Mestrado Integrado em Engenharia Informática e Computação

Concepção e Análise de Algoritmos

VaccineRouter: transporte de vacinas entre centros de aplicação

2MIEIC05_G2

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

16 abril 2021

Conteúdo

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*

```
G = (V, E)

function DFS(G)
    for v ∈ V do
        visited(v) ← false
    for v ∈ V do
        if !visited(v) then
            DFS-VISIT(G, v)

function DFS-VISIT(G, v)
    visited(v) ← true
    pre-process(v)
    for w ∈ Adj(v) do
        if !visited(w) then
            DFS-VISIT(G, w)
    post-process(v)
```

3.1.2 *Breath-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 *Breath-First-Search*

```
G = (V, E)

function BFS(G)
    choose s from G
    visit(s)
    insert(Q, s)
    while !empty(Q) do
        v ← extract(Q)
        for e ∈ edges(v) do
            w ← destiny(e)
            if !visited(w) then
                visit(w)
                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*

$G = (V, E)$
stack stack
vector SCC
int component

function DFS1(G, s)
    visit(s)
    for $v \in Adj(s)$ **do**
        if $!visited(v)$ **then**

```
            DFS1(v)

function DFS2(G, component, s)
    visit(s) ← false
    SCC(component).push(s)
    for v ∈ Adj(s) do
        if !visited(v) then
            DFS2(G, component, s)

function KOSARAJU()
    for v ∈ V do
        visit(s) ← false
    for v ∈ V do
        DFS1(G, s)
    for v ∈ V do
        visit(s) ← false
    while !stack.empty do
        v ← stack.pop()
        if !visited(v) then
            DFS2(G, component, v)
        component + +
    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 duma 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*

```
G = (V, E)
stack stack
id ← 0
sccCount ← 0
```

```
function DFS(G, s)
    stack.push(s)
    id(s) ← id++
    low(s) ← id(s)
    onstack(s) ← true
    for v ∈ Adj(s) do
        if id(v) = NULL then
            DFS(v)
        if onstack(v) then
            low(s) ← min(low(s), low(v))
    if low(s) = id(s) then
        while (v ← stack.pop()) ≠ s do
            onstack(v) ← false
            low(v) ← id(s)
            SCC(v) ← s
        SCC(s) ← s

function Tarjan(G)
    for u ∈ V do
        id(u) ← NULL
        SCC(u) ← NULL
    for u ∈ V do
        if id(u) = NULL then
            DFS(G, u)
    return SCC
```

3.3 Caminho mais curto

Para resolvermos 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*

```
G = (V, E)
s ∈ V

function DIJKSTRA(G, s)                          ▷ using a priority queue
    for v ∈ V do
        dist(v) ← INF
        path(v) ← NULL
    dist(s) ← 0
    Q ← V                                        ▷ Q is a min-priority queue of dist
    while |Q| > 0 do
        v ← Q.extract_min()
        for u ∈ Adj(v) do
            dist_temp ← dist(v) + weight(v, u)
            if dist_temp < dist(u) then
                dist(u) ← dist_temp
                path(u) ← v
                Q.decrease_key(u, dist(u))
    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**

```
G = (V, E)
s ∈ V

function ASTAR(G, s)                              ▷ using a priority queue
    for v ∈ V do
        dist(v) ← INF
        hdist(v) ← INF
        path(v) ← NULL
    dist(s) ← 0
    hdist(s) ← h(s)
    Q ← V                                 ▷ Q is a min-priority queue of hdist
    while |Q| > 0 do
        v ← Q.extract_min()
        for u ∈ Adj(v) do
            dist_temp ← dist(v) + weight(v, u)
            if dist_temp < dist(u) then
                dist(u) ← dist_temp
                path(u) ← v
                Q.decrease_key(u, dist(u) + h(u))
    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*

$G = (V, E)$

function FLOYDWARSHALL(G)
  $distM \leftarrow matrix(|V|, |V|)$
  $fill(distM, INF)$
  for $e \in E$ **do**
    $distM[orig(e), dest(e)] \leftarrow dist(e)$
  for $i = 1$ to $|V|$ **do**
    $distM[i, i] \leftarrow 0$
  for $k = 1$ to $|V|$ **do**
    for $i = 1$ to $|V|$ **do**
      for $j = 1$ to $|V|$ **do**
        if $distM[i, j] > distM[i, k] + distM[k, j]$ **then**
          $distM[i, j] \leftarrow distM[i, k] + distM[k, j]$
  $return\ distM$

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*

```
G = (V, E)

function BellmanHeldKarp(G)
    s ← inicial vertex
    for v ∈ V do
        OPT[{v}, v] ← weight(s, v)
    for i = 2 to |V| do
        for S ⊆ V − s with |S| = i do
            for v ∈ S do
                OPT[S, v] ← min{OPT[S − v, u] + weight(u, v) — u ∈ S − v})
    return min{OPT[V − s, v] + weight(v, s) — v ∈ 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 algoritmo 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*

```
G = (V, E)

function NN(G)
    for v ∈ V do
        visited(v) ← false
    v ← select(V, 0)
    visited(v) ← true
    S ← ∅
    insert(S, v)
    count ← 0
    while count < size(V) do
        Q ← ∅
        for e ∈ edges(v) do
            insert(Q, (destiny(e), distance))
        while size(Q) > 0 do
            w ← extractMin(Q)
            if !visited(w) then
                increment(count)
                visited(w) ← true
                v ← w
                insert(S, v)
                break
    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. O método a ser utilizado será um algoritmo genético.

Com a distribuição feita (e até mesmo para a função de *fitness* para o algoritmo genético) 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*

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*

$G = (V, E)$
$S \subseteq V$

function MULTIDIJKSTRA(G, S)
  for $v \in V$ **do**
    $dist(v) \leftarrow INF$
    $path(v) \leftarrow NULL$
  for $s \in S$ **do**
    $dist(s) \leftarrow 0$
  $Q \leftarrow V$ ▷ Q is a min-priority queue of dist
  while $|Q| > 0$ **do**
    $v \leftarrow Q.extract\_min()$
    for $u \in Adj(v)$ **do**
      $dist\_temp \leftarrow dist(v) + weight(v, u)$
      if $dist\_temp < dist(u)$ **then**

$dist(u) \leftarrow dist\_temp$
$path(u) \leftarrow v$
$Q.decrease\_key(u, dist(u))$
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

Conclusão

Após análise do problema, este foi subdividido em fases que representam subproblemas mais simples. Sendo que, o nível de dificuldade de cada fase vai aumentando.

Através desta divisão surgiram então subproblemas mais simples como acessibilidade, conectividade forte e caminho mais curto, mas também problemas mais complexos (*NP-hard*) como Caixeiro Viajante (*TSP*) e roteamento de veículos (*VRP*) que só poderão ser resolvidos em tempo útil fazendo uso de heurísticas, abordagens gananciosas que mesmo não garantindo a solução ótima, garantam uma solução razoável, e ainda de técnicas mais avançadas, nomeadamente algoritmos genéticos. Avaliámos ainda as diferentes complexidades, temporais e espaciais, por forma a escolher os melhores algoritmos para resolver cada um dos subproblemas.

O desenvolvimento deste projeto irá fazer-se pela progressiva implementação das fases enunciadas, fazendo uso dos algoritmos previamente estudados.

Capítulo 6

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 7

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.*