



MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA E
COMPUTAÇÃO

CONCEPÇÃO E ANÁLISE DE ALGORITMOS

VaccineRouter: transporte de vacinas entre centros de aplicação

ENTREGA 2

2MIEIC05_G2

Diogo Costa up201906731@fe.up.pt
Francisco Colino up201905405@fe.up.pt
Rui Alves up201905853@fe.up.pt

21 maio 2021

Conteúdo

Lista de Figuras	3
1 Descrição do problema	4
1.1 Primeira Fase	4
1.2 Segunda Fase	5
1.3 Terceira Fase	5
2 Formalização do problema	6
2.1 Dados de entrada	6
2.1.1 Restrições dos dados de entrada	7
2.2 Dados de saída	7
2.2.1 Restrições dos dados de saída	8
2.3 Funções Objetivo	8
3 Perspetivas de solução	9
3.1 Acessibilidade	9
3.1.1 <i>Depth-First-Search algorithm</i>	9
3.1.2 <i>Breadth-First-Search algorithm</i>	10
3.2 Conectividade forte	11
3.2.1 <i>Kosaraju algorithm</i>	11
3.2.2 <i>Tarjan's algorithm</i>	12
3.3 Caminho mais curto	13
3.3.1 <i>Dijkstra algorithm</i>	14
3.3.2 <i>A* algorithm</i>	14
3.3.3 <i>Floyd-Warshall algorithm</i>	15
3.4 <i>Travelling Salesman Problem</i>	16
3.4.1 <i>Bellman-Held-Karp algorithm</i>	16
3.4.2 <i>Nearest neighbor (NN) algorithm</i>	17
3.5 <i>Vehicle routing Problem</i>	18
3.6 <i>Multi-depot Vehicle routing Problem</i>	19
3.6.1 <i>Multi-source Dijkstra algorithm</i>	19

4 Casos de utilização	21
5 Funcionalidades e cenários implementados	22
5.1 <i>Load</i> de mapas	22
5.2 Determinar as componentes fortemente conexas de um grafo . .	23
5.3 Determinar caminho mais curto entre dois vértices	24
5.3.1 <i>A* algorithm</i>	24
5.3.2 <i>Dijkstra algorithm</i>	25
5.4 Fazer a distribuição das vacinas	25
6 Estruturas de dados utilizadas	27
7 Algoritmos efetivamente implementados	28
8 Análise de complexidade dos algoritmos implementados	30
8.1 Depth-First-Search algorithm	30
8.2 Breath-First-Search algorithm	31
8.3 Kosaraju algorithm	32
8.4 Dijkstra algorithm	32
8.5 <i>A* algorithm</i>	33
8.6 Nearest Neighbor algorithm	34
8.7 Multi-source Dijkstra algorithm	34
8.8 Solução do Problema	35
9 Análise da conectividade do grafo	37
9.1 Mapa do Porto	38
9.2 Mapa da componente fortemente conexa do Porto	38
9.3 Mapa de Espinho	39
9.4 Mapa da componente fortemente conexa de Espinho	40
9.5 Mapa de Penafiel	41
9.6 Mapa da componente fortemente conexa de Penafiel	42
10 Conclusão	43
11 Contribuição	44
12 Bibliografia	45

Listas de Figuras

1	Visualização após <i>load</i> do mapa do <i>Porto</i>	23
2	Visualização das componentes fortemente conexas do Porto.	23
3	Visualização da execução do algoritmo <i>A*</i>	24
4	Visualização da execução do algoritmo <i>Dijkstra</i>	25
5	Visualização da distribuição das vacinas no Porto.	26
6	Tempo de execução do algoritmo DFS em ordem a $ V + E $. . .	31
7	Tempo de execução do algoritmo BFS em ordem a $ V + E $. . .	31
8	Tempo de execução do algoritmo de Kosaraju em ordem a $ V + E $	32
9	Tempo de execução do algoritmo de Dijkstra em ordem a $(V + E) \cdot \log V $	33
10	Tempo de execução do algoritmo <i>A*</i> em ordem a $(V + E) \cdot \log V $	33
11	Tempo de execução do algoritmo Nearest Neighbor em ordem a $ POI \cdot (V + E) \cdot \log V $	34
12	Tempo de execução do algoritmo Multi-Source Dijkstra em ordem a $(V + E) \cdot \log V $	35
13	Tempo de execução da solução final em ordem a $O(\log POI \cdot POI \cdot ((V + E) \cdot \log V))$	36
14	Componentes fortemente conexas no mapa do Porto completo. .	38
15	Componentes fortemente conexas no mapa da maior componente fortemente conexa do Porto.	38
16	Componentes fortemente conexas no mapa de Espinho completo. .	39
17	Componentes fortemente conexas no mapa da maior componente fortemente conexa de Espinho.	40
18	Componentes fortemente conexas no mapa de Penafiel completo. .	41
19	Componentes fortemente conexas no mapa da maior componente fortemente conexa de Penafiel.	42

Capítulo 1

Descrição do problema

De forma a controlar a distribuição de vacinas contra a COVID-19, a aplicação VaccineRouter deve ser capaz de determinar os itinerários de distribuição de vacinas desde os centros de armazenamento até aos centros de aplicação. Como tal, pretende-se que estes itinerários não sejam demasiado extensos de forma a prevenir que o tempo de distribuição ultrapasse o tempo de conservação das vacinas.

1.1 Primeira Fase

Numa primeira instância, pretende-se que a aplicação seja capaz de definir um percurso para um único veículo, a sair de um único centro de armazenamento, não tendo como preocupação o tempo despendido para a distribuição das vacinas.

Por outro lado, nesta primeira fase, não há preocupação acerca da capacidade dos veículos, de forma a garantir que apenas um veículo consiga distribuir vacinas para todos os centros de aplicação.

Para que tal distribuição seja possível, é necessário que existam caminhos que relacionem todos estes pontos de interesse, isto é, que existam ligações (estradas) que liguem o centro de distribuição com os vários centros de aplicação. Visto que estas ligações são representadas a partir de um grafo, este grafo terá de ser fortemente conexo (com ligações entre todos os vértices, no nosso caso sendo estes vértices os pontos de interesse), o que vai de acordo com a situação real à qual estamos a aplicar o problema.

O objetivo desta fase trata-se, portanto, de encontrar a melhor rota, isto é, o percurso mais rápido, que uma carrinha tem de percorrer de forma a distribuir as vacinas em todos os centros de aplicação.

1.2 Segunda Fase

Numa segunda fase, é acrescentado ao problema o facto de termos vários centros de armazenamento, ou seja, teremos que distribuir os centros de aplicação pelos centros de armazenamento. Cada um dos centros de armazenamento terá, ainda, apenas um veículo com capacidade ilimitada.

Este ponto irá levar a uma diminuição no tempo de distribuição das vacinas uma vez que teremos vários centros de armazenamento que, posteriormente, estarão mais perto de certos centros de aplicação. Este aumento de proximidade entre os vários centros (de aplicação e armazenamento) leva a que o tempo necessário para um dado veículo chegar a um centro de aplicação, saindo de um centro de armazenamento, seja menor do que o que tínhamos na primeira fase, onde uma carrinha tinha de distribuir as vacinas em todos os centros de aplicação.

1.3 Terceira Fase

Na terceira fase será adicionado o fator de tempo limite de distribuição e capacidade limitada das carrinhas. Tal como descrito anteriormente, as vacinas têm um tempo limitado de transporte, isto é, não podem estar muito tempo nas carrinhas que as irão distribuir (em transporte). Por outro lado, adicionando a restrição da capacidade limitada das carrinhas, poderão ocorrer casos em que mesmo que uma carrinha consiga percorrer os vários centros de aplicação no tempo pretendido, não tenha capacidade para levar as vacinas suficientes para os centros a que estava predestinada a distribuir. Isto leva, portanto, a que seja necessário adicionar mais veículos para fazer a sua distribuição a partir de um dado centro de armazenamento.

O valor concreto de veículos por centro de armazenamento dependerá da performance de cada um destes, isto é, o tempo que demora a completar a distribuição. Para controlar esta condicionante, a solução passa por adicionar veículos que irão distribuir a partir desse centro, diminuindo, assim, o tempo de distribuição aos centros de aplicação associados a esse centro e, ainda, o número de veículos utilizados.

