Índice

- 1. Introdução
  - 1.1 Descrição do Tema
- 2. Formalização
  - 2.1 Problematização
  - 2.2 Estratificação
    - 2.2.1 Fase I
    - 2.2.2 Fase II
    - 2.2.3 Fase III
    - 2.2.4 Fase IV
  - 2.3 Dados de Entrada
  - 2.4 Dados de Saída
  - 2.5 Restrições
    - 2.5.1 Dados de Entrada
    - 2.5.2 Dados de Saída
  - 2.6 Função Objetivo
- 3. Perspetiva de Implementação
  - Algoritmos considerados
    - 3.1.1 Dijkstra Unidirecional
    - 3.1.2 Dijkstra Bidirecional
    - 3.1.3 Algoritmo A*
    - 3.1.4 Algoritmo Floyd-Warshall
    - 3.1.5 Abordagens consideradas na Fase III
- 4. Casos de utilização a serem suportados
- 5. Implementação
  - 5.1 Estruturas de dados
  - 5.2 Grafos e Mapas usados
  - 5.3 Algoritmos efetivamente implementados
  - 5.4 Análise empírica
    - 5.4.1 Fase I
    - 5.4.2 Fase II
    - 5.4.3 Fase III
    - 5.4.4 Fase IV
- 6. Algoritmos e Estratégias implementadas
  - 6.1 Um ou vários pedidos por ordem temporal
  - 6.2 Pedido de múltiplos restaurantes
  - 6.3 Um estafeta - Vários pedidos no mesmo deslocamento (variante do TSP)
  - 6.4 Nota geral acerca dos algoritmos implementados
- 7. Principais casos de uso implementados
  - 7.1 Realização de um pedido
  - 7.2 Simulação de múltiplos pedidos
    - 7.2.1 Um estafeta - Vários pedidos no mesmo deslocamento (variante do TSP)
    - 7.2.2 Múltiplos estafetas - distribuição de pedidos por ordem temporal
- 8. Conclusão
- 9. Referências

1. Introdução

1.1 Descrição do Tema

A EatExpress é um sistema de entrega de comida entre os restaurantes registados na plataforma e os utilizadores da sua aplicação. Quando um utilizador acede à aplicação, escolhe o restaurante e o prato que quer, e o seu pedido é entregue no local onde o utilizador se encontra, por um estafeta que utiliza o seu próprio meio de transporte para lá chegar (a pé, bicicleta, mota, carro etc).

Neste projeto procuramos criar um sistema para gerar os percursos mais curtos a ser percorridos pelos estafetas, entre as localizações dos restaurantes e dos clientes.

Como base para o estudo, formalização, interpretação, e garantia da eficácia dos algoritmos usados, apoiar-nos-emos nas bases da teoria dos grafos. Adotaremos uma estratégia *bottom-up*, começando com variantes mais simples do problema, e avançaremos, à medida que forem testados e escolhidos novos e mais eficientes algoritmos, até ao sistema mais complexo de funcionamento da aplicação.

Para garantir a plausibilidade da resolução, informação real e atualizada será usada, para garantirmos a sua aplicabilidade em problemas verdadeiramente reais, do nosso mundo.

2. Formalização

2.1 Problematização

Em primeiro lugar, convém dividir todas as questões relativas aos vários componentes da aplicação.

O que se pretende com esta análise é garantir a máxima eficiência na implementação de um algoritmo que permita criar e gerir, de forma rápida e precisa, as rotas dos estafetas de entrega de comida, entre as localizações dos restaurantes e as localizações dos clientes que efetuaram os pedidos.

Não serão equacionadas as implementações da interface da aplicação, ao nível do utilizador, nem serão alvo de análise todos os aspetos não relacionados com grafos, ou algoritmos não abordados.

Assim sendo, o problema reduzir-se-á a encontrar os caminhos mais curtos entre pontos/vértices, num mapa de atuação pré-escolhido, e, por fim, tendo em conta as diversas variantes do problema, definir rotas que permitam navegar pelos percursos, percorrendo a menor distância possível.

2.2 Estratificação

A estratégia a seguir, para a resolução deste problema, sugere a divisão do conceito em fases de análise, de implementação e de deliberação. No entanto, como pré-estabelecida tomamos a ideia de que um estafeta em trabalho, antes de receber a sua rota, pode estar localizado num qualquer ponto do mapa. Esta e a generalização de que existirão sempre, no mínimo, um restaurante e uma localização de um cliente como pontos integrantes da rota, serão o ponto de partida para a análise que se segue.

Convém definir, também, o perfil do estafeta, enquanto entidade que se move pelo percurso estabelecido, com características próprias que serão definidas pelo meio de transporte em que se desloca e pela capacidade máxima que pode transportar. Estas condicionantes influenciarão a escolha da rota e serão tidas em conta, antes de o estafeta iniciar a entrega dos pedidos.
É imperativo realçar também que, em qualquer percurso, o levantamento dos pedidos nos restaurantes terá de ser efetuado antes da sua entrega aos clientes, por uma questão de fiabilidade e praticabilidade real da situação.

Outro aspeto importante é a existência de zonas de desconectividade, nos mapas - a existência de obstáculos à circulação, particularmente, a ocorrência de obras na via pública, que pode inviabilizar o acesso a certas moradas e restaurantes, ao tornar zonas inacessíveis. Também isto será tido em conta, aquando da esquematização dos algoritmos.

Numa última nota, destaca-se a possível predeterminação de todos os restaurantes e estafetas registados na plataforma. Este detalhe será substancialmente importante para a adoção de determinados algoritmos, sobretudo se se impuser, ou não, com base na estratégia a utilizar, um pré-processamento dos dados, em particular, a cada novo registo, de uma destas entidades, na plataforma.

2.2.1 Fase I

Na sua simplicidade, consideramos a existência de apenas um estafeta, que realiza, sequencialmente, os percursos que lhe são impostos. Nesta fase, de modo a focar a análise na essência do problema, será considerado um meio de transporte qualquer e desvalorizada a capacidade de transporte. Na

sua medida, o único funcionário terá rotas na qual concretizará o levantamento de pedidos, nos restaurantes, e a sua entrega, na morada dos clientes. Será, portanto, considerado o caso atómico de um estafeta que entrega apenas um pedido, entre um restaurante e a morada de um cliente.

2.2.2 Fase II

Nesta fase, serão considerados vários estafetas que utilizam o mesmo meio de transporte e para os quais é delineado um percurso. Aqui, estaremos perante a distribuição simultânea de pedidos, com a avaliação de todas as condicionantes relacionadas com o problema, nesta fase.
Os pedidos estarão, logicamente, organizados numa escala temporal, ficando encarregue por um dado pedido o estafeta que se encontre mais perto do restaurante a ele associado.

2.2.3 Fase III

Esta fase coincidirá com a implementação de variados meios de transporte, o que poderá corresponder à utilização de diferentes mapas, para atender às características das múltiplas entidades que entregarão os pedidos e, em simultâneo, às características das diversas vias. Será igualmente imprescindível ter em conta a capacidade máxima que cada estafeta pode transportar, pois a dimensão da encomenda passa a ser determinante na atribuição dos pedidos.
Assim, nesta fase, a atribuição de um pedido a um estafeta fica a depender não só da sua proximidade ao restaurante, mas também das características próprias do meio de transporte por ele utilizado, preferindo-se os meios mais rápidos para encomendas que impliquem deslocações maiores e atribuindo as encomendas de maior dimensão aos estafetas com maior capacidade de transporte.

Para agilizar todo o processo, tendo em conta os múltiplos fatores envolvidos na seleção de um estafeta para a realização de um pedido, optar-se-á, numa pré-seleção, por excluir estafetas que não tenham capacidade para o transportar. Seguidamente, como não se pretende ocupar estafetas com encomendas de dimensão muito inferior à sua capacidade máxima de transporte, por poderem ser necessários para os pedidos seguintes, terá que se avaliar, simultaneamente, a capacidade máxima de transporte de cada estafeta e a sua distância ao restaurante. Para isso, será atribuído um peso de 50% a cada um destes fatores, aquando da escolha entre estafetas para a realização de um pedido.
Para além disso, ao avaliar a possibilidade de escolha dos estafetas, será necessário definir uma distância máxima para aqueles que se desloquem a pé, ou de bicicleta, já que a utilização destes meios de transporte para longas distâncias, apesar de não se refletir na distância total percorrida, resultaria, em situações reais, num tempo de entrega extremamente longo.

Por fim, consideraremos também a possibilidade de um pedido englobar vários restaurantes. Mais uma vez, isto implicará escolher, em primeiro lugar, os estafetas elegíveis, com base na sua disponibilidade para transportar a dimensão do pedido. Posteriormente, delinear-se-á a melhor opção de percurso para cada estafeta recolher o pedido dos vários restaurantes e entregá-lo ao cliente, considerando todas as restrições descritas anteriormente, no que diz respeito à capacidade, distância e tipo de transporte utilizado.

2.2.4 Fase IV

Aqui, entrará em consideração a existência de vários obstáculos nas vias, identificados em cima, o que levará a uma seleção mais restritiva do percurso e do tipo de estafeta encarregue de determinado pedido. Além disso, o tratamento dos grafos poderá sofrer alterações, sobretudo ao nível do pré-processamento.

2.3 Dados de Entrada

Os dados a recolher, antes da execução de qualquer dos algoritmos, para os percursos de cada um dos estafetas, são generalizáveis na seguinte lista:

- Grafo $G(V, E)$, dirigido, representando o mapa de vias (poderemos estar perante múltiplos grafos como opção para vários tipos de veículo), a percorrer pelos estafetas, onde estão localizados, como parte integrante, os pontos de recolha e entrega dos pedidos.
  - Cada vértice $v \in V$ terá os seguintes atributos:
    - Uma lista de arestas $Adj(v) \in E$, que partam desse vértice;
    - Um id, identificativo;
    - Um par de coordenadas $coords$, representando a localização real do respetivo ponto.
  - Cada aresta $e(v, u) \in E$, que parta de um dado vértice v será caracterizada por:
    - Um vértice $u \in V$ de destino;
    - Um peso $weight$ associado, relacionado com o seu comprimento real, expresso numa medida de comprimento espacial;
    - Um estado $state$, que representará a transitabilidade da via;
    - Um campo $name$, sem valor expressivo, servindo de identificador.
- Conjunto de pontos R, representando os restaurantes registados na plataforma:
  - Restaurante $r \in R$ com atributos:
    - O seu id, único, identificativo;
    - A referência para o vértice v do grafo, que representa a sua posição.
- Conjunto de estafetas $Employees$:
  - Estafeta $employee \in Employees$, com a informação sobre:
    - A sua posição inicial/no momento, s, $s \in V$;
    - A sua capacidade máxima de transporte, $maxCargo$;
    - O seu meio de transporte, $type$;
    - A sua disponibilidade, $ready$, para iniciar uma nova tarefa;

        - A sua velocidade média, $avgSpeed$, para permitir a apresentação do tempo estimado de entrega.
- Lista ordenada de Pedidos P:
    - Pedido $p \in P$ com a informação detalhada sobre:
        - Data e Hora, $time$, em que foi realizado;
        - Lista, $checkPoints$, de pontos/vértices que façam, obrigatoriamente, parte do percurso (referência apenas aos restaurantes e localização dos clientes);
        - Carga total, $cargo$, ocupada pelos itens que o constituem.

