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

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 ↔ 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

$\mathbf{R_i}$ – raio de ação de cada centro de armazenamento

$\mathbf{V_i}$ – 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

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

$\mathbf{S_i}$ – centro de armazenamento que enviou a viatura responsável pela distribuição

$\mathbf{T_i}$ – tempo total de cada percurso P_i

$\mathbf{N_i}$ – número de vacinas a ser entregue em cada centro de aplicação

$\mathbf{V_s}$ – 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

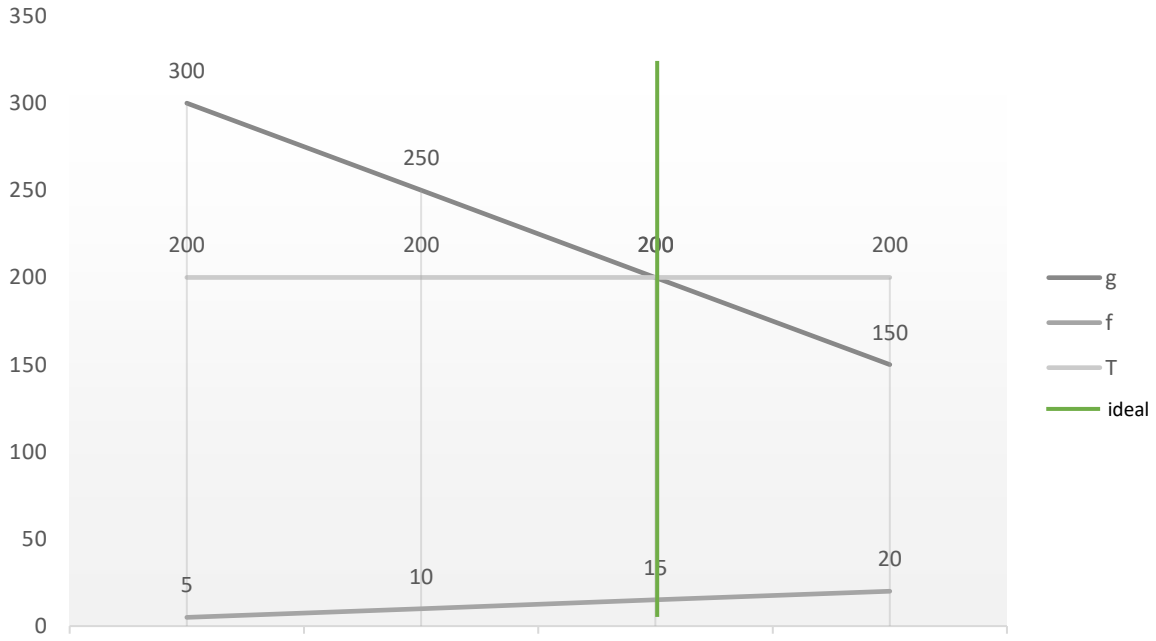
$$f = |Vs|$$

Função 2

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

sendo e ∈ 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 ← NodeD ←NodeA
Fornecida pela unidade curricular, com adaptações requeridas pelos algoritmos.

Edge
Fornecida pela unidade curricular.

Source & Header files

GraphProcesser
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.

MutablePrioryQueue.h
Fornecida pela unidade curricular.