Capítulo 2

Formalização do problema

2.1 Dados de entrada

- $G_i = (V_i, E_i)$ - Grafo dirigido pesado, representado por:
 - V_i - Vértices, onde $V_i(i)$ representa o i-ésimo vértice, entre os quais se encontram os centros de distribuição e de aplicação, com:
 - * Id - identificador do vértice, único
 - * $Adj \subseteq E$ - Arestas que partem do vértice
 - * Lat - latitude real do ponto no mapa
 - * Long - longitude real do ponto no mapa
 - E_i - Arestas, onde $E_i(i)$ representa a i-ésima aresta, representando estradas da rede rodoviária, com:
 - * Id - identificador da aresta
 - * W - Peso da aresta, neste caso representando o tempo que demora a ser percorrida essa estrada
 - * $Dest \in V_i$ - vértice ao qual a aresta liga o vértice inicial
- D - Conjunto dos Centros de Armazenamento, onde cada elemento $D_i(i)$ é um vértice V que representa o i-ésimo Centro de Armazenamento caracterizado por:
 - Nva - número de vacinas armazenadas
- A - Conjunto dos Centros de Administração, onde cada elemento $A_i(i)$ é um vértice V que representa o i-ésimo Centro de Administração caracterizado por:

- N - Número de vacinas que serão administradas nesse centro e, portanto, que têm de ser distribuídas para o mesmo
- T - Tempo que as vacinas podem estar em transporte (nas carrinhas durante a distribuição desde os Centros de Armazenamento aos Centros de Administração)

2.1.1 Restrições dos dados de entrada

- $\forall E \in E_i, W(E) > 0$, uma vez que w representa o tempo necessário para percorrer uma aresta
- $\forall E \in Adj(V), Dest(E) \neq V$, uma vez que isso seria uma estrada de um ponto para ele próprio
- $|D| > 0$
- $|A| > 0 \wedge \forall a \in A, N(a) > 0$, uma vez que caso seja 0, já não teremos de ir a esse centro de aplicação pois não se enquadra no nosso objetivo
- $T > 0$
- $\sum_{d \in D} Nva(d) \geq \sum_{a \in A} N(a)$ (número de vacinas dos centros de armazenamento é maior ou igual ao número de vacinas pedido nos centros de administração)

2.2 Dados de saída

- $G_f = (V_f, E_f)$ - Grafo dirigido pesado, sendo V_f e E_f os mesmos atributos que V_i e E_f , à exceção de atributos que poderão ser acrescentados de forma a aplicar certos algoritmos.
- C_f - Sequência das carrinhas, onde $C_f(i)$ representa a i-ésima carrinha, contendo o mesmo tipo de elementos que C_i à exceção de que estes terão mais quatro atributos:
 - T - Tempo que a carrinha demorou a distribuir as vacinas
 - P - Sequência de arestas E que irá percorrer de forma a distribuir as vacinas desde o Centro de Armazenamento, onde P_i representa a i-ésima aresta percorrida
 - N - Número de vacinas que distribuiu
 - C - Centro de armazenamento usado

2.2.1 Restrições dos dados de saída

- $\forall C \in C_f, N(C) > 0$, isto é, todas as carrinhas que foram distribuir têm de distribuir pelo menos uma vacina
- $|C_f| > 0$
- $\forall c \in C_f, (T(c) > 0 \wedge T(c) \leq T)$
- $\forall c \in C_f \forall p \in P(c), p \in E$
- $|P| \leq |E|$
- $\forall c \in C_f$, número de vacinas por distribuir = 0, isto é, todas as carrinhas distribuem todas as vacinas que transportam

2.3 Funções Objetivo

O principal objetivo deste problema é que as vacinas sejam todas distribuídas num dado tempo limite. Contudo, a melhor solução do mesmo passa por, após ter encontrado uma solução que entregue as vacinas dentro desse tempo limite, minimizar o número de carrinhas que as vão distribuir, de forma a tornar esta distribuição menos dispendiosa.

Assim, primeiro pretende-se minimizar o tempo despendido pela carrinha que demora mais tempo (função f), de forma que as vacinas não estejam em transporte mais do que o tempo limite (T), e, secundariamente, minimizar o número de carrinhas que irão transportar as vacinas (função g) aos centros de administração.

$$f = \max([T(c) \forall c \in C_f]) < T$$

$$g = |C_f|$$

Capítulo 3

Perspetivas de solução

O problema proposto é semelhante ao *Travelling Salesman Problem*, colocando agora mais opções para o percurso tomado, tal como a existência de várias carrinhas e de vários centros de armazenamento de onde estas saem. Este problema (*TSP*) enquadra-se nos problemas *NP-Hard* cujo objetivo é encontrar o menor caminho (em termos de tempo e custo) que passe por todos os pontos de interesse.

O facto de termos vários centros de distribuição leva a que este problema vá ao encontro da generalização do *Vehicle routing problem*, ou seja, entramos no âmbito de *Multi-depot vehicle routing problem*.

3.1 Acessibilidade

Acessibilidade diz respeito à possibilidade de, num dado grafo, aceder aos seus vértices a partir de outros.

Para a analisar, poderemos usar algoritmos tal como o *depth-first-search* (*DFS*) ou *breadth-first-search* (*BFS*), que nos indicam os componentes conexos de um grafo.

3.1.1 *Depth-First-Search algorithm*

Método de busca não-informada que avança através da expansão do primeiro nó encontrado no grafo avançando em profundidade até o objetivo ser encontrado ou já não existirem nós não visitados que sejam adjacentes; chegando a este ponto faz-se *backtracking* e ao encontrar um nó com arestas cujos destinos não tenham sido ainda visitados o processo repete-se. Desta forma, este algoritmo tem uma complexidade temporal de $O(|V| + |E|)$.

Algorithm 1 *Depth-First-Search*

```
1:  $G = (V, E)$ 
2: function DFS( $G$ )
3:   for  $v \in V$  do
4:      $visited(v) \leftarrow false$ 
5:   for  $v \in V$  do
6:     if  $!visited(v)$  then
7:       DFS-VISIT( $G, v$ )
8: function DFS-VISIT( $G, v$ )
9:    $visited(v) \leftarrow true$ 
10:  pre-process( $v$ )
11:  for  $w \in Adj(v)$  do
12:    if  $!visited(w)$  then
13:      DFS-VISIT( $G, w$ )
14:  post-process( $v$ )
```

3.1.2 *Breadth-First-Search algorithm*

É também um método de busca não-informada que expande sequencialmente todos os vértices do grafo. Por cada vértice o algoritmo itera por todas as arestas deste e garante que nenhum vértice é visitado mais do que uma vez. Desta forma, este algoritmo tem uma complexidade temporal de $O(|V|+|E|)$.

Algorithm 2 *Breadth-First-Search*

```
1:  $G = (V, E)$ 
2: function BFS( $G$ )
3:   choose  $s$  from  $G$ 
4:    $visit(s)$ 
5:    $insert(Q, s)$ 
6:   while  $!empty(Q)$  do
7:      $v \leftarrow extract(Q)$ 
8:     for  $e \in edges(v)$  do
9:        $w \leftarrow destiny(e)$ 
10:      if  $!visited(w)$  then
11:         $visit(w)$ 
12:         $insert(Q, w)$ 
```

3.2 Conectividade forte

De forma a podermos utilizar certos tipos de algoritmos, temos de, primeiro, averiguar a conectividade do grafo de entrada. Este, sendo um grafo que representa um mapa real, será um grafo esparso, pelo que teremos de confirmar se é possível navegar de um dado vértice para os outros, descartando os vértices que não são acessíveis.

Para tal, poderemos usar o algoritmo de *Kosaraju* ou o algoritmo de *Tarjan*, cujo objetivo é encontrar as componentes fortemente conexas de um grafo dirigido, isto é, encontrar conjuntos de vértices onde todos os vértices desse conjunto são atingíveis a partir de qualquer outro vértice pertencente a esse mesmo conjunto.

3.2.1 *Kosaraju algorithm*

O algoritmo de *Kosaraju* é um algoritmo de complexidade linear cujo objetivo é encontrar componentes fortemente conexas de um grafo. O algoritmo foi sugerido por *Sambasiva Kosaraju* em 1978, contudo apenas foi publicado independentemente em 1981 por *Micha Sarir*. [1]

Este algoritmo consiste em fazer primeiro uma pesquisa em profundidade (*DFS*), colocando os vértices visitados numa *stack*, após ter sido feita a chamada recursiva *DFS* aos seus vértices adjacentes; posteriormente, fazer o reverso do grafo, isto é, mudar a direção de todas as arestas do grafo; numa terceira fase, retirar elementos um a um da *stack* (*pop*) enquanto esta não estiver vazia. Sendo *V* o elemento retirado (*popped*), aplicar a pesquisa em profundidade *DFS* começando no vértice *V*; após este 2.^º *DFS*, todos os vértices visitados irão formar uma componente fortemente conexa.

Este algoritmo tem complexidade temporal $O(|V| + |E|)$, pelo que se revela um bom algoritmo para este objetivo.

Algorithm 3 *Kosaraju*

```
1:  $G = (V, E)$ 
2: stack stack
3: vector SCC
4: int component

5: function DFS1( $G, s$ )
6:   visit( $s$ )
7:   for  $v \in Adj(s)$  do
8:     if  $!visited(v)$  then
```

```
9:           DFS1(v)  
  
10: function DFS2(G, component, s)  
11:   visit(s)  $\leftarrow$  false  
12:   SCC(component).push(s)  
13:   for v  $\in$  Adj(s) do  
14:     if !visited(v) then  
15:       DFS2(G, component, s)  
  
16: function KOSARAJU()  
17:   for v  $\in$  V do  
18:     visit(s)  $\leftarrow$  false  
19:   for v  $\in$  V do  
20:     DFS1(G, s)  
21:   G = getTranspose(G)  
22:   for v  $\in$  V do  
23:     visit(s)  $\leftarrow$  false  
24:   while !stack.empty do  
25:     v  $\leftarrow$  stack.pop()  
26:     if !visited(v) then  
27:       DFS2(G, component, v)  
28:     component ++  
29:   return SCC
```

3.2.2 Tarjan's algorithm

O algoritmo de *Tarjan* é outro algoritmo que poderemos usar para estudar a conectividade do grafo. Proposto em 1972 por *Robert Tarjan* [2], o algoritmo consiste em encontrar as componentes fortemente conexas a partir dumha exploração em profundidade (*DFS*) começando num vértice arbitrário. Atribuindo *ids* e um *low-link value* aos vértices utilizando uma *stack*, o algoritmo agrupa os vértices que pertencem à mesma componente fortemente conexa na chamada recursiva da *DFS*.