Estes dados poderão fazer parte de um pré-processamento existente, consoante o algoritmo escolhido e o problema a resolver. Se esses cálculos iniciais existirem, no caso de um algoritmo que estabeleça as distâncias mínimas entre cada par de vértices, seguir-se-á, então, a inevitável instanciação dos pedidos, com a sua organização em tarefas e a requisição de estafetas para as realizar, terminando no cálculo do caminho final a ser percorrido pelos mesmos.

2.4 Dados de Saída

Tendo sido o grafo analisado, tratado e traduzido numa das várias formas plausíveis de representação, poderá ser retornado, como dado de saída.
O importante, realmente, aqui, é a entrega das tarefas aos estafetas, o que resultará nos seguintes dados:

- Conjunto de tarefas $Tasks$:
    - Tarefa $task \in Tasks$, com informação sobre:
        - A lista completa e ordenada dos vértices consecutivos, $path$, por onde passará o estafeta;
        - O pedido P, que faz parte da tarefa;
        - O estafeta $employee$, encarregue da sua realização;
        - A distância total, $totalDistance$, a percorrer pelo estafeta;
        - o tempo estimado $estimatedTime$ para a entrega, com base na velocidade média do $employee$ e na distância a percorrer.

O que se segue, nomeadamente, o ato de percorrer o caminho, com a identificação dos pontos já visitados, da entrega dos pedidos, do cálculo da distância percorrida, é um tanto independente desta análise algorítmica inicial, mas poderá ser, eventualmente, alvo de estudo e reflexão, para interpretação de resultados.

2.5 Restrições

A primeira restrição prende-se com o tamanho do grafo. Em termos reais, aplicações deste género, por uma questão de praticabilidade, limitam as zonas de atuação a áreas urbanas, onde exista um número razoável de restaurantes registados e de estafetas em operação. Assim sendo, os grafos aqui analisados também terão a sua área limitada.

As outras restrições são como se seguem:

2.5.1 Dados de Entrada

- $\forall\, v_1, v_2 \in V,\ v_1 \neq v_2$, no sentido em que não haverá vértices repetidos;
- $\forall\, e_1, e_2 \in E,\ e_1 \neq e_2$, no sentido em que não haverá arestas repetidas;
- $\forall\, e \in E,\ e.weight > 0,\ e.state \in \{\ 0,\ 1\}$, já que uma aresta - via/rua - tem um comprimento definido e encontra-se transitável, ou não transitável;
- $\forall\, r_1, r_2 \in R,\ r_1 \neq r_2$, no sentido em que não haverá restaurantes repetidos;
- $\forall\, r \in R,\ r.v \neq null$, sendo que um restaurante tem que ter uma posição estabelecida;
- $\forall\, employee_1, employee_2 \in Employees,\ employee_1 \neq employee_2$, no sentido em que não haverá estafetas repetidos;
- $\forall\, employee \in Employee,\ employee.s \neq null$, sendo que um estafeta tem que ter uma posição estabelecida;
- $\forall\, employee \in Employee,\ employee.maxCargo > 0,\ employee.type \in \{\ 'car',\ 'bike',\ 'foot'\ \}\ ,\ employee.ready \in \{0,\ 1\}$;
- $\forall\, p_1, p_2 \in P,\ p_1 \neq p_2$, no sentido em que não haverá pedidos repetidos;
- $\forall\, p \in P,\ p.time \neq null,\ |p.checkPoints| > 1,\ p.cargo > 0$;

2.5.2 Dados de Saída

- $\forall\, task_1, task_2 \in Tasks,\ task_1 \neq task_2$;
- $\forall\, task \in Tasks,\ task.path \neq null, task.P \neq null,\ task.employee \neq null\ , |task.path| > 1$;
- $\forall\, task_1, task_2 \in Tasks,\ task_1.P \neq task_2.P$, no sentido em que nenhum pedido pode pertencer a mais do que uma $Task$;
- $\forall\, task_1, task_2 \in Tasks,\ task_1.employee \neq task_2.employee$, no sentido em que nenhum estafeta pode estar associado a mais do que uma $Task$ a ocorrer em simultâneo;
- $\forall\, task \in Tasks, \text{em}\ task.path$, os restaurantes terão de ser visitados antes da localização do cliente, em relação ao pedido associado a ambos;

2.6 Função Objetivo

A solução ótima para este problema reside, genericamente, na minimização da distância percorrida, em cada rota, por cada estafeta, numa dada tarefa. Assim, será necessário encontrar o mínimo de:

$$f(task) = \sum_{e(v_k, v_{k+1})} e.weight \ , \ v_k, v_{k+1} \in task.path \ \wedge \ e \in E$$

Adicionalmente, para múltiplas tarefas a realizar em simultâneo, num dado intervalo de tempo, poderá surgir a necessidade de minimizar, também, as funções:

$$g = \sum_{task} f(task) \ , \ task \in Tasks$$
$$h = |Tasks|$$

Para a função g, representando a distância total percorrida por todos os estafetas, relacionados com um dado conjunto de $Tasks$, o mínimo poderá, eventualmente, ser atingido pela simples minimização de f, em todas as instâncias, sendo, por isso, a função f de maior prioridade, entre todas. A função h terá mais interesse quando implementada a funcionalidade extra de um cliente poder englobar no seu pedido vários pratos de vários restaurantes, compactando o que seriam diferentes pedidos num pedido e numa tarefa apenas.

3. Perspetiva de Implementação

Em cada uma das fases de análise anteriormente deliberadas pode-se, seguramente, realçar como problema comum a determinação do *caminho mais curto entre dois vértices numa rede viária*. Em particular, enquadra-se perfeitamente nesta problemática a necessidade de, aquando da realização de um pedido, determinar o percurso que permite proceder à sua entrega, recorrendo ao trajeto de menor distância. Isto implicará:

- determinar o percurso transitável mais curto, que permita a cada um dos estafetas alcançar um ou mais restaurantes dos quais o pedido foi solicitado;
- calcular, no caso de pedidos realizados num só restaurante, o trajeto de menor distância entre o restaurante e o ponto de entrega especificado pelo cliente.

Assim, é pertinente analisar a variedade de algoritmos e alternativas de implementação disponíveis para o cálculo do caminho mais curto entre dois vértices. Só a realização desta análise prévia tornará possível perceber o impacto das vantagens e desvantagens de cada algoritmo e das suas variantes, no contexto da temática a ser trabalhada.
Salienta-se, portanto, que nenhum algoritmo a ser considerado na fase de implementação deve ser, ainda numa fase experimental, tomado como solução ideal, pois a sua aplicabilidade está dependente da dimensão e tipologia dos grafos de entrada e de uma análise cuidada da sua eficiência temporal e espacial, para cada caso.

Destaca-se ainda que, numa terceira fase, os resultados obtidos através da implementação dos algoritmos a ser analisados vão ser conciliados com as múltiplas restrições impostas pela temática. Pretende-se que, no processo de escolha de um estafeta para a realização de um pedido, a distância mínima que este precisa de percorrer para entregar o pedido em questão não seja o único fator determinante na sua escolha em detrimento da escolha de outro estafeta. Assim, o estafeta cuja distância seja a mínima obtida para a concretização de uma encomenda, não será obrigatoriamente selecionado para o realizar, sendo, também, fatores relevantes para a escolha do estafeta a capacidade que este pode transportar e o tipo de veículo por ele utilizado.

Algoritmos considerados

3.1.1 Dijkstra Unidirecional

A primeira alternativa de solução que pretendemos testar passa essencialmente pela implementação de uma das múltiplas variantes do algoritmo de Dijkstra.
Este algoritmo ganancioso poderá ser vantajoso, na medida em que procura, a cada iteração, obter o melhor resultado possível, minimizando a distância percorrida. Para além disso, é logicamente aplicável ao grafo que será utilizado como dado de entrada, já que este se trata de um grafo dirigido - cada aresta representa uma via unidirecional; e pesado - o peso de uma aresta representa uma distância.
A variante a implementar permite, recorrendo a uma fila de prioridades (heap com mínimo à cabeça) como estrutura auxiliar, obter a distância mínima desde o vértice de origem até todos os outros vértices do grafo, garantindo-se um tempo de execução de $O((|V| + |E|) \times log|V|)$. Este tempo resulta das $|V|$ operações executadas na fila de prioridade (inserções/extrações), realizadas, cada uma, em tempo logarítmico, e da utilização da operação "Decrease-Key", realizada, no máximo, uma vez por aresta, com um tempo também logarítmico, para a fila de $|V|$ elementos.
Note-se que, nas fases previamente definidas, há a necessidade de aplicar o algoritmo mais do que uma vez:

- considerando como vértice de origem a posição atual de cada um dos estafetas disponíveis para efetuar a entrega de um pedido;
- assumindo que o vértice de origem é o restaurante do qual foi solicitado o pedido.

No entanto, o que se pretende realmente é utilizar o algoritmo de Dijkstra como base para encontrar o caminho mais curto entre dois pontos, pelo que se termina o algoritmo quando se for processar o vértice que procuramos, uma otimização que evita continuar a processar vértices, quando já se descobriu o caminho pretendido.

Assim, como, nas duas primeiras fases de implementação, a escolha do estafeta a realizar o pedido depende exclusivamente da sua proximidade ao restaurante do qual este foi solicitado, é necessário determinar, em primeiro lugar, o caminho mais curto para cada um dos estafetas se deslocar até aos respetivos estabelecimentos, sendo o restaurante o vértice de destino. De seguida, aplica-se, novamente, o algoritmo, para encontrar o percurso mais curto do restaurante à morada escolhida para o ato de entrega, usando o vértice de destino correspondente à localização indicada pelo cliente.

Espera-se, ao aplicar este algoritmo, em alguns casos específicos, entre os quais, mapas de estradas com poucos vértices e distâncias curtas, obter tempos de execução razoáveis. No entanto, para trajetos longos e mapas de estradas complexos, a utilização do algoritmo de Dijkstra, na sua variante unidirecional, gerará, certamente, tempos de execução demasiado longos, o que nos levará, provavelmente, a experimentar otimizações e a optar por utilizar outros algoritmos na implementação final, procurando sempre obter o caminho mais curto para a entrega de um pedido com a maior eficácia temporal possível.

3.1.2 Dijkstra Bidirecional

Uma das otimizações que se procura explorar com mais detalhe na fase de implementação, para o algoritmo de Dijkstra, tendo em conta que o resultado esperado para a variante unidirecional não é o desejado, é a variante bidirecional. A execução alternada do algoritmo de Dijkstra, no sentido do vértice de origem para o de destino, e no sentido inverso, poderá trazer vantagens no que diz respeito ao tempo de execução, esperando-se que este seja reduzido para metade, em comparação com a variante do algoritmo que utiliza pesquisa unidirecional.
Esta melhoria deriva do facto de a área processada diminuir para metade, já que a pesquisa passa a ser executada nas duas direções, mantendo-se sempre a distância mais curta descoberta até ao momento e verificando-se, ao processar uma aresta já processada previamente, na direção oposta, se foi descoberta uma distância menor.
Deste modo, a variante bidirecional do algoritmo de Dijkstra vai ser aplicada com o intuito de melhorar o tempo de cálculo do percurso mais curto entre o estafeta e o restaurante e, posteriormente, entre o restaurante e o local de entrega, no caso de pedidos que envolvam um só restaurante.