Diagrama UML

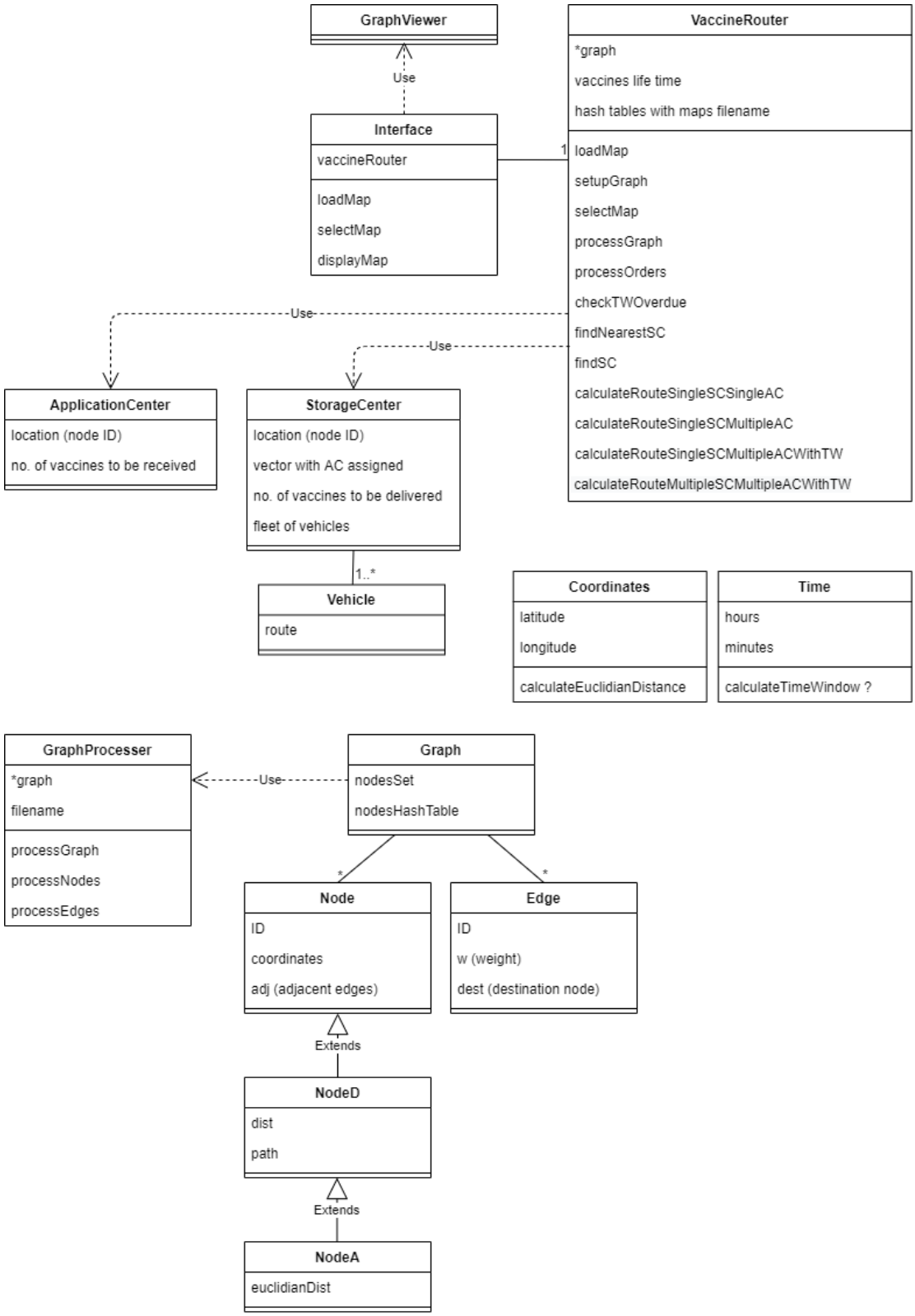


Figura 1

Disclaimer: O diagrama é apenas um *template*. Eventuais mudanças poderão ocorrer na implementação do código.

Solução Conceptual

Pré-processamento

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

Construção/Inicialização do grafo

A informação contida no ficheiro de texto correspondente à localidade selecionada 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 constrições 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 subgrafos[2]. 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 constrições 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 respetivas *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-me 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 pesos 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 constrições 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 constrições 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 constrição 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.

