U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Projeto de Conceção e Análise de Algoritmos 2020/2021 – Parte I

Tema 9: O Padeiro da Vila em Época Covid

T1G1

André Pereira up201905650
João Marinho up201905952
Matilde Oliveira up201906954

16 de Abril de 2021

Índice

Descrição do Tema

Um padeiro, necessita de vender o seu produto levando-o diretamente a casa dos seus clientes com a sua carrinha. Parte da sua padaria, e entrega o produto na morada especificada por cada cliente. No final das entregas, terá que regressar à padaria. Cada cliente especifica a hora preferencial para a receção da entrega, com algum tempo de tolerância, pelo que é necessário ter em conta esta restrição.

A definição do trajeto que uma carrinha tem que ter, precisa de otimizar vários objetivos, principalmente e numa etapa inicial: minimizar o tempo total do itinerário e equilibrar o tempo de atraso nas entregas.

De forma a ter uma implementação incremental das variantes do problema podemos subdividi-los em subproblemas de diferentes graus de complexidade:

1. Uma carrinha que realiza apenas uma entrega

Nesta abordagem, considera-se que o padeiro tem disponível apenas uma carrinha que terá de realizar uma entrega. Nesta situação, o problema reduz-se a encontrar o trajeto mais curto, com início na padaria, que passe na morada de entrega, no horário pretendido, e termine na padaria.

2. Uma carrinha que realiza todas as entregas

Semelhante ao caso anterior a padaria continua a deter apenas uma carrinha, capaz de transportar todos os produtos, mas agora existem várias entregas para os diversos clientes. Assim, o problema passa por encontrar o trajeto mais curto com início na padaria que passe por todas as moradas pretendidas, dentro do horário pretendido, equilibrando o tempo de atraso das entregas e terminando na padaria.

3. Frota de carrinhas que realizam entregas

Podemos também assumir o problema em que a padaria tem ao seu dispor uma frota de carrinhas. É importante salientar que, ao ter disponível várias carrinhas, não implica que as tenha de usar a todas, sendo do interesse do padeiro precisamente o contrário, entenda-se, o objetivo será minimizar o percurso tomado por cada carrinha e utilizar o mínimo número de carrinhas para a realização de todas as entregas no horário pretendido. Em geral, o objetivo é diminuir os custos incorridos nas entregas. Isto tem de ser complementado com as condições já enunciadas nas implementações anteriores.

4. Frotas de carrinhas com capacidade limitada

Por último podemos impor a restrição de cada carrinha ter uma capacidade limitada de transporte de pão. Esta obriga a ter em conta as implicações anteriores assim como otimizar o volume de entregas realizada por carrinha.

Formalização do Problema

A solução deste problema passa pela formalização do mesmo em grafos e resolução recorrendo a algoritmos com estas estruturas.

Dados de Entrada

G(V,E) – grafo dirigido pesado constituído por:

- **V** – os vértices que representam as localizações geográficas do mapa e terão os seguintes atributos:
  - adj – ⊆ E lista de arestas que saem deste vértice;
  - coords – localização geográfica nas unidades pretendidas, de cada vértice.
- **E** – as arestas que representam as estradas que interligam os vértices.
  - orig – ⊆ V e representa o vértice de partida desta aresta;
  - dest – ⊆ V e representa o vértice de chegada desta aresta;
  - weight – tempo que demora a percorrer desde a origem até ao destino.

Clients – estrutura com todos os clientes, tendo cada um:

- id – identificação do cliente.
- vertex – ⊆ V indicativo de uma localização de entrega;
- scheduledTime – hora pretendida para a realização da entrega;
- breadsQuantity – número de pães pretendido.

Bakery – ⊆ V representa a localização geográfica da padaria (identifica o vértice inicial e final).

previousTolerance – intervalo de tempo de tolerância antes da hora, pretendida pelo cliente.

afterTolerance – intervalo de tempo de tolerância depois da hora pretendida pelo cliente.

Baker – estrutura que representa o padeiro e as suas informações.

- id – identificação do padeiro;
- startTime – horário a que o padeiro sai da padaria para começar as entregas;
- capacity – número de pães que a sua carrinha consegue transportar;
- deliveryTime – tempo gasto no processamento de uma entrega.

(os dois últimos atributos tornam-se relevantes na implementação 4 descrita na Descrição do Tema)

Dados de Saída

Informação relacionada a cada Baker:

- **Baker ID** – identificador do padeiro;
- **Route –** estrutura ordenada de vértices e respetivos tempos de entrega, que delineam o percurso a seguir pelo padeiro. Cada elemento deve especificar o arrivalTime a um vertexID e o leaveTime deve vértice indicando qual a aresta a seguir, edgeID (arrivalTime – vertexID – leaveTime - edgeID);

- **totalTime** – tempo previsto de duração do processamento da totalidade das entregas atribuídas (diferença de tempo desde que saiu da padaria até ao momento em que regressou);
- **averageDeliveryDelay**– Média do atraso das entregas realizadas
- **maximumDelay** – entrega com o maior atraso
- **minimumDelay** – entrega com o menor atraso

Restrições

- $\text{afterTolerance} \geq 0$, a tolerância (intervalo de tempo) após a hora pretendida para a entrega tem que ser positiva podendo ser 0;
- $\text{previousTolerance} \geq 0$, a tolerância (intervalo de tempo) para realização da entrega antes da hora pretendida tem que ser positiva, podendo ser 0;
- $\forall e \in E: e.weight > 0$, todas as arestas percorridas têm um weight maior que 0, isto é, todas têm um custo de tempo gasto a atravessá-las;
- $\forall c \in Clients\ c.scheduledTime \geq 0$, o tempo pretendido para a entrega tem que ser positivo e ser válido;
- $\forall b \in Baker: b.startTime \geq 0 \wedge b.deliveryTime > 0$, o horário para o início da entrega tem que ser positivo e válido dentro do formato horário escolhido assim como o intervalo de tempo gasto na realização de uma entrega, não pode ser negativo;
- $\forall c \in Clients: c.breadsQuantity > 0$, uma entrega apenas é realizada se o número de pães a entregar for maior que 0;
- $\forall b \in Baker: b.capacity > 0$, a carrinha de um baker tem capacidade maior que 0;
- $\forall c \in Clients\ \exists b \in Baker: c.scheduledTime \geq b.startTime$, todos os pedidos a serem realizados têm que ter um horário pretendido superior ao horário de saída de pelo menos um padeiro da padaria, sendo que são removidas, no pré-processamento, as entregas que não cumprirem este requisito;
- $\forall c \in Clients\ \forall b \in Baker: c.breadsQuantity \geq b.capacity$, não serão consideradas as entregas que ultrapassem a capacidade da maior carrinha em número de pães, sendo estas removidas no pré-processamento;
- $\exists! v \in V: v = Bakery$, existirá uma e uma só Bakery;
- $\forall v \in V: v.adj.orig = v$, todas arestas que pertencem ao vetor de adj de um vértice terão de ter como orig o próprio vértice;
- $\forall e \in E: e\ pode\ ser\ usada\ pela\ carrinha$, todas as arestas (caminhos) podem ser usadas pela carrinha, caso não se verifique serão removidos;
- $\forall c \in Clients: c.vertex \neq Bakery$, o vértice associado a um cliente nunca poderá ser a Bakery, pois a bakery não é uma localização de cliente valida;
- $\forall c \in Clients: \exists\ caminho\ para\ a\ padaria\ (gráfico\ conexo)$, é necessário que o Baker tenha sempre um caminho para a Bakery, visto que é o seu destino final;