3.1.3 Algoritmo A*

O algoritmo A* será, também, objeto de estudo, na fase de implementação, não só por ser mais um algoritmo que permite, igualmente, calcular o caminho mais curto entre dois vértices, mas, essencialmente, pela sua performance se diferenciar do algoritmo de Dijkstra, pelo facto de recorrer a heurísticas para manipular os pesos das arestas e, assim, afetar os nós que são expandidos.
Antes de mais, é essencial escolher, cuidadosamente, a heurística a utilizar, porque esta desempenhará um papel determinante no comportamento do algoritmo e nos resultados obtidos.

Posto isto, tendo em conta as propriedades dos vértices dos grafos a tratar, foram consideradas três funções de heurística diferentes, que farão parte do processo de implementação e análise deste algoritmo em concreto. Todas elas atuarão em função das coordenadas dos pontos no mapa e darão, sempre, um valor aproximado menor que o custo real do percurso, entre os vértices escolhidos.

A primeira estratégia tem o nome de **Manhattan Distance** e calcula a distância, como número de quadrados percorridos, em ambas as direções, desde o ponto inicial, até ao final. Tem particular interesse em mapas com a forma de grelha e usa, para o efeito, a seguinte fórmula generalizada:

$$h = \mid x_{start} - x_{destination} \mid + \mid y_{start} - y_{destination} \mid$$

Outra estratégia, conhecida como **Euclidean Distance**, um pouco mais precisa que a anterior, explora o percurso em linha reta, demorando, em contrapartida, mais tempo a executar, por necessitar de explorar uma área maior. A sua função é representada na forma:

$$h = \sqrt{(x_{start} - x_{destination})^2 + (y_{start} - y_{destination})^2}$$

Por fim, existe também a **Diagonal Distance**. Esta última perde interesse real, quando comparada com as outras funções, porque tem a limitação de só poder ser usada em movimentos realizados numa direção apenas:

$$h = \max(abs(x_{start} - x_{destination}), abs(y_{start} - y_{destination}))$$

Este algoritmo em particular é conhecido por não garantir uma solução ótima em muitos casos. No entanto, os seus resultados serão avaliados, a par com os restantes algoritmos, tendo em conta, também, as diferentes funções de heurística aqui referidas.

3.1.4 Algoritmo Floyd-Warshall

O último algoritmo analisado será o de Floyd-Warshall. Este algoritmo de programação dinâmica atua diretamente no problema da distância mais curta, entre vértices, e baseia-se numa matriz, onde se encontra a menor distância entre cada par de vértices do grafo. O algoritmo retorna, como resultado, uma tabela de distâncias mínimas e, com uma simples modificação, garante, facilmente, informação adicional, como a reconstrução da matriz de predecessores de um determinado caminho, entre dois pontos.

Os custos são mais notórios em termos de memória, mas a compensação temporal permite que este seja um algoritmo ideal para ser usado no pré-processamento dos grafos. Depois de efetuado o pressuposto, com um tempo de execução $O(|V|^3)$, qualquer cálculo de distância se resumirá a uma simples *table lookup* na matriz de distâncias mínimas, o que o torna vantajoso, a longo prazo, em oposição aos algoritmos anteriores, que necessitariam de ser executados, de novo, a cada situação. A única inconveniência deste algoritmo poderá ser, nos casos específicos de mudança da constituição atómica dos grafos, como quando se registam arestas intransitáveis, com obras, ou obstáculos na via pública, ter que se pré-processar e reestruturar, de novo, os respetivos mapas, de maneira a identificar outros trajetos que evitem essas arestas.
Apesar disso, este algoritmo será, certamente, o escolhido para ser aplicado na fase final de implementação da aplicação, como descrito de seguida.

3.1.5 Abordagens consideradas na Fase III

Na terceira fase, o caso em que se consideram múltiplos restaurantes no mesmo pedido, poderá ser visto como uma generalização do típico "Travelling Salesman Problem", pois o objetivo é encontrar o menor caminho possível para que cada estafeta visite todos os restaurantes associados a um pedido apenas uma vez, acrescentando-se a restrição de existir um ponto de origem e de destino predeterminados, respetivamente a posição inicial do estafeta e a morada de entrega do pedido.

A alternativa mais direta para resolver o problema seria aplicar um algoritmo "brute-force" que testasse todas os percursos possíveis, preservando, a cada iteração, aquele caminho que apresentasse uma distância mais curta. No entanto, esta solução é impraticável pelo seu elevado tempo de execução, $O(n!)$, até para grafos pouco densos.

Também são conhecidas alternativas de solução para o problema recorrendo a algoritmos de programação dinâmica que, apesar de revelarem melhorias, em comparação com a solução "brute-force", ainda apresentam um tempo exponencial $O(2^n \times n^2)$, tendo, para além disso, complexidade espacial elevada, $O(2^n \times n)$, o que nos leva a descartá-los, para o efeito desejado.

A alternativa, que, à partida, nos parece a mais viável, tendo em conta os conhecimentos adquiridos até agora, baseia-se numa abordagem gananciosa, já que procura, em cada iteração, escolher a solução ótima, ou seja, neste caso, escolher o restaurante mais perto do anterior. Assim, tendo como vértice de origem a posição de cada estafeta, procurar-se-ia construir o percurso até à morada de entrega do pedido, escolhendo, a cada iteração, o restaurante de menor distância. No entanto, para esta abordagem gananciosa será necessário calcular previamente as distâncias entre os vários pontos do grafo. Para isso, pensamos recorrer ao algoritmo de Floyd-Warshall, referido anteriormente, obtendo, assim, o caminho mais curto entre todos os pares de vértices, que será usado, posteriormente, para determinar o caminho de cada estafeta, alcançando todos os restaurantes e, por fim, a morada do cliente.

4. Casos de utilização a serem suportados

Se a perspetiva se cumprir, a aplicação final conseguirá, na sua camada superior, receber os pedidos dos utilizadores e, na camada inferior, proceder à sua gestão e delineação da respetiva rota, a percorrer pelos estafetas responsáveis por atender os pedidos.

Como idealização dos casos de utilização, prevê-se, no final da implementação, a existência de vários mapas, de várias cidades, cada uma com um conjunto de estafetas a operar e um conjunto de restaurantes já registados. O utilizador terá, simplesmente, de identificar a cidade e proceder com a encomenda, escolhendo a sua localização, o restaurante e elaborando, por fim, o seu pedido. O sistema atribuir-lhe-á um estafeta livre e com capacidade para o transportar, calculando as distâncias mínimas e estabelecendo o percurso. Esta informação, nomeadamente, o estafeta responsável e a rota a percorrer, irá ser retornada, como resultado. Para além disso, permitirá simular a existência de vários pedidos e a sua distribuição pelos estafetas.

Numa última nota, pretender-se-á utilizar, também, uma API de visualização de grafos, como auxílio à compreensão e perceção das várias circunstâncias e consequências dos algoritmos abordados. O percurso final do estafeta poderá ser apresentado num mapa real, para que o utilizador possa ter a noção real da utilização e capacidade do nosso sistema na gestão das rotas e cálculo efetivo das distâncias mínimas.

5. Implementação

Iniciada a parte de implementação prática de tudo o que foi discutido anteriormente, os dados de partida, mapas e código genérico foram recolhidos e as novas classes foram criadas, de acordo com tudo o que foi estabelecido como dados de entrada. É de notar que as soluções, disponibilizadas em código nas aulas práticas, para os grafos, vértices e arestas e respetivos algoritmos foram adaptadas e reorganizadas, para melhor satisfazerem as nossas necessidades.

5.1 Estruturas de dados

Para representar toda a informação necessária e construir a aplicação final, em conformidade com o que foi especificado, dividimos a implementação nas classes relativas. São elas:

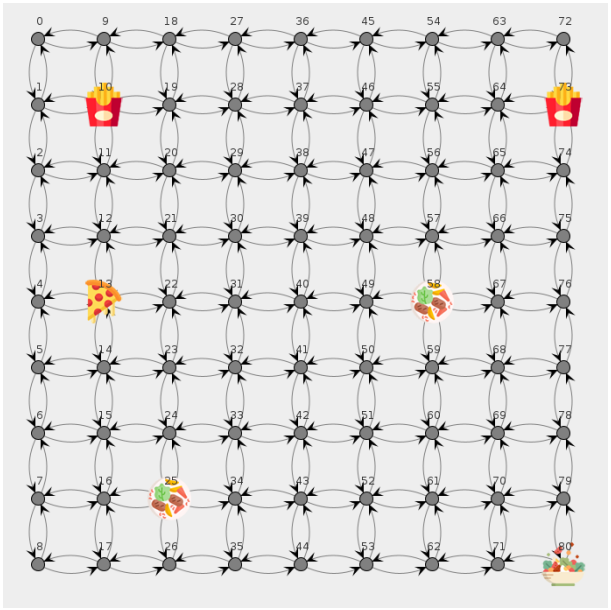
- Classe **Graph**, presente no ficheiro *Graph.h* para representar o Grafo.
- Classes **Edge** e **Vertex**, para representar os constituintes do grafo, respetivamente as arestas e os vértices.
- Classe **Coordinates**, usada como implementação da classe *Template T* das estruturas anteriores, representando as coordenadas dos vértices que servem de identificação e são usadas para o cálculo das distâncias reais e visualização dos mapas.
- Classe **Employee**, que representa um Estafeta, com todos os atributos anteriormente descritos.
- Classe **Request**, concretizando e contendo a informação de um pedido realizado por um cliente.
- Classe **Task**, com subclasses **SingleTask** e **SpecialTask** - a primeira representando uma tarefa simples de atribuição e entrega de um pedido por um dado estafeta; a segunda, para o caso especial de implementação da situação de entrega de vários pedidos de uma só vez por um único estafeta. Ambos os ficheiros são acompanhados de utilitários que permitem, entre outros aspetos, determinar, por exemplo, as distâncias para cada estafeta até ao restaurante do pedido, distribuir os pedidos pelos estafetas, determinar se um dado estafeta é elegível para a realização de um pedido, etc.