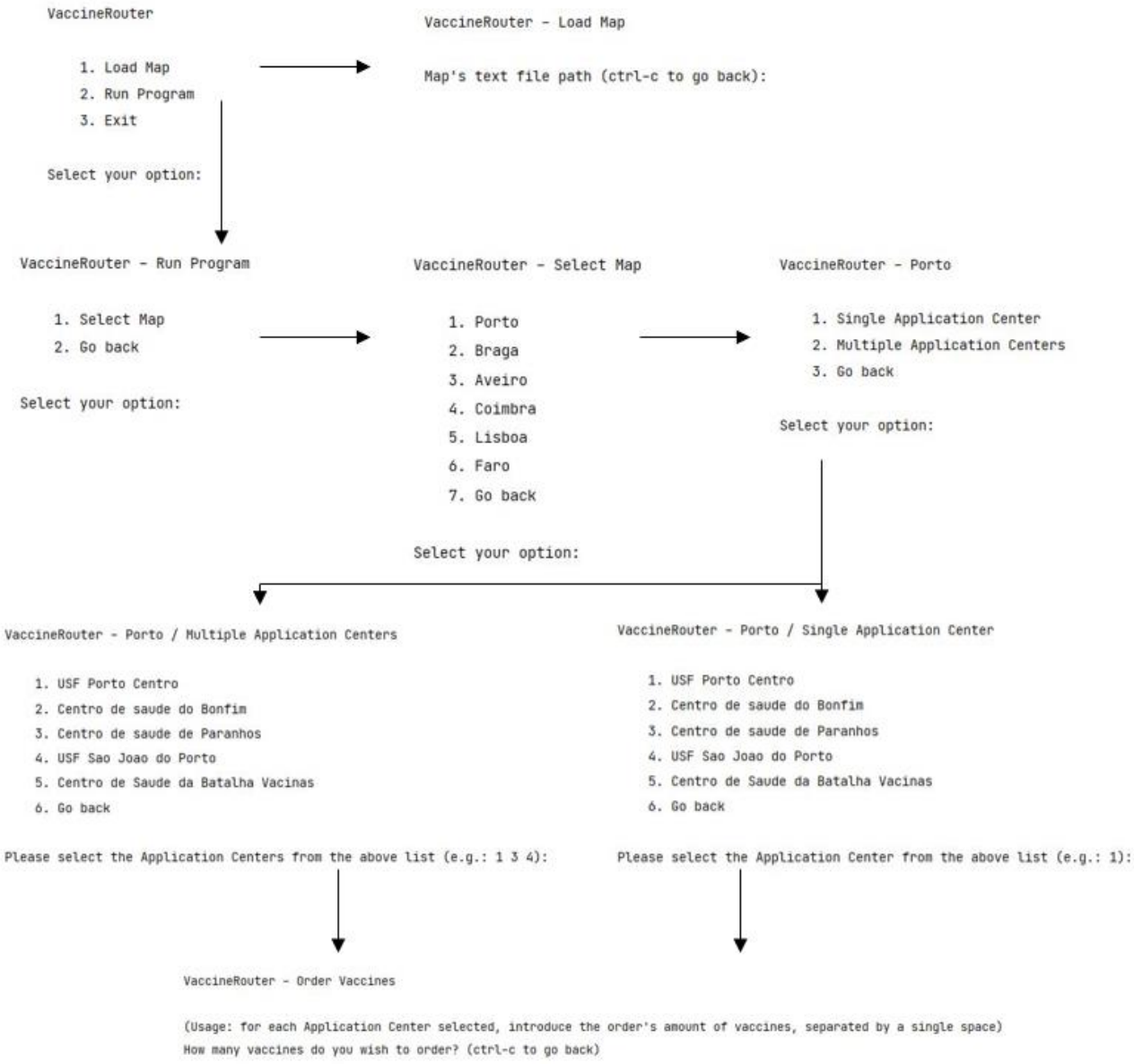
A primeira etapa passa por verificar, até que não existam mais conflitos, para cada centro de aplicação, se este faz parte do grafo de outro centro de armazenamento. No caso de existir mais do que um centro de armazenamento candidato a ficar a cargo da entrega, tentar-se-á optar pelo mais próximo ao 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


VaccineRouter
1. Load Map
2. Run Program
3. Exit
Select your option:
VaccineRouter - Load Map
Map's text file path (ctrl-c to go back):
VaccineRouter - Run Program
1. Select Map
2. Go back
Select your option:
VaccineRouter - Select Map
1. Porto
2. Braga
3. Aveiro
4. Coimbra
5. Lisboa
6. Faro
7. Go back
Select your option:
VaccineRouter - Porto
1. Single Application Center
2. Multiple Application Centers
3. Go back
Select your option:
VaccineRouter - Porto / Multiple Application Centers
1. USF Porto Centro
2. Centro de saude do Bonfim
3. Centro de saude de Paranhos
4. USF Sao Joao do Porto
5. Centro de Saude da Batalha Vacinas
6. Go back
Please select the Application Centers from the above list (e.g.: 1 3 4):
VaccineRouter - Porto / Single Application Center
1. USF Porto Centro
2. Centro de saude do Bonfim
3. Centro de saude de Paranhos
4. USF Sao Joao do Porto
5. Centro de Saude da Batalha Vacinas
6. Go back
Please select the Application Center from the above list (e.g.: 1):
VaccineRouter - Order Vaccines
(Usage: for each Application Center selected, introduce the order's amount of vaccines, separated by a single space)
How many vaccines do you wish to order? (ctrl-c to go back)