Algorithm 4 Tarjan

```
1: G = (V, E)  
2: stack stack  
3: id  $\leftarrow$  0
```

```
4: sccCount  $\leftarrow 0$ 

5: function DFS( $G, s$ )
6:   stack.push( $s$ )
7:    $id(s) \leftarrow id + +$ 
8:    $low(s) \leftarrow id(s)$ 
9:   onstack( $s$ )  $\leftarrow true$ 
10:  for  $v \in Adj(s)$  do
11:    if  $id(v) = NULL$  then
12:      DFS( $v$ )
13:    if onstack( $v$ ) then
14:       $low(s) \leftarrow min(low(s), low(v))$ 
15:    if  $low(s) = id(s)$  then
16:      while ( $v \leftarrow stack.pop()$ )  $\neq s$  do
17:        onstack( $v$ )  $\leftarrow false$ 
18:         $low(v) \leftarrow id(s)$ 
19:         $SCC(v) \leftarrow s$ 
20:       $SCC(s) \leftarrow s$ 

21: function TARJAN( $G$ )
22:   for  $u \in V$  do
23:      $id(u) \leftarrow NULL$ 
24:      $SCC(u) \leftarrow NULL$ 
25:   for  $u \in V$  do
26:     if  $id(u) = NULL$  then
27:       DFS( $G, u$ )
28:   return SCC
```

3.3 Caminho mais curto

Para resolvemos o problema em questão, teremos, também, de calcular o caminho mais curto entre dois vértices do grafo, de forma a, posteriormente, conseguirmos conectar os vários centros de administração ao centro de distribuição a si referenciado.

Desta forma, pretende-se, com os seguintes algoritmos, determinar o caminho mais curto entre um vértice de origem e um vértice de destino.

3.3.1 Dijkstra algorithm

O algoritmo de *Dijkstra* é talvez o mais conhecido algoritmo para calcular o caminho mais curto, sendo que parte do pressuposto que não há arestas negativas.

Primeiramente, todos os vértices são marcados com distância infinita até ao vértice de origem, exceto o próprio, que é marcado com distância 0. Após isso, fazem-se sucessivas visitas aos vértices adjacentes V_{adj} do vértice V_p com distância menor e atualiza-se a distância mínima a eles passando pelo vértice considerado V_p e a aresta que os conecta. Assim, este algoritmo, utilizando uma fila de prioridade de mínimos, tem uma complexidade temporal $O((|V| + |E|) \cdot \log|V|)$, apesar de poder ser melhorada para $O(|V| \cdot \log|V|)$ utilizando *Fibonacci Heaps*.

Algorithm 5 Dijkstra

```
1:  $G = (V, E)$ 
2:  $s \in V$ 

3: function DIJKSTRA( $G, s$ )                                 $\triangleright$  using a priority queue
4:   for  $v \in V$  do
5:      $dist(v) \leftarrow INF$ 
6:      $path(v) \leftarrow NULL$ 
7:    $dist(s) \leftarrow 0$ 
8:    $Q \leftarrow V$                                           $\triangleright Q$  is a min-priority queue of dist
9:   while  $|Q| > 0$  do
10:     $v \leftarrow Q.extract\_min()$ 
11:    for  $u \in Adj(v)$  do
12:       $dist\_temp \leftarrow dist(v) + weight(v, u)$ 
13:      if  $dist\_temp < dist(u)$  then
14:         $dist(u) \leftarrow dist\_temp$ 
15:         $path(u) \leftarrow v$ 
16:         $Q.decrease\_key(u, dist(u))$ 
17:   return G
```

3.3.2 A * algorithm

O algoritmo *A** [3] é também um algoritmo bastante conhecido de caminho mais curto, sendo este uma extensão do algoritmo de *Dijkstra* usando métodos heurísticos. Trata-se de uma pesquisa informada que averigua primeiro os caminhos mais promissores, fazendo uso de uma função heurística

h.

Em termos simples, enquanto que o algoritmo de *Dijkstra* seleciona o próximo vértice a averiguar pelo mínimo de distância até ele, o algoritmo A^* escolhe o vértice v que minimiza $f(v) = dist(v) + h(v)$ sendo $h(v)$ a função heurística. Para $h(v)$ pode ser usado, por exemplo, a distância euclidiana de v até ao vértice final. Confirma-se, ainda, que o algoritmo de *Dijkstra* é um caso particular do algoritmo A^* em que $h(v) = 0$. Desta forma, a complexidade do algoritmo A^* é semelhante à complexidade do algoritmo de *Dijkstra* apenas com uma pequena ponderação dependendo da função $h(v)$ usada.

Algorithm 6 A^*

```

1:  $G = (V, E)$ 
2:  $s \in V$ 

3: function ASTAR( $G, s$ )                                 $\triangleright$  using a priority queue
4:   for  $v \in V$  do
5:      $dist(v) \leftarrow INF$ 
6:      $hdist(v) \leftarrow INF$ 
7:      $path(v) \leftarrow NULL$ 
8:    $dist(s) \leftarrow 0$ 
9:    $hdist(s) \leftarrow h(s)$ 
10:   $Q \leftarrow V$                                           $\triangleright Q$  is a min-priority queue of hdist
11:  while  $|Q| > 0$  do
12:     $v \leftarrow Q.extract\_min()$ 
13:    for  $u \in Adj(v)$  do
14:       $dist\_temp \leftarrow dist(v) + weight(v, u)$ 
15:      if  $dist\_temp < dist(u)$  then
16:         $dist(u) \leftarrow dist\_temp$ 
17:         $path(u) \leftarrow v$ 
18:         $Q.decrease\_key(u, dist(u) + h(u))$ 
19:  return  $G$ 

```

3.3.3 *Floyd-Warshall algorithm*

O algoritmo de *Floyd-Warshall*, publicado independentemente por *R. W. Floyd* e *S. Warshall* em 1962 difere nos mencionados anteriormente no facto de encontrar o caminho mais curto entre todos os vértices numa única execução. O algoritmo mantém uma matriz de distâncias que vai sendo atualizada à medida que vão sendo considerados diferentes vértices intermédios.

Este algoritmo tem uma complexidade temporal de $O(|V|^3)$ e espacial de $O(|V|^2)$.

Algorithm 7 *Floyd-Warshall*

```

1:  $G = (V, E)$ 
2: function FLOYDWARSHALL( $G$ )
3:    $distM \leftarrow matrix(|V|, |V|)$ 
4:    $fill(distM, INF)$ 
5:   for  $e \in E$  do
6:      $distM[\text{orig}(e), \text{dest}(e)] \leftarrow dist(e)$ 
7:   for  $i = 1$  to  $|V|$  do
8:      $distM[i, i] \leftarrow 0$ 
9:   for  $k = 1$  to  $|V|$  do
10:    for  $i = 1$  to  $|V|$  do
11:      for  $j = 1$  to  $|V|$  do
12:        if  $distM[i, j] > distM[i, k] + distM[k, j]$  then
13:           $distM[i, j] \leftarrow distM[i, k] + distM[k, j]$ 
14:   return  $distM$ 

```

3.4 Travelling Salesman Problem

Este é o problema de encontrar o caminho mais curto e eficiente para o Salesman de forma a passar em todos os seus pontos de interesse. Existem diferentes métodos usados para resolver este problema, entre eles:

- *Brute-Force*
- *Branch and Bound*
- *Nearest Neighbor*

3.4.1 Bellman-Held-Karp algorithm

Uma possível solução exata para este problema é a utilização do algoritmo de *Bellman-Held-Karp*.

Este algoritmo foi proposto independentemente em 1962 por *Richard Bellman*, *Michael Held* e por *Richard Karp* e consiste na utilização de programação dinâmica, resolvendo o exercício geral e mais complexo a partir da

resolução de subproblemas do mesmo, isto é, a solução do problema pode ser obtida a partir das soluções dos seus subproblemas.

Este algoritmo oferece-nos uma complexidade temporal relativamente boa (inferior à complexidade da abordagem *brute-force* que é de ordem fatorial) de $O(|V|^2 \cdot 2^{|V|})$ mas, por outro lado, tem uma complexidade espacial $O(|V| \cdot 2^{|V|})$, ambas no pior caso. [4]

Assim sendo, e uma vez que os grafos com os quais vamos lidar serão relativamente grandes, não vai ser útil um algoritmo de ordem exponencial e como tal, não será usado na implementação. Ainda assim, achamos por bem fazer referência a ele dado que se trata de um algoritmo exato.

Algorithm 8 Bellman-Held-Karp

```

1:  $G = (V, E)$ 
2: function BELLMANHELDKARP( $G$ )
3:    $s \leftarrow$  inicial vertex
4:   for  $v \in V$  do
5:      $OPT[\{v\}, v] \leftarrow weight(s, v)$ 
6:   for  $i = 2$  to  $|V|$  do
7:     for  $S \subseteq V - s$  with  $|S| = i$  do
8:       for  $v \in S$  do
9:          $OPT[S, v] \leftarrow min\{OPT[S - v, u] + weight(u, v) \mid u \in S - v\}$ 
10:      return  $min\{OPT[V - s, v] + weight(v, s) \mid v \in V - s\}$ 
```

3.4.2 Nearest neighbor (NN) algorithm

O *Nearest neighbor (NN) algorithm* [5] foi um dos primeiros métodos usados para resolver o *TSP*. Passa por escolher um ponto onde começar e escolhe o vértice adjacente a este cuja aresta que os ligue tenha menor peso e ainda não tenha sido visitado.

No caso em concreto deste trabalho, em vez de escolher o vértice adjacente mais próximo, é escolhido o ponto de interesse mais próximo, que pode ser, por exemplo, obtido usando o algoritmo A^* com origem no vértice atual. Outra possibilidade será a de condensar o grafo de input para um outro, cujos únicos vértices são os pontos de interesse (*POI*), ou seja, os centros de armazenamento e os centros de aplicação.