> Informação adicional

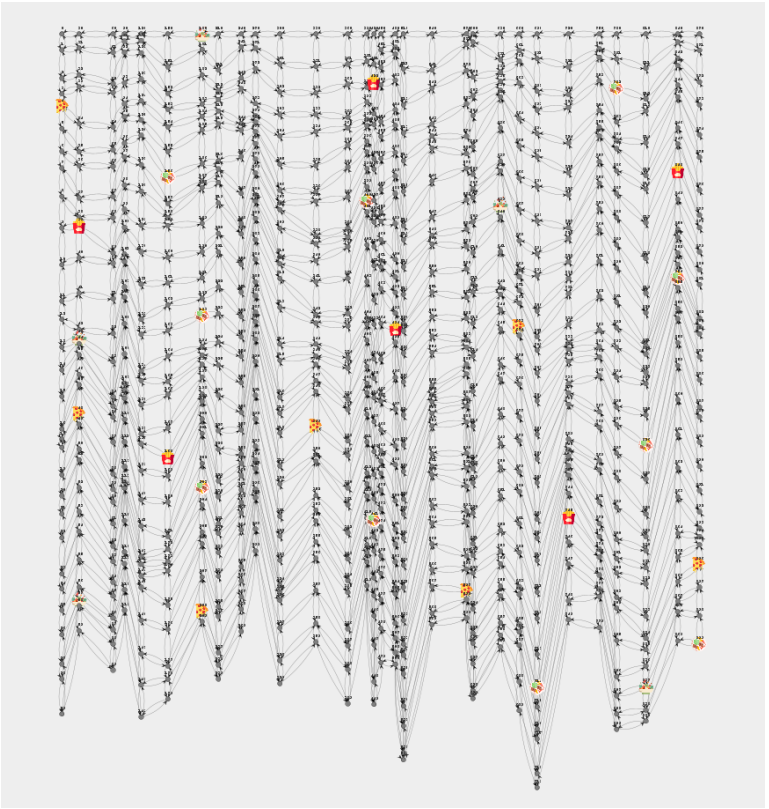
- Ficheiro *utils.h* - contém classes utilitárias, como **Hour** e **Date**, e várias funções de carregamento e construção de grafos e geração de informação aleatória, além de funções de ligação e utilização da API *GraphViewer*.
- Ficheiro *app.h* - contém as funções que permitem o funcionamento da aplicação final, gerindo toda a interação com o utilizador.
- Ficheiro *simulations.h* - onde residem as múltiplas simulações de implementação, incrementalmente e em conformidade com as várias fases descritas nas secções anteriores.
- Ficheiros *tests.cpp* e ficheiros contidos no diretório *phase_tests* - contendo, sobretudo, testes unitários e testes temporais empíricos a grafos e aos algoritmos e fases consideradas.

5.2 Grafos e Mapas usados

Em matéria de testes, para uma possível readaptação futura em grafos com representação real de áreas urbanas, foram usados grafos em forma de grelha, de diversas dimensões e alguns com distâncias e arestas aleatoriamente distribuídas. A utilização de diferentes mapas com dimensões, em número de vértices, $N \times N$, fortemente conexos, permitiu uma mais fácil visualização de todo o processo e uma fundamentação mais cuidada na escolha de determinados algoritmos. Exemplos de grafos utilizados na implementação são os seguintes:



Grafo 8×8



Grafo 30×30

Todos os grafos presentes na aplicação têm as localizações dos seus restaurantes definidas inicialmente e os seus vértices, na API de visualização, distinguidos com diferentes ícones, para os diferentes tipos de restaurante. Para o grafo 8×8, por exemplo, a lista de restaurantes existentes é:

```
Type: Fast Food         Vertex id: 10
Type: Pizzeria          Vertex id: 13
Type: Restaurant        Vertex id: 25
Type: Restaurant        Vertex id: 58
Type: Fast Food         Vertex id: 73
Type: Vegetarian        Vertex id: 80
```

A conectividade dos grafos usados é analisada seguidamente, durante a Fase III de implementação, onde foram criados grafos reduzidos a partir destes em grelha, representando mapas para outros meios de transporte, mais restritivos.

5.3 Algoritmos efetivamente implementados

Dadas as limitações temporais e logísticas deste projeto, chegado o momento de eleger algoritmos de cálculo de distâncias e caminhos entre vértices, por questões mais relacionadas com uma análise interpretativa de resultados distintos, decidimos focar-nos em dois algoritmos apenas.

Cada um representa uma generalização de um subproblema do caminho mais curto, o **Dijkstra unidirecional** como algoritmo representativo do problema *Single Source Multiple Destinations*, que, com a utilização de uma *fila de prioridade*, apresenta uma complexidade temporal de $O((V +$

$E) \log V)$ e espacial de $O(V)$

```
1  function Dijkstra(Graph, source):
2      dist[source] ← 0                           // Initialization
3
4      create vertex priority queue Q
5
6      for each vertex v in Graph:
7          if v ≠ source
8              dist[v] ← INFINITY                 // Unknown distance from source to v
9              prev[v] ← UNDEFINED                // Predecessor of v
10
11         Q.add_with_priority(v, dist[v])
12
13
14     while Q is not empty:                      // The main loop
15         u ← Q.extract_min()                    // Remove and return best vertex
16         for each neighbor v of u:              // only v that are still in Q
17             alt ← dist[u] + length(u, v)
18             if alt < dist[v]
19                 dist[v] ← alt
20                 prev[v] ← u
21                 Q.decrease_priority(v, alt)
22
23     return dist, prev
```

e o **Floyd-Warshall** como algoritmo abrangendo a questão *Multiple Sources Multiple Destinations*, com complexidade temporal na ordem de $O(V^3)$ e espacial na ordem de $O(V^2)$.

```
1 let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
2 function FloydWarshall():
3    for each edge (u, v) do
4      dist[u][v] ← w(u, v)  // The weight of the edge (u, v)
5    for each vertex v do
6         dist[v][v] ← 0
7    for k from 1 to |V|
8         for i from 1 to |V|
9             for j from 1 to |V|
10                if dist[i][j] > dist[i][k] + dist[k][j]
11                    dist[i][j] ← dist[i][k] + dist[k][j]
12                end if
```

Como alicerce para a aplicação final, dada a análise exaustiva a todas as fases de implementação, o algoritmo **Floyd-Warshall** foi o escolhido para vigorar e funcionar em pleno.

Enuncia-se, também, a implementação do algoritmo de **Pesquisa em Profundidade (DFS)**, na sua versão recursiva, para a análise da conectividade dos grafos, com uma complexidade temporal de $O(V + E)$ e espacial de $O(V)$:

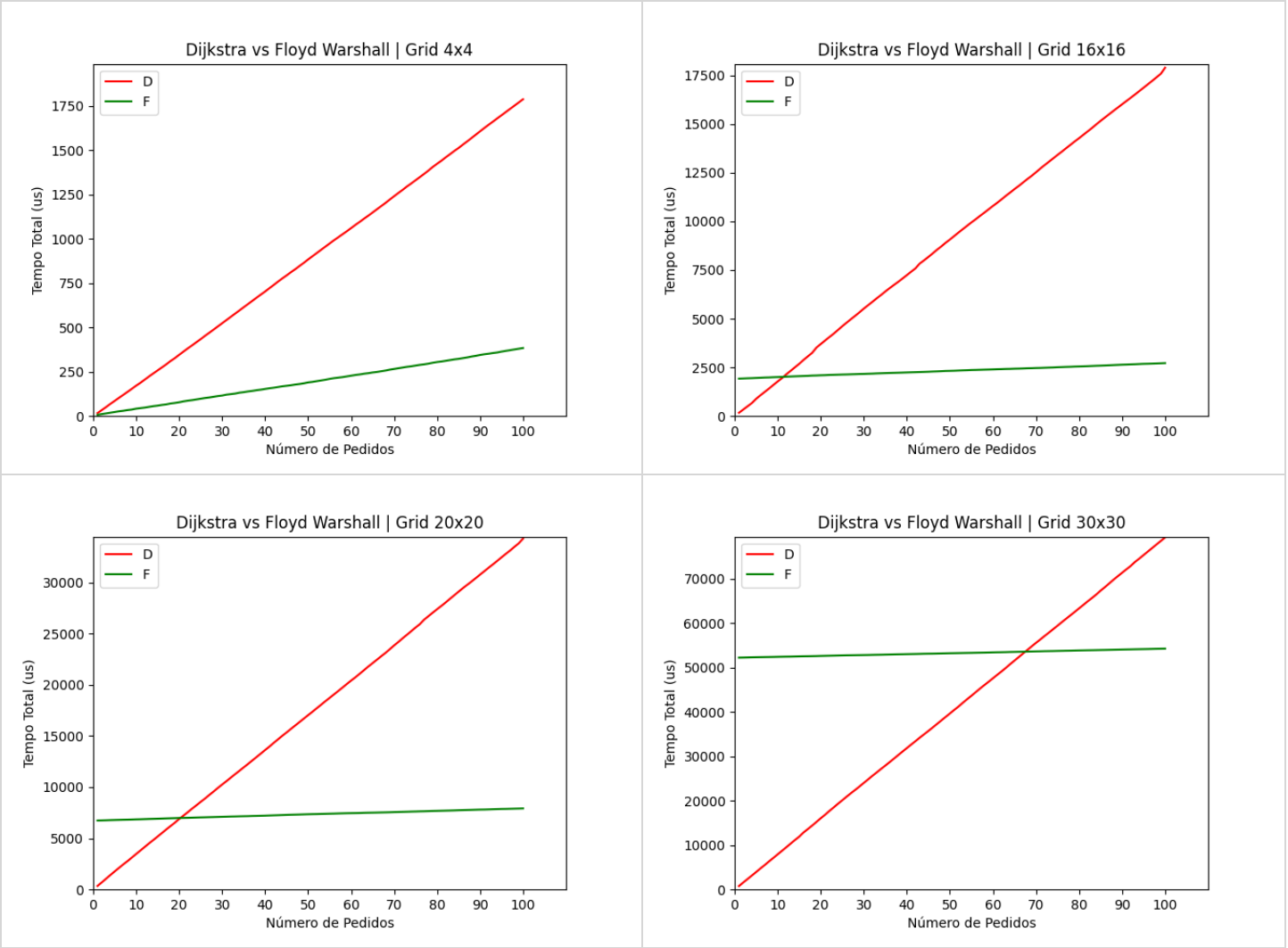
```
1 function DFS-recursive(G, s):
2       mark s as visited
3       for all neighbours w of s in Graph G:
4           if w is not visited:
5               DFS-recursive(G, w)
```

5.4 Análise empírica

Para uma análise generalizada e abrangente dos algoritmos tidos em consideração, estes foram testados em todos os grafos grid elegíveis e à nossa disposição. Os testes foram efetuados em conformidade com as fases de elaboração determinadas anteriormente e a sua eficiência temporal foi avaliada.