- $Percurso[0].VertexID = Bakery.id$, o primeiro vértice do percurso terá de ser a Bakery visto que o Baker partirá de lá.
- $Percurso[0].arrivelTime = NULL$, o ponto vértice inicial do percurso não tem tempo de chegada.

- $n = Percurso.size, Percurso[n-1].vertexID = Bakery.id$, como o Baker tem de voltar á Bakery no final do percurso, portanto o último vértice do percurso é a Bakery;
- $n = Percurso.size,\ Percurso[n-1].leaveTime = NULL$, Baker após retornar á Bakery não volta a sair;
- $totalTime \geq 0$, sendo totalTime uma variação de tempo este valor terá de ser positivo. Caso tempo total $= 0$, baker não saiu da padaria;
- $0 \leq Média\ de\ atraso \leq afterTolerance$, a média de atrasos deverá variar entre 0 e afterTolerance;
- $maximumDelay \geq minimumDelay$, atraso máximo deverá sempre ser maior ou igual ao atraso mínimo, consequentemente atraso mínimo deverá sempre ser menor ou igual ao atraso máximo;
- $minimumDelay \geq 0$, atraso mínimo deverá ser maior ou igual a 0;
- $maximumDelay \leq afterTolerance$, como o Baker só poderá chegar no máximo afterTolerance depois da hora indicada, o atraso máximo nunca poderá ser maior que o valor de afterTolerance.

Funções Objetivo

Para se alcançar a solução ótima é necessário **minimizar o tempo total despendido em entregas**:

$n = Percurso.size$

- $Percurso[n-1].arrivelTime - Percurso[0].leaveTime = totalTime$, tempo total de viagem deve ser o menor possível.

Equilibrar os tempos de atraso entre os diversos clientes:

- $maximumDelay - minimumDelay$, quanto mais próximo de 0 menor é diferença entre atrasos.
- $\frac{\sum Atrasos}{nEntregas} = averageDeliveryDelay$, deverá ser o mais próximo de 0 possível.

Maximizar o número de entregas:

- $Clients.size - nEntregas$, deverá ser o mais próximo de 0 possível, sendo este o objetivo que iremos priorizar.

Perspetiva de Solução

Técnicas de Conceção

Preparação dos Dados

Nesta fase é pretendido realizar o carregamento de todos os dados. Tais como criar um grafo com todos os vértices e arestas pretendidos, indicar a localização da padaria, indicar a localização dos vários clientes e criar o/os padeiro/s. A tentativa de inserção de todas as arestas não úteis, (caminhos em obras) devem ser ignoradas.

Pré-processamento do grafo – Conetividade do Grafo

Dado que queremos trabalhar com um grafo dirigido pesado fortemente conexo, contendo apenas os dados essenciais à resolução do nosso problema, é essencial a sua preparação, fazendo um pré-processamento do grafo construído com os dados inseridos, para o obtermos da forma necessária, completa e apenas com as suas componentes essenciais.

Em primeiro lugar, o grafo é dirigido, dado que iremos trabalhar com um sistema de estradas que podem ter apenas um sentido ou ambos, por exemplo, temos dois pontos V1 e V2, a estrada que os une pode apenas permitir ir de V1 a V2 e não de V2 a V1, via de sentido único.

Como já foi especificado, o grafo pesado indica que cada aresta terá um peso associado, em que neste caso, representará o tempo gasto ao atravessar cada aresta.

O grafo fornecido inicialmente poderá não cumprir a necessidade de trabalharmos com um grafo fortemente conexo. Após o pré-processamento do grafo, teremos que ter, uma componente fortemente conexa deste, que contenha todos os vértices necessários a um percurso de uma carrinha, isto é, que contenha a padaria e todos os vértices correspondentes a entregas e onde seja possível regressar de novo à padaria.

As alterações ao grafo fornecido, serão realizadas por algoritmos que verificam a conetividade em grafos. Neste trabalho a conetividade terá sempre que partir de um vértice inicial que corresponde à padaria, pois dela partem todos percursos de entrega de encomendas e a ela têm que regressar. Utilizaremos então o vértice correspondente a esta como ponto inicial e de partida para o estudo da conetividade do grafo inicialmente fornecido.

A análise da conetividade terá que permitir reconhecer:

a) Quais os vértices, se estes existirem, que, partindo do vértice inicial acima referido (a padaria), não são alcançáveis, isto é, que não existe nenhum percurso possível que os una.
b) Os possíveis vértices que mesmo sendo alcançáveis a partir do vértice inicial escolhido, não consigam a partir deles mesmos chegar à padaria, ou seja, que não façam parte da mesma componente fortemente conexa do grafo.

Uma vez feito este estudo, os vértices mencionados serão descartados e removidos do grafo pois não poderão fazer parte da solução do nosso problema. Assim, após execução do algoritmo, o grafo obtido terá que ser fortemente conexo e terá que incluir o vértice inicial.

Existem vários algoritmos estudados que nos permitem realizar este processamento. Destacamos três, com as respetivas adaptações ao problema:

