mostrar a visualização de um caminho no mapa (caminho respresentado a vermelho, com os vértices inicial/final a verde).

Estruturas de dados utilizadas

Grafo

Utilizada a classe Grafo que consiste essencialmente num vetor de Vertex (vértices). Visto que o acesso a memória por apontadores possui tempo constante, a iteração pelo vetor de vértices possui complexidade temporal O(n) e o acesso à informação destes possui complexidade temporal constante O(1).

A classe grafo possui ainda funções auxiliares e específicas dos algoritmos implementados, como por exemplo, funções para avaliar a sua conetividade e para eliminar a componente não conexa. O grafo possui também um vetor que regista aquando da importação os ids dos nós repetidos, visto que é mais eficiente iterar por este vetor aquando da adição de novas arestas para utilizar apenas um nó no caso de existirem nós duplicados. Como o número de vértices duplicados é substancialmente inferior ao de vértices efetivamente importados, o tempo de iteração é negligenciável bem como o espaço ocupado.

Vértices

Os vértices por sua vez possuem a informação sobre o próprio id, as suas coordenadas e um vetor de arestas que partem deles para outros nós.

Arestas

As arestas guardam informação sobre o seu vértice de destino e sobre o seu peso. Todas as arestas são direcionais, visto que, tratando-se de um mapa, as ruas poderão ser efetivamente de sentidos únicos.

Representação gráfica de mapas e caminhos

A representação gráfica recorre à API graphViewer desenvolvida em Java. Foram desenvolvidas funções para representação dos mapas com recurso a esta API, mostrando os ids dos vértices para que o utilizador possa facilmente escolher os pontos de origem e destino, sendo que após calculado o caminho estas funções recebem o conjunto de vértices percorridos, mostrando os vértices e arestas percorridas a cor diferente.

Condutores e Passageiros

Para representar os condutores e os passageiros recorremos, respetivamente, às classes Driver e Passenger.
No caso da classe Passenger, esta contém o id do passageiro, o seu ponto de origem e o seu ponto de destino no mapa, e também os tempos de partida que representam o intervalo de disponibilidade do passageiro. Para todos estes atributos da classe foram criados os respetivos métodos get.
Por sua vez, a classe Driver contém, para além dos mesmos atributos que a

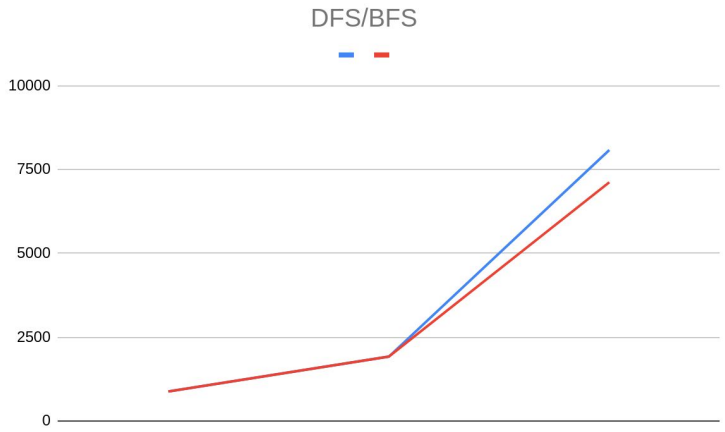
classe Passenger, atributos representantes da maxDetourDistance e vehicleCapacity. Tal como antes também foram implementados todos os métodos get respetivos.

Aplicação

Por último, foi também criada uma classe Application. Esta classe tem, entre outras estruturas, 2 vetores muito relevantes para o funcionamento da aplicação em si, nomeadamente o vetor de passageiros e o vetor de condutores. Na classe Application estão também criadas todas as funções que permitem adicionar e remover quer passageiros quer condutores dos respetivos vetores, preencher estes dois vetores com os dados guardados anteriormente em ficheiros texto, e também atualizar esse registo no ficheiro.

Algoritmos efetivamente implementados e análise de complexidade

Conectividade



	Penafiel	Espinho	Porto
DFS	878,2	1919,8	8082,675
BFS	878,4	1921,5	7121,975

Foram realizados testes para determinar empiricamente a performance destes dois algoritmos. O processo utlizado foi a medição do tempo de execução do algoritmo para o mesmo vértice de início. Após repetir este processo um número elevado de vezes foi feita uma média do tempo de execução. Todos os tempos apresentados na tabela e no gráfico estão em microsegundos.