O resultado é a ordem pela qual os vértices foram visitados.

Este sendo um algoritmo que procura obter a solução a partir da melhor escolha em cada momento (indo ao vértice mais próximo), revela-se um al-

goritmo ganancioso. Ainda que o problema não possua subestrutura ótima e, como tal, o algoritmo não garanta a solução ótima, o *NN* apresenta bons resultados.

Este algoritmo tem, no pior caso, uma complexidade temporal de $O(|V|^2)$ e espacial de $O(|V|)$.

Algorithm 9 *Nearest neighbor*

```
1:  $G = (V, E)$ 
2: function NN( $G$ )
3:   for  $v \in V$  do
4:      $visited(v) \leftarrow false$ 
5:      $v \leftarrow select(V, 0)$ 
6:      $visited(v) \leftarrow true$ 
7:      $S \leftarrow \emptyset$ 
8:      $insert(S, v)$ 
9:      $count \leftarrow 0$ 
10:    while  $count < size(V)$  do
11:       $Q \leftarrow \emptyset$ 
12:      for  $e \in edges(v)$  do
13:         $insert(Q, (destiny(e), distance))$ 
14:      while  $size(Q) > 0$  do
15:         $w \leftarrow extractMin(Q)$ 
16:        if  $!visited(w)$  then
17:           $increment(count)$ 
18:           $visited(w) \leftarrow true$ 
19:           $v \leftarrow w$ 
20:           $insert(S, v)$ 
21:          break
22:    return  $G$ 
```

3.5 *Vehicle routing Problem*

Vehicle routing problem pode ser considerado um *Travelling Salesman Problem* (*TSP*) com vários *Salesman*, sendo os *Salesman* os veículos e como o tradicional *TSP* tem diversos destinos pelos quais tem de passar. O objetivo é distribuir pelos diversos veículos os destinos de forma que todos sejam visitados, sendo esta distribuição otimizada de forma que o tempo total de

distribuição (de todos os percursos aquele que demorar mais tempo) seja o menor possível.

Para fazer esta distribuição de destinos a veículos existem diversos métodos, entre eles os heurísticos.

Com a distribuição feita para cada instância de veículo e respetivos destinos temos um subproblema de *Travelling Salesman*. Resolve-se este problema de forma a obter o caminho mais pequeno possível.

3.6 *Multi-depot Vehicle routing Problem*

Multi-depot vehicle routing problem é uma generalização do *Vehicle routing problem* no qual, para além de haver vários veículos, existem também vários armazéns, sendo este o problema com que estamos a lidar para fazer a distribuição das vacinas.

A abordagem que seguiremos será fazer *clustering* dos centros de aplicação pelos centros de armazenamento de forma a subdividir este problema mais complexo em múltiplos subproblemas do *Vehicle Routing problem*.

3.6.1 *Multi-source Dijkstra algorithm*

O método usado para fazer os clusters será fazer múltiplas pesquisas em largura “simultaneamente” a partir de todos os centros de armazenamento até todos os centros de distribuição estarem no *cluster* de um centro de armazenamento. Este método acaba por ser uma variante do algoritmo de *Dijkstra* com múltiplas origens.

Algorithm 10 *Multi-source Dijkstra*

```

1:  $G = (V, E)$ 
2:  $S \subseteq V$ 

3: function MULTIDIJKSTRA( $G, S$ )
4:   for  $v \in V$  do
5:      $dist(v) \leftarrow INF$ 
6:      $path(v) \leftarrow NULL$ 
7:   for  $s \in S$  do
8:      $dist(s) \leftarrow 0$ 
9:    $Q \leftarrow V$                                  $\triangleright Q$  is a min-priority queue of dist
10:  while  $|Q| > 0$  do
11:     $v \leftarrow Q.extract\_min()$ 

```

```
12:     for  $u \in Adj(v)$  do
13:          $dist\_temp \leftarrow dist(v) + weight(v, u)$ 
14:         if  $dist\_temp < dist(u)$  then
15:              $dist(u) \leftarrow dist\_temp$ 
16:              $path(u) \leftarrow v$ 
17:              $Q.decrease\_key(u, dist(u))$ 
18:     return G
```

Neste pseudocódigo, por questões de simplicidade, o algoritmo só termina quando todos os vértices forem processados, sendo que o *cluster* de um dado centro de distribuição *CD* pode ser obtido a partir de $path(CD)$. Para além disso, no programa terá que se ter em conta a quantidade de vacinas que cada centro de armazenamento tem, dado que, não adianta adicionar um centro de administração ao *cluster* do centro de armazenamento mais próximo dele se este já não tiver vacinas suficientes.

Capítulo 4

Casos de utilização

O nosso programa deve permitir as seguintes funcionalidades:

- Fazer a distribuição dos centros de administração pelos centros de armazenamento (*clustering*).
- Definir o número de veículos necessários para a distribuição de vacinas, as suas rotas e tempo utilizado em viagem por cada um, respeitando as restrições do problema.
- Possivelmente, fazer uso de uma ferramenta externa de visualização de grafos de forma a observar a rota de cada veículo.
- Obter as componentes fortemente conexas do grafo de entrada.
- Obter, caso existam, os centros de administração inacessíveis.

Capítulo 5

Funcionalidades e cenários implementados

O nosso programa vai acompanhado dos seguintes mapas: Porto, Espinho, Penafiel e correspondentes componentes fortemente conexas de cada um dos referidos sendo que para a sua visualização dos mesmos é utilizada a ferramenta recomendada *GraphViewerCpp*.

Nestes mapas é possível realizar as seguintes operações:

5.1 *Load* de mapas

Inicialmente o programa permite escolher entre um conjunto de mapas, sendo que, mesmo após dar load a um deles, é possível dar load a outro. Após cada load o utilizador visualiza o mapa sendo possível identificar claramente os centros de distribuição e os centros de aplicação.

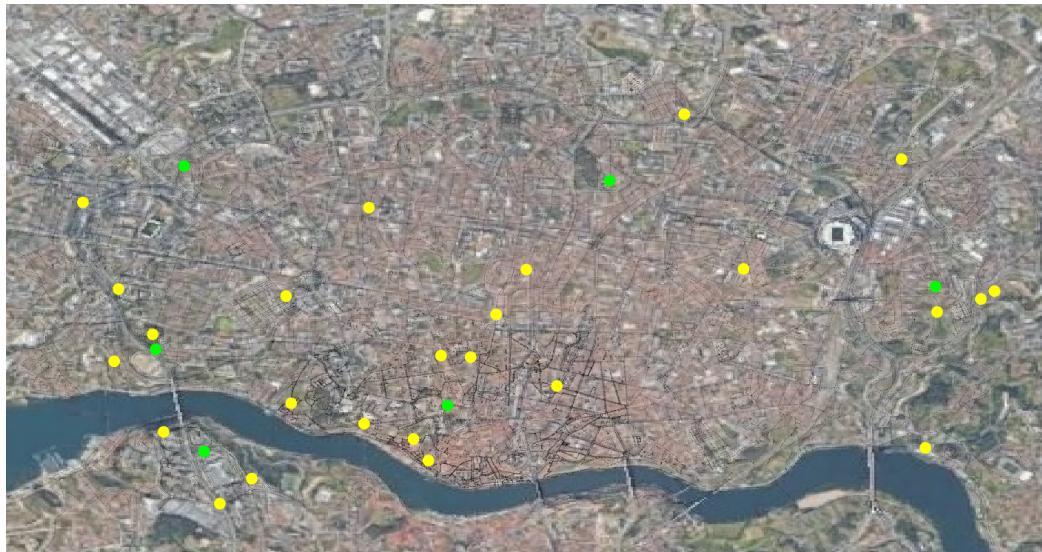


Figura 1: Visualização após *load* do mapa do *Porto*.

5.2 Determinar as componentes fortemente conexas de um grafo

O programa permite determinar as componentes fortemente conexas de um dado grafo, sendo mostrado posteriormente a amarelo a componente maior e, a azul, todas as restantes componentes.

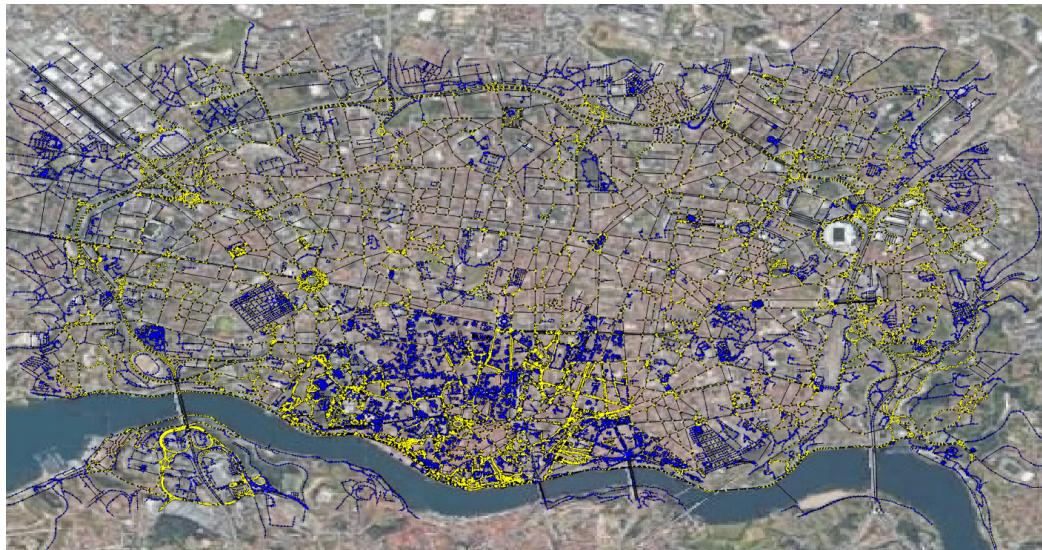


Figura 2: Visualização das componentes fortemente conexas do Porto.