1. A primeira forma possível seria utilizar o algoritmo **Breath-First Search**. A melhor abordagem para a utilização deste algoritmo nos nossos dados, sem realizar cálculos desnecessários, seria em primeiro lugar realizar esta pesquisa partindo do vértice inicial. Sendo que daí, obtemos todos os vértices alcançáveis pela padaria, podendo então excluir os vértices não alcançáveis (a)). De seguida, com o objetivo de obter apenas uma componente conexa do grafo, é necessário realizar esta pesquisa, partindo de todos os vértices que não foram descartados, isto é, todos os vértices alcançáveis pela padaria, para verificar se a partir destes se consegue chegar ao vértice inicial, podendo-se interromper o algoritmo mal este seja encontrado. Os que não chegarem ao vértice inicial serão então removidos (b)).
2. O algoritmo de **Depth-First Search** é igualmente razoável para uso, sendo que a abordagem aos dados seguiria moldes semelhantes ao exposto em cima. Primeiramente, esta pesquisa partiria do vértice inicial e, em profundidade, chegaria a todos os vértices a este ligado, removendo do nosso grafo todos aqueles vértices que a este não estivessem direta ou indiretamente ligados (a)). Como referido em 1, para perceber se existe um retorno destes vértices para o vértice inicial, esta pesquisa teria que ser realizada em todos os vértices que ainda permaneciam no grafo, até ser encontrado o vértice correspondente ao vértice inicial. Se este não fosse encontrado, o vértice seria descartado (b)). Será importante comparar o uso destas duas diferentes pesquisas no nosso grafo e perceber qual das duas se torna mais benéfica. (? Ver vantagens de cada um e quando se usa mais cada um)
3. Outro algoritmo estudado e de possível uso, neste momento de processamento do grafo fornecido, pode ser o algoritmo de **Floyd-Warshall**. Este algoritmo descobre o caminho mais curto entre todos os pares de vértices do nosso grafo, com recurso a uma matriz de distâncias/adjacências, por exemplo A, e uma matriz de precedências que indicaram o percurso a seguir. Para além do estudo da conetividade, este algoritmo resolve logo o problema das distâncias entre todos os vértices pelo que pode vir a facilitar o trabalho que seria resolvido mais à frente. Se todas as distâncias entre dois vértices diferentes tiverem sido inicializadas com ∞, a análise da matriz revelará importantes informações sobre a conetividade do grafo para além de simultaneamente indicar o caminho mais curto entre todos os vértices. Note-se que, na matriz de adjacências obtida após execução do algoritmo, se existirem valores da matriz ainda com o valor: ∞, o significado deste valor revela que não existe qualquer ligação direta ou indireta entre os vértices correspondentes aos índices daquele valor na matriz. Assim, para todos os vértices da matriz em que $A[vi][v] = \infty$, isto significa que não existe uma conexão entre a padaria e esse vértice, logo pode ser descartado (a)). Nos restantes vértices se o inverso se der, isto é, se $A[v][vi] = \infty$, o percurso de retorno desses vértices para a padaria também não existe e assim terão também que ser descartados. Obtendo no final deste processo uma matriz de adjacências e um grafo apenas com os vértices membros da componente fortemente conexa do grafo inicial.

Será importante, no momento de implementação, realizar um conjunto de testes que nos permitam identificar quais os algoritmos e formas de implementação nos poderão dar um melhor resultado. Sendo que há ainda a possibilidade de implementarmos, por exemplo, um dos dois primeiros algoritmos referidos para excluir os vértices não alcançáveis (a) pela

padaria, e um algoritmo diferente para verificar quais os vértices que não alcançam a padaria (b).

1) Uma carrinha que realiza apenas uma entrega

A resolução deste problema resume-se ao cálculo do caminho mais curto entre a padaria, ponto de origem inicial, e o local da entrega, respetivo destino, e, posteriormente, o mesmo cálculo entre o local de entrega e a padaria, invertendo os papéis e obtendo o caminho de regresso.

Se for usado o algoritmo de **Floyd-Warshall** para o teste da conetividade do grafo no pré-processamento, a obtenção do percurso realizado pela carrinha é obtido através da matriz de precedência, já preenchida. Apenas seria necessário, então, encontrar os vértices perecedores que indicam o percurso da padaria para o local de entrega e o percurso contrário, que, pode efetivamente ser diferente, sendo que não seriam efetuados cálculos adicionais.

No caso do uso de **Floyd-Warshall** não se revelar "eficiente", outra possibilidade será usar um algoritmo de pesquisa do caminho mais curto entre dois pontos. Sendo que, será sempre necessária a aplicação deste algoritmo, para os percursos entre a padaria e o local de entrega (ida e volta).

Iremos explorar tanto o uso de **Dijkstra** como de **A*** e as suas respetivas derivações bidirecionais, comparando no momento da implementação, os seus resultados.

Ambos os algoritmos são muito semelhantes, embora o algoritmo de **A*** recorra ao uso de uma função heurística que se reflete num algoritmo de **Dijkstra**, mas com prioridades para outros caminhos podendo ou não resultar num ganho (speedup). A principal diferença reside na alteração que a função pode introduzir no custo/peso de uma aresta e o modo como isso interfere com a fila de prioridade usada. Sendo que a distância/custo da aresta no nosso grafo é o intervalo de tempo gasto a percorrê-la, será desta forma que iremos analisar e comparar os dois algoritmos.

Como nota, gostaríamos de referir que não utilizaremos o algoritmo estudado de **Bellman-Ford**, já que o algoritmo de **Dijkstra** resolve igualmente este problema do caminho mais curto entre dois pontos, com uma complexidade temporal base de $O(E + log(V))$, menor que $O(V \times E)$ do primeiro. Para além disso, o principal acrescento no uso de **Bellman-Ford** é que este permite arestas de peso negativo, que não temos qualquer intenção de utilizar já que o peso das arestas será referente ao intervalo de tempo gasto entre dois vértices.

Para além do problema e objetivo do nosso trabalho, de minimizar o tempo gasto por um padeiro na realização da entrega, é dada a possibilidade do cliente escolher um horário para realização da entrega. Assim, nesta primeira fase, esta revela-se de fácil implementação, obrigando o padeiro a sair da padaria o número de minutos que demora a realizar o percurso, da padaria para o local de entrega, antes do horário pretendido pelo cliente, desde que este seja após o começo do seu ofício.

2) Uma carrinha que realiza todas as entregas

Este problema revela-se muito mais complicado do que o anterior já que é preciso idealizar um percurso que percorra todos os locais indicados para as entregas, idealmente num menor intervalo de tempo e o mais próximo dos horários pretendidos pelos clientes, chegando sempre ao maior número de clientes possível.

A solução mais direta deste problema seria calcular todos caminhos e horários possíveis para realização das entregas e regresso à padaria, verificando no fim qual era a mais eficaz e que cumpria os objetivos impostos. Isto seria, usar uma **Brute-Force Search**. No entanto,

é evidente a impraticabilidade desta implementação dada a elevada complexidade temporal, ***O(V!)***, gasta nesta abordagem. Desta forma, torna-se indispensável a procura de melhorias.

A heurística do algoritmo do **Nearest Neighbour** apresenta uma melhoria significativa à implementação descrita em cima. Novamente, considerando as distâncias intervalos de tempo, como pesos das arestas, este é de fácil implementação e execução. No entanto, pode por vezes não retornar o percurso mais curto dada a sua natureza gananciosa. De notar que esta implementação não teria qualquer atenção aos horários pretendidos nem sempre nos iria obter a solução do nosso primeiro objetivo.