Os algoritmos apresentados começam por percorrer o vetor de vértices, marcando-os como não tendo sido visitados, complexidade temporal **O(n)**. No entanto enquanto que o algoritmo dfs irá visitar recursivamente todos os nós que conseguir alcançar a partir da origem, fazendo uma pesquisa em profundidade, o algoritmo de bfs faz uma pesquisa lateral, sem utilizar recursividade. Isto permite-lhe não despender tanta memória pois não tem de alocar espaço para uma nova *stack* nem tem o *overhead* necessário ao invocar uma função. Desta forma, quanto maior o grafo, maior é esta diferença de desempenho entre os algoritmos. A complexidade temporal dos algoritmos é a mesma pois no caso em que percorrem todos os vértices e todas as arestas é **O(V+E)**.

O código implementado para estas duas funções segue o pseudo-código apresentado anteriormente na secção “Pré-processamento do grafo” (figura 1 e figura 3).

Pré-Processamento

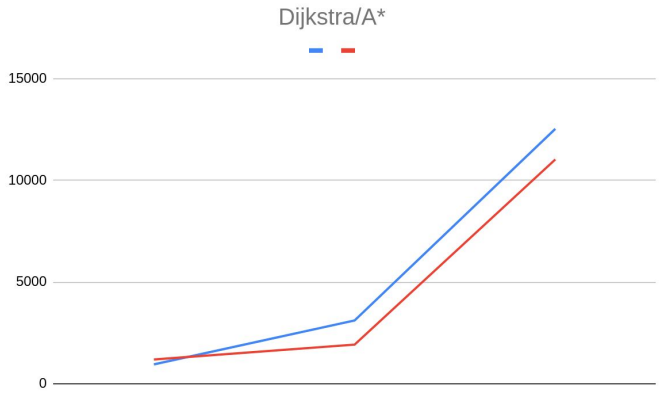
Utiliza-se o algoritmo bfs para analisar a conectividade (devido às vantagens relativamente ao algoritmo dfs enunciadas anteriormente). Extraem-se, assim, os vértices pertencentes à componente conexa do grafo, sendo todos os outros eliminados. A operação de eliminação apresenta complexidade **O(|V|^2)** pois implica:

- Percorrer todo o vetor de vértices;
- Ao apagar um elemento do vetor, mover todos os outros elementos subsequentes desse vetor uma posição;

Assim, o pré-processamento é uma etapa de complexidade temporal **O(|V|^2).**

A sua **complexidade espacial** é de, no pior caso (quando todos os vértices são processados), $O(|V|)$ onde $|V|$ representa o número total de vértices no grafo, como referido em secções anteriores (“Pré-processamento do grafo”).

Path finding



	Penafiel	Espinho	Porto
DIJKSTRA	953,4	3107,9	12526,9
A*	1189,7	1924,2	11024,3

Foi utilizado o mesmo processo que nos algoritmos de conectividade para a determinação da performance de cada algoritmo.

Esta estratégia corresponde à estratégia 2 abordada na secção “Estratégias de implementação”. Após a implementação do algoritmo Floyd-Warshall, notámos que a sua performance era bastante pior quando comparada ao A*/Dijkstra, pois não corria em tempo útil. Assim, descartamos a opção 1 e ficou decidido que usaríamos ou o Dijkstra ou o A* o para a determinação de caminhos.

A maior diferença entre os dois algoritmos testados é a maneira como determinam os próximos vértices a processar. O algoritmo A* é mais seletivo que o Dijkstra, que processa todos os vértices inicialmente e depois, com base nisso, retorna o melhor caminho entre dois pontos. Em vez disso o A* utiliza uma heurística (no nosso caso a distância euclidiana) para o auxiliar na pesquisa pelo melhor caminho entre dois pontos, pelo que diminui a quantidade de vértices a processar.

A análise da complexidade destes algoritmos encontra-se na secção “Estratégias de implementação” na Opção 2.

