



Projeto de Conceção de algoritmos - CAL

VaccineRouter

Turma 1 – Grupo 2

Beatriz Aguiar

up201906230@fe.up.pt

Margarida Vieira

up201907907@fe.up.pt

Índice

| | |
|--|-----------|
| Introdução | 4 |
| Descrição do problema | 4 |
| Modularização do problema | 5 |
| <i>Porquê modularizar?</i> | <i>5</i> |
| <i>Pré-processamento de dados</i> | <i>5</i> |
| <i>Iterações</i> | <i>5</i> |
| Formalização do problema | 7 |
| Dados de entrada | 7 |
| <i>Restrições aos dados de entrada</i> | <i>7</i> |
| Dados de saída | 8 |
| <i>Restrições aos dados de saída</i> | <i>8</i> |
| Função objetivo | 9 |
| Estruturação de código | 10 |
| <i>Classes</i> | <i>10</i> |
| <i>Source & Header files</i> | <i>11</i> |
| <i>Diagrama UML</i> | <i>12</i> |
| Solução Conceptual | 13 |
| Pré-processamento | 13 |
| <i>Construção/Inicialização do grafo</i> | <i>13</i> |
| <i>Pré-processamento do grafo</i> | <i>13</i> |
| <i>Pré-processamento dos dados</i> | <i>14</i> |
| Problemas e Algoritmos | 15 |
| <i>Pré-processamento</i> | <i>15</i> |
| <i>1ª iteração</i> | <i>15</i> |
| <i>2ª iteração</i> | <i>17</i> |
| <i>3ª iteração</i> | <i>18</i> |
| <i>4ª iteração</i> | <i>19</i> |
| Interface | 20 |
| Análise e Resultados | 21 |
| Divisão do grafo em subgrafos | 21 |
| Obtenção dos componentes fortemente conexos de um grafo | 23 |
| Cálculo da rota de duração mínima entre dois pontos | 23 |

| | |
|--|----|
| Cálculo da rota de duração mínima passando por vários pontos | 24 |
| Conclusão – 1ª entrega | 26 |
| Detalhes do projeto – 1ª entrega | 26 |
| Estruturas de dados utilizadas | 27 |
| Algoritmos implementados | 28 |
| Análise da conectividade – DFS | 28 |
| <i>Pseudocódigo e análise da complexidade</i> | 28 |
| 1ª iteração – Entregas de um Centro de Armazenamento a um Centro de Aplicação | 29 |
| <i>Pseudocódigo e análise da complexidade</i> | 29 |
| 2ª iteração – Entregas de um Centro de Armazenamento a vários Centros de Aplicação | 30 |
| <i>Pseudocódigo e análise da complexidade</i> | 30 |
| | 30 |
| 3ª iteração - Entregas de um Centro de Armazenamento a vários Centros de Aplicação com restrição temporal | 32 |
| <i>Pseudocódigo e análise da complexidade</i> | 32 |
| 4ª iteração - Entregas de vários Centros de Armazenamento a vários Centros de Aplicação com restrição temporal | 33 |
| <i>Pseudocódigo e análise da complexidade</i> | 33 |
| | 34 |
| Casos de utilização e principais funcionalidades | 35 |
| Conclusão – 2ª entrega | 46 |
| Detalhes do projeto – 2ª entrega | 47 |
| Referências | 48 |

Introdução

Descrição do problema

A logística relacionada com a distribuição de vacinas contra a COVID-19 requer um cuidado especial para o seu transporte. As vacinas ficam armazenadas em centros específicos de armazenamento, determinados previamente, capazes de cumprir com um conjunto de condições de segurança e conservação. Assim, é necessário que o transporte seja realizado de forma controlada, tendo em consideração as doses a serem aplicadas nos vários centros de aplicação.

O *VaccineRouter* é uma aplicação capaz de determinar os itinerários de distribuição de vacinas desde os centros de armazenamento até aos centros de aplicação. O itinerário completo não deve demorar mais do que um determinado tempo previamente estabelecido, tendo em consideração as restrições de conservação das vacinas.

A cada dia, o centro operacional de distribuição recebe o número de inscritos a serem vacinados em cada centro de aplicação e determina os centros de aplicação que serão servidos por cada centro de armazenamento. No caso do itinerário total de serviço de um centro de armazenamento exceder uma dada estimativa temporal, será necessário dividir a distribuição com outros centros de armazenamento ou aumentar-se o número de veículos a utilizar, dividindo-se o itinerário em percursos mais curtos.

Modularização do problema

Porquê modularizar?

A resolução do problema em mãos será subdividida em módulos de dificuldade incremental, os quais serão, mais detalhadamente, abordados nas secções abaixo.

Respondendo à questão que dá mote a esta secção, optou-se por modularizar o código de forma a detetar, o mais precocemente possível, eventuais problemas que possam vir a surgir aquando da implementação do código. Assim sendo, as várias iterações que abaixo se propõem, correspondem a sub-problemas do problema final que se visa resolver.

Pré-processamento de dados

Esta etapa passa por tratar o grafo sobre o qual incidirão os vários algoritmos e determinar o escalonamento centros de armazenamento \leftrightarrow centros de aplicação, nomeadamente quantas carrinhas terão que ser despachadas por cada centro de armazenamento para fazer face a todas as encomendas de vacinas, tendo em consideração o tempo limite disponível para o transporte, associado ao prazo de conservação das mesmas. No entanto, há que referir que estes detalhes apenas começarão a ser tidos em conta a partir da 3ª iteração, inclusive.

Iterações

Abaixo segue um resumo das várias iterações que se procurará implementar, cada uma das quais será abordada com mais detalhe na secção [Solução Conceptual](#)¹, nomeadamente com a respetiva análise dos algoritmos que visam resolver os problemas que surgem em cada uma.

- **Pré-processamento de dados**
- **1ª iteração:**
 - Entrega de um centro de armazenamento a um único centro de aplicação.
 - Apenas o caminho entre ambos é tido em consideração – instância do problema do caminho mais curto entre dois pontos de um grafo.
- **2ª iteração:**
 - Entregas de um centro de armazenamento a vários centros de aplicação.

¹ ctrl-click para seguir o link.

- Necessidade de calcular a melhor rota que passa por todos os centros de aplicação – instância do problema *Travelling Salesman - TSP*.
- Nesta iteração ainda não será tido em consideração o tempo de conservação das vacinas.
- **3ª iteração:**
 - Entregas de um centro de armazenamento a vários centros de aplicação.
 - Nesta iteração já será tido em consideração o tempo de conservação das vacinas – instância do problema *Travelling Salesman*, com *constraint* temporal.
 - No caso do tempo da melhor rota calculada usando apenas uma carrinha ser superior ao tempo de conservação das vacinas, há que avaliar a melhor estratégia – manter o centro de armazenamento em causa encarregue de todas distribuições, mas recorrer a mais veículos para o efeito ou reatribuir os centros de aplicação a outros centros de armazenamento.
- **4ª iteração:**
 - Entregas de vários centros de armazenamento a vários centros de aplicação, com necessidade de calcular quais dos primeiros servirão cada um dos centros de aplicação.

Formalização do problema

Dados de entrada

$G(N, E)$ – Grafo dirigido e pesado com:

- N – *nodes*
 - ID – *node ID*
 - **coordenadas** – latitude e longitude no mapa
 - $adj \subseteq E$ – *edges* correspondentes ao *node*
- E – *edges* que ligam vários *nodes*
 - ID – *edge ID*
 - w – peso que representa a distância entre dois *nodes* ligados através do *edge*
 - $dest \in N$ – *node* de destino do *edge*

$S \subseteq N$ – *set* de *nodes* que representam os centros de armazenamento

$A \subseteq N$ – *set* de *nodes* que representam os centros de aplicação

R_i – raio de ação de cada centro de armazenamento

V_i – número de vacinas a ser entregue aos centros de aplicação $\in S$, i representando a viatura que parte do centro de armazenamento

T – intervalo de tempo para a distribuição das vacinas (não tido em consideração nas 1ª e 2ª iterações)

Restrições aos dados de entrada

- $\forall e \in E, w(e) > 0$, considerando que o peso de uma *edge* representa a distância entre dois pontos do mapa
- $V_i > 0$, o que significa que um veículo precisa de uma quantidade de vacinas a entregar para justificar a sua partida de um centro de armazenamento
- $T > 0$
- Se o caminho mais rápido para qualquer centro de aplicação exceder o intervalo de tempo para a distribuição da vacina, o centro de armazenamento é removido de S , ficando inativo pelo facto do seu raio de ação não abranger nenhum centro de aplicação

Dados de saída

$P_i \subseteq \mathbf{N}$ – *set* de *nodes* que representa o caminho percorrido pela viatura, ordenado pela ordem de visita

S_i – centro de armazenamento que enviou a viatura responsável pela distribuição

T_i – tempo total de cada percurso P_i

N_i – número de vacinas a ser entregue em cada centro de aplicação

V_s – número de veículos por cada centro de armazenamento

Restrições aos dados de saída

- $p \in P_i \wedge p_0 = S_i$, o ponto de partida é um centro de armazenamento que fornece as vacinas a todos os centros de aplicação contidos no percurso P_i
- $p \in P_i \wedge p_n = A_i$, o ponto final é o último centro de aplicação visitado pela viatura no percurso P_i
- $|P_i| \geq 2$, um itinerário contém no mínimo um centro de armazenamento e um ou mais centros de aplicação
- $T_i \leq T$
- $N_i = V_i$

Função objetivo

A solução ótima do problema é garantida quando todas as vacinas são entregues (dentro do intervalo de tempo da sua transportação), requerendo o número mínimo de veículos. Sendo assim, a solução ótima passa pela maximização da função 2, dentro de um intervalo limite, e pela minimização da função 1.

Função 1

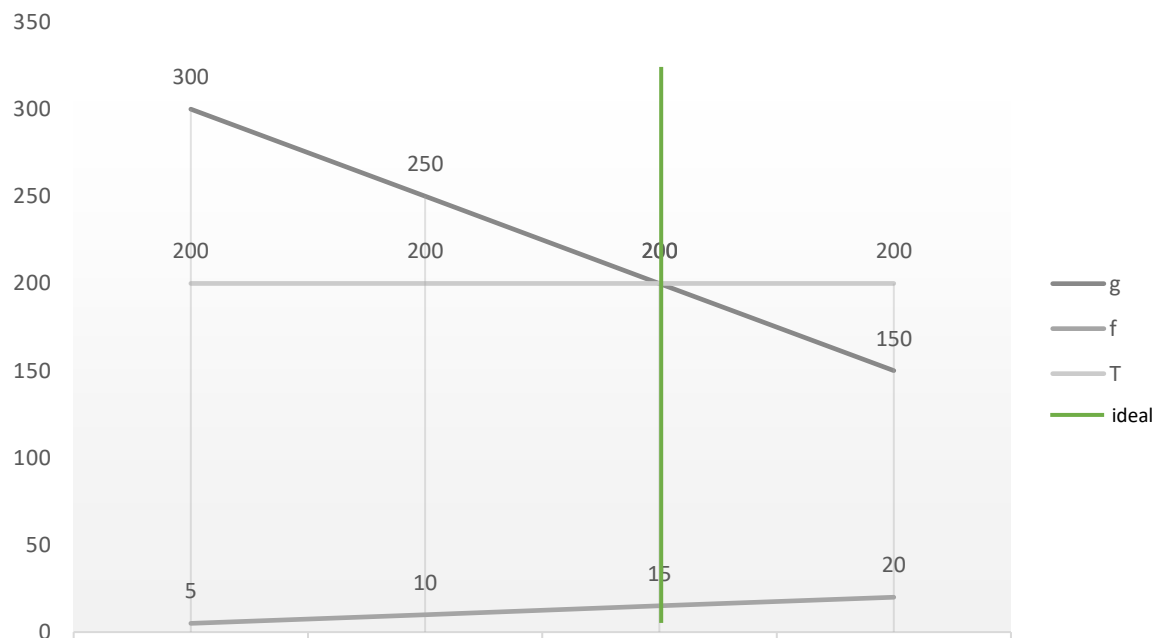
$$f = |Vs|$$

Função 2

$$g = \left(\sum_{i=0}^{n-1} w(e) * velocidade \right) \leq T$$

sendo $e \in E$, o *edge* que liga os *nodes* n_i e n_{i+1} no percurso P_i

Exemplo:



Estruturação de código

Disclaimer: A seguinte estruturação de código é apenas um *template*. Eventuais mudanças poderão ocorrer na implementação do código.