A segunda abordagem **greedy** que poderíamos impor seria ordenar e escolher as entregas a realizar mediante o horário pretendido para a entrega. Esta podia utilizar um algoritmo que calculasse os caminhos mais curtos entre dois vértices, sendo a sequência destes da ordem sequencial pela qual seriam ordenados. Partindo da padaria, realizar-se-ia a primeira entrega no primeiro horário e descobrindo o percurso menos demorado entre estes, seguindo, posteriormente, para o segundo cliente com o horário mais perto deste, calculando novamente a melhor rota recorrendo a algoritmos como **Dijkstra**, **A*** ou **Floyd-Warshall**. Embora esta abordagem possa trazer bons resultados em algumas execuções pode nem sempre chegar ao melhor resultado. Concluímos então que é essencial pensarmos noutro modo de implementação.

Assim, apresentamos outra possível conduta que poderá resolver melhor este problema. A ideia será uma abordagem com recurso a **backtracking** que permitirá não só maximizar o número de entregas dentro do horário, mas simultaneamente minimizar o tempo para a execução das entregas. Detalhando o procedimento que tentaremos implementar: a estratégia passará por incluir cada cliente ao percurso do padeiro, combinando os outros clientes e, por fim, avaliar a possibilidade de se conseguir realizar essa entrega ou não, continuando a cumprir os objetivos do problema. Os percursos mais curtos entre a padaria e os clientes e o percurso de retorno serão calculados pelos algoritmos já especificados como, por exemplo, **Floyd-Warshall** ou **Dijkstra**.

3) Frota de carrinhas que realizam entregas

Não temos um algoritmo específico previsto para este ponto, visto que temos perante nós um elevado número de decisões a tomar durante a implementação do programa. Assim, após pesquisar, o mais adequado a fazer nesta fase do problema seria implementar um algoritmo de **machine learning**, com o objetivo de estudar o sucedido para futuramente retirar as melhores decisões no desenrolar das entregas. Visto que este tipo de algoritmos não é estudado no âmbito desta unidade curricular, não iremos optar por esse caminho.

Desta forma, e mediante os resultados obtidos nas fases anteriores descritas, teremos que optar por uma abordagem de **Brute-Force** ou **Backtraking,** entre outras, que combinará e complementará a implementação descrita em cima, em que o objetivo será atender o maior número de clientes possível.

Um dos problemas que podemos antecipar será resultante de horários de entrega muito apertados, para tal, uma solução que temos em mente será enviar um padeiro, assim que possível, para atender os seus clientes. No momento em que um cliente não puder ser atendido, será enviado um padeiro que esteja disponível. Para minimizar os tempos das viagens de cada padeiro, é necessário, com regularidade, e assim que for possível terminar as viagens dos padeiros que estão há mais tempo a trabalhar.

Como já referido em cima, nesta fase será necessário analisar o que já alcançamos de modo a retirar as melhores decisões.

4) Frota de carrinhas com capacidade limitada

Neste ponto, as alterações relevantes em comparação com o ponto 3, serão em termos de alteração de valores e variáveis e/ou acréscimo/alteração de condições usadas nos algoritmos. Os algoritmos usados neste ponto serão idênticos aos usados no ponto anterior. Novas abordagens poderão ser testadas nesta fase.

Principais Algoritmos

Breath-first search

O nome deste algoritmo consegue logo de início fazer uma contextualização do modo como atua. Fundamentalmente, realiza uma pesquisa em largura sobre um grafo, isto é, partindo de um ponto inicial explora todos os vértices vizinhos desse mesmo e só depois explora os vértices vizinhos destes. Assim, continuamente chegará a todos os vértices alcançáveis pelo primeiro.

Este algoritmo, se utilizado, permitirá analisar a conetividade do grafo em questão, como referido em cima na secção de Pré-Processamento do grafo. Utilizaremos neste projeto um grafo dirigido que coincide e representa uma estrutura rodoviária. Assim sendo, é necessário preparar as estruturas de dados para tornar o algoritmo eficiente, quanto à possível e comum existência de ciclos. De forma a fugir aos ciclos infinitos que se poderão originar ao percorrer o grafo iremos usar a variável *visited* em cada vértice para nos indicar se este já foi visitado. Este valor de *visited* será, também, usado para nos indicar se o vértice foi alcançado. Para tal, é necessário inicializar este valor como não visitado, seguindo como exemplo o pseudocódigo, com complexidade temporal de ***O(V)***:

```
ClearVertexes(){

    for( v : V ) v.visited = false;

}
```

De forma a utilizar este algoritmo corretamente será necessário indicar um vértice onde a pesquisa vai começar. De seguida, todos os seus vértices adjacentes serão adicionados a uma estrutura do tipo **FIFO** (ex: *queue*) pela ordem em que foram encontrados, sendo esta a principal característica do algoritmo em questão. A análise dos vértices é, continuamente, feita enquanto a estrutura não estiver vazia. Sempre que se retira um vértice desta, o seu valor de *visited* é alterado, ficando-se a saber que o vértice já foi visitado. De seguida, e já referido anteriormente, todos os vértices não visitados e adjacentes a este final são adicionados à estrutura para posterior análise. Temos como exemplo do algoritmo o seguinte pseudocódigo de complexidade ***O(V+E)***:

```
BFS(vertex){

    vertex.visited = true;
    queue = QUEUE;
    queue.push(vertex);

    while(!queue.empty){
        v = queue.pop;

        for (edge : v.adj){
            if(edge.dest.visited == false){
                queue.push(edge.dest);
                edge.dest.visited=true;
            }
        }
    }
}
```

Além disso, é possível tirar proveito do algoritmo para outro tipo de análises com recurso a certas condições de paragem. Como foi referido, na análise da conectividade do grafo no seu pré-processamento, quando este algoritmo é aplicado iniciando nos vértices que são alcançáveis pela padaria, se o vértice encontrado na estrutura for a padaria não é necessário continuar a iterar sobre os restantes vértices, pois verificamos a dualidade de alcance dos dois vértices, padaria alcança vértice e vice-versa.

Como foi referido na secção de pré processamento as utilizações pretendidas para este algoritmo serão:

1) Verificar os vértices alcançáveis partindo da padaria.
2) Verificar a conectividade do grafo.

No ponto 1) o algoritmo terá uma complexidade de ***O((V) + (V + E))*** visto que será necessário limpar os vértices e correr o algoritmo BFS apenas uma vez.

No pronto 2) o algoritmo terá uma complexidade de ***O((V)*** $\times$ ***((V) + (V + E)))*** visto que será necessário limpar os vértices e correr o algoritmo BFS para todos os vértices válidos após o ponto 1). É importante referir que neste último ponto o número de vértices se encontra reduzido devido á primeira verificação, mas como o número de dados em análise será bastante elevado e tem em vista um percurso de estradas real, o número de remoções não deve ser suficientemente relevante.