5.3 Determinar caminho mais curto entre dois vértices

É possível determinar o caminho mais curto entre dois vértices utilizando dois algoritmos: A* ou Dijkstra, sendo mostrado posteriormente o caminho, os vértices visitados pelo algoritmo, e a contagem de vértices visitados.

Desta forma é possível comparar os resultados obtidos pelos dois algoritmos nomeadamente o número de vértices que cada um processa.

5.3.1 *A* algorithm*



Figura 3: Visualização da execução do algoritmo A^* .

5.3.2 Dijkstra algorithm



Figura 4: Visualização da execução do algoritmo *Dijkstra*.

5.4 Fazer a distribuição das vacinas

O programa permite ainda fazer a distribuição das vacinas completamente. É pedido como input o tempo limite de conservação em minutos das vacinas. Neste programa é assumido que a velocidade das carrinhas utilizadas no transporte das vacinas é tal que é numericamente igual às distâncias percorridas. Isto aconteceu por não termos dados relativos às velocidades máximas em cada uma das arestas.

É feita a atribuição de rotas para carrinhas desde os centros de armazenamento até aos centros de aplicação, sendo posteriormente mostrada o tempo dispendido por cada carrinha. Para além disto, é mostrado o grafo utilizado da seguinte forma: Centros de aplicação a amarelo quando na sua componente fortemente conexa está presente pelo menos um centro de armazenamento, vermelho caso contrário; Centros de armazenamento a verde quando na sua componente fortemente conexa está presente pelo menos um centro de aplicação, laranja caso contrário; A cada carrinha é atribuída uma cor aleatoriamente que é usada para identificar as arestas por onde ela passa. Se numa aresta passar mais que uma vez uma carrinha, mesmo que seja a mesma, essa aresta é colorida a vermelho para dar essa indicação.

O programa recebe ainda um outro input que permite ao utilizador escolher se deseja visualizar os caminhos mais rápidos possíveis mesmo que

o tempo limite seja ultrapassado. Esta situação pode acontecer quando é introduzido um tempo limite extremamente baixo.

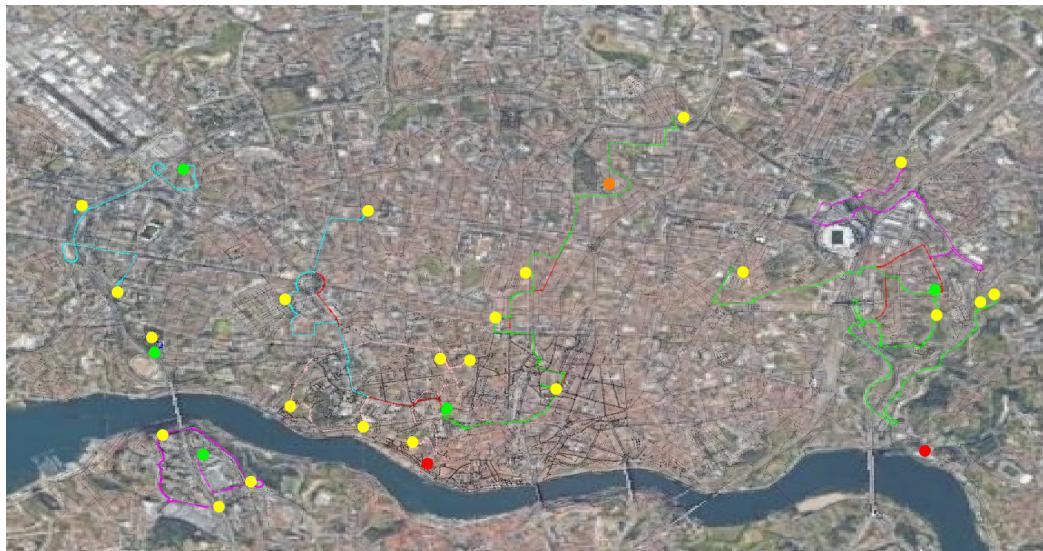


Figura 5: Visualização da distribuição das vacinas no Porto.

Capítulo 6

Estruturas de dados utilizadas

De modo a resolver o problema em mãos, foi necessário recorrer a diversas estruturas de dados. Para obter uma representação dos grafos foi utilizado um *hash-set* dos vértices sendo utilizado o seu id, único, como hash. Com esta representação dos vértices num *hash-set* foi possível reduzir o tempo de pesquisa de um vértice específico de $O(N)$, conseguido usando uma representação baseada em vetores, para $O(1)$.

Para guardar as arestas decidiu-se manter em cada vértice o conjunto de arestas que têm esse vértice como origem e outro conjunto de arestas que têm esse vértice como destino. Dada a natureza dos grafos (grafos de cidades), estes conjuntos são sempre bastante pequenos e, uma vez que a operação mais comum sobre as arestas é iterar por elas todas, um simples vetor mostra-se eficaz para a nossa representação das mesmas.

Tanto nos vértices como nas arestas, para além dos atributos "obrigatórios" como ids e pesos, tornou-se necessário manter atributos relacionados com o problema, nomeadamente: id da componente conexa forte a que pertence cada vértice, coordenadas dos vértices em latitude e longitude, tipo de vértice (se é um centro e, se sim, que tipo de centro). Por outro lado, foi também necessário que cada vértice tivesse o id do cluster (de notar que cluster \neq componente conexa forte) em que se encontra e que cada aresta mantivesse o registo das carrinhas que passaram por ela, de modo a facilitar a representação visual posterior do grafo na interface utilizada.

De modo a tornar os algoritmos que necessitam de filas prioridades (*queues*) mais eficazes, foi utilizada uma implementação de uma fila de prioridades mutável, *MutablePriorityQueue.h*, fornecida nas aulas práticas. Esta estrutura mostrou ser eficaz e útil na implementação de algoritmos de caminho mais curto, nomeadamente o *Dijkstra* e o *A**.

De forma geral, foram ainda implementadas diversas classes como *ApplicationCenter* e *StorageCenter* de modo a facilitar a resolução do problema.

Capítulo 7

Algoritmos efetivamente implementados

Para a solução do problema proposto, foram implementados parte dos algoritmos descritos em Perspetivas de solução.

Para efetuar a análise da Acessibilidade, foram implementados os algoritmos DFS e BFS.

Para a análise da complexidade da Conectividade forte do grafo, foi implementado o algoritmo Kosaraju uma vez que se mostrou, em relação ao algoritmo Tarjan, mais adequado pela sua simplicidade e eficácia. Por outro lado, devido à estrutura do Grafo utilizada, foi possível implementar este algoritmo sem a necessidade de criar o grafo transposto descrito no pseudocódigo, uma vez que eram guardadas as arestas pela sua direção.

Para o cálculo do Caminho mais curto, foram implementados os algoritmos Dijkstra e A*. Uma vez que estes algoritmos são algoritmos gananciosos, foram feitas alterações nos mesmos de modo a que, quando se atinge o ponto de interesse mais próximo, estes terminam, de forma a evitar a análise de vértices sem interesse e, portanto, melhorar a sua eficiência. Uma vez que o algoritmo Floyd-Warshall tem uma complexidade muito elevada $O(|V|^3)$, descartou-se a sua utilização para a resolução do problema proposto.

De forma a resolver o Travelling Salesman Problem descartou-se a utilização do algoritmo Bellman-Held-Karp uma vez que apresenta uma complexidade muito elevada de $O(|V|^2 \cdot 2^{|V|})$, pelo que foi implementado o algoritmo Nearest neighbor. Tal como descrito no tópico do algoritmo Nearest neighbor, este algoritmo foi adaptado e, em vez de ser escolhido o vértice adjacente mais próximo, é escolhido o ponto de interesse mais próximo como o demonstra o seguinte pseudo código:

Algorithm 11 *Nearest neighbor adapted*

```

1:  $G = (V, E)$ 
2:  $POI \subseteq V$ 
3:  $s \in V$  ▷ s is the starting vertex

4: function DIJKSTRA_NARESTAC( $G, s, POI$ )
5:   for  $v \in V$  do
6:      $dist(v) \leftarrow INF$ 
7:      $path(v) \leftarrow NULL$ 
8:    $dist(s) \leftarrow 0$ 
9:    $Q \leftarrow V$  ▷ Q is a min-priority queue of dist
10:  while  $|Q| > 0$  do
11:     $v \leftarrow Q.extract\_min()$ 
12:    if  $v \in POI$  then
13:      return  $v$ 
14:    for  $u \in Adj(v)$  do
15:       $dist\_temp \leftarrow dist(v) + weight(v, u)$ 
16:      if  $dist\_temp < dist(u)$  then
17:         $dist(u) \leftarrow dist\_temp$ 
18:         $path(u) \leftarrow v$ 
19:         $Q.decrease\_key(u, dist(u))$ 
20:    return  $NULL$ 

21: function NN( $G, s, POI$ )
22:    $runningPath \leftarrow \emptyset$ 
23:    $count \leftarrow 0$ 
24:   while  $count < size(POI)$  do
25:      $v \leftarrow DijkstraNearestAC(G, s, POI)$  ▷ v is the nearest POI
26:      $count \leftarrow count + 1$ 
27:      $visitedAC(v) \leftarrow true$ 
28:      $runningPath.push(G.getPathFromTo(s, v))$ 
29:      $POI.remove(v)$ 
30:      $s \leftarrow v$ 
31:   return  $runningPath$ 

```

Por outro lado uma vez que pode haver vários centros de distribuição, foi necessário resolver o Multi-depot Vehicle Routing Problem. Tal como descrito na seção 3.6, para resolver o problema foi implementado o algoritmo Multi-source Dijkstra.