Classes

Classes principais

Interface

Responsável pela interação *user*-programa.

GraphViewer

Classe fornecida pela unidade curricular, utilizada para o *display* do mapa escolhido pelo *user*.

VaccineRouter

Classe principal que regista e lida com todos os dados.

De acordo com os inputs do *user*, esta classe irá estar responsável por todas as chamadas de funções e fluxo do programa.

Classes de objetos

StorageCenter

ApplicationCenter

Vehicle

Classes auxiliares

Coordinates

Regista as coordenadas do node e implementa todas as operações de cálculo de distâncias.

Time

Engloba todas as funções relacionadas com o tempo, com destaque nos operadores.

Classes relacionadas com os algoritmos

Graph

Fornecida pela unidade curricular.

Node \leftarrow *NodeD* \leftarrow *NodeA*

Fornecida pela unidade curricular, com adaptações requeridas pelos algoritmos.

Edge

Fornecida pela unidade curricular.

Source & Header files

GraphProcessor

Encarregue de criar o grafo e processá-lo. A escolha do ficheiro provém do input do *user*.

Algorithms

Todos os algoritmos a ser utilizados no programa.

MutablePriorityQueue.h

Fornecida pela unidade curricular.

Diagrama UML

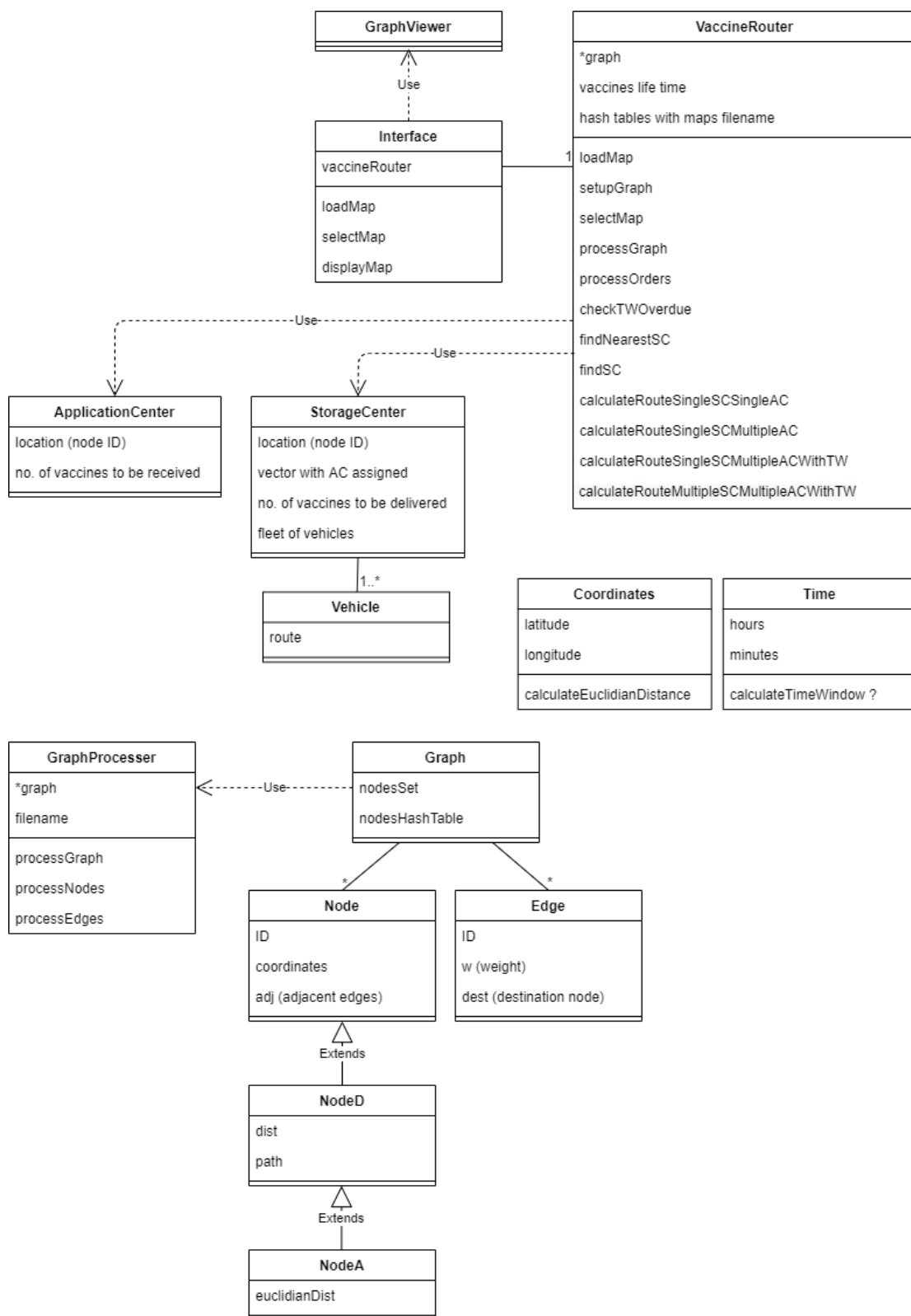


Figura 1

Solução Conceptual

Pré-processamento

Apresentam-se, de seguida, os problemas encontrados ao longo das várias iterações, para os quais serão identificadas perspectivas de resolução bem como os algoritmos a serem implementados.

Construção/Inicialização do grafo

A informação contida no ficheiro de texto correspondente à localidade seleccionada pelo utilizador será convertida para um grafo. Este tratamento está, no entanto, dependente do formato da informação disponibilizada pela plataforma *OpenStreetMap*.

Pré-processamento do grafo

Após a construção do grafo propriamente dito, este passará por uma fase de pré-processamento que visa, sobretudo, reduzir o número de *nodes* e *edges*, de forma a aumentar a eficiência temporal dos algoritmos que irão incidir sobre o mesmo.

A primeira etapa passa por, a partir do grafo construído, gerar um número de grafos correspondente ao número de centros de armazenamento existentes no mesmo.

A geração dos vários grafos será feita com base no raio de ação, R_a , que corresponde à distância máxima, em linha reta, que um veículo pode percorrer, desde um centro de armazenamento, de forma a não exceder o tempo de conservação das vacinas, e passará por remover todos os *nodes* e respetivas *edges* incidentes que se encontrem a uma distância maior do que R_a do centro de armazenamento a ser tido em consideração. A escolha deste critério deve-se ao facto de que, qualquer *node* que se encontre a uma distância maior do que R_a corresponderá, necessariamente, a um *node* que, por restrições temporais, não pode ser alcançado por um veículo transportador de vacinas.

De seguida, devido à natureza circular das viagens – os veículos têm que ser capazes de regressar ao centro de armazenamento do qual partiram após terem concluído todas as entregas das vacinas – *nodes* e respetivas *edges* incidentes que não pertençam ao mesmo componente fortemente conexo dos grafos devem também ser eliminados.

A vantagem mais evidente da abordagem da geração dos vários grafos é que, desta forma, a segunda etapa poderá ser realizada recorrendo a *threads*, dado que se trata de aplicar o mesmo algoritmo a todos os grafos.

No entanto, no sentido de apurar se, de facto, esta estratégia seria vantajosa, foram realizados testes, cuja análise e exposição dos resultados se encontram na secção *Análise e Resultados – Divisão do*

grafo em subgrafos². Dado que os resultados se revelaram favoráveis à adoção da estratégia descrita, esta será implementada na segunda parte deste projeto.

Com as duas etapas acima, garante-se que, na aplicação dos algoritmos, o grafo não terá *nodes* e/ou *edges* que não podem ser percorridos pelos veículos.

Pré-processamento dos dados

Esta etapa passa por determinar o número de veículos necessários para suprir todos os pedidos de vacinas realizados.

Portanto, assim que as restrições temporais comecem a ser tidas em conta – a partir da 3ª iteração, inclusive - pode dar-se o caso de, com apenas um veículo, não ser possível realizar a entrega de todas as encomendas.

Face a este problema surgem duas alternativas – recorrer a um maior número de carrinhas e manter as entregas a cargo do centro de armazenamento originalmente encarregue das mesmas, ou reatribuir algumas das entregas a outro centro de armazenamento, se tal não impossibilitar a realização das entregas já atribuídas ao mesmo.

² ctrl-click para seguir o link.

Problemas e Algoritmos

Pré-processamento

Devido à natureza circular das viagens – os veículos têm que ser capazes de regressar ao centro de armazenamento do qual partiram, após terem concluído a entrega de todas as vacinas – *nodes* e respectivas *edges* incidentes que não pertençam ao componente fortemente conexo do centro de armazenamento relativo ao grafo a ser tido em consideração devem ser eliminados, para evitar que um veículo se veja impossibilitado de regressar ao ponto do qual partiu.

Para a obtenção de todos os componentes fortemente conexos de cada grafo serão ponderados dois algoritmos – **Kosaraju** e **Tarjan**.

Kosaraju

O algoritmo de **Kosaraju** consiste em fazer uma pesquisa em profundidade no grafo, numerando os *nodes* em pós-ordem; de seguida, inverter todas as *edges* do grafo e realizar uma segunda pesquisa em profundidade, começando pelos *nodes* de numeração superior ainda por visitar.

No final da execução do algoritmo, cada árvore obtida é um componente fortemente conexo.

No entanto, interessa apenas o componente fortemente conexo correspondente ao centro de armazenamento relativo ao grafo a ser processado no momento. Partindo deste princípio, o algoritmo pode ser otimizado e parar aquando da descoberta deste.

Tarjan

O algoritmo de **Tarjan** é semelhante ao acima descrito, mas recorre apenas a uma pesquisa em profundidade, pelo que se afigura mais eficiente e, portanto, mais apelativo na resolução do problema em mãos.

1ª iteração

Nesta iteração, na qual apenas se considera uma entrega de um centro de armazenamento a um único centro de aplicação, o problema reduz-se a encontrar o caminho mais curto entre estes dois pontos.

Para a resolução deste problema, serão avaliados os algoritmos de **Dijkstra** e **A-Star**. O algoritmo de **Bellman-Ford** também poderia resolver o problema, no entanto, a única vantagem que este oferece relativamente aos anteriores é a de não obrigar a que as *edges* tenham p

esos positivos. No entanto, tal como já foi referido, só serão considerados grafos cujas *edges* apresentem pesos positivos, devido ao significado do peso no contexto do problema, pelo que não vimos vantagem no recurso a este algoritmo.