Por fim, ao utilizar esta aproximação num pré processamento do grafo iremos ter uma complexidade temporal de ***O((V) + (V + E) + (V)*** $\times$ ***((V) + (V + E)))*** que se pode resumir a ***O((V)*** $\times$ ***((V) + (V + E)))*** e mais sucintamente a $O(2V^2+VE)$.

Em relação á complexidade espacial do algoritmo, no pior caso em que a padaria se encontra ligada a todos os outros vértices, estamos perante uma complexidade de ***O(V)***, visto que a estrutura terá de conter todos os vértices do grafo.

Depth-first search

Mais uma vez, o nome do algoritmo explica por si só, o tipo de pesquisa que é feita, esta que parte de um vértice inicial, explorando em profundidade os vértices que consegue alcançar. Assim sendo, para cada vértice explora o primeiro vértice adjacente e de seguida os seus vértices adjacentes, só retornando ao segundo vértice adjacente do inicial após ter percorrido todo o ramo do primeiro vértice adjacente.

A abordagem a ser tomada por este algoritmo é bastante idêntica ao anterior, sendo assim necessário ter em atenção, a existência de ciclos no grafo. Deste modo iremos também usar a variável *visited* em cada vértice. Esta restrição implicará o uso da função "*ClearVertexes()*" com complexidade temporal de ***O(V)***.

O uso deste algoritmo no contexto do nosso problema obriga-nos a indicar o vértice em que se deve começar a pesquisa, visto que queremos apenas trabalhar com os vértices alcançáveis a partir de uma determinada origem.

Passando agora à sua implementação concreta, existem duas variações para este algoritmo, utilizando uma estrutura do tipo **LIFO** (ex: stack) ou recursivamente. Na nossa implementação iremos optar pelo uso da recursividade visto que foi esta a metodologia

referida no contexto das aulas teóricas. A seguir, segue-se um exemplo de uma possível implementação deste algoritmo com complexidade temporal de ***O(V+E)***:

```
DFS(vertex){

    vertex.visited = true;

    for (edge : vertex.adj){
        if(edge.dest.visited == false)
            DFS(edge.dest);
}
```

Assim, quando chamada, a função, sobre um vértice identifica-o como visitado. De seguida percorre todas as suas adjacências fazendo uma chamada recursiva para todos os vértices que ainda não foram visitados.

Pretende-se usar este algoritmo para as mesmas análises que o algoritmo acima. Como as complexidades das suas implementações são idênticas e as condições também são semelhantes, as conclusões a retirar serão as mesmas. Portanto a complexidade temporal deste algoritmo no contexto do nosso problema será $O((V) \times ((V) + (V + E)))$ mais sucintamente a $O(2V^2+VE)$.

Em relação á complexidade espacial do algoritmo, no pior caso em que cada vértice está ligado apenas e necessariamente a um outro vértice não visitado, estamos perante uma complexidade de ***O(V)***, visto que a função será chamada para todos os seus vértices sem se finalizar nenhuma chamada.

Floyd-Warshall

O algoritmo de **Floyd-Warshall** é um algoritmo usado para encontrar o caminho mais curto entre todos os pares de vértices de um grafo. Sendo este um algoritmo de programação dinâmica, os dados calculados ao longo do algoritmo dependem de uma matriz de adjacências que irá resultar em duas matrizes 2D, a matriz de tempos e a matriz de precedência.

É possível usar este algoritmo em grafos orientados unidirecionais ou bidirecionais. No contexto do nosso problema ambos os tipos de ligações existem, portanto, o seu uso é possível. Uma complicação que este algoritmo apresenta é a incapacidade de lidar com ciclos negativos. No nosso caso a existência de ciclos negativos é de todo impossível pois o peso das arestas é caraterizado pelo tempo que o condutor a demora a percorrer, sendo este, necessariamente, positivo. Logo, não existindo arestas com pesos negativos não há ciclos negativos.

A seguir segue-se uma possível implementação deste algoritmo:

```
DIST[V.size()][V.size()];
PATH[V.size()][V.size()];

for (int i=0 ; i<V.size() ; i++)
    for (int j=0 ; j<V.size() ; j++)
        i == j ? DIST[i][j] = 0 , DIST[i][j] = ∞ ;

for (vertex : V)
    for (edge : vertex.adj)
        DIST[vertex.index][edge->vertex.index] = edge.weight;
        PATH[vertex.index][edge->vertex.index] = vertex.index:

for (int k=0 ; k<V.size() ; k++)
    for (int i=0 ; i<V.size() ; i++)
        for (int j=0 ; j<V.size() ; j++)
            if (DIST[i][k] + DIST[k][j] < DIST[i][j]){
                DIST[i][j] = DIST[i][k] + DIST[k][j];
                PATH[i][j] = PATH[k][j];
            }
```

Analisemos agora a complexidade temporal do algoritmo apresentado. Podemos dividir este algoritmo em duas partes.

Inicialmente, é feita a inicialização das matrizes *Dist* e *Path*. Nesta fase a matriz de adjacências (transformada em *Dist*) é inicializada com os pesos das arestas contidas no grafo. Sendo i e j os índices dos vértices do grafo, cada célula da matriz representada por $[i][j]$ é preenchida com a distância do i^{th} vértice até ao j^{th} vértice. Caso não exista nenhuma aresta que ligue estes dois vértices a distância é mantida a infinito. Se $i = j$ (índices iguais), os vértices são os mesmos logo o valor a colocar na matriz deverá ser 0. Nesta fase, o algoritmo apresenta uma complexidade temporal de ***O(V²)***, já que o vetor dos vértices do grafo é percorrido 2 vezes de forma encadeada.

Na segunda parte do algoritmo, procede-se ao preenchimento dos valores corretos nas matrizes. Todas as possibilidades de caminhos entre i e j que passem por k, sendo k um vértice do grafo, são analisadas. Caso o caminho que passa por k,soma dos valores $[i][k]$ e $[k][j]$ da matriz, seja menor que o valor $[i][j]$ atualmente na matriz *Dist* o seu valor é alterado, tanto nesta como na matriz *Path*. Nesta fase o algoritmo apresenta uma complexidade de ***O(V³)***.

Uma característica deste algoritmo é a necessidade de ser executado apenas uma vez, já que as matrizes resultantes contêm os índices e pesos mínimos entre qualquer par de vértices. Segue-se um exemplo de uma possível matriz de adjacências para um grafo com 4 vértices:

$$\begin{bmatrix} 0 & 4 & 2 & \infty \\ 2 & 0 & 7 & 6 \\ \infty & 3 & 0 & 4 \\ 2 & \infty & 6 & 0 \end{bmatrix}$$

A leitura de um valor localizado em $[i][j]$, sendo i a representação das linhas e j a representação das colunas, deverá ser do género: “A distância mínima entre o ponto i e o ponto j é $[i][j]$”.

Semelhante aos algoritmos anteriores as utilizações pretendidas para o algoritmo de **Floyd Warshall** serão:

Verificar quais os vértices alcançáveis partindo da padaria.