A*

```
h() - heuristic

Queue Q;

for each vertex v:
        v.distance ← ∞
        v.path ← ∅

ENQUEUE(Q, origin)

while Q ≠ ∅ do:
        DEQUEUE(Q, v)
        if v == destination:
                break;
        for each edge e in v.outgoing:
                f = h(origin, destination)
                if e.destination.distance > f:
                        d = e.destination.distance;
                        e.destination.distance = f;
                        e.destination.path = v;
                        if d == infinito:
                                ENQUEUE(Q, destination)
                        else
                                DECREASE_PRIORITY(Q, e)
```

Dijkstra

```
Queue Q;

for each vertex v:
        v.distance ← ∞
        v.path ← ∅
        v.visited ← false

ENQUEUE(Q, origin)

while Q ≠ ∅ do:
        DEQUEUE(Q, v)
        for each edge e in v.outgoing:
                if (e.destination.distance > v.distance + e.weight) do:
                        e.destination.distance ← v.distance + e.weight
                        e.destination.path ← v
                        if e.destination is visited:
                                DECREASE_PRIORITY(Q, e)
                        else
                                ENQUEUE(Q, destination)
                                e.destination.visited ← true
```

1ª iteração

Pretende-se determinar se é possível efetuar uma viagem da origem ao destino apresentados. Caso esta viagem seja possível, é determinado um percurso a seguir, utilizando o algoritmo A*, sendo este caminho representado a cor diferente no grafo apresentado.

2ª interação

O condutor dará boleia a um passageiro. Para isso são calculados os caminhos mais rápidos desde a posição inicial do condutor, até à posição inicial do passageiro, da posição inicial do passageiro até ao destino do passageiro e do destino do passageiro até ao destino do condutor. A junção destes 4 caminhos mais rápidos dará o caminho mais rápido total.

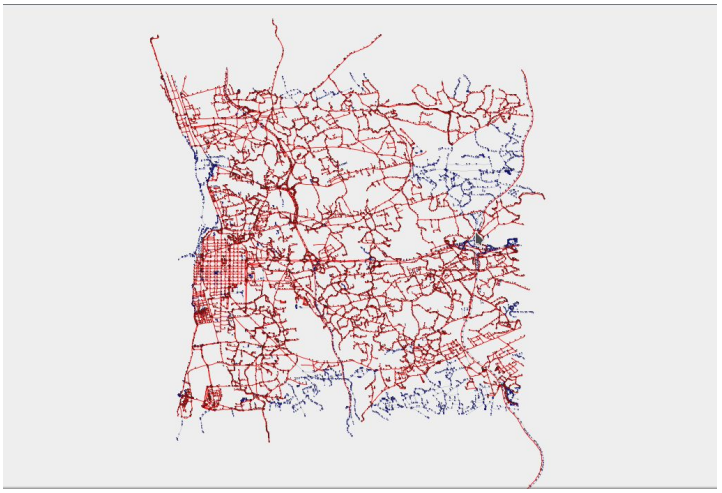
Para esse processamento pretendeu-se utilizar o algoritmo nearest neighbour. No entanto, apesar de o algoritmo ter sido implementado e de termos essa função declarada no ficheiro Graph.h, ela não foi integrada com o resto do programa.

Nearest neighbour

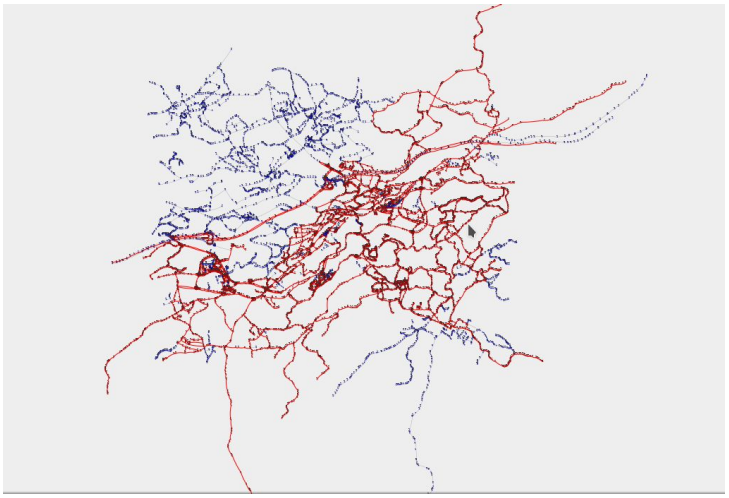
```
<T>
vector<T> Graph<T>::nearestNeighbour(const T &origin, const T &destiny){
    auto orig=findVertexByInfo(origin);
    auto dest=findVertexByInfo(destiny);
    vector<T>path;

    for (auto w:this->vertexSet){
        w->visited=false;
    }
    nearestNeighbourAux(orig, dest, &path);
    return path;
}

<T>
int Graph<T>::nearestNeighbourAux(Vertex<T> *origin, Vertex<T> *dest, vector<T> *res){
    while(origin->info!=dest->info){
        origin->visited=true;
        res->push_back(origin->info);
        auto shortest_path=origin->outgoing[0];
        for (auto w:origin->outgoing){
            if (w.weight<shortest_path.weight){
                if(!(w.dest->visited)) {
                    shortest_path = w;
                }
            }
        }
        origin=shortest_path.dest;
        if (origin->visited){
            return -1;
        }
    }
}
```

Conectividade dos grafos utilizados

Os pontos azuis representam pontos desconexos. Os pontos a vermelho representam pontos conexos do grafo.