Dijkstra

O algoritmo de **Dijkstra** destaca-se pela sua facilidade de implementação e, embora tenha sido originalmente concebido para solucionar o problema de encontrar o caminho mais curto desde um *node* de origem até qualquer outro do grafo, é possível otimizá-lo, fazendo cessar o algoritmo quando se encontra o *node* de destino.

No entanto, uma desvantagem deste algoritmo é o de realizar uma espécie de pesquisa em largura no sentido em que esta é efetuada num círculo que se vai expandindo em torno do *node* definido como origem até que o *node* de destino seja encontrado. Este comportamento tem implicações, sobretudo em grafos de maior dimensão, tal como os que virão, eventualmente, a ser considerados neste programa, devido à quantidade de *nodes* não promissores que serão explorados.

Para a implementação do algoritmo de **Dijkstra**, será feita uma extensão à classe *Node*, **NodeD**, que possuirá, para além dos atributos herdados, os atributos adicionais:

- dist – distância mínima até à origem
- path – node antecessor no caminho mais curto

A-Star

Numa tentativa de corrigir as desvantagens do algoritmo de **Dijkstra**, mencionadas acima, surge o algoritmo **A-Star**. Este recorre à heurística da distância euclidiana entre dois *nodes* para guiar a sua pesquisa, o que se traduz na obtenção de melhores resultados, sobretudo em grafos mais densos. O que isto significa é que apenas serão explorados *nodes* efetivamente promissores, ou seja, *nodes* que não constituem um afastamento em termos de distância euclidiana do *node* de destino.

Visto ter que analisar menos *nodes*, este algoritmo torna-se mais eficiente e rápido na obtenção do caminho mais curto, pelo que se afigura uma abordagem mais vantajosa.

Para a implementação do algoritmo de **A-Star**, será feita uma extensão à classe *NodeD*, **NodeA**, que possuirá, para além dos atributos herdados, os atributos adicionais:

- euclidianDist – distância euclidiana ao destino

2ª iteração

Nesta iteração, dado que já se consideram entregas de um centro de armazenamento a vários centros de aplicação, com necessidade de calcular a melhor rota que passa por todos os centros de aplicação, o problema afigura-se similar ao do *Travelling Salesman - TSP*.

O *TSP* é um problema de otimização *NP-hard*, inspirado na necessidade dos vendedores em realizar entregas em diversos locais, percorrendo o menor caminho possível e reduzindo ao máximos os custos associados à realização do percurso. O problema consiste, portanto, na procura de um circuito que possua a menor distância possível, começando num determinado ponto de origem, passando por quaisquer pontos necessários uma única vez e regressando ao ponto de origem.

Ora, à semelhança do supra descrito, também o problema desta iteração passa por determinar a melhor rota desde um centro de armazenamento, passando por todos os centros de aplicação que tenham efetuado uma encomenda, para já sem quaisquer restrições temporais.

Para a resolução do problema, temos ao nosso dispor tanto algoritmos que alcançam soluções exatas – **força bruta** e **Held-Karp** – como algoritmos que alcançam soluções aproximadas através do recurso a heurísticas de decisão – heurística “*Nearest Neighbour*”.

Solução exata - Força bruta

A abordagem de força bruta, também conhecida como abordagem ingénua, calcula e compara todas as permutações possíveis de rotas, com o intuito de determinar uma solução ótima única.

O algoritmo passa por calcular todas as rotas possíveis, passando pelos pontos necessários, e calcular a distância de cada rota computada de forma a garantir que, no final, se opte pela rota mais curta, correspondente à solução ótima.

Solução exata - Held-Karp

O algoritmo **Held-Karp** é um algoritmo de programação dinâmica que foi desenvolvido com o intuito de solucionar o *TSP*.

Solução aproximada - Heurística “*Nearest Neighbour*”

A heurística de decisão “*Nearest Neighbour*” é, provavelmente, a mais simples para a resolução, ainda que aproximada, do *TSP*.

Seguindo esta abordagem, a estratégia traduzir-se-ia em, partindo do centro de armazenamento encarregue da distribuição das encomendas, ir optando sucessivamente pelo centro de aplicação mais

próximo, relativamente ao ponto atual onde se encontrasse o veículo. Assim que todos os centros de aplicação tivessem sido servidos, o veículo retorna ao centro de armazenamento do qual partiu.

De notar que esta heurística de decisão, de facto, não garante o caminho mais curto no trajeto circular que consiste em partir de um centro de armazenamento e regressar ao mesmo no final da entrega de todas as encomendas, embora constitua uma boa aproximação. No entanto, no problema em mãos, não há necessidade de garantir que o trajeto circular acima é o melhor, mas sim que o trajeto até ao último centro de aplicação a ser servido é o melhor, por restrições temporais que serão precisas ter em atenção, tal como se verá na análise da próxima iteração. Nesta fase, há apenas que salientar que, este relaxamento face ao *TSP* original, acaba por tornar esta abordagem bastante apelativa.

3ª iteração

Esta iteração é uma extensão à iteração anterior dado que o problema é, essencialmente, o mesmo, calcular a melhor rota que passa por todos os centros de aplicação, partindo do centro de armazenamento, mas com restrição temporal imposta pelo período limitado de conservação das vacinas.

No caso de não ser possível, com apenas um veículo, efetuar as entregas a todos os centros de aplicação, terá que se optar por uma das seguintes alternativas:

- I. recorrer a um maior número de veículos e manter as entregas a cargo do centro de armazenamento originalmente encarregue das mesmas, ou
- II. reatribuir algumas das entregas a outro centro de armazenamento, se tal não impossibilitar a realização das entregas já atribuídas ao mesmo – pormenor que só poderá ter impacto na próxima iteração, uma vez que a atual apenas considera um centro de armazenamento e, como tal, todos os outros candidatos a realizar a entrega estarão livres.

Considerando a função objetivo, tentar-se-á optar, sempre que possível, pela segunda alternativa, uma vez que é a que minimiza o número de carrinhas utilizadas na distribuição.

Consideremos o grafo correspondente ao centro de armazenamento que está a causar conflitos.

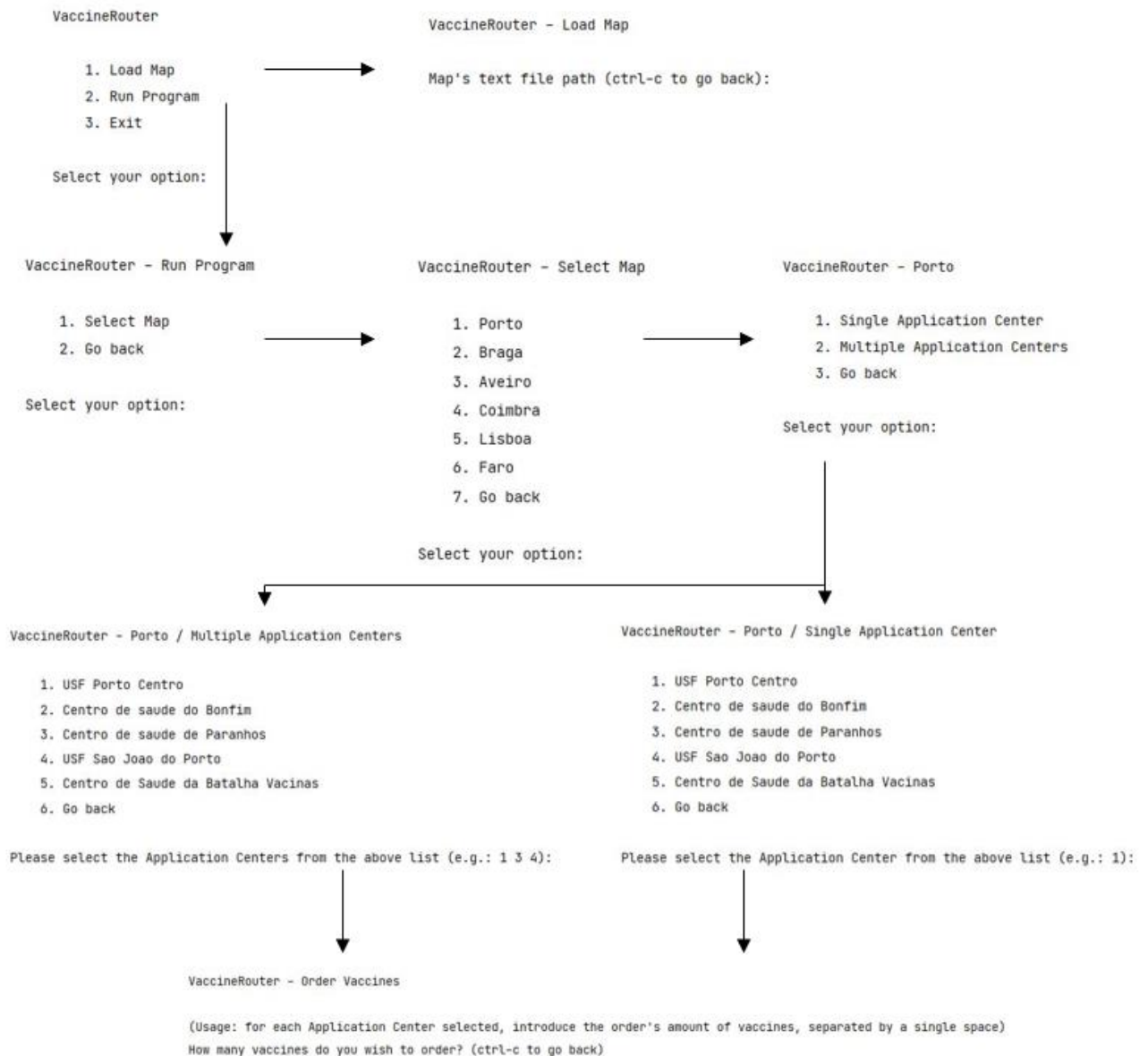
A primeira etapa passa por verificar, até que não existam mais conflitos, para cada centro de aplicação, se este faz parte do grafo de outro centro de armazenamento. No caso de existir mais do que um centro de armazenamento candidato a ficar a cargo da entrega, tentar-se-á optar pelo mais próximo ao centro de aplicação ou, no caso de tal ser impossível pelos motivos expostos acima, pelo centro de armazenamento capaz de receber a nova entrega.

No caso do problema não ser resolúvel pela estratégia acima, optar-se-á pela outra alternativa. Aqui, a estratégia passa por, começando por aquele que seria o último centro de aplicação na rota do veículo com conflitos, e assim sucessivamente até que tal deixe de se verificar, ir preenchendo um novo veículo, obviamente com a verificação da viabilidade da respetiva rota.

4ª iteração

Esta iteração é, na sua essência, uma instância do *Vehicle Routing Problem*. No entanto, considerando a estratégia de resolução do problema explicada até então, acaba por se traduzir numa aplicação *multi-threaded* da iteração anterior, ou seja, de forma resumida, o procedimento explicado acima será realizado para todos os centros de armazenamento, paralelamente.