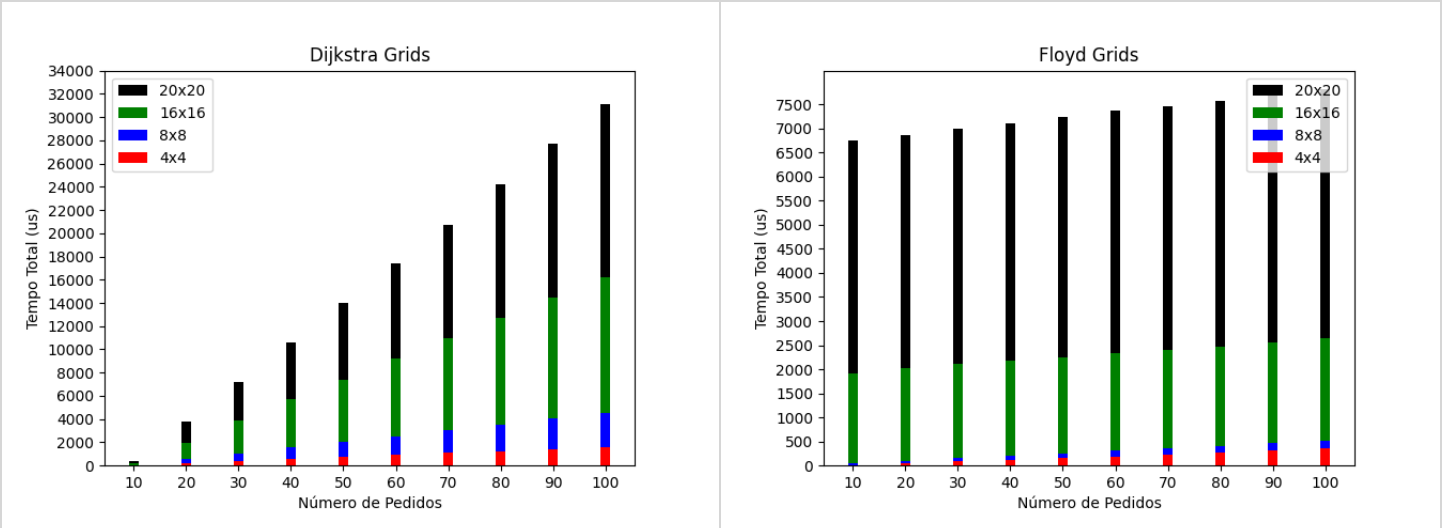
5.4.1 Fase I

Na Fase I de implementação, considerado o caso atómico de um estafeta que entrega apenas um pedido, entre um restaurante e a morada de um cliente, realizando todas as tarefas em sequência, foi calculada a média do tempo demorado na realização de 1 a 100 pedidos, para os algoritmos mais genéricos, neste caso, o **Dijkstra unidirecional** e o **Floyd-Warshall**. Os tempos foram medidos em micro-segundos e os grafos considerados tinham dimensões variando entre 4×4 e 30×30, como relatam os seguintes dados:



Um olhar mais atento aos dados do grafo de dimensões 20×20, por exemplo, revela um pormenor interessante relativamente à concorrência entre estes dois algoritmos e à sua rentabilidade, para uso na aplicação. A explicação para os valores reside no facto de o algoritmo de **Floyd-Warshall** demorar ligeiramente mais tempo, inicialmente, devido ao pré-processamento necessário, mantendo, de seguida, um tempo médio de execução de cada pedido muito inferior, sendo logo ultrapassado, em matéria de tempo acumulado total, pelo algoritmo de **Dijkstra**, com um comportamento linear mais acentuado.

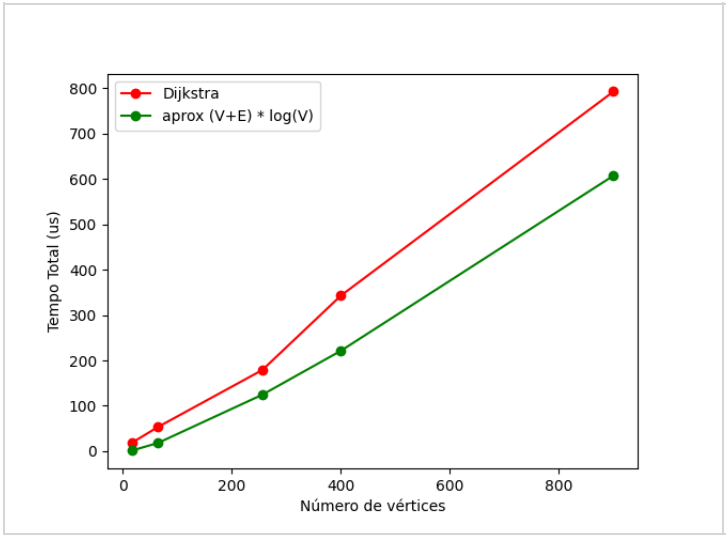
A partir de um certo número de pedidos, o algoritmo de **Floyd-Warshall** torna-se o mais viável, o que satisfaz as necessidades, a longo prazo, de uma aplicação deste tipo, tanto em grafos grandes, como em grafos mais pequenos, já que o custo de operação do algoritmo de **Dijkstra** aumenta consideravelmente.

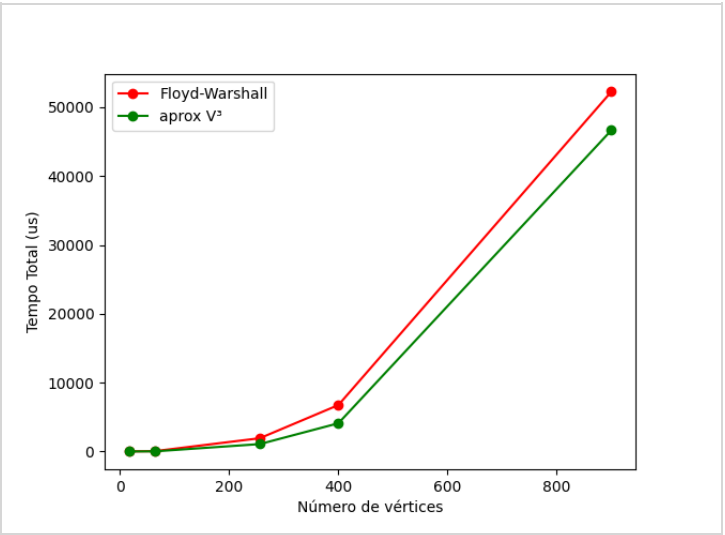


Os valores mais precisos, em micro-segundos, para alguns destes gráficos, encontram-se na tabela seguinte:

Número de Pedidos	Dijkstra 8x8	Dijkstra 16x16	Dijkstra 20x20	Floyd 8x8	Floyd 16x16	Floyd 20x20
5	53	170	353	50	1923	6743
10	305	1069	2113	77	1968	6804
15	557	1950	3817	102	2013	6862
20	806	2882	5524	130	2059	6930
25	1063	3877	7223	157	2102	6997
30	1314	4793	8887	182	2137	7055
35	1561	5690	10586	208	2172	7116
40	1804	6568	12260	231	2213	7169
45	2044	7411	13965	256	2247	7233
50	2286	8351	15687	281	2287	7307
55	2529	9242	17362	305	2328	7367
60	2782	10104	19053	331	2370	7418
65	3037	10951	20728	358	2405	7469
70	3287	11815	22447	383	2438	7518
75	3537	12705	24180	407	2473	7574
80	3794	13573	25927	438	2516	7637
85	4047	14439	27702	467	2554	7693
90	4304	15325	29432	496	2598	7757
95	4556	16171	31092	524	2645	7812
100	4809	17032	32784	553	2687	7875

O tempo médio gasto a cada pedido, para o algoritmo de **Dijkstra**, revelou, assim, o esperado e aproximado tempo computacional de proporções $O((E+V)\log V)$. O pré-processamento realizado pelo algoritmo de **Floyd-Warshall**, contabilizado apenas como parte do primeiro pedido, revelou a sua complexidade na ordem $O(V^3)$ e um tempo de execução mínimo para a reconstrução dos caminhos, na ordem $O(E)$, com E somente o número de arestas entre os dois vértices em consideração.





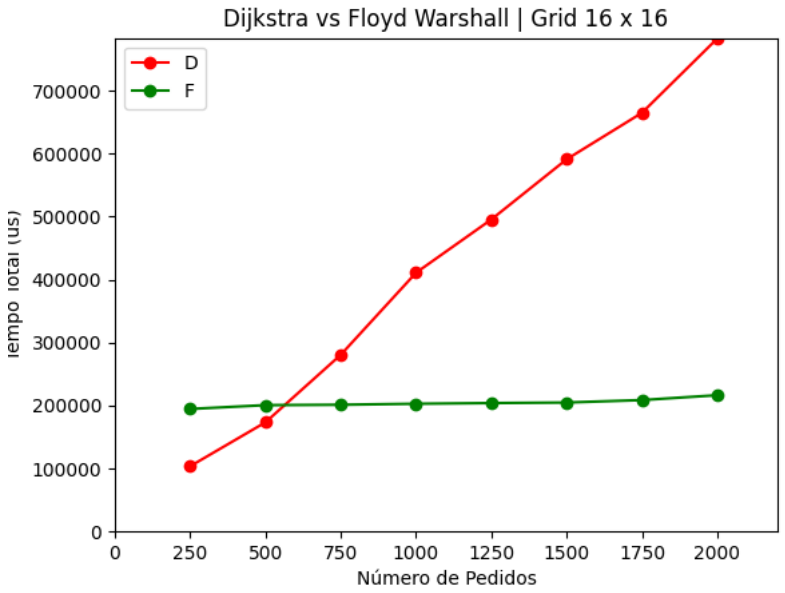
Com isto, confirmamos também a complexidade temporal destes algoritmos e reafirmamos e fundamentamos as escolhas para as fases seguintes.

5.4.2 Fase II

Nesta segunda fase, voltámos a analisar a execução dos algoritmos, desta vez, tendo em conta a existência de vários estafetas e a distribuição de vários pedidos pelos funcionários disponíveis e elegíveis para os entregar. É de notar que a escolha do estafeta encarregue de determinado pedido era diretamente influenciada pela sua proximidade ao restaurante, o que implicava ainda mais cálculos intermédios e mais execuções dos algoritmos usados. Foram gerados, portanto, pedidos e a cada tarefa concluída, os estafetas atualizavam a sua posição.

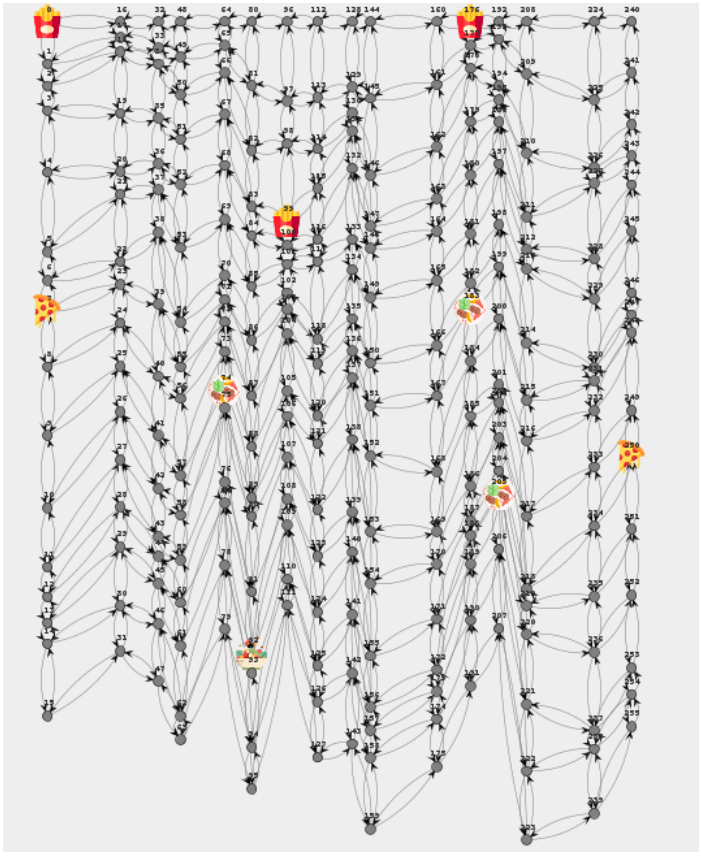
```
// Add Employees
vector<Employee> employees;
Employee employee1(0,Coordinates(10),1000,CAR,true);
employees.push_back(employee1);
Employee employee2(1,Coordinates(20),1000,CAR,true);
employees.push_back(employee2);
Employee employee3(2,Coordinates(30),1000,CAR,true);
employees.push_back(employee3);

// Add Requests
vector<Request> requests = getRandomRequests(graph,2000);
queue<Request> requestsQueue;
for(Request request : requests){
    requestsQueue.push(request);
}

// ... Same logic for the Floyd Algorithm

while(!requestsQueue.empty()){
    vector<SingleTask*> tasks = distributeRequestsByCloseness_Dijkstra(graph, requestsQueue, employees);

    // Couldn't find any Employees to fulfill the remaining requests
    if(tasks.empty())
        return 0;

    // Get requests distribution and paths
    for(int i = 0; i < tasks.size(); i++){
        tasks[i]->setDijkstraPath(graph);
    }
}
```

Para efeitos de teste desta fase de implementação, considerou-se a existência de 3 estafetas, distribuídos aleatoriamente por um grafo, representativo de uma área urbana, de dimensões 16×16, e geraram-se até 2000 pedidos também aleatórios.