Verificar a conectividade do grafo.

Adicionalmente:

Indicar os vértices a percorrer para fazer uma determinada viagem.

No ponto 1) os vértices relevantes serão todos aqueles que tem um valor diferente de infinito, na linha correspondente ao vértice da padaria na matriz de tempos ou de melhores caminhos. No exemplo da matriz acima, caso a padaria seja identificada pela primeira linha da matriz, o valor localizado em $[0][4]$, último valor da primeira linha, é igual a ∞, portanto, no nosso contexto este vértice poderia ser removido, visto que não existe caminho da padaria para o ponto em questão. Ficando a matriz *Dist* igual a:

$$\begin{bmatrix} 0 & 4 & 2 \\ 2 & 0 & 7 \\ \infty & 3 & 0 \end{bmatrix}$$

No ponto 2) todos os vértices que apresentem um valor igual a infinito na coluna do vértice da padaria deveram ser removidos, pois não apresentam um caminho válido para a padaria, colocando em causa a conectividade do grafo. Mantendo o exemplo da matriz indicada, caso a padaria esteja identificada pela primeira coluna da matriz, o valor localizado em $[3][0]$, último valor da primeira coluna, é igual a ∞, portanto, no nosso contexto este vértice poderia ser removido, visto que não existe caminho do ponto em questão para a padaria. Ficando a matriz igual a:

$$\begin{bmatrix} 0 & 4 \\ 2 & 0 \end{bmatrix}$$

Como podemos ver após este pré-processamento foram reduzidos os dados em análise.

Em consequência deste algoritmo surge também uma matriz, anteriormente referida, de precedência, muito parecia à matriz de tempos, em que para cada célula $[i][j]$, o valor indica o índice do vértice a percorrer após i em direção a j, de forma a percorrer o menor caminho entre estes dois pontos. De referir que os valores calculados através da matriz *Dist* podem ser calculados usando a matriz *Path* e a respetiva soma dos pesos de cada aresta percorrida entre vértices, contudo, e de forma a não executar esta chamada mais que uma vez, a melhor escolha é guardar ambas as matrizes. A matriz *Path* será principalmente usada para resolver o ponto "3) indicar os vértices a percorrer para fazer uma determinada viagem".

Concluindo, a complexidade temporal que este algoritmo trará para o nosso programa, caso usado, será $O(V^3 + V^2)$, podendo ser simplificado para $O(V^3)$.

No que toca à complexidade espacial, este algoritmo apresenta uma complexidade de $O(V^2+V^2)$, simplifica-se para $O(2V^2)$ visto que é necessário guardar duas matrizes de duas dimensões, que representam o caminho mais curto entre todos os pares de vértices e respetivos índices entre estes.

Dijkstra

Ao contrário do algoritmo de **Floyd-Warshall**, o algoritmo de **Dijkstra** pretende encontrar o melhor percurso numa rede entre dois pontos. Escolhido o vértice de início de busca, este algoritmo calcula o custo mínimo do vértice para os demais pertencentes ao grafo.

Trata-se de um algoritmo ganancioso e com um bom nível de performance. Funcionalmente parte de uma estimativa para o custo mínimo entre o vértice inicial e os restantes, que no nosso caso simboliza o tempo, e vai sucessivamente ajustando esta estimativa de forma a maximizar o ganho imediato.

Para o problema em questão é possível aplicar uma modificação à qual apenas interessa obter o caminho mínimo entre um certo ponto e outro, específico. Tal pode ser conseguido terminando o algoritmo mal o vértice a retirar da fila de prioridade seja o vértice de destino.

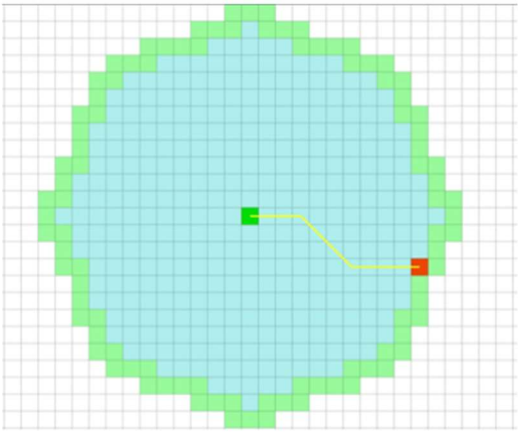
Em seguida é apresentado uma possível implementação do algoritmo:

```
Dijstra(origin, destiny) {

    for ( vertex:V ){
        vertex.dist = ∞;
        vertex.path = -1;
    }

    origin.distance = 0;
    priorityQueue queue = PRIORITYQUEUE;

    queue.insert(origin);

    while(!queue.empty()){
        Vertex vert = queue.extractMin();

        if( vert=destiny ) return;

        for(Edge edge:vert.adj){
            if(edge.dest.dist > vertex.dist + edge.weight){

                double oldDist = edge.dest.dist;
                edge.dest.dist = vertex.dist + edge.weight;
                edge.dest.path = vertex.index;
                if (oldDist == LONG_MAX) {
                    queue.insert(edge.dest);
                } else {
                    queue.decreaseKey(edge.dest);
                }
            }
        }
    }
}
```

Analisando as complexidades temporal e espacial, é possível obter diferentes complexidades de acordo com as estruturas usadas. Caso seja usado um vetor para armazenar os vértices obtemos complexidade temporal ***O(V²)***. Por outro lado, se optarmos por uma fila de prioridade como é usado no pseudocódigo apresentado, a complexidade obtida é ***O((V+E) * log(V))***, devido à inserção nesta ser logarítmica. Além disso, é possível otimizar o tempo para ***O(V * log(V) + E)***, com o uso de *Fibonacci Heaps* e ***O(V * log(V))*** com *Binary Heaps*.

Em relação á complexidade espacial esta será de ***O(V)***, pois no pior caso a estrutura utilizada poderá conter todos os vértices pertencentes ao grafo.

No contexto do nosso problema este algoritmo irá ser usado, principalmente, para encontrar o menor percurso entre dois pontos. Por fim apresentamos uma execução do algoritmo de **Dijkstra**, que melhor ilustra o seu comportamento e desempenho.



Dijkstra Bidirecional

Falemos agora do algoritmo de **Dijkstra bidirecional** um algoritmo baseado em **Dijkstra**. O funcionamento deste algoritmo consiste em executar o algoritmo de **Dijkstra** no ponto de origem e no ponto de destino. Deste modo ambas as partes ficam com o dever de encontrar o menor caminho entre o ponto de origem associado e todos os outros. Assim que uma das partes encontre um vértice já visitado pela outra execução o algoritmo irá parar. O percurso mínimo entre o ponto de origem e o de destino será dado pela união do caminho mais curto da origem ao ponto de interseção com o caminho mais curto do ponto de interseção ao ponto de destino.