Interface



Análise e Resultados

Divisão do grafo em subgrafos

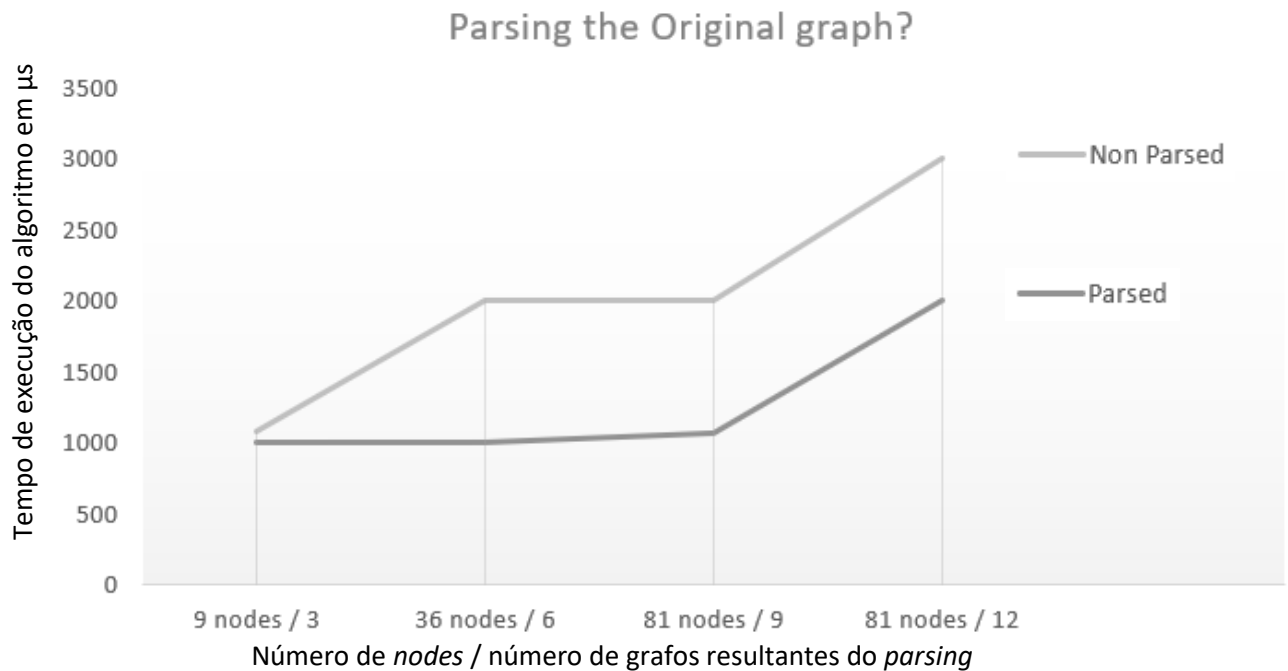


Gráfico 1

Numa tentativa de verificar a eficiência do *parsing*, foram efetuados vários testes, tendo como variável independente o número de *nodes* e o número de grafos resultantes do *parsing*.

Os testes consistiram no cálculo do tempo de execução do algoritmo de *Dijkstra*, com recurso a *threads*, comparando as eficiências num grafo com x *nodes* e em n grafos com x/n *nodes*.

Em praticamente todos os testes, houve uma discrepância significativa entre os tempos de execução – 50%, em média –, destacando-se sempre a rapidez do *ParsingTime*, com tempo de execução pertencente ao intervalo $[0, 100] \mu s$, em comparação com o *NotParsingTime*, cujos valores se mantiveram no intervalo dos $[2000, 3000] \mu s$.

Exemplo de código utilizado:

```
int NotParsingTime(){
    Graph<int> bigGraph;
    bigGraph = BigGraph();

    auto start = std::chrono::high_resolution_clock::now();
    std::thread t1 ([&](){
        bigGraph.dijkstraShortestPath( origin: 6);
    });
    std::thread t2 ([&](){
        bigGraph.dijkstraShortestPath( origin: 12);
    });
    std::thread t3 ([&](){
        bigGraph.dijkstraShortestPath( origin: 18);
    });
    std::thread t4 ([&](){
        bigGraph.dijkstraShortestPath( origin: 24);
    });
    std::thread t5 ([&](){
        bigGraph.dijkstraShortestPath( origin: 30);
    });
    std::thread t6 ([&](){
        bigGraph.dijkstraShortestPath( origin: 36);
    });
    std::thread t7 ([&](){
        bigGraph.dijkstraShortestPath( origin: 42);
    });
    std::thread t8 ([&](){
        bigGraph.dijkstraShortestPath( origin: 48);
    });
    std::thread t9 ([&](){
        bigGraph.dijkstraShortestPath( origin: 54);
    });
    std::thread t10 ([&](){
        bigGraph.dijkstraShortestPath( origin: 60);
    });
    std::thread t11 ([&](){
        bigGraph.dijkstraShortestPath( origin: 66);
    });
    std::thread t12 ([&](){
        bigGraph.dijkstraShortestPath( origin: 81);
    });
    t1.join(); t2.join(); t3.join();
    t4.join(); t5.join(); t6.join();
    t7.join(); t8.join(); t9.join();
    t10.join(); t11.join(); t12.join();

    auto finish = std::chrono::high_resolution_clock::now();
    auto elapsed = std::chrono::duration_cast<std::chrono::microseconds>(finish - start).count();
    std::cout << "Dijkstra processing average time (micro-seconds)=" << (elapsed) << std::endl;
}

int ParsingTime(){
    Graph<int> miniGraph1, miniGraph2, miniGraph3, miniGraph4, miniGraph5, miniGraph6, miniGraph7, miniGraph8, miniGraph9, miniGraph10, miniGraph11, miniGraph12;
    std::vector<Graph<int>> vect = getParsedGraphs();
    miniGraph1 = vect[0]; miniGraph2 = vect[1]; miniGraph3 = vect[2];
    miniGraph4 = vect[3]; miniGraph5 = vect[4]; miniGraph6 = vect[5];
    miniGraph7 = vect[6]; miniGraph8 = vect[7]; miniGraph9 = vect[8];
    miniGraph10 = vect[9]; miniGraph11 = vect[10]; miniGraph12 = vect[11];

    auto start = std::chrono::high_resolution_clock::now();
    std::thread t1 ([&](){
        miniGraph1.dijkstraShortestPath( origin: 6);
    });
    std::thread t2 ([&](){
        miniGraph2.dijkstraShortestPath( origin: 12);
    });
    std::thread t3 ([&](){
        miniGraph3.dijkstraShortestPath( origin: 18);
    });
    std::thread t4 ([&](){
        miniGraph4.dijkstraShortestPath( origin: 24);
    });
    std::thread t5 ([&](){
        miniGraph5.dijkstraShortestPath( origin: 30);
    });
    std::thread t6 ([&](){
        miniGraph6.dijkstraShortestPath( origin: 36);
    });
    std::thread t7 ([&](){
        miniGraph7.dijkstraShortestPath( origin: 42);
    });
    std::thread t8 ([&](){
        miniGraph8.dijkstraShortestPath( origin: 48);
    });
    std::thread t9 ([&](){
        miniGraph9.dijkstraShortestPath( origin: 54);
    });
    std::thread t10 ([&](){
        miniGraph10.dijkstraShortestPath( origin: 60);
    });
    std::thread t11 ([&](){
        miniGraph11.dijkstraShortestPath( origin: 66);
    });
    std::thread t12 ([&](){
        miniGraph12.dijkstraShortestPath( origin: 81);
    });
    t1.join(); t2.join(); t3.join();
    t4.join(); t5.join(); t6.join();
    t7.join(); t8.join(); t9.join();
    t10.join(); t11.join(); t12.join();
}
```

Figura 2

Obtenção dos componentes fortemente conexos de um grafo

Kosaraju vs Tarjan

O algoritmo de **Kosaraju** baseia-se em duas pesquisas em profundidade, sendo também necessário inverter todas as arestas do grafo. Cada uma destas operações possui uma complexidade temporal $O(|N| + |E|)$ e, dado que são realizadas sequencialmente, a complexidade temporal efetiva do algoritmo é, igualmente, $O(|N| + |E|)$, ou seja, executa em tempo linear de acordo com a soma do número de *nodes* e *edges* do grafo.

O algoritmo de **Tarjan** apresenta a mesma complexidade temporal linear do algoritmo de **Kosaraju**.

No entanto, considerando que o algoritmo de **Tarjan** apenas requer a realização de uma pesquisa em profundidade, em contraste com as duas pesquisas em profundidade requeridas pelo algoritmo de **Kosaraju**, afigura-se mais apelativo na resolução do problema da obtenção da conectividade do grafo.

Cálculo da rota de duração mínima entre dois pontos

Dijkstra vs A-Star

O algoritmo de **Dijkstra** requer a inicialização de todos os *nodes* com valores *INF* e *NULL* para os atributos *dist* e *path*, respetivamente, o que pode ser alcançado em tempo linear relativamente ao número de *nodes*, $O(|N|)$. Para além disso, exige que sejam percorridos todos os *nodes* e *edges*, $O(|N| + |E|)$, sendo que, a cada passo, podem ser realizadas operações de inserção, extração ou *decrease-key*, associadas à fila de prioridade - *MutablePriorityQueue*. Qualquer uma destas operações, por sua vez, apresenta uma complexidade temporal $O(\log |N|)$ uma vez que $|N|$ é o tamanho máximo da fila de prioridade. Assim, a complexidade temporal do algoritmo de **Dijkstra** é $O((|N| + |E|) * \log |N|)$.

O algoritmo de **A-Star**, por recorrer à heurística da distância euclidiana para determinar os *nodes* promissores, requer a exploração de um número significativamente menor de *nodes* e, portanto, é mais rápido quando comparado com o algoritmo de **Dijkstra**. No entanto, ao contrário deste, o **A-Star** não alcança garantidamente a solução ótima.

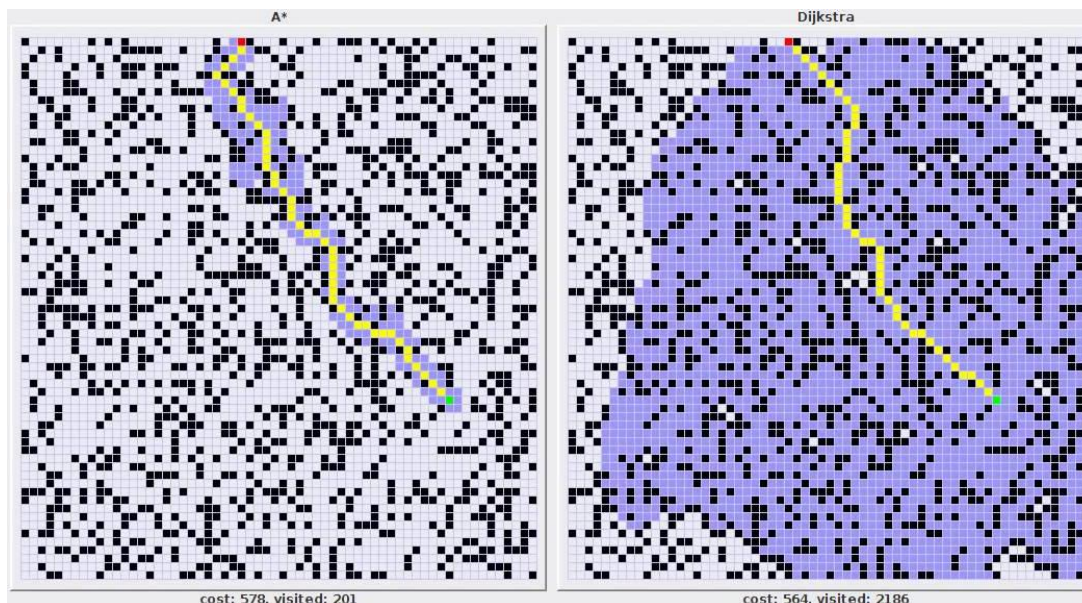


Figura 3

Analisando a imagem acima conclui-se que, de facto, o algoritmo **A-Star** visitou um número consideravelmente menor de *nodes* quando comparado com o **Dijkstra**. No entanto, o caminho mais curto encontrado pelo **A-Star** possui um custo de 578, maior do que o custo do caminho mais curto encontrado pelo **Dijkstra**, de 564.