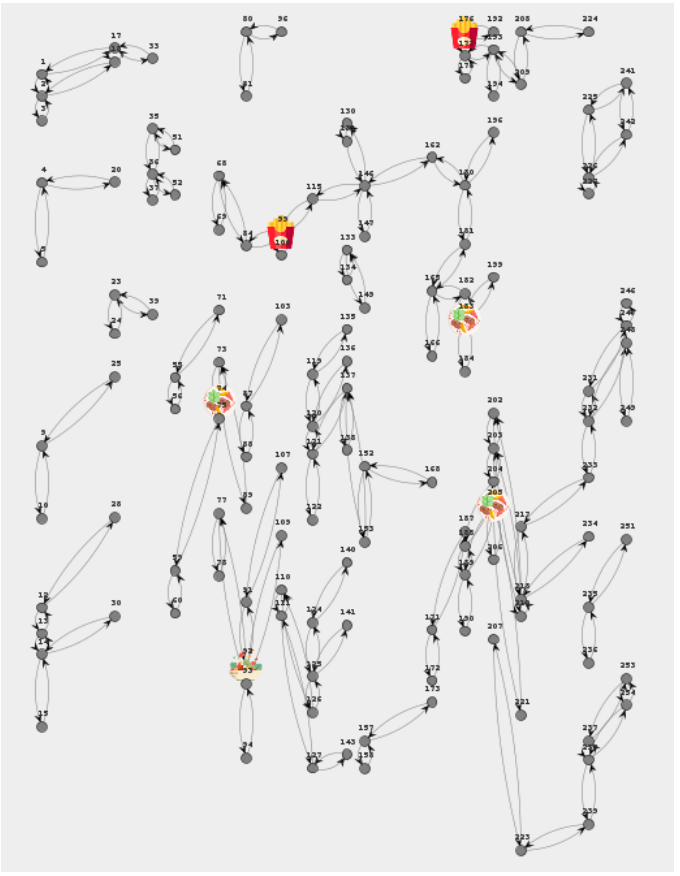


5.4.3 Fase III

Aqui, variados meios de transporte começaram já a ser considerados, o que correspondeu à necessidade de criar mapas adaptados, partindo dos já existentes. A solução, em conformidade com as simulações efetuadas, foi reconverter os mapas em grelha para grafos com um número inferior de vértices e arestas disponíveis, simulando a transitabilidade de certas vias e a sua acessibilidade, conforme o meio de transporte. Um exemplo de um grafo adaptado para servir estafetas que se desloquem a pé, ou de bicicleta, é o seguinte:



Grafo original para veículos motorizados



Grafo reduzido para estafetas a pé ou de bicicleta

Neste ponto, tornou-se pertinente a análise paralela à conetividade dos grafos, sobretudo dos grafos maiores e dos respetivos grafos reduzidos, que revelaram pormenores também visíveis nas imagens anteriores. Obteve-se, assim, a média e o máximo do número de vértices descobertos, partindo de um dado vértice, através do algoritmo de **Pesquisa em Profundidade (DFS)**:

Grafo	Número de Vértices	Média de Vértices Encontrados	Máximo de Vértices Encontrados
16x16	256	256	256
16x16 reduzido	142	9	20
20x20	400	400	400
20x20 reduzido	206	10	17
30x30	900	900	900
30x30 reduzido	498	10	26

Desta tabela, retira-se, apenas, a informação da super conectividade dos grafos originais, que contrasta com a mais fraca conectividade dos grafos adaptados/reduzidos, usados para entregas a pé, ou de bicicleta.

5.4.4 Fase IV

Chegada esta fase, decidiu-se a sua não inclusão prática na aplicação, pelo seu desajuste em relação à situação real, sendo apenas representativa de algo hipotético. A sua implementação resumir-se-ia a duas estratégias, após a deteção de um obstáculo na via - aresta inacessível:

- pré-processar, de novo os grafos envolvidos, com o algoritmo de Floyd-Warshall;
- ou optar, nestas situações específicas, por usar o algoritmo de Dijkstra.

Ambas teriam as vantagens e desvantagens já enunciadas e a sua demonstração não traria mais valor ao que já foi simulado.

6. Algoritmos e Estratégias implementadas

A primeira e segunda fase planeadas na primeira parte do trabalho serviram de base para desenvolver a fase que realmente consideramos crucial, a fase III, na qual já se tem em conta as várias condicionantes do problema, entre as quais as distâncias dos estafetas aos restaurantes dos pedidos, as capacidades máximas de transporte, os meios de transporte utilizados e a disponibilidade dos estafetas.
Como tal, todos estes fatores que foram ignorados numa fase inicial do desenvolvimento do trabalho, na qual só se pretendia abordar o problema na sua essência, são, na aplicação final, trabalhados tanto para o caso de um estafeta como para o caso em que existem vários.

Para melhor analisar a implementação, optamos, então, por descrever sucintamente as estratégias adotadas para cada caso de utilização, acompanhadas de pseudo-código.

6.1 Um ou vários pedidos por ordem temporal

Para o caso em que é feito um pedido ou vários, cada um de um único restaurante, e existem múltiplos estafetas para o realizar, seguimos a estratégia descrita na fase III para a escolha do estafeta.

A implementação deste caso tem como base a função:
`vector<SingleTask*> distributeRequests(Graph<Coordinates> & graph, Graph<Coordinates> & reducedGraph, min_priority_queue & requests,` - definida em `SingleTask.cpp` ; que devolve um conjunto de tarefas, consistindo uma tarefa na atribuição de um pedido a um estafeta.

Começa-se por verificar que tipos de meio de transporte podem ser utilizados para completar cada pedido, tendo em conta que são utilizados grafos diferentes para representar os caminhos transitáveis a pé/bicicleta e aqueles que podem ser percorridos de carro/motociclo. Para isso, são atualizadas as variáveis `deliverableByCar` e `deliverableByFoot` do pedido, consoante exista ou não um caminho entre o vértice do restaurante e a morada de entrega no grafo correpondente a cada meio de transporte. Utiliza-se a função
`setRequestsDeliverability(const Graph<Coordinates> & graph, const Graph<Coordinates> & reducedGraph, min_priority_queue & requests)` para realizar esta verificação.

Seguidamente, realizando os pedidos por ordem temporal, para a escolha de um estafeta para cada pedido, começa-se por definir as distâncias dos estafetas ao restaurante, armazenadas na matriz de distâncias calculada previamente com o algoritmo de Floyd Warshall - `setDistancesToCheckpoint(Graph<Coordinates> & graph, Graph<Coordinates> & reducedGraph, vector<Employee*> & employees, Request & req` . Aí, verifica-se também se existe um caminho entre estes dois pontos no grafo de cada meio de transporte e impõe-se uma distância limite para a realização de pedidos de bicicleta ou a pé, evitando trajetos excessivamente longos para estafetas que utilizem este meio de transporte.

O passo seguinte, é excluir todos os estafetas que não têm possibilidade de entregar o pedido, não só devido aos fatores anteriores mas também por não terem capacidade para o transportar ou por não estarem disponíveis - `getEligibleEmployees(vector<Employee*> & employees, const Request & request)` .

Finalmente, de todos os estafetas disponíveis, a escolha passa pela conjugação de três fatores: distância mais curta até ao restaurante, carga e velocidade média do meio de transporte, dando prioridade à distância, com um peso de 50% na decisão. Para isso utiliza-se o operador menor da classe `Employee` para escolher o estafeta a realizar o pedido.

```
bool Employee::operator<(const Employee &rhs) const {
    return dist * 0.5 + maxCargo * 0.3 + avgVelocity * 0.2 <
        rhs.getDist() * 0.5 + rhs.getMaxCargo() * 0.3 + rhs.getAvgVelocity() * 0.2;
}
```

Definido um estafeta para um dado pedido, este é marcado como indisponível, sendo necessário distribuir os restantes pedidos pelos estafetas disponíveis.
Como existem pedidos que só podem ser realizados por alguns estafetas específicos, devido à sua carga ou fraca acessibilidade dos seus vértices, alguns pedidos não serão atribuídos à primeira tentativa, utilizando-se uma fila de prioridade auxiliar `pendingRequests` para os guardar.

Assim que uma ronda de distribuição de pedidos termina, isto é, assim que todos os estafetas tenham um pedido atribuído, volta-se a disponibilizar todos os estafetas para a distribuição dos restantes pedidos e constrói-se o caminho mais curto para cada estafeta realizar o pedido que lhe foi atribuído - `setFloydWarshallPath(Graph<Coordinates> & graph)` - em `SingleTask.cpp` ; obtendo o caminho mais curto entre a posição do estafeta e o restaurante e entre o restaurante e a morada de entrega.