Análise e Resultados

Divisão do grafo em subgrafos

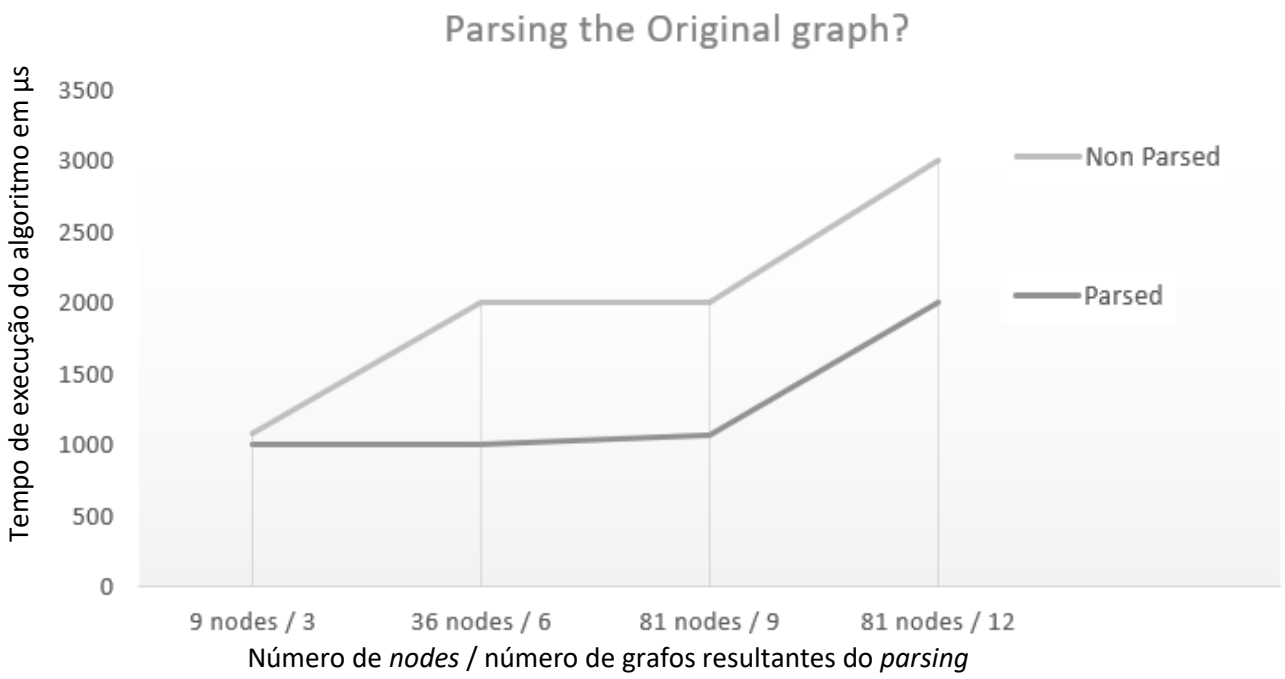


Gráfico 1

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] µs, em comparação com o *NotParsingTime*, cujos valores se mantiveram no intervalo dos [2000, 3000] µs.

Exemplo de código utilizado:

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

    auto start = std::chrono::high_resolution_clock::no
    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,mi
    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 - *MutablePrioryQueue*. 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

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 programa[3]. 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 excluímos a possibilidade de ser necessário efetuar alterações, na segunda parte do projeto, ao planeamento original aqui exposto.

Detalhes do projeto

O projeto foi desenvolvido, durante as semanas de 5 e 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.

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_IfyticWKQ

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.