Devido aos aspetos expostos acima, a decisão de implementar um e/ou outro será novamente ponderada na próxima fase do projeto.

Cálculo da rota de duração mínima passando por vários pontos

Solução exata - Força bruta vs Held-Karp

Dos algoritmos de solução exata aplicáveis ao *TSP* destacam-se o de **força bruta** e o de **Held-Karp**.

O primeiro apresenta uma complexidade temporal $O(n!)$ e o segundo $O(n^2 2^n)$. Constata-se, portanto, que possuem ambos uma complexidade temporal significativamente elevada, sendo que o algoritmo de **Held-Karp** chega, inclusive, a ser menos eficiente do que a solução de **força bruta**, quando $n \geq 8$, ou seja, quando o número de *nodes* do grafo é maior ou igual a 8.

Solução aproximada - Heurística “*Nearest Neighbour*”

É possível otimizar o algoritmo de força bruta através do recurso a uma heurística de decisão que, tal como o próprio nome indica, consiste em ir selecionando, sucessivamente, o *node* mais próximo ao da atual iteração, o “***Nearest Neighbour***”, para a construção da rota ótima que passa nos vários pontos.

Esta abordagem gananciosa alcança uma solução em tempo reduzido, no entanto, nem sempre alcança a solução ótima.

De notar que esta abordagem, originalmente, passa por selecionar um *node* aleatório como ponto de partida e que a solução obtida varia, por vezes significativamente, conforme o *node* selecionado. No entanto, no contexto do problema em mãos, a aleatoriedade desse passo seria eliminada, visto que o ponto de partida duma determinada rota é sempre o centro de armazenamento que envia o veículo.

Há que referir, por último, que existem muitas outras heurísticas que poderiam ser tidas em consideração durante o processo de escolha de qual o algoritmo a implementar. No entanto, esta revelou ser a mais interessante, com destaque para a facilidade da implementação.

Solução exata vs Solução aproximada

Dada a maior complexidade algorítmica das soluções exatas à disposição e considerando que, tal como fora explicado, a solução aproximada recorrendo à heurística se revela, de facto, apelativa, optar-se-á pela mesma na resolução do problema em causa.

Conclusão – 1ª entrega

O projeto faz uso de uma abordagem *bottom-up*, dado que tentamos ao máximo modularizar o problema por iterações, da mais simples à mais complexa (que corresponde ao problema na sua totalidade), sobretudo com o intuito de melhor encaminhar o fio de pensamento e detetar, o mais precocemente possível, eventuais problemas que pudessem surgir seguindo uma determinada abordagem.

Achamos por bem referir que optamos por não impor limite de capacidade das carrinhas uma vez que, em termos práticos, tal não acrescentaria nada de significativo ao programa³. No entanto, destacamos a importância da restrição temporal, que é prioritária na resolução do problema.

Sem mais a acrescentar, julgamos que a explicação de qualquer um dos passos envolventes na dita abordagem fora bem conseguida. No entanto, não podemos deixar de referir que não excluimos a possibilidade de ser necessário efetuar alterações, na segunda parte do projeto, ao planeamento original aqui exposto.

Detalhes do projeto – 1ª entrega

O projeto foi desenvolvido durante os dias 5 a 12 de abril, por ambos os elementos do grupo, cuja contribuição consideramos ter sido equitativa - 50%.

³ Suscetível a mudança, caso os docentes da unidade curricular acharem importante criar essa restrição.

Estruturas de dados utilizadas

Nesta secção serão mencionadas algumas das estruturas utilizadas na implementação do programa.

No entanto, chama-se à atenção para o facto de algumas estruturas apenas serem mencionadas na secção [Casos de utilização e principais funcionalidades](#)⁴, dado que é aí que a sua referência se revela efetivamente pertinente.

As estruturas de dados implementadas servem, essencialmente, para melhorar a eficiência temporal do programa.

→ ***Mutable priority queue***

Fornecida pela UC.

Utilizada nos algoritmos *Dijkstra* e *A-Star*.

→ ***Map***

Para um fácil e eficiente acesso aos *GVNodes* do *Graph Viewer*, é utilizado um *map* que guarda um *Node* e o seu correspondente *GVNode*. Desta forma, na criação dos *GVEdges*, a procura pelos seus correspondentes *GVNodes* é $O(1)$.

→ ***Unordered Map***

Utilizada para guardar os *nodes* do grafo e permitir que a sua procura, através do ID, seja efetuada em tempo constante, $O(1)$.

→ ***Priority queue***

Na procura pelos *Centers* (*Storage Center/Application Center*) mais próximos de um determinado *Center X*, recorre-se a esta estrutura de dados, redefinindo o operador $<$, de tal forma a que compare os *Centers* alocados pela sua distância a *X*. O acesso ao *Center* mais próximo de *X* consiste num `pop()` de $O(1)$.

⁴ ctrl-click para seguir o link.

Algoritmos implementados

Análise da conectividade – DFS

Pseudocódigo e análise da complexidade

```
class Graph {
    void DFSConnectivity(Node start) {
        if (not start->wasVisited())
            DFSRecursive(start);
    }

    DFSRecursive(Node start) {
        start.setVisited();
        for (each Edge e in start->getAdj()) {
            if (not e->getDest()->wasVisited())
                DFSRecursive(e->getDest());
        }
    }
}

analyseConnectivityMenu() {
    // ...

    // process graph correspondent to user's selected city
    graph = processGraph(mapFilename, false); O(|N|)

    vaccineRouter->setGraph(graph); O(1)

    // set the storage centers existent in the user's selected city
    vaccineRouter->setUpSCs(mapFilename); O(|SCs|)

    // save a copy of the graph before its connectivity analysis
    Graph copy = *graph; O(1)

    // analyse the connectivity for each node correspondent to a storage center
    for (each StorageCenter sc in vaccineRouter->getSCs()){ O(|SCs|)
        graph->DFSConnectivity(sc->getNode()); O(|N|)
    }

    // remove all nodes that weren't visisted, when starting from a storage center
    graph->removeUnvisitedNodes(); O((|N| x |E|) x |N| x |E|)

    // displays both graphs
    displayConnectivityAnalysis(copy, graph);
}
```

1ª iteração – Entregas de um Centro de Armazenamento a um Centro de Aplicação

Pseudocódigo e análise da complexidade

```
calculateRouteSingleSCSingleAC() {  
    // since there is a single application center, one can assume this  
    ac = selectedACs[0]; O(1)  
  
    // find nearest storage center relatively to the application center  
    nearestSC = findNearestSC(ac); O(|SCs|)  
  
    // get last available vehicle in nearest storage center's fleet  
    vehicle = sc->getAvailableVehicle(); O(1)  
  
    // effectively assign application center to the nearest storage center  
    nearestSC->assignAC(ac); O(1)  
  
    // get vector containing the nodes visited by the shortest path  
    path = AStar(graph, nearestSC.node, ac.node); O(|E| + |N| x log(|N|))  
  
    // false indicating that time constraint is not yet being taken into account  
    vehicle->setVehicleRoute(path, false); O(|path|)  
  
    // for each one of the vehicles in the fleet, display its path  
    displayVehiclesPath();  
}
```

Total = $O(|SCs| + (|E| + |N| \times \log(|N|)) + |path|)$

```
findNearestSC(ApplicationCenter ac) {  
    minDist = INF;  
    dist = 0.0;  
    nearest = NULL;  
    acCoords = ac->getNode()->getCoordinates(); O(1)  
  
    for (each StorageCenter sc in SCs) { O(|SCs|)  
        scCoords = sc->getNode()->getCoordinates(); O(1)  
        dist = scCoords.calculateEuclideanDistance(acCoords);  
        if (dist < minDist) {  
            minDist = dist;  
            nearest = sc;  
        }  
    }  
  
    return nearest;  
}
```

Total = $O(|SCs|)$

2ª iteração – Entrega de um Centro de Armazenamento a vários Centros de Aplicação

Pseudocódigo e análise da complexidade

```
calculateRouteSingleSCMultipleAC() {  
    // find nearest storage center relatively to all application centers  
    sc = findNearestSC(); O(|selectedACs| x |SCs|)  
  
    // assign all selected application centers to previously computed storage center  
    for (each ApplicationCenter ac in selectedACs) sc->assignAC(ac); O(|selectedACs|)  
  
    // get last available vehicle in fleet  
    vehicle = sc->getAvailableVehicle(); O(1)  
  
    // find next point to visit  
    nextPoint = sc->findNextNearestAC(sc); O(|selectedACs|) → note que, neste caso, |assignedACs| = |selectedACs|  
  
    // continue until all assigned application centers have been visited  
    while ((not sc->checkACsVisited()) and (nextPoint not NULL)) { O(|assignedACs| x (|E| + |N| x log(|N|))) → note que, no pior caso, o loop termina apenas quando todos os  
        // mark center as visited                                centros de aplicação em assignedACs são alocados na rota  
        startingPoint->setVisited(); O(1)  
  
        // get vector containing the nodes visited by the shortest path  
        path = AStar(graph, startingPoint->getNode(), nextPoint->getNode()); O(|E| + |N| x log(|N|))  
  
        // false indicating that time constraint is not yet being taken into account  
        vehicle->setVehicleRoute(path, false); O(|path|) → note que, esta complexidade deve-se ao facto de não ser tida em  
        conta a restrição temporal, pelo que, para dar set à rota do  
        startingPoint = nextPoint;                                veículo, apenas se tem que percorrer todos os pontos da rota  
  
        // nearest neighbor heuristic  
        // route's next point is the nearest one in terms of euclidean distance  
        nextPoint = sc->findNextNearestAC(startingPoint); O(|selectedACs|)  
    }  
  
    // for each one of the vehicles in the fleet, display its path  
    displayVehiclesPath();  
}
```

Total = $O(|\text{selectedACs}| \times |\text{SCs}| + |\text{selectedACs}| + |\text{selectedACs}| \times (|E| + |N| \times \log(|N|)) + |\text{path}| + |\text{selectedACs}|)$

```
findNearestSC() {  
    // initialize a priority queue to hold the nearest storage center relatively to each application center  
    // and the correspondent distance between both (ordered by this parameter)  
    queueSCs;  
  
    for (each ApplicationCenter ac in selectedACs) { O(|selectedACs|)  
        // find nearest storage center relatively to the application center of the current iteration  
        nearestSC = findNearestSC(ac); O(|SCs|)  
  
        // calculate distance between the storage center and the application center  
        dist = nearestSC->getNode()->calculateDist(ac->getNode()); O(1)  
  
        // insert in the queue a pair containing the nearest storage and the distance  
        queueSCs.insert((nearestSC, dist)); O(1)  
    }  
  
    // return the optimal storage center to serve all the application centers  
    return dists.top().nearestSC; O(1)  
}
```