Assim que não existirem mais pedidos para distribuir, havendo a possibilidade de existirem pedidos que nenhum estafeta tenha condições para realizar - por os seus vértices serem inacessíveis ou por nenhum estafeta ter capacidade para o transportar; são devolvidas todas as tarefas que fazem corresponder cada pedido com o estafeta que ficou responsável por ele.

```
distributeRequests(Graph G1, Graph G2, PriorityQueue<Request> R, vector<Employee> E){
    vector<SingleTask> task // Para devolver a distribuição dos pedidos pelos estafetas
    vector<SingleTask> roundTasks // A cada ronda de distribuição guarda as tarefas

    PriorityQueue<Request> pendingRequests ← ∅ // Guardar pedidos pendentes
    R ← setRequestsDeliverability(G1,G2,R)

    while R != ∅ :
        Request request ← R.top() // Tentar atribuir o 1º pedido da fila
        R.pop()

        setDistancesToCheckpoint(G1, G2, E, request)
        eligibleEmployees ← getEligibleEmployees(E, request)

        if E == ∅ :
            pendingRequests.push(request) // Nenhum estafeta eligível
        else :
            sort(E) // de acordo com o operador <
            ready(E[0]) ← false // Ocupar o estafeta com o pedido
            SingleTask task(E[0], request) // Associar o estafeta ao pedido

        if Q == ∅ :
            //Verificar caso em que o pedido não pode ser entregue por nenhum estafeta
            // ...
            // Construção do caminho dos estafetas
            for each task ∈ roundTasks :
                if VehicleType(task) == CAR || VehicleType(task) == MOTORCYCLE :
                    task.setFloydWarshallPath(G1)
                if VehicleType(task) == BIKE || VehicleType(task) == FOOT :
                    task.setFloydWarshallPath(G2)

            tasks.insert(roundTasks)
            roundTasks.clear()
    return tasks
}
```

6.2 Pedido de múltiplos restaurantes

Um outro caso de utilização consiste na atribuição de um pedido que inclui refeições de mais do que um restaurante, cuja implementação se centra essencialmente na função `SingleTask * multipleRestaurantsRequest(Graph<Coordinates> & graph, Graph<Coordinates> & reducedGraph, vector<Employee*> & employees` - definida em `SingleTask.cpp` .
Para isso, determinam-se, mais uma vez, quais os estafeta eligíveis para realizar o pedido. Neste caso, no entanto, só podemos descartar, numa fase inicial, aqueles estafetas que não possuam capacidade para transportar a carga do pedido.
Seguidamente, determina-se para cada estafeta o caminho a seguir para recolher o pedido dos restaurantes, escolhendo sempre o restaurante mais perto - `int getNearestRestaurant(Graph<Coordinates> & graph, const Coordinates & origin, vector<Coordinates> & restaurants)` ; (consultando a matriz de distâncias pré calculada com o algoritmo de Floyd Warshall). Mantém-se sempre guardada a melhor distância, o estafeta ao qual essa distância corresponde e a ordem de visita dos restaurantes, tendo sempre em atenção que pode não existir caminho possível, no caso de alguns meios de transporte (dependendo do grafo de entrada). No fim, é escolhido o estafeta ao qual corresponde a melhor distância.

Mais uma vez, para definir o caminho completo a percorrer pelo estafeta, utiliza-se a função `setFloydWarshallPath(Graph<Coordinates> & graph)` - em `SingleTask.cpp` ; que calcula o caminho mais curto entre restaurantes consecutivos cuja ordem de visita já fora determinada no algoritmo descrito.

```
multipleRestaurantsRequest(Graph G1, Graph G2, vector<Employee> E, Request request){
    vector<Coordinates> restaurants // Restaurantes por ordem de visita
    int nearestEmployeePos // Estafeta ao qual corresponde a melhor distância
    int nearestRestaurantPos // Restaurante mais próximo dos que faltam visitar
    double nearestEmployeeDist ← INF // Inicializar a melhor distância com valor elevado

    employees ← getEligibleEmployeesMultipleRestaurants(E, request);

    for i = 0 to E.size(): // Repetir procedimento para cada estafeta disponível
        double totalDist ← 0
        Coordinates orig ← coords(E[i]) // Posição inicial do estafeta
        vector<Coordinates> requestRestaurants ← checkpoints(request) // Restaurantes por visitar
        vector<Coordinates> restaurantsPath // Ordem de visita dos restaurantes

        for j = 0 to checkpoints(request).size() :
            double dist ← 0; // Calcular distância a percorrer pelo estafeta
            if vehicle(E[i]) == CAR ||vehicle(E[i]) == MOTORCYCLE :
                // Restaurante por visitar mais perto do anterior
                nearestRestaurantPos ← getNearestRestaurant(G1, orig, requestRestaurants)

                dist ← G1.getDist(orig, requestRestaurants[nearestRestaurantPos]);
                //...
                if dist == INF : // Não foi encontrado caminho
                    totalDist ← INF
                    break

            else if vehicle(E[i]) == BIKE ||vehicle(E[i]) == FOOT : //...

            totalDist ← totalDist + dist // Acumular distância
            // Guardar restaurantes por ordem de visita
            restaurantsPath.push_back(requestRestaurants[nearestRestaurantPos])
            // Atualizar posição anterior
            orig ← requestRestaurants[nearestRestaurantPos]
            // Eliminar dos restaurantes por visitar
            requestRestaurants.erase(requestRestaurants.begin() + nearestRestaurantPos)

        // Não escolher o estafeta
        if totalDist == INF ||
        (totalDist > 6000 && (vehicle(E[i]) == BIKE || vehicle(E[i]) == FOOT))):
            continue

        // Adicionar a distância à morada de destino
        if vehicle(E[i]) == CAR || vehicle(E[i]) == MOTORCYCLE :
            double dist ← G.getDist(restaurantsPath[restaurantsPath.size()-1]),deliveryAddr(request))
            //...
            if(dist == INF) break
            totalDist ← totalDist + dist

        else if vehicle(E[i]) == BIKE || vehicle(E[i]) == FOOT) :
            double dist ← G.getDist(restaurantsPath[restaurantsPath.size()-1]),deliveryAddr(request))
            //...
            if(dist == INF || totalDist + dist > 6000) break;
             totalDist ← totalDist + dist

        // Melhor distância -> mudar estafeta escolhido
        if totalDist < nearestEmployeeDist :
            nearestEmployeePos ← i
            nearestEmployeeDist ← totalDist
            restaurants ← restaurantsPath
    }

    // Se não foi encontrado nenhum estafeta para realizar o pedido retornar tarefa vazia
    if nearestEmployeeDist == INF : return SingleTask(nullptr, request)

    request.setCheckpoints(restaurants) // Guardar os restaurantes por ordem de visita
    SingleTask task ← SingleTask(E[nearestEmployeePos], request) // Associar pedido e estafeta

    if vehicle(task) == CAR || vehicle(task) == MOTORCYCLE:
        task.setFloydWarshallPath(G1) // Construção do caminho do estafeta
    if vehicle(task) == BIKE || vehicle(task) == FOOT :
        task.setFloydWarshallPath(G2)

    return task
}
```

6.3 Um estafeta - Vários pedidos no mesmo deslocamento (variante do TSP)

Outra alternativa que pensamos ser essencial incluir na nossa implementação, tendo em conta a essência do problema, foi o caso em que um só estafeta deve, num só trajeto, recolher todos os pedidos dos restaurantes e entregá-los nas respetivas moradas de entrega, tendo em atenção que o restaurante de um pedido deve sempre ser visitado antes da morada de entrega e que podem existir múltiplos pedidos do mesmo restaurante.

Para isso, na função `SpecialTask * simultaneousRequests(Graph<Coordinates> & graph, vector<Request> & requests, Employee* employee)` - definida em `SpecialTask.cpp` ; utilizam-se dois vetores auxiliares que acabam por funcionar como filas, uma com o restaurante não visitado mais próximo da posição atual do estafeta à cabeça e outra com a morada de entrega não visitada mais próxima da posição atual do estafeta à cabeça.
Estas posições mais próximas são atualizadas nas funções:
`setNearestRestaurant(Graph<Coordinates> & graph, vector<Request> & requests, Coordinates origin)`
e
`setNearestDeliveryAddress(Graph<Coordinates> & graph, vector<Request> & requests, Coordinates origin)` , chamadas a cada iteração.

Enquanto existirem restaurantes ou moradas de entrega por visitar, o próximo ponto a visitar é escolhido do seguinte modo:

- Na primeira iteração e sempre que não há pedidos para entregar é necessário recolher um pedido de um restaurante;
- Quando a carga máxima é atingida ou já foram recolhidos todos os pedidos é necessário entregar um pedido;
- Se existir capacidade para tal, recolhe-se múltiplos pedidos de restaurantes dos quais tenha sido feito mais do que um pedido;
- No caso de haver possibilidade de entregar ou recolher um pedido, opta-se por escolher o deslocamento de menor distância;

A ordem de visita dos vários pontos é guardada num vetor, que mantém a associação das coordenadas do ponto com o pedido ao qual este está associado, de modo a facilitar cálculos posteriores de tempos estimados.

Finalmente, é definido o caminho completo e também os tempos estimados para cada pedido na função
`setFloydWarshallPath(Graph<Coordinates> & graph, const vector<pair<Coordinates,unsigned long>> & checkpoints)` - em `SpecialTask.cpp` ; que procede de modo similar à função do mesmo nome descrita anteriormente, calculando o caminho mais curto entre cada par de pontos consecutivos do caminho parcial.

```
simultaneousRequests(Graph G, vector<Request> R, Employee employee){
    vector<Coordinates> path // Pontos por ordem de visita
    Coordinates orig ← coordinates(employee) // Ponto de origem

    vector<Request> pick ← R // Pedidos a recolher do restaurante
    setNearestRestaurant(G, pick, orig) // Restaurante mais perto da origem na frente
    vector<Request> deliver; // Pedidos recolhidos, a ser entregues nas moradas

    int totalCargo ← 0;
    while deliver == ∅ || pick == ∅ :
        // Estafeta não tem mais capacidade / Não há mais pedidos para recolher
        if totalCargo == maxCargo(e) || pick == ∅ :
            // Entrega pedido já recolhido (da morada mais perto)
            orig ← deliveryAddr(deliver.front())
            path.push_back(orig)
            totalCargo ← totalCargo - cargo(deliver.front())
            deliver.erase(deliver.begin())

        // Não há mais pedidos para entregar
        else if deliver == ∅ :
            // Recolher pedido(s) do restaurante mais perto da localização atual
            orig ← restaurant(pick.front())
            path.push_back(orig)
            totalCargo ← totalCargo + cargo(pick.front())

            // Recolher pedidos do mesmo restaurante, se possível(verifica carga)
            repeatedRestaurants(pick,deliver,totalCargo,maxCargo(employee));

            // Adicionar aos pedidos a entregar, remover dos pedidos a recolher
            deliver.push_back(pick.front())
            pick.erase(pick.begin())

        // Escolher a opção mais próxima - recolher ou entregar um pedido
        // Se não tiver capacidade entrega um pedido
        else:
            if(totalCargo + cargo(pick.front()) > maxCargo(employee) ||
                G.getDist(orig,deliveryAddr(deliver.front())) < G.getDist(orig,restaurant(pick.front()))):
                //...Entrega um pedido...
            else: //...Recolhe pedido(s) do restaurante mais perto...

        // Manter na frente o restaurante mais perto da nova origem
        setNearestRestaurant(G,pick,orig)
        // Manter na frente a morada (dos pedidos recolhidos) mais perto da origem
        setNearestDeliveryAddress(G,deliver,orig)

    SpecialTask s(employee,R) // Atribui pedidos ao estafeta
    // Constrói caminho total a partir da ordem de visita dos restaurantes e moradas
    s.setFloydWarshallPath(G,path)
    return s
}
```