No caso do nosso problema, como o grafo é dirigido, a execução do algoritmo de **Dijkstra** a partir do ponto de destino trará algumas complicações provenientes do facto de ser necessário realizar um **backwards search**. Procuramos resolver este problema calculando um grafo revertido adicional. Este grafo deve conter todos os vértices do grafo original, mas todas as arestas nele presentes devem ser invertidas. Realizando este processamento será possível usar o algoritmo de **Dijkstra bidirecional** no nosso problema. A seguir apresentamos uma possível implementação do mesmo:

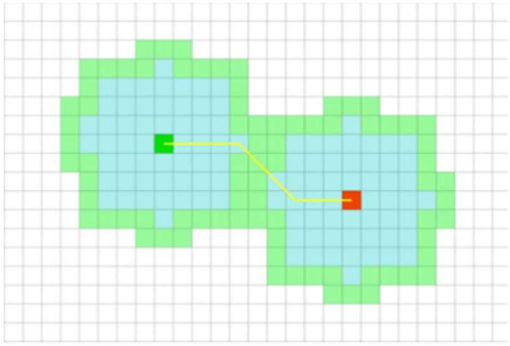
```
BiDijkstra(queueOrig, queueDest){

    minDist = MAX_INT; // minimum orig->dest path distance
    while (!queueOrig.isEmpty() and !queueDest.isEmpty()){

        u = queueOrig.extractMin();
        v = queueDest.extractMin();
        u.normalVisited = true;
        v.reverseVisited = true;

        if (u.dist + v.dist >= minDist)
            return;

        for x : adj(u){
            if (x.dest.dist > u.dist + x.weight)
                x.dest.dist = u.dist + x.weight;
            if (x.reverseVisited and ( u.dist + x.weight + x.dist ) < minDist)
                minDist = u.dist + x.weight + x.dist;
        }

        for x : adj(v){
            if (x.dest.dist > v.dist + x.weight)
                x.dest.dist = v.dist + x.weight;
            if (x.normalVisited and ( v.dist + x.weight + x.dist ) < minDist)
                minDist = v.dist + x.weight + x.dist;
        }
    }
}
```

No pior caso possível e utilizando uma fila de prioridade este algoritmo trará para o nosso problema uma complexidade temporal de ***O(V * log(V) + E)***. É relevante referir que o pior caso possível deste problema raramente irá acontecer, portanto pensamos que será preferível a sua utilização ao algoritmo de **Dijkstra** normal. Pois com o uso de **Dijkstra bidirecional**, caso os pontos estejam distantes, o número de vértices visitado irá reduzir, diminuindo assim o tempo na execução do algoritmo.

Em relação á complexidade espacial esta será de ***O(2V + E)***, sendo necessário guardar os vértices a percorrer numa fila prioridade, bem como um grafo revertido.

No contexto do nosso problema este algoritmo irá ser usado, principalmente, para encontrar o menor percurso entre dois pontos.

Por fim apresentamos uma execução do algoritmo de **Dijkstra Bidirecional**, que melhor ilustra o seu comportamento e desempenho.



A*

O algoritmo **A-estrela**, trata-se de um melhoramento heurístico do algoritmo de **Dijkstra**. Desta forma, é um algoritmo ganancioso cujo objetivo consiste em encontrar o caminho mais curto entre dois pontos.

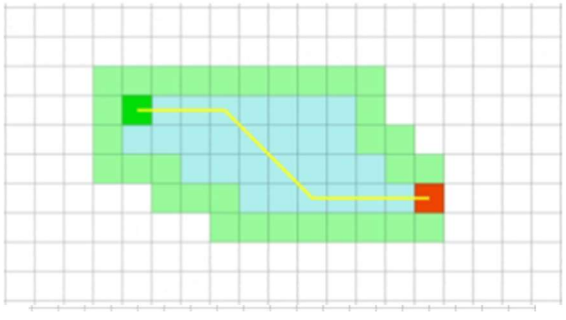
Como não possuem o mesmo nome, claramente existe uma diferença, diferença que reside no facto deste algoritmo aplicar uma alteração aos pesos das arestas, no nosso caso os intervalos de tempo, de maneira que a ordem dos vértices na fila de prioridade seja diferente da do algoritmo em cima exposto.

Os novos valores são calculados com base na seguinte equação, $f(v) = g(v) + h(v)$, em que $g(v)$ é o custo atual desde o vértice inicial até v, já a função $h(v)$ é a real responsável pela alteração dos respetivos pesos, estimando a distância deste ponto ao final. Existem várias possibilidades para definir a função heurística, $h(v)$, sendo que a que pretendemos usar no nosso projeto será com base no cálculo da distância **euclidiana** entre o ponto atual e o destino. Com tudo outras funções heurísticas poderão ser testadas.

No pior caso possível a complexidade temporal deste algoritmo será idêntica á do algoritmo de **Dijkstra**, admitindo que usamos uma fila de prioridade, $O((V+E) * log(V))$. É de notar que de acordo com as características do nosso grafo, que representa uma rede rodoviária, esperamos que o algoritmo de **A*** quase nunca seja usado com a sua pior complexidade. Concluímos, portanto, que este algoritmo nos trará uma melhoria em comparação com o algoritmo de **Dijkstra**, devido á sua melhoria heurística.

Em relação á complexidade espacial esta será de $O(V)$, pois no pior caso a estrutura utilizada poderá conter todos os vértices pertencentes ao grafo. No contexto do nosso problema este algoritmo irá ser usado, principalmente, para encontrar o menor percurso entre dois pontos.

Por fim apresentamos uma execução do algoritmo de **A***, que melhor ilustra o seu comportamento e desempenho.



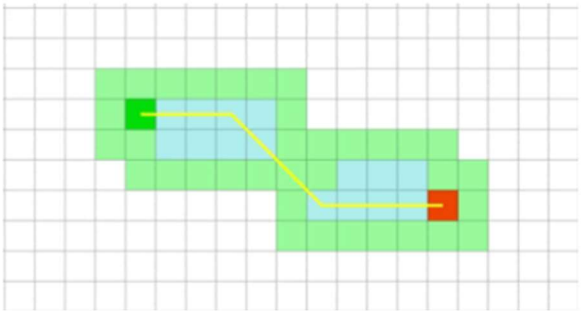
A* Bidirecional

O algoritmo **A-estrela bidirecional** é a junção do algoritmo de **A*** com uma abordagem bidirecional, igual á abordagem usada em **dijkstra bidirecional**. De modo a podermos usar este algoritmo na nossa resolução será preciso, uma vez mais, ciar um grafo invertido para podermos realizar uma **Backwards Search**. A ordenação dos vértices na fila de prioridade será com base no resultado da função **heurística** usada. A condição de paragem será a mesma: assim que for encontrado, por uma das execuções, um vértice já visitado pela outra execução.