Capítulo 8

Análise de complexidade dos algoritmos implementados

De forma a avaliar a qualidade da solução implementada, procedeu-se à análise da complexidade das soluções algorítmicas implementadas. Para tal, foram feitas análises teóricas e empíricas.

8.1 Depth-First-Search algorithm

Tal como descrito na secção 3.1.1, a complexidade temporal deste algoritmo é da ordem de $O(|V| + |E|)$ e a complexidade espacial é $O(V)$ uma vez que a implementação utilizada deste algoritmo necessita de guardar a informação de se os nós já foram visitados ou não. Para se fazer a análise empírica do algoritmo, foi utilizado o algoritmo quinze vezes em cada mapa (Porto, PortoSCC, Espinho, EspinhoSCC, Penafiel, PenafielSCC). O gráfico representado de seguida mostra os resultados obtidos em ordem à complexidade temporal teórica:

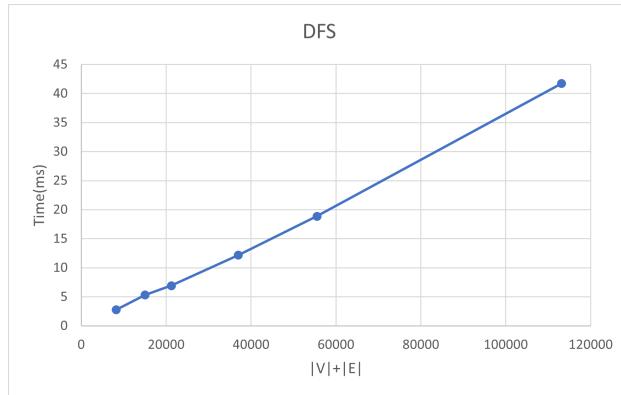


Figura 6: Tempo de execução do algoritmo DFS em ordem a $|V| + |E|$.

Como o gráfico indica, obteve-se uma linearidade dos resultados em relação à sua complexidade temporal, pelo que se confirma a complexidade teórica analisada face aos grafos utilizados.

8.2 Breath-First-Search algorithm

Tal como descrito na secção 3.1.2, a complexidade temporal deste algoritmo é da ordem de $O(|V| + |E|)$ e a sua complexidade espacial é de ordem constante $O(V)$ uma vez que a implementação deste algoritmo necessita de guardar os vértices ainda não visitados numa queue. Para se fazer a análise empírica do algoritmo, foi utilizado o algoritmo quinze vezes em cada mapa (Porto, PortoSCC, Espinho, EspinhoSCC, Penafiel, PenafielSCC). O gráfico representado de seguida mostra os resultados obtidos em ordem à complexidade temporal teórica:

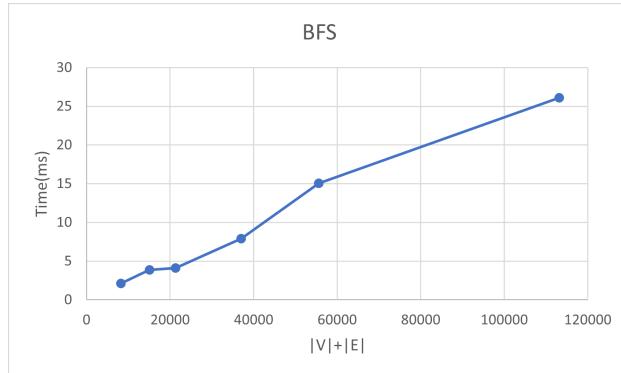


Figura 7: Tempo de execução do algoritmo BFS em ordem a $|V| + |E|$.

Como o gráfico indica, obtiveram-se resultados aproximadamente lineares em relação à sua complexidade temporal, pelo que se confirma a complexidade teórica analisada face aos grafos utilizados.

8.3 Kosaraju algorithm

Tal como descrito na secção 3.2.1, a complexidade temporal deste algoritmo é da ordem de $O(|V| + |E|)$ e a sua complexidade espacial é da ordem $O(|V|)$ uma vez que consiste na execução de duas pesquisas em profundidade (DFS) e guarda os vértices visitados numa stack. Para se fazer a análise empírica do algoritmo, foi utilizado o algoritmo quinze vezes em cada mapa (Porto, PortoSCC, Espinho, EspinhoSCC, Penafiel, PenafielSCC). O gráfico representado de seguida mostra os resultados obtidos em ordem à complexidade temporal teórica:

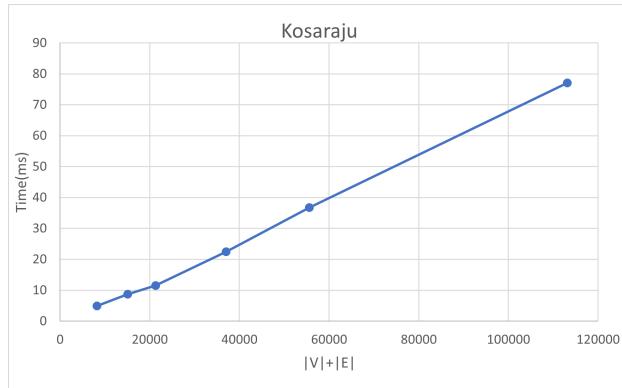


Figura 8: Tempo de execução do algoritmo de Kosaraju em ordem a $|V| + |E|$.

Como o gráfico indica, obteve-se uma linearidade dos resultados em relação à sua complexidade temporal, pelo que se confirma a complexidade teórica analisada face aos grafos utilizados.

8.4 Dijkstra algorithm

Tal como descrito na secção 3.3.1, a complexidade temporal deste algoritmo é da ordem de $O((|V|+|E|)\cdot\log |V|)$ e a complexidade espacial da ordem $O(|V|)$ uma vez que se tem de guardar o path e a distância dos vértices. Para se fazer a análise empírica do algoritmo, foi utilizado o algoritmo onze vezes em cada mapa (Porto, PortoSCC, Espinho, EspinhoSCC, Penafiel, PenafielSCC). O

gráfico representado de seguida mostra os resultados obtidos em ordem à complexidade temporal teórica:

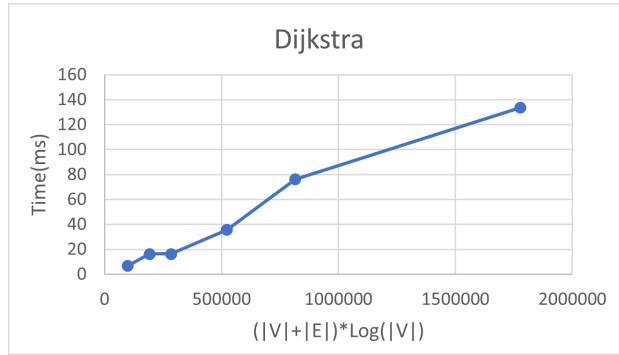


Figura 9: Tempo de execução do algoritmo de Dijkstra em ordem a $(|V| + |E|) \cdot \log |V|$.

Como o gráfico indica, obtiveram-se resultados aproximadamente lineares em relação à sua complexidade temporal, pelo que se confirma a complexidade teórica analisada face aos grafos utilizados.

8.5 A* algorithm

Tal como descrito na secção 3.3.2, a complexidade temporal deste algoritmo é da ordem de $O((|V| + |E|) \cdot \log |V|)$ e a complexidade espacial é da ordem $O(|V|)$ uma vez que, tal como no algoritmo de Dijkstra, tem de se guardar o path e a distância dos vértices. Para se fazer a análise empírica do algoritmo, foi utilizado o algoritmo onze vezes em cada mapa (Porto, PortoSCC, Espinho, EspinhoSCC, Penafiel, PenafielSCC). O gráfico representado de seguida mostra os resultados obtidos em ordem à complexidade temporal teórica:

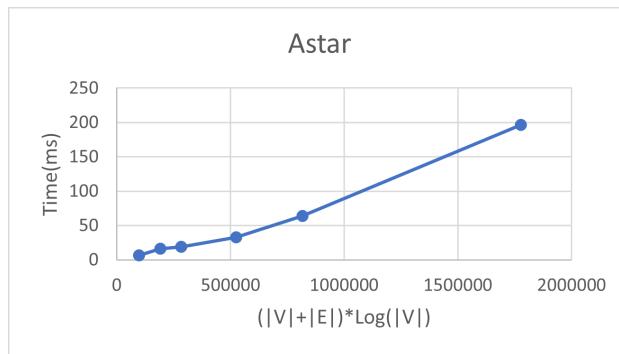


Figura 10: Tempo de execução do algoritmo A* em ordem a $(|V| + |E|) \cdot \log |V|$.

Como o gráfico indica, obteve-se uma linearidade dos resultados em relação à sua complexidade temporal, pelo que se confirma a complexidade teórica analisada face aos grafos utilizados.