6.4 Nota geral acerca dos algoritmos implementados

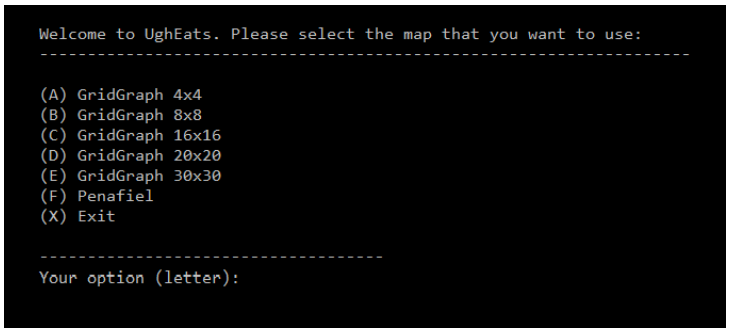
Por fim, é de salientar a vantagem proporcionada pelo pré-processamento com o algoritmo de Floyd Warshall quando utilizados algoritmos como os descritos acima. Para efeitos de comparação, no entanto, podem ser consultados em `simulations.cpp` e em `SingleTask.cpp` os algoritmos que testamos na Fase II e que utilizam o algoritmo de Dijkstra na sua implementação. Também a análise das funções `virtual void setFloydWarshallPath(Graph<Coordinates> & graph)` e `void setDijkstraPath(Graph<Coordinates> & graph)` permite facilmente perceber que, ao utilizar o Floyd Warshall, é evitado o cálculo repetitivo das distâncias e do caminho mais curto para cada novo ponto de origem, uma desvantagem do Dijkstra. *Parte da função utilizada para definir o caminho final de um estafeta, nos algoritmos 1 e 2:*

```
void SingleTask::setFloydWarshallPath(Graph<Coordinates> & graph){
    //...
    // Percorrer coordenadas do pedido - por ordem de visita
    for(Coordinates checkpoint: request.getCheckpoints()){
        // Caminho mais curto desde o ponto anterior
        tempPath = graph.getfloydWarshallPath(orig, checkpoint);
        // Acumular a distância
        totalDistance += graph.getDist(graph.findVertexIdx(orig), graph.findVertexIdx(checkpoint));
        //...
        // Adicionar caminho calculado ao caminho construído até então
        path.insert(path.end(),tempPath.begin()+1, tempPath.end());
        orig = checkpoint; // Definir novo ponto de origem
    }
    //...
}
```

7. Principais casos de uso implementados

Depois de todos os algoritmos estarem implementados e estudadas todas as questões relacionadas com a análise temporal e espacial, passamos para o desenvolvimento da solução para o contexto do projeto.

A aplicação desenvolvida, poderá ser utilizada em vários mapas (4x4, 8x8, 16x16, 20x20, 30x30 e Penafiel), cabendo ao utilizador escolher o mapa que deseja quando a aplicação se inicia.



Escolhido o mapa, é perguntado ao utilizador se pretende fazer apenas um pedido ou simular múltiplos pedidos.

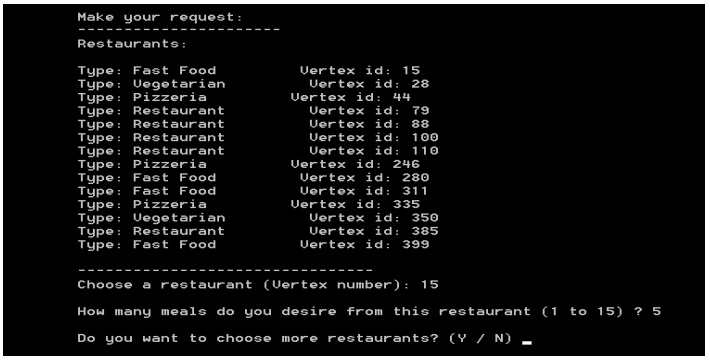
Escolha do tipo de pedido

De seguida, irá ser perguntado ao utilizador se deseja visualizar o mapa escolhido, onde será possível visualizar a localização dos vários restaurantes.

Seguidamente, no caso de grafos maiores, será mostrada uma mensagem ao utilizador durante alguns segundos, indicando que o mapa escolhido está a ser processado. É de notar que, no caso do grafo da cidade de Penafiel, o processamento já foi previamente realizado (com o algoritmo de Floyd Warshall), sendo este tempo de espera provocado pela leitura das matrizes de distâncias e de predecessor no caminho mais curto dos respetivos ficheiros. Em todos os outros casos o processamento é realmente feito no momento, recorrendo ao mesmo algoritmo.

7.1 Realização de um pedido

Quando a opção escolhida é fazer apenas um pedido, é possível fazer um pedido que pode conter apenas um restaurante entre os apresentados na lista ou então vários, sendo que para cada restaurante se escolhe quantas refeições se deseja deste, estando o tamanho total do pedido limitado a 15 refeições.

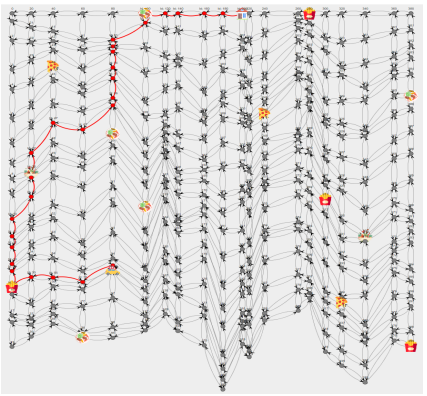


Menu de escolha dos restaurantes para o grafo 20x20

Depois de o pedido estar completo, serão apresentadas na aplicação as informações sobre o pedido, sobre o estafeta que o vai entregar, o caminho percorrido pelo estafeta desde a sua localização inicial até ao local de entrega, a distância total percorrida e o tempo estimado de espera. É ainda apresentado o grafo com o caminho percorrido pelo estafeta.

```
Request: Id = 0; Restaurants = 15 ; Delivery address = 200; Cargo = 5; Date: 19/5/2020; Hour: 18:2
Employee: Id = 6 Vehicle = Car; MaxCargo = 15; Avg Velocity = 40
Initial Employee's Position: 96
Path: 96 95 75 55 35 15 14 13 12 11 10 30 29 28 27 47 67 87 86 85 84 83 82 81 101 100 120 140 160 180 200
Total distance: 2100.7 m
Estimated time: 3 min

Enter to exit!
```

Informações sobre o pedido e respetiva entrega



Caminho percorrido pelo estafeta

*Para a implementação deste caso de utilização são usados os algoritmos **1** e **2**, sendo o segundo usado se forem definidos múltiplos restaurantes e o primeiro quando é escolhido apenas um.*

7.2 Simulação de múltiplos pedidos

No caso de a opção escolhida ser a de simular vários pedidos, existem 2 opções.

```
(A) Simultaneous Requests - One employee
(B) Requests in Temporal Order - One/Many employees
(X) Exit

Choose an option:
```

Opções de simulação de pedidos

7.2.1 Um estafeta - Vários pedidos no mesmo deslocamento (variante do TSP)

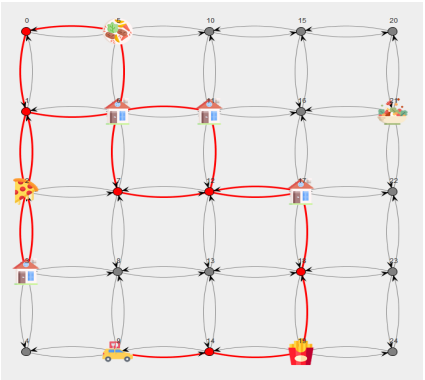
Caso o utilizador opte pela **opção A**, será apresentado ao utilizador o estafeta disponível para entregar os pedidos simultâneos e será pedido ao utilizador quantos pedidos quer simular.

```
Our employee:
Employee: Id = 0 Vehicle = Car; MaxCargo = 25; Avg Velocity = 40; Initial Position: 9

Number of requests (1 to 5) ?:
```

Introdução dos dados para simulação de pedidos simultâneos

Depois de escolhidos, irão aparecer as informações de cada um dos pedidos simulados, bem como sobre a sua entrega, tal como foi descrito anteriormente.

É também mostrado ao utilizador o grafo com o caminho percorrido pelo estafeta para entregar os pedidos.



Caminho percorrido pelo estafeta

*Para a implementação deste caso de utilização é usado o algoritmo **3**.*

7.2.2 Múltiplos estafetas - distribuição de pedidos por ordem temporal

Se o utilizador preferir a **opção B**, a seguir terá de escolher um número de estafetas, entre 1 e 10, e o número de pedidos a simular.

```
int maxRequests = 15;
do{
        cout << "\t Number of employees (1 to 10) ?: ";
        cin >> employeesNum;
        cin.ignore(1000, '\n');
        cout << endl;
        if(employeesNum > 10 || employeesNum < 1) cout << "Try again!" << endl;
} while(employeesNum > 10 || employeesNum < 1);
```

```
Number of employees (1 to 10) ?:

Number of requests (1 to 15) ?:
```

Introdução dos dados para simulação de pedidos

De seguida, são apresentadas as informações sobre os pedidos gerados e sobre os estafetas criados, assim como as informações sobre a entrega de cada um dos pedidos, tal como acima foi descrito.

No final da lista, aparece um menu onde podemos escolher entre algumas possíveis visualizações dos grafos:

```
(A) View a Request in city map
(B) View an Employee's complete route in city map
(C) View all Employees complete routes in city map
(X) Exit

Choose an option:
```

Opções de visualização dos grafos

Opção A - Visualizar o caminho percorrido para realizar um dado pedido;

Opção B - Visualizar o percurso total efetuado por um determinado estafeta;

Opção C - Visualizar os percursos efetuados por todos os estafetas.

*Para a implementação deste caso de utilização é usado o algoritmo **1**.*

8. Conclusão

9. Referências

- “Introduction to Algorithms”, 3rd Edition, T.H. Cormen, C. E. Leiserson, R. L. Rivest , C. Stein., MIT Press, 2009
- “Data Structures and Algorithm Analysis in Java”, Second Edition, Mark Allen Weiss, Addison Wesley, 2006
- Algoritmo A*,
  https://brilliant.org/wiki/a-star-search/#heuristics
- A*’s Use of the Heuristic,
  http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html
- Algoritmo de Dijkstra,
  https://brilliant.org/wiki/dijkstras-short-path-finder/
- Dijkstra’s shortest path algorithm | Greedy Algo-7
  https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/
- Algoritmo de Floyd-Warshall,
  https://brilliant.org/wiki/floyd-warshall-algorithm/
- Floyd Warshall Algorithm | DP-16
  https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/
- Comparison of Dijkstra’s and Floyd–Warshall algorithms
  https://www.geeksforgeeks.org/comparison-dijkstras-floyd-warshall-algorithms/
- Comparison of Dijkstra’s algorithm and Floyd–Warshall algorithm
  https://www.selmanalpdundar.com/comparison-of-dijkstras-algorithm-and-floyd-warshall-algorithm.html
- Am I right about the differences between Floyd-Warshall, Dijkstra and Bellman-Ford algorithms?
  https://cs.stackexchange.com/questions/2942/am-i-right-about-the-differences-between-floyd-warshall-dijkstra-and-bellman-fo
- Dijkstra vs. Floyd-Warshall: Finding optimal route on all node pairs
  https://stackoverflow.com/questions/4212431/dijkstra-vs-floyd-warshall-finding-optimal-route-on-all-node-pairs
- Travelling salesman problem,
  https://en.wikipedia.org/wiki/Travelling_salesman_problem