Total = $O(|\text{selectedACs}| \times |\text{SCs}|)$

```

class StorageCenter {
    findNextNearestAC(Center startingPoint) const {
        // initialize a priority queue to hold the nearest application center relatively to the storage center
        // and the correspondent distance between both (ordered by this parameter)
        queueACs;

        for (each ApplicationCenter ac in assignedACs) { O(|assignedACs|)
            if ((not ac.visited) and (ac != startingPoint)) {
                // insert in the queue a pair containing the application center
                // and the distance relatively to the storage center
                queueACs.insert((ac, dist)); O(1)
            }
        }

        // return the nearest application center
        return queueACs.top().ac; O(1)
    }
}

```

Total = $O(|assignedACs|)$

3ª iteração - Entregas de um Centro de Armazenamento a vários Centros de Aplicação com restrição temporal

Pseudocódigo e análise da complexidade

```
calculateRouteSingleSCMultipleACWithTW() {  
    // find nearest storage center relatively to all application centers  
    sc = findNearestSC(); O(|selectedACs| x |SCs|)  
  
    // assign all selected application centers to the previously calculated storage center  
    for (each ApplicationCenter ac in selectedACs) sc->assignAC(ac); O(|selectedACs|)  
  
    // find next point to visit  
    nextPoint = sc->findNextNearestAC(sc); O(|assignedACs|)  
  
    // continue until all assigned application centers have been visited  
    while ((not sc->checkACsVisited()) and (nextPoint not NULL)) { O(|assignedACs| x (|E| + |N| x log(|N|))) → note que, no pior caso, o loop termina apenas quando todos os  
        // get last available vehicle in fleet                                centros de aplicação em assignedACs são alocados na rota  
        vehicle = sc->getAvailableVehicle(); O(1)  
  
        // mark center as visited  
        startingPoint->setVisited(); O(1)  
  
        // get vector containing the nodes visited by the shortest path  
        path = dijkstra(graph, startingPoint->getNode(), nextPoint->getNode()); O(|E| + |N| x log(|N|))  
  
        // try to set the optimal sub-route, this time considering the time constraint  
        if (vehicle->setVehicleRoute(path, true)) { O(|path|)  
            startingPoint = nextPoint;  
  
            // nearest neighbor heuristic - route's next point is the nearest one in terms of euclidean distance  
            nextPoint = sc->findNextNearestAC(startingPoint); O(|assignedACs|)  
        }  
        // if the optimal sub-route could not fit in the route already assigned to the vehicle  
        // add a new vehicle to the storage center's fleet and try again to assign the sub-route  
        else {  
            sc->addVehicle(); O(1)  
        }  
    }  
}  
  
// for each one of the vehicles in the fleet, display its path  
displayVehiclesPath();  
}
```

Total = $O(|\text{selectedACs}| \times (|E| + |N| \times \log(|N|)))$

4ª iteração - Entregas de vários Centros de Armazenamento a vários Centros de Aplicação com restrição temporal

Pseudocódigo e análise da complexidade

```
calculateRouteMultipleSCMultipleACWithTW() {
    // assign application centers to optimal storage centers
    // in terms of euclidean distance between each other
    for (each ApplicationCenter ac in selectedACs) { O(|selectedACs|)
        // find nearest storage center relatively to the application center of the current iteration
        sc = findNearestSC(ac); O(|SCs|)

        // assign the application center to the previously calculated storage center
        sc->assignAC(ac); O(1)
    }

    // iterate over storage centers to calculate its optimal primary route
    for (each StorageCenter sc in SCs) { O(|SCs|)
        if (sc->getAssignedAC() is empty) continue;

        // calculate storage center's route considering its pre-assigned application centers
        calculateSCRoute(sc); O(|assignedACs| x (|E| + |N| x log(|N|)))
    }

    // delete the already handled application centers from the vaccine router's system
    // however, note that the application centers remain associated to the storage center
    deleteDispatchedACs(); O(|selectedACs|)

    // handle the application centers that could not fit in its optimally assigned storage center route
    handleACsNotVisited(); O(|selectedACs| x |viableNextSCOptions| x (|assignedACs| x (|E| + |N| x log(|N|))))

    // for each one of the vehicles in the fleet, display its path
    displayVehiclesPath();
}
```

Total = $O(|\text{selectedACs}| \times |\text{viableNextSCOptions}| \times (|\text{assignedACs}| \times (|E| + |N| \times \log(|N|))))$

```
calculateSCRoute(StorageCenter sc) {
    startingPoint = sc;

    // find next point to visit
    nextPoint = sc->findNextNearestAC(sc); O(|assignedACs|)

    // get last available vehicle in nearest storage center's fleet
    vehicle = sc->getAvailableVehicle(); O(1)

    // initialize counter of the assigned application centers effectively visited
    visited = 0;

    while ((not sc->isOptimalState()) and (nextPoint not NULL)) { O(|assignedACs| x (|E| + |N| x log(|N|))) →
        startingPoint->setVisited(); O(1)

        // get vector containing the nodes visited by the shortest path
        path = dijkstra(graph, startingPoint->getNode(), nextPoint->getNode()); O(|E| + |N| x log(|N|))

        // try to set the vehicles path, considering the time constraint
        if (vehicle->setVehicleRoute(path, true)) { O(|path|)
            // update startingPoint and nextPoint to next iteration
            startingPoint = nextPoint;
            nextPoint = sc->findNextNearestAC(startingPoint); O(|assignedACs|)

            // increment the number of assigned application centers effectively visited
            visited++;
        } else {
            // check if some application center could even fit in route
            // it could have been the case where the storage center could not fulfill none of its assigned
            if (not vehicle->hasEmptyPath()) O(1)
                // if it has managed to fulfill some of its assigned
                // but fulfilling this one would require dispatching another vehicle
                // mark as in optimal state
                sc->setOptimalState(); O(1)
        }
    }

    // true if visited all assigned applications centers, false otherwise
    return (visited == sc->getAssignedAC().size()); O(1)
}
```

note que, no pior caso, o loop termina apenas quando todos os centros de aplicação em assignedACs são alocados na rota

Total = $O(|\text{assignedACs}| \times (|E| + |N| \times \log(|N|)))$

```

deleteDispatchedACs() {
    it = selectedACs.begin;
    while (it is not selectedACs.end) { O(|selectedACs|)
        if ((*it)->isVisited())
            it = selectedACs.erase(it); O(1)
        else
            it++;
    }
}

```

Total = $O(|\text{selectedACs}|)$

```

handleACsNotVisited() {
    allocatedAC = false;
    viableSC = NULL;

    // by this function call, the only application centers that remain in selectedACs are the unhandled ones
    for (each ApplicationCenter ac in selectedACs) { O(|selectedACs|)
        // initialize a priority queue to hold the viable storage centers to allocate the application center
        // and the correspondent distance between both (ordered by this parameter)
        viableNextSCOptions;

        for (each StorageCenter sc in SCs) { O(|SCs|)
            // a storage center being in an optimal state means that the last dispatched vehicle is complete and so,
            // allocating another application center in its route, would require another vehicle being dispatched

            // if the storage center of the current iteration is already in an optimal state,
            // trying to allocate the application center to it would be counterproductive once that, by doing so,
            // it would have been better to allocate the application center to its first/best storage center alternative
            if (not sc->isOptimalState()) { O(1)
                dist = sc->getNode()->calculateDist(ac->getNode()); O(1)
                viableNextSCOptions.insert((sc, dist)); O(1)
            }
        }

        do {
            // get best viable option (meaning that, considering the remaining alternatives, this is the closest)
            viableSC = viableNextSCOptions.top().sc; O(1)

            // save viable storage center original path in case it needs to be restored later
            originalPath = viableSC->getAvailableVehicle()->getPath(); O(1)

            viableSC->assignAC(ac); O(1)

            // computes the route, now with the newly assigned application center, to check if it fits
            // allocatedAC holds the success of the operation
            // true if the application center could fit in route, false otherwise
            allocatedAC = calculateSCRoute(viableSC); O(|assignedACs| x (|E| + |N| x log(|N|)))

            if (allocatedAC) break;

            // in case the application center could not be allocated
            // one needs to reset storage center's original path and undo the assignment
            viableSC->reset(); O(1)
            viableSC->resetPath(originalPath); O(1)

            // removes storage center from the queue since it has shown not to be a viable option
            viableNextSCOptions.pop(); O(1)
        } while (viableNextSCOptions is not empty); O(|viableNextSCOptions|)

        // has not been allocated yet but there are no more storage centers to try
        if (viableNextSCOptions is not empty and not allocatedAC) {
            // viableSC is now the first/best storage center to which the application center was assigned
            viableSC = findNearestSC(ac); O(|SCs|)
            viableSC->addVehicle(); O(1)
            calculateSCRoute(viableSC); O(|assignedACs| x (|E| + |N| x log(|N|)))
        }

        allocatedAC = false;
    }
}

```

Total = $O(|\text{selectedACs}| \times |\text{viableNextSCOptions}| \times (|\text{assignedACs}| \times (|E| + |N| \times \log(|N|))))$

Casos de utilização e principais funcionalidades

A aplicação implementada comunica com o utilizador através de uma interface simples de texto. Para tal, implementou-se um sistema de menus que permitem ao utilizador navegar por entre as várias opções disponibilizadas.

Seguem abaixo as funcionalidades implementadas, acompanhadas de uma captura de ecrã significativa.

1. Menu Inicial

```
----- INITIAL MENU -----  
1. Run Program  
2. Analyse Connectivity  
3. Modify Data  
4. Exit  
  
Please select your option:
```

2. Modificação dos dados

2.1 Seleção do mapa para a modificação dos dados

| | |
|--|---|
| <pre>----- MODIFY DATA MENU - MAP SELECTION ----- 1. porto 2. penafiel 3. espinho 4. Go Back Please select your option:</pre> | <pre>----- MODIFY DATA MENU ----- 1. Add Application Center 2. Add Storage Center 3. Remove Application Center 4. Remove Storage Center 5. Go Back Please select your option: </pre> |
|--|---|

No menu da seleção do mapa onde incidirá a modificação dos dados, destaca-se o recurso a um *map<int, string>* para mapear o índice da opção para o mapa que este representa.

Após selecionar em qual mapa incidirão as modificações, o utilizador pode selecionar uma das quatro opções apresentadas – adicionar/remover Centros de Aplicação/Armazenamento.

2.2 Adição de um Centro de Aplicação/Armazenamento

----- ADD APPLICATION CENTER MENU -----

Introduce the ID of the node about to become an Application Center:

----- ADD STORAGE CENTER MENU -----

Introduce the ID of the node about to become a Storage Center:

Para adicionar um Centro de Aplicação/Armazenamento basta que o utilizador introduza o número correspondente ao ID do *node* que pretende que passe a constituir um Centro. Optou-se por esta abordagem dado que, considerando a quantidade significativa de *nodes* de cada grafo, seria impensável mostrar todas as opções no ecrã.

2.3 Remoção de um Centro de Aplicação/Armazenamento

----- REMOVE APPLICATION CENTER MENU -----

1. AC_A
2. AC_B
3. AC_C
4. AC_D
5. AC_E
6. AC_F
7. AC_G
8. AC_H
9. AC_I
10. AC_J
11. AC_K
12. AC_L
13. AC_M
14. AC_N
15. AC_O
16. AC_P
17. AC_Q
18. AC_R
19. Go Back

Select one of the following Application Centers to remove:

----- REMOVE STORAGE CENTER MENU -----

1. SC_A
2. SC_B
3. SC_C
4. SC_D
5. SC_E
6. SC_F
7. SC_G
8. SC_H
9. SC_I
10. SC_J
11. Go Back

Select one of the following Storage Centers to remove:

Ao contrário do que acontece para a funcionalidade acima, ao remover um Centro de Aplicação/Armazenamento, o utilizador tem acesso a todas as suas opções.