8.6 Nearest Neighbor algorithm

O Algoritmo Nearest Neighbor implementado tem complexidade espacial de $O(|V|)$ tal como descrito na secção 3.4.2, contudo não tem a complexidade temporal teórica de $O(|V|^2)$ analisada na mesma, uma vez que foi feita a adaptação descrita na seccão Algoritmos efetivamente implementados. Desta forma, o algoritmo irá percorrer $|POI|$ vezes o algoritmo de Dijkstra para obter o ponto de interesse mais próximo não visitado ao ponto de interesse atual. Assim, terá uma complexidade da ordem de $O(|POI| \cdot ((|V| + |E|) \cdot \log |V|))$. Para se fazer a análise empírica do algoritmo, foi utilizado o algoritmo onze vezes em cada mapa (Porto, PortoSCC, Espinho, EspinhoSCC, Penafiel, PenafielSCC). O gráfico representado de seguida mostra os resultados obtidos em ordem à complexidade temporal teórica:

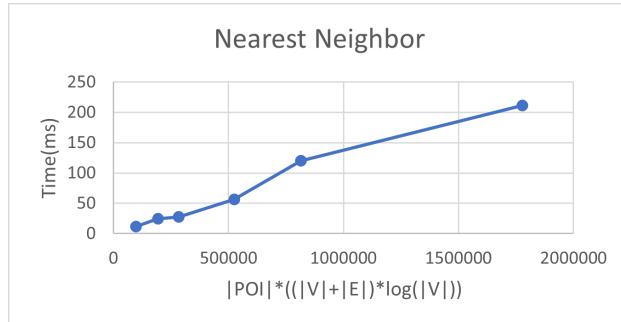


Figura 11: Tempo de execução do algoritmo Nearest Neighbor em ordem a $|POI| \cdot (|V| + |E|) \cdot \log |V|$.

Como o gráfico indica, obtiveram-se resultados aproximadamente lineares em relação à sua complexidade temporal, pelo que se confirma a complexidade teórica analisada face aos grafos utilizados.

8.7 Multi-source Dijkstra algorithm

Tal como descrito na secção 3.6.1, a complexidade temporal deste algoritmo é da ordem de $O((|V| + |E|) \cdot \log |V|)$ e a sua complexidade espacial é da ordem de $O(|V|)$, uma vez que guarda os vértices não visitados e cuja distância

foi atualizada. Para se fazer a análise empírica do algoritmo, foi utilizado o algoritmo nove vezes em cada mapa (Porto, PortoSCC, Espinho, EspinhoSCC, Penafiel, PenafielSCC). O gráfico representado de seguida mostra os resultados obtidos em ordem à complexidade temporal teórica:

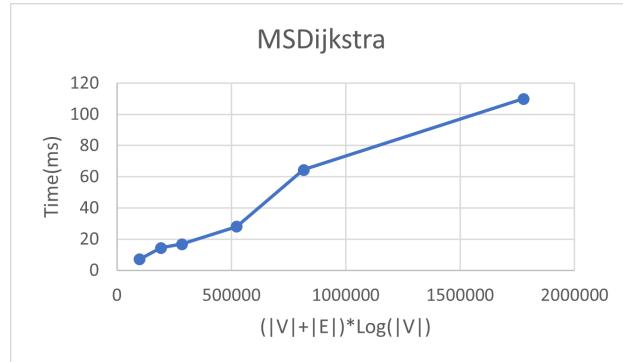


Figura 12: Tempo de execução do algoritmo Multi-Source Dijkstra em ordem a $(|V| + |E|) \cdot \log |V|$.

Como o gráfico indica, obtiveram-se resultados aproximadamente lineares em relação à sua complexidade temporal, pelo que se confirma a complexidade teórica analisada face aos grafos utilizados.

8.8 Solução do Problema

De forma a resolver o problema proposto, foram conjugados os algoritmos descritos acima: de modo a verificar se temos caminho de ida e de volta entre os centros de distribuição e aplicação aplica-se o algoritmo Kosaraju, obtendo-se as componentes fortemente conexas; De seguida, fez-se a distribuição dos centros de aplicação aos respetivos centros de distribuição, formando-se clusters, com a utilização do algoritmo Multi-source Dijkstra; posteriormente, para se obter o percurso desde os centros de distribuição aos respetivos centros de aplicação onde as carrinhas irão distribuir as vacinas, utilizou-se o algoritmo Nearest Neighbor. Uma vez que esta implementação final é uma junção, em sequência, de vários algoritmos, a ordem da sua complexidade temporal será igual à ordem da complexidade temporal do algoritmo com maior complexidade temporal. Assim, a complexidade temporal da implementação será da ordem de $O(\log |POI| \cdot |POI| \cdot ((|V| + |E|) \cdot \log |V|))$. Para confirmar esta complexidade, foi utilizado o algoritmo nove vezes em cada mapa (Porto, PortoSCC, Espinho, EspinhoSCC, Penafiel, PenafielSCC). O gráfico representado de seguida mostra os resultados obtidos

em ordem à complexidade temporal teórica esperada:

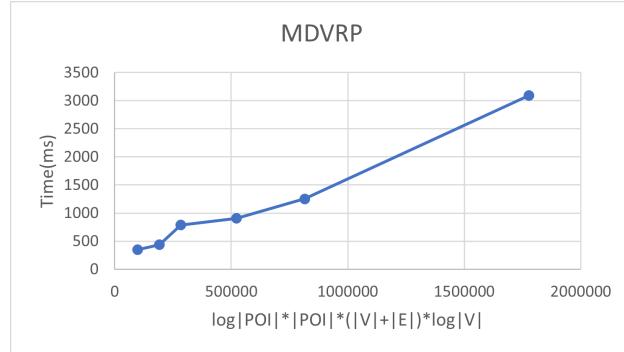


Figura 13: Tempo de execução da solução final em ordem a $O(\log |POI| \cdot |POI| \cdot ((|V| + |E|) \cdot \log |V|))$.

Os resultados obtidos foram aproximadamente lineares, pelo que se confirma a complexidade teórica analisada face aos grafos utilizados.

Capítulo 9

Análise da conectividade do grafo

De acordo com a informação dada no problema, pode haver obras na via pública que podem tornar algumas zonas inacessíveis. Mais do que isto, torna-se ainda necessário que, não só uma carrinha de distribuição consiga chegar aos centro de aplicação aos quais se pretende levar as vacinas, mas também que tenha caminho de volta. Para este fim, recorreu-se ao algoritmo de Kosaraju que divide o grafo dado em componentes fortemente conexas. Aplicando isto ao problema em questão, é possível associar a cada centro de distribuição os respetivos Centros de aplicação que, devido ao descrito em cima, têm de pertencer à mesma componente conexa.

Utilizando a interface do programa, é possível visualizar de forma distinta os vértices pertencentes à componente fortemente conexa maior dos vértices que pertencem às restantes componentes conexas, sendo estas últimas em grande número mas cada uma delas com reduzido número de vértices.

Nas imagens que se seguem, observam-se as componentes fortemente conexas do Porto, de Espinho e de Penafiel. Como já mencionado em Funcionalidades e cenários implementados, a componente maior é mostrada a amarelo e as restantes a azul, pelo que nos mapas da componente fortemente conexa os vértices estão todos a amarelo.

9.1 Mapa do Porto

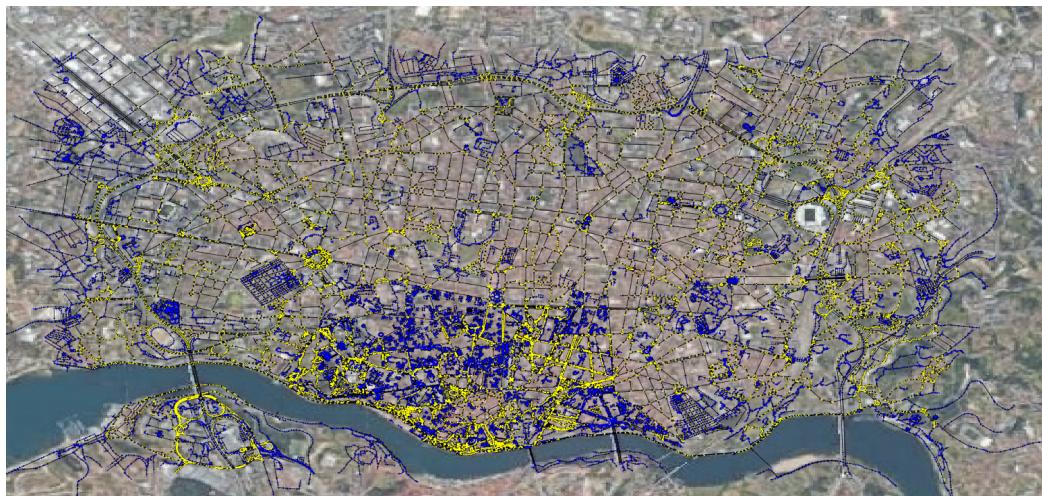


Figura 14: Componentes fortemente conexas no mapa do Porto completo.

9.2 Mapa da componente fortemente conexa do Porto



Figura 15: Componentes fortemente conexas no mapa da maior componente fortemente conexa do Porto.