Grafo de Espinho apresenta elevada conectividade



Grafo de Penafiel não aparenta ser muito conexo pois cerca de 15% dos vértices estão a azul, mostrando que não pertencem à componente conexa do grafo.

Grafo do Porto apesar de apresentar bastantes vértices a azul, pode ser considerado ainda assim bastante conexo pois a elevada densidade de nós permite descartar estes nós não conexos e ainda assim obter uma boa aproximação a qualquer destino da cidade, sem comprometer significativamente o cálculo de distâncias e percursos.

Conclusão

O problema de RideSharing apresentado foi compreendido e analisado com sucesso. Para a solução foram propostas três etapas cujo incremento de dificuldade foi semelhante e cuja complexidade foi crescente, apresentando-se ainda, na última fase, funcionalidades extra à resolução direta do problema, nomeadamente a rotatividade de passageiros ao longo da viagem e a tentativa de reduzir o número de veículos utilizados como medida ecológica.

Para a resolução das diferentes fases, foi analisada a utilização de vários algoritmos abordados na unidade curricular de Concepção e Análise de Algoritmos, tal como descrito na Perspetiva de Solução.
Ao implementar os vários algoritmos foi testada a sua eficiência e tempo de execução, sendo que foram escolhidos os que se revelaram mais eficientes.

Utilizamos como recurso grafos representativos de situações reais recorrendo a ferramentas como o Open Street Maps, de forma que a solução está mais ajustada à realidade.

Relativamente às estruturas de dados utilizadas e aos grafos utilizados implementamos soluções para alguns possíveis problemas como a necessidade de pré processar os grafos devido à presença de elementos redundantes e situações que não podem ser utilizadas (por exemplo estradas bloqueadas) de forma a aumentar a eficiência temporal.

Os principais desafios que encontramos foram redefinição de novos grafos utilizando só a componente conexa dos grafos anteriores, a integração do programa com o GraphViewer, a complexidade de certos algoritmos e a dificuldade ao testar devido ao tempo que demorava a carregar certos mapas.
O processo de debugging foi também bastante moroso já que os dados de entrada do openStreetMaps possuíam algumas limitações que não eram cobertas pelos mapas de teste mais pequenos que possuíamos. A dificuldade em compreender se o resultado dos processamentos seria o esperado tornou-se uma dificuldade adicional mas facilmente colmatada.

Esforço dedicado e divisão de tarefas:

A divisão do trabalho foi bastante equilibrada entre todo o grupo. Não houve nenhuma clara divisão de tarefas visto que trabalhamos sempre os três ao mesmo tempo em chamada o que permitiu mesmo a quem não estava a escrever determinada parte do código esclarecer dúvidas e participar nos processos de debugging e teste. Além disso, a interação entre a equipa foi determinante para tomar decisões fulcrais que afetam toda a estrutura do trabalho.

Bibliografia

https://moodle.up.pt/pluginfile.php/115848/mod_label/intro/07.grafos2.pdf?time=1584068435647

https://moodle.up.pt/pluginfile.php/115850/mod_label/intro/09.grafos4.pdf?time=1585235738967

https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/

https://www.khanacademy.org/computing/computer-science/algorithms/breadth-first-search/a/analyhttps://www.geeksforgeeks.org/bidirectional-search/sis-of-breadth-first-search

https://pt.wikipedia.org/wiki/Busca_em_largura

https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/

https://towardsdatascience.com/a-data-scientists-guide-to-data-structures-algorithms-part-2-6bc27066f3fe

https://iq.opengenus.org/dijkstras-algorithm-finding-shortest-path-between-all-nodes/

https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

https://ijcsmc.com/docs/papers/July2015/V4I7201599a23.pdf

https://www.geeksforgeeks.org/a-search-algorithm/

https://en.wikipedia.org/wiki/A*_search_algorithm

https://en.wikipedia.org/wiki/2-opt

https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/

https://en.wikipedia.org/wiki/Nearest_neighbor_search

https://clementmihailescu.github.io/Pathfinding-Visualizer/#

https://qiao.github.io/PathFinding.js/visual/