Cada mapa disponível tem a si associados dois ficheiros, cujo propósito é meramente o de guardar os dados relativos aos Centros de Aplicação e Armazenamento – mais concretamente, o ID do *node* do grafo e o nome de cada Centro.

Assim, quando o utilizador seleciona o mapa que pretende manipular, o programa acede ao ficheiro correspondente, quer se trate da remoção de um Centro de Aplicação ou de um Centro de Armazenamento, e mostra no ecrã quais são removíveis.

Destaca-se também, aqui, o recurso a um *map<int, pair<int, string>>* para mapear o índice da opção para o *pair<int, string>* que guarda o ID e o nome, tal como suprarreferido, do Centro a ser removido.

Após a seleção por parte do utilizador, o programa apenas tem que eliminar do *map* a entrada correspondente à opção selecionada e efetuar o *output* das opções restantes para o ficheiro respetivo, que é aberto em modo *truncate*, para garantir que não há repetição de dados.

3. Execução do programa

```
----- RUN PROGRAM MENU -----  
1. Select Map  
2. Go Back  
  
Please select your option:|
```

3.1 Seleção do mapa para execução do programa

```
----- AVAILABLE CITIES -----  
1. porto  
2. penafiel  
3. espinho  
4. Go Back  
  
Please select your option:|
```

Neste menu, a seleção do mapa dá-se tal como no menu de seleção do mapa para a modificação dos dados, exposto em 2.1.

No entanto, neste são efetivamente invocados os métodos responsáveis por construir o grafo relativo ao mapa selecionado, que servirá de base à execução dos algoritmos propriamente ditos.

3.2 Seleção dos Centros de Aplicação

```
----- SINGLE OR MULTIPLE APPLICATION CENTER SELECTION -----  
1. Single Application Center  
2. Multiple Application Centers  
3. Go Back  
  
Please select your option:|
```

Em primeiro lugar, o utilizador seleciona se pretende efetuar encomendas para um único Centro de Aplicação – *Single Application Center*, ou para vários – *Multiple Application Center*.

----- SINGLE APPLICATION CENTER -----

Please note that by selecting a single Application Center, you will not be given the chance of taking into account the vaccine's lifetime.

1. AC_A
2. AC_B
3. AC_C
4. AC_D
5. AC_E
6. AC_F
7. AC_G
8. AC_H
9. AC_I
10. AC_J
11. AC_K
12. AC_L
13. AC_M
14. AC_N
15. AC_O
16. AC_P
17. AC_Q
18. AC_R

Please select the Application Center from the above list [usage: >2]:|

----- MULTIPLE APPLICATION CENTERS -----

After the selection, you will be redirected to a menu where you will be given the chance of taking into account the vaccine's lifetime.

1. AC_A
2. AC_B
3. AC_C
4. AC_D
5. AC_E
6. AC_F
7. AC_G
8. AC_H
9. AC_I
10. AC_J
11. AC_K
12. AC_L
13. AC_M
14. AC_N
15. AC_O
16. AC_P
17. AC_Q
18. AC_R

Please select the Application Center from the above list [usage: >2 3 4]:

De seguida, é-lhe solicitado que selecione para qual/quais Centro(s) de Aplicação pretende efetuar uma encomenda de vacinas.

3.3 Encomenda de vacinas para os Centro de Aplicação selecionados

```
Please select the Application Center from the above list [usage: >2 3 4]:6 7 15
```

```
----- ORDER VACCINES MENU -----
```

```
Vaccine's order for AC_F:78
```

```
Vaccine's order for AC_G:125
```

```
Vaccine's order for AC_O:200
```

O próximo passo passa por, tal como seria de esperar, indicar, para cada um dos Centros de Aplicação selecionados no passo anterior, a quantidade de vacinas que pretende encomendar.

3.4 Seleção de detalhes adicionais

```
Do you want to consider the vaccine's lifetime? [y/n]y
```

```
Enter the vaccine's lifetime: [hh:mm:ss]04:30:50
```

```
Do you want to force the deliveries to be made by a single Storage Center? [y/n]n
```

Por último, resta apenas ao utilizador selecionar se pretende ter em conta o prazo de validade das vacinas e, em caso afirmativo, qual o respetivo prazo, e se pretende forçar a que as encomendas sejam todas entregues pelo mesmo Centro de Armazenamento.

3.5 Cálculo da(s) rota(s)

O cálculo da(s) rota(s) é feito com base nas seleções efetuadas durante os passos acima e invocando os respetivos métodos da classe *VaccineRouter*.

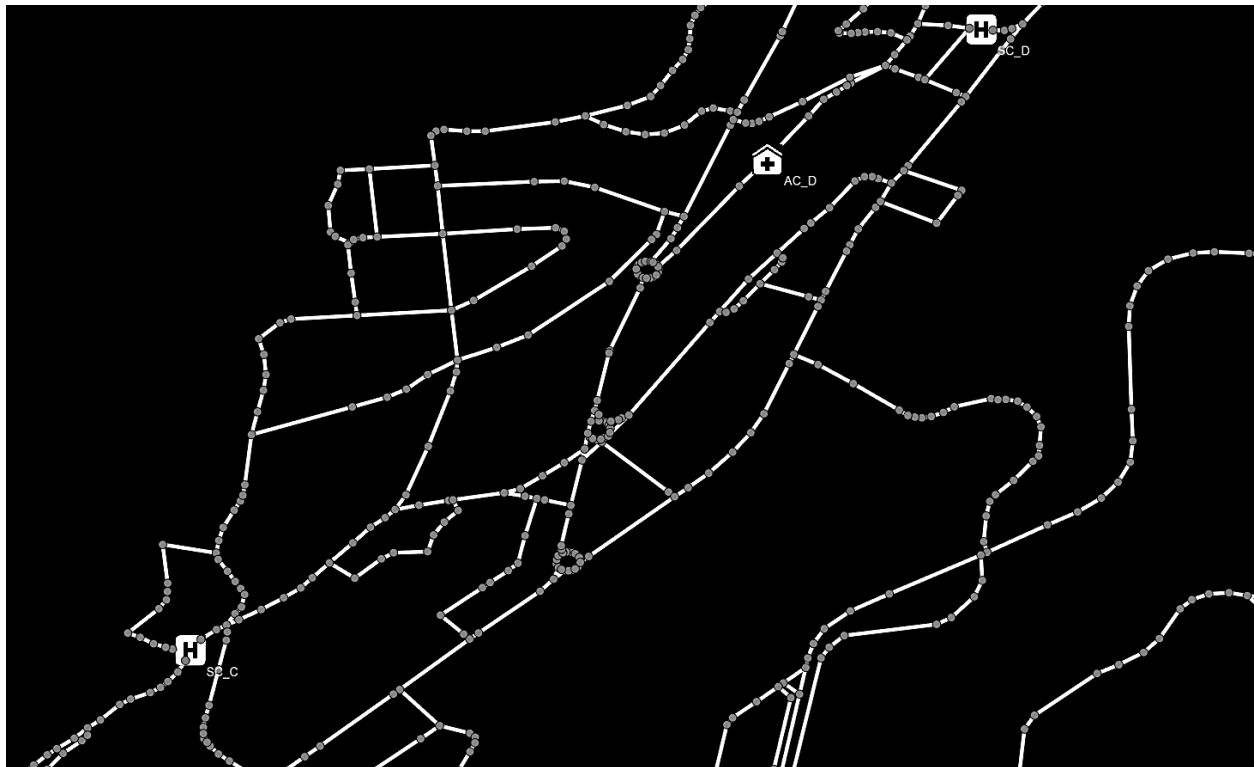
3.6 Display da(s) rota(s) calculadas

Seguem abaixo capturas de ecrã das rotas calculadas para os vários cenários de utilização.

De notar os seguintes aspetos:

- Centros de Armazenamento (*StorageCenter*) estão representados, geralmente, por SC_*;
- Centros de Aplicação (*ApplicationCenter*) estão representados, geralmente, por AC_*;
- Fazem-se ambos acompanhar de um ícone representativo, tal como é possível observar na imagem abaixo, do mapa de Penafiel.

Estes pormenores contribuem também para facilitar e tornar mais intuitiva a interação do utilizador com o programa.



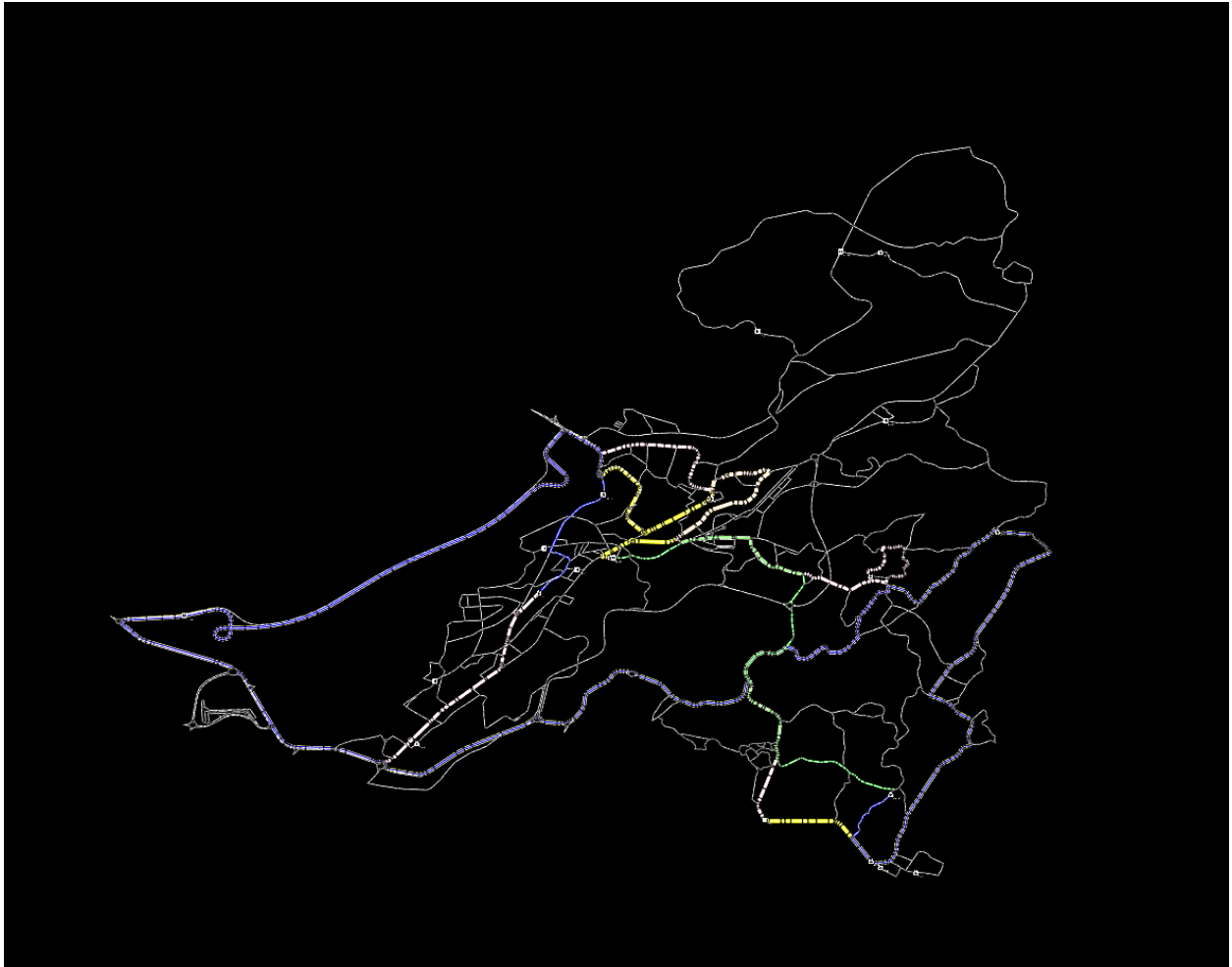
3.6.1 *Single Storage Center → Single Application Center*