9.3 Mapa de Espinho

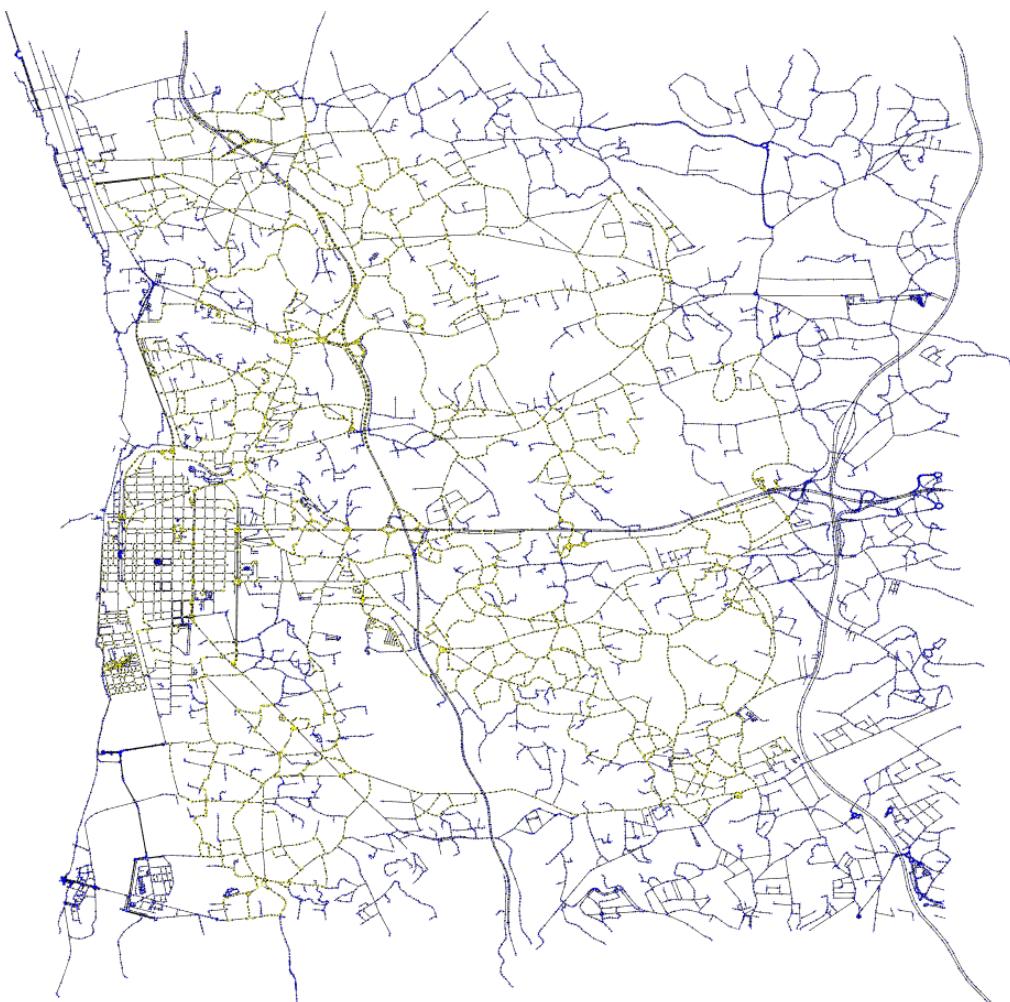


Figura 16: Componentes fortemente conexas no mapa de Espinho completo.

9.4 Mapa da componente fortemente conexa de Espinho



Figura 17: Componentes fortemente conexas no mapa da maior componente fortemente conexa de Espinho.

9.5 Mapa de Penafiel

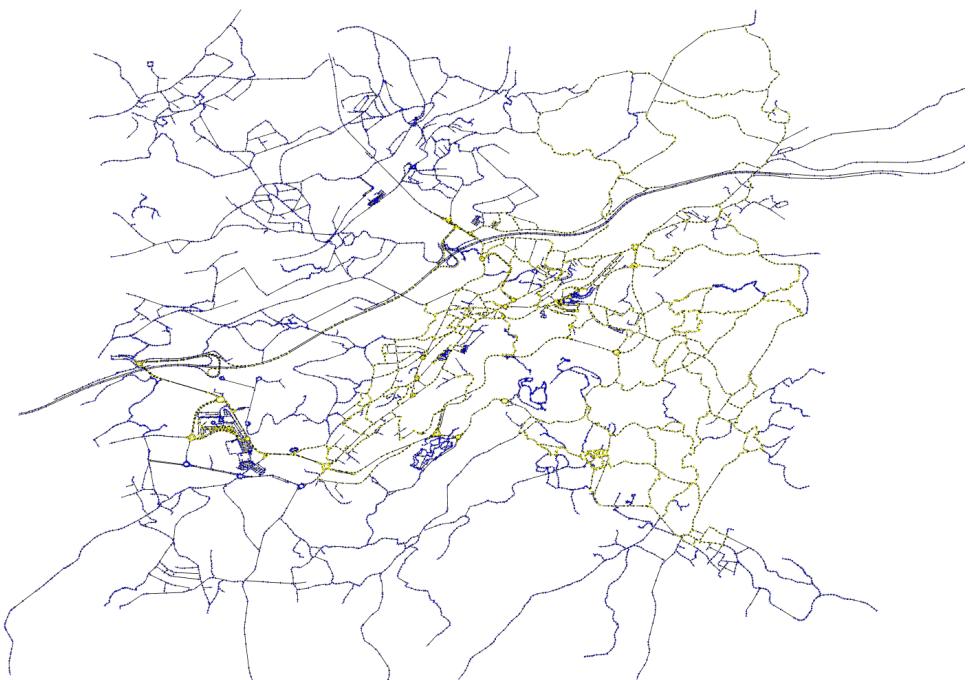


Figura 18: Componentes fortemente conexas no mapa de Penafiel completo.

9.6 Mapa da componente fortemente conexa de Penafiel

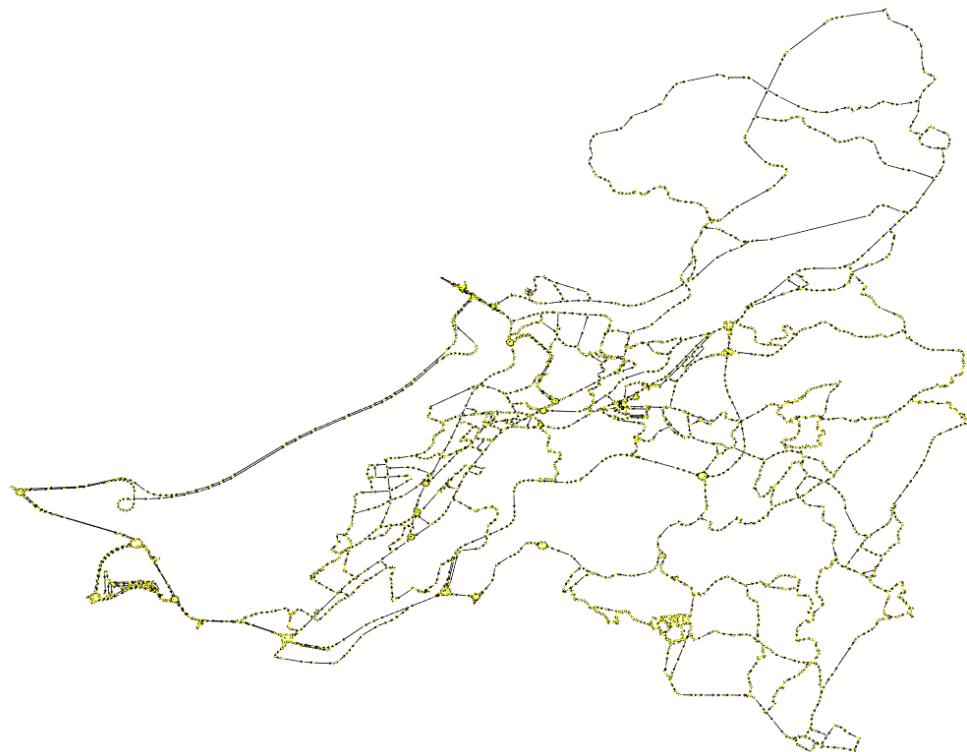


Figura 19: Componentes fortemente conexas no mapa da maior componente fortemente conexa de Penafiel.

Capítulo 10

Conclusão

// TODO Mudar esta conclusão De forma a resolver o problema proposto foram conjugados os algoritmos descritos acima, como se explica de seguida: de modo a verificar se temos caminho de ida e de volta entre os centros de distribuição e aplicação aplica-se o algoritmo Kosaraju, obtendo-se as componentes fortemente conexas; de seguida, faz-se a distribuição dos centros de aplicação pelos centros de distribuição em clusters, utilizando o algoritmo Multi-source Dijkstra; posteriormente, para se obter o percurso final, onde as carrinhas saem dos centros de distribuição e distribuem as vacinas àos respetivos centros de aplicação, utiliza-se o algoritmo Nearest Neighbor.

Capítulo 11

Contribuição

Todos os membros do grupo participaram ativamente na elaboração do trabalho, pelo que a cada um dos três membros é atribuída uma contribuição de $\frac{100}{3}\%$.

Capítulo 12

Bibliografia

- [1] Sharir, M. 1981. "A strong-connectivity algorithm and its application in data flow analysis." *Computers & Mathematics with Applications* 67-72.
- [2] Tarjan, R. E. 1972. "Depth-first search and linear graph algorithms." *SIAM Journal on Computing* 146-160.
- [3] Hart, Peter E., Nils J. Nilsson, and Bertrand Raphael. 1968. "A Formal Basis for the Heuristic determination of Minimum Cost Paths." *IEEE Transactions on Systems Science and Cybernetics* 4 (2): 100-107.
- [4] Spoerhase, Joachim, Thomas van Dijk, and Alexander Wolff. 2019/20. "Lecture 1. Introduction & Held-Karp-algorithm for TSP." *Slides for Advanced Algorithms course in Universität Würzburg*, 39-55.
- [5] Chanzy, P. 1993 "Range Search and Nearest Neighbor Search." *Master's Thesis, McGill University*.
- [6] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, e Clifford Stein. 2009. *Introduction to Algorithms*. 3rd. MIT Press.
- [7] Weiss, Mark Allen. 2013. *Data Structures and Algorithm analysis in C++*. 4th. Pearson.
- [8] Rossetti, R., A. P. Rocha, L. Ferreira, J. P. Fernandes, F. Ramos, G. Leão. 2021 *Slides for Concepção e Análise de Algoritmos course in FEUP, MIEIC*.