No pior caso a complexidade temporal do algoritmo será ***O((V+E) * log(V))***, caso seja usada uma fila de prioridade.

A complexidade espacial será ***O(2V + E)***, resultante do uso da fila de prioridade e do grafo invertido.

Por fim apresentamos uma execução do algoritmo de **A* bidirecional**, que melhor ilustra o seu comportamento e desempenho.



Brute-Force Search

Como próprio nome indica um algoritmo de força bruta irá sistematicamente, enumerar todas as soluções candidatas analisando se estas satisfazem os objetivos do problema. Ao usar este algoritmo na nossa resolução do problema iremos procurar reduzir ao máximo a sua complexidade temporal. Contudo não temos grandes esperanças de que será a abordagem escolhida no resultado final do nosso projeto, consequência do elevado tempo de processamento associado. Por outro lado, como este algoritmo irá enumerar todas as possíveis combinações de resolução podemos concluir que o resultado final será bastante assertivo.

Nearest Neighbour

Esta abordagem vai procurar encontrar o cliente mais próximo a visitar de modo a reduzir ao máximo o seu tempo de viagem de modo ganancioso. Assim que um cliente for encontrado ele será atendido imediatamente. Esta abordagem irá apresentar rapidamente um caminho a seguir, preocupando-se menos com os horários validos de cada cliente, podendo mesmo assim, nem chegar ao exato caminho mais curto.

Greedy

Este algoritmo irá tomar partido de uma abordagem gananciosa. Essa abordagem consiste em cada estado escolher a solução ótima, com isto quero dizer que após cada entrega a entrega mais próxima, temporalmente, será escolhida. Caminhos ou entregas que podem vir a ser benéficos não serão analisados. Podemos concluir que o caminho escolhido

poderá não ser o melhor, logo o resultado final deste algoritmo não será muito assertivo. Contudo, como só nos preocupamos com a melhor escolha o tempo gasto em processamento por parte deste algoritmo será bastante reduzido.

Backtracking

O algoritmo de **backtracking** é apresentado, normalmente, como um melhoramento do algoritmo de força bruta, visto que ele irá analisar múltiplas soluções, mas algumas irão ser eliminadas a meio do processamento. Portanto este algoritmo irá apresentar uma redução do tempo de processamento, comparando com o algoritmo de força bruta. Por outro lado, como o algoritmo de **backtracking** é um melhoramento do algoritmo de força bruta é importante referir que sua assertividade não irá ser reduzida. Portanto este é um candidato a ser usado no resultado final.

Casos de Utilização

Queremos implementar possíveis soluções para as quatro fases de implementação descritas inicialmente no nosso problema, assim como a interface que permitirá testar e apresentar os nossos resultados.

Em primeiro lugar, é de esperar que seja possível escolher um mapa existente ou fazer download de outro. Será então especificado qual o grafo, vértices e arestas por onde o/os padeiro/os se deslocarão.

Concluindo o set-up do grafo, será possível indicar possíveis estradas que não estão válidas.

A próxima fase da execução do programa passará por indicar o local da padaria, bem como as tolerâncias de tempo para as entregas e o horário desejado destas. Posteriormente será possível escolher qual dos problemas (1,2,3,4) a resolver:

1. No primeiro caso, o padeiro deverá indicar o horário de início do trabalho e o tempo de processamento de uma entrega, sendo que apenas é necessário indicar complementarmente o local da entrega, bem como o número de pães a entregar e o horário pretendido para a entrega.
2. Na segunda opção, é imprescindível indicar toda a informação de cada cliente, isto é, local de entrega, número de pães e horário, bem como as informações do horário do padeiro, descritas em 1.
3. Nesta opção, será dada a opção de ir incrementalmente adicionando padeiros com diferentes horários. Seguidamente, terá de ser introduzida a informação dos clientes tal e qual como nos pontos anteriores.
4. Como já foi especificado, neste ponto, em comparação com o anterior, obrigará a introdução do limite máximo de cada carrinha usada por cada padeiro.

Adicionalmente, gostávamos de implementar um modo que nos permita escolher quais os algoritmos usar para a conetividade e para o cálculo dos caminhos mínimos entre dois pontos, para que seja possível comparar tempos de execução.

Deste modo, achamos que esta implementação irá permitir desde logo fazer boas escolhas para o desenvolvimento do trabalho.

Conclusão

A realização deste trabalho revelou-se trabalhosa e diferente do que esperávamos. Dado ser a primeira vez a trabalharmos com um problema deste género, foi difícil colocar em palavras aquilo que serão os nossos objetivos sem antes tentar e experimentar.

A complexidade do problema e as muitas opções de resolução não permitem desde o início optar por uma estratégia sem antes compararmos resultados e execuções. Assim, sentimo-nos obrigados a expor as abordagens pela qual o nosso trabalho passará no momento da implementação.

Estamos conscientes que foi um método desafiante para planear um projeto que provavelmente seria elaborado apenas em código e comparado desse modo. Este relatório permitiu que ficássemos a perceber melhor os algoritmos usados na resolução destes problemas. Ainda que abstratamente pensássemos na sua resolução, tornou-se real a possibilidade do planeamento e pensamento por de trás de um projeto deste âmbito desta forma.

O desenvolvimento deste relatório, bem como o pensamento da sua estrutura, organização e conteúdo foi realizado sempre em grupo de modo que todos os seus elementos tenham contribuído igualmente para o seu crescimento. Assim, e visto que foi realizado em tempo de pausa letiva, trabalhamos maioritariamente sempre em conjunto em chamada online e não foi necessário propriamente dividir tarefas.

Referências Bibliográficas

Slides informativos das aulas teóricas.

https://www.programiz.com/dsa/graph-dfs

https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

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

https://www.youtube.com/watch?v=4OQeCuLYj-4

https://www.programiz.com/dsa/floyd-warshall-algorithm

https://cse442-17f.github.io/A-Star-Search-and-Dijkstras-Algorithm/

https://pt.wikipedia.org/wiki/Algoritmo_de_Bellman-Ford

https://pt.wikipedia.org/wiki/Algoritmo_do_vizinho_mais_pr%C3%B3ximo

https://pubsonline.informs.org/doi/pdf/10.1287/opre.43.2.367

https://drive.google.com/file/d/1vu2JBV_9jFDnVoDiMk44ZGZRohjaURNT/view?usp=sharing

https://blog.engeman.com.br/o-que-e-um-algoritmo-de-decisao-veja-exemplos/

https://www.homepages.ucl.ac.uk/~ucahmto/math/2020/05/30/bidirectional-dijkstra.html

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

https://pt.wikipedia.org/wiki/Problema_de_roteamento_de_ve%C3%ADculos