3.6.2 *Single Storage Center → Multiple Application Center*



3.6.3 Single Storage Center → Multiple Application Center (w/ time constraint)

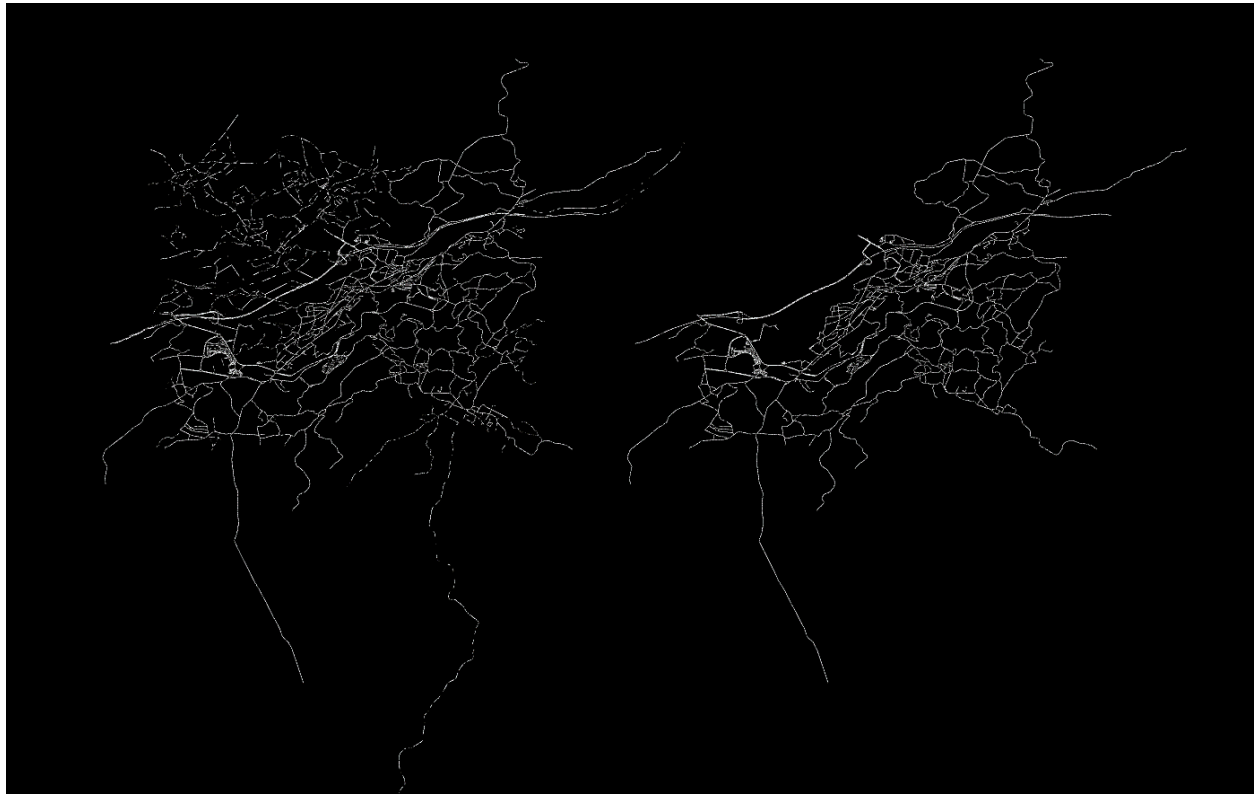


3.6.4 Multiple Storage Center → Multiple Application Center (w/ time constraint)



4. Menu da análise da conectividade

Segue abaixo uma captura de ecrã do antes e depois de ter sido aplicada a análise da conectividade ao grafo do mapa de Penafiel, com remoção dos *nodes* não visitados.



Conclusão – 2ª entrega

Consideramos que o problema em causa, do planeamento da rota tomada por veículos de distribuição de vacinas, partindo de Centros de Armazenamento, até Centros de Aplicação, foi analisado com sucesso, tendo sido, tal como proposto aquando da 1ª entrega do projeto, decomposto em quatro iterações, que visaram incrementar progressivamente o nível de dificuldade.

Houve, de facto, um esforço, por parte de ambos os elementos do grupo, em, não só, encontrar e implementar a melhor solução para o problema em mãos, como em tentar aplicar vários conceitos relevantes na área da conceção de algoritmos, lecionados nas aulas da unidade curricular ao longo do semestre.

Ao longo do desenvolvimento do projeto, deparamo-nos com algumas dificuldades, nomeadamente:

- na implementação de *threads*, na 4ª iteração, como havia sido planeado na 1ª parte do projeto. Aparecia o erro “*terminate called without an active exception*”, devido ao qual, após uma pesquisa intensiva, nos apercebemos que não poderiam ser usadas *threads* no contexto em que as estávamos a tentar implementar;

- na utilização da ferramenta *GraphViewer*, disponibilizada pela unidade curricular, que, embora reconheçamos estar devidamente documentada, achamos que deveria ter sido disponibilizada mais do que uma aula teórico-prática para permitir aos alunos adquirirem um domínio mais profundo da mesma;

- no que diz respeito à sobrecarga de projetos, sobretudo considerando que o grupo era constituído por menos um elemento.

Outro aspeto que consideramos importante referir é o de que, nos cenários de utilização que envolvem restrição temporal, optou-se por recorrer ao algoritmo *Dijkstra*, visto que este alcança, garantidamente, a solução ótima, ao contrário do algoritmo *A-Star* que, por depender da heurística da distância euclidiana para decidir qual o próximo *node* a visitar, nem sempre a alcança. Ora, nos cenários em que é tida em conta a dita *vaccineLifetime*, é imperativo que se consiga alcançar, de facto, o caminho mais curto, uma vez que tal é decisivo na gestão dos veículos que partem dos Centros de Armazenamento.

Para além disso, no que diz respeito à análise da conectividade, e indo de encontro à sobrecarga de projetos já referida acima, acabamos por implementar tal funcionalidade recorrendo ao algoritmo de pesquisa em profundidade, lecionado nas aulas teóricas, para cada Centro de Armazenamento, ao invés do algoritmo *Tarjan*, que havia sido planeado na 1ª entrega.

Conseguimos ir de encontro à função objetivo definida, o que se torna evidente, sobretudo, na 4ª iteração, na qual priorizamos a minimização do número de veículos e tentamos sempre preencher o máximo de vagas possíveis em veículos disponíveis de outros Centros de Armazenamento.

Dado o carácter incremental, em termos de nível de dificuldade, da resolução do problema, consideramos que o projeto foi útil, no sentido em que nos permitiu adquirir a capacidade de pensar modularmente. Para além disso, permitiu aplicar a teoria aprendida ao longo do semestre a um cenário de aplicação real.

Sem mais considerações, gostaríamos de concluir este relatório dizendo que, não obstante a carga de trabalho inerente à realização do projeto, a sua realização veio mostrar-se surpreendentemente aprazível.

Detalhes do projeto – 2ª entrega

O projeto foi desenvolvido durante os dias 17 a 24 de abril, por ambos os elementos do grupo, cuja contribuição consideramos ter sido, à semelhança da entrega anterior, equitativa - 50%.

Referências

- Computerphile. (2017, fevereiro 15). *A* (A Star) Search Algorithm - Computerphile*. Retrieved from Youtube: <https://www.youtube.com/watch?v=ySN5Wnu88nE&t=5s>
- Computerphile. (2017, Janeiro 4). *Dijkstra's Algorithm - Computerphile*. Retrieved from Youtube: <https://www.youtube.com/watch?v=GazC3A4OQTE>
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. Cambridge, Massachusetts: MIT Press. Retrieved from Introduction to Algorithms: https://edutechlearners.com/download/Introduction_to_algorithms-3rd%20Edition.pdf
- Dantzig, B. G., & Ramser, H. J. (2008, setembro 28). *The Truck Dispatching Problem*. Retrieved from andresjaquep: <https://andresjaquep.files.wordpress.com/2008/10/2627477-clasico-dantzig.pdf>
- Eddie Woo. (2015, setembro 9). *The Travelling Salesman (1 of 3: Understanding the Problem)*. Retrieved from Youtube: <https://www.youtube.com/watch?v=CPetTODX-FA>
- Eddie Woo. (2015, setembro 10). *The Travelling Salesman (2 of 3: Nearest Neighbour & SFCs)*. Retrieved from Youtube: https://www.youtube.com/watch?v=R_lfyticWKQ
- GeeksforGeeks. (2020, outubro 29). *Comparision between Tarjan's and Kosaraju's Algorithm*. Retrieved from GeeksforGeeks: <https://www.geeksforgeeks.org/comparision-between-tarjans-and-kosarajus-algorithm/>
- GeeksforGeeks. (2020, setembro 2). *Tarjan's Algorithm to find Strongly Connected Components*. Retrieved from GeeksforGeeks: <https://www.geeksforgeeks.org/tarjan-algorithm-find-strongly-connected-components/>
- GeeksforGeeks. (2021, Março 31). *Dijkstra's shortest path algorithm | Greedy Algo-7*. Retrieved from GeeksforGeeks: <https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/>
- Klarreich, E. (2020, outubro 8). *Computer Scientists Break Traveling Salesperson Record*. Retrieved from Quantamagazine: <https://www.quantamagazine.org/computer-scientists-break-traveling-salesperson-record-20201008/>
- Ma, S. (2020, janeiro 2). *Understanding The Travelling Salesman Problem (TSP)*. Retrieved from Routific: <https://blog.routific.com/travelling-salesman-problem>
- Wikipedia. (2020, novembro 16). *Tarjan's strongly connected components algorithm*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm
- Wikipedia. (2021, fevereiro 8). *A* search algorithm*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/A*_search_algorithm
- Wikipedia. (2021, Março 29). *Dijkstra's algorithm*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

- Wikipedia. (2021, fevereiro 22). *Held–Karp algorithm*. Retrieved from Wikipedia:
https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm
- Wikipedia. (2021, abril 8). *Kosaraju's algorithm*. Retrieved from Wikipedia:
https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm
- Wikipedia. (2021, abril 5). *Travelling salesman problem*. Retrieved from Wikipedia:
https://en.wikipedia.org/wiki/Travelling_salesman_problem
- Wikipedia. (2021, janeiro 21). *Vehicle routing problem*. Retrieved from Wikipedia:
https://en.wikipedia.org/wiki/Vehicle_routing_problem#Exact_solution_methods
- WilliamFiset. (2018, janeiro 2). *Travelling Salesman Problem | Dynamic Programming | Graph Theory*. Retrieved from Youtube: <https://www.youtube.com/watch?v=cY4HiiFHO1o>
- Wu, Q., Qin1, G., & Li2, H. (2015). *An Improved Dijkstra's algorithm application*. Changchun: MetalJournal.