U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Concepção e Análise de Algoritmos

Mestrado Integrado em Engenharia Informática e Computação

TEMA 7

HealthCarrier

Transportes de doentes entre centros de saúde

Turma 1, Grupo E
João Paulo Gomes Torres Abelha – **up201706412**
João Rafael Gomes Varela – **up201706072**
Vítor Hugo Pereira Barbosa – **up201703591**

Data de Entrega:
28 de maio, 2019

Índice

1. Estruturas de Dados utilizadas

Estrutura de dados utilizadas para representação do grafo:

1. Vertex

Na classe **Vertex** é usada para representar os *nodes* do grafo. Tem como atributos a sua **posição** no mapa lido do ficheiro, a **info**, que no nosso caso representa o identificador único do vértice, e um vetor de **arestas adjacentes**. Desta forma, a estrutura usada para a representação de um mapa é da forma da lista de adjacências, uma vez que que a partir de cada vértice tem-se acesso aos respetivos adjacentes, sendo o espaço usado para tal O(|E| + |V|) .

Após serem analisados vários mapas, concluiu-se facilmente que o número de arestas aproxima-se relativamente bem ao número de vértices e, por esta razão, a estrutura de dados usada foi a anterior usada, em detrimento da matriz de adjacências. Nesta última o espaço ocupado é de O(|V|^2) , sendo este apropriado para grafos densos, o que foi demonstrado não ser o caso. A título exemplificativo, nos mapas que foram dados, calculando Number Of nodes/Number of Edges ≅ 1.

As classes que derivam do vértice são **HealthCenter e Hospital**, uma vez que estes não passam de um tipo de vértice particular e com especial relevância no contexto do problema. A classe HealthCenter representa um vértice onde os **doentes** devem ir ser recolhidos e o Hospital o vértice **destino** destes. Por último, haverá um vértice que corresponderá à garagem que representa o local de partida das ambulâncias para irem buscar posteriormente os doentes nos pontos de interesse assinalados.

2. Edge

A classe **Edge** representa a forma como os vértices se relacionam, sendo estes unidos pelas arestas. Cada aresta tem a si associada um **peso** que, dentro do problema, representa o **tempo** necessário para ir de um vértice para o outro (foi considerada uma relação linear entre o tempo e a distância entre vértices). Para além disso cada aresta contém um apontador para um vértice de **destino**, o local a que essa aresta leva.

3. Graph

A classe **Graph** é essencialmente constituída por um **conjunto de vértices** e por algumas **operações** de manipulação desta estrutura de dados como adicionar ou eliminar vértices e arestas.

Outras classes usadas:

Outras classes importante no contexto do exercício, para além de Hospital e HealthCenter, são a class Ambulance e a classe Patient. A **Ambulance** irá conter o **capacidade** da ambulância, os **doentes** que esta transporta, o vértice de **partida** (garagem), podendo ainda apresentar um **tipo**, transportando apenas um tipo específico de paciente, e , consequentemente, apenas irá a recolher doentes a centros de saúde específicos e o mesmo pode ser dito para os hospitais, visto que estes devem ter o tipo de tratamento compatível com o tipo de problema do paciente (notar que esta última parte apenas torna-se relevante para a última iteração). Por último um **Patient** corresponderá a um doente que espera ser recolhido num determinado centro de saúde e levado a um hospital. Na terceira iteração é tido em conta o seu tipo de problema, sendo que este pode ter a si associado um tempo específico que corresponde ao tempo máximo que pode resistir sem ter chegado ao destino. Isto leva a que, na terceira iteração, a recolha nos HealthCenter e a sua entrega num Hospital tenha de ser bem gerida de forma a que sobreviva o maior número de doentes.

A classe **CompleteGraph** possui um **grafo original** e o **grafo pré-processado**, bem como as funções que realizam as **iterações** em que o problema foi dividido. É esta classe que se abstrai da implementação dos algoritmos "primitivos", tentando resolver o problema apresentado.

A classe **GUI** representa uma interface para o utilizador poder manipular grafos (através dos algoritmos implementados) existindo uma série de caso de utilização, que o próprio pode escolher.

Por último, foram também criadas várias classes que representam os **algoritmos** implementados (como a class DFS, AStar, etc), que contêm toda a informação necessária para que o algoritmo posso correr com sucesso.

É de notar que, de modo a aumentar a eficiência e velocidade dos algoritmos, tiveram-se em conta alguns fatores como o tipo de estruturas usadas para guardar os elementos do problema, dando-se prioridade ao uso de ***maps*** e ***sets*** (quando pertinente).

2. Conectividade dos Grafos

A conectividade dos grafos é um fator de relevância elevada na resolução de problemas de grafos. Um grafo que seja uma boa representação de um mapa real, caso seja dirigido, deve ser fortemente conexo, para garantir que se pode a partir de um local qualquer chegar a outro qualquer, situação presente nos mapas. Mesmo que nem todo o mapa seja assim devemos no mínimo garantir que os pontos de interesse podem ser alcançados para que se possa resolver o algoritmo. Para este estudo foram realizados dois algoritmos diferentes, sabendo que o grafo é dirigido:

(1) **Kosaraju** : verifica se o grafo é todo ele fortemente conexo.
(2) **DFS** : permite verificar se é possível chegar de um determinado ponto de interesse a outro.

Os algoritmos deverão ter em conta que os mapas podem não permitir a conexão entre vários ponto de interesse, por restrição de sentido ou inacessibilidade e deve ser tomada uma decisão em conformidade. Estes algoritmos serão explicados com maior detalhe num dos pontos seguintes.

3. Casos de Utilização

Implementou-se uma interface simples mas bastante completa, que permite ao utilizador interagir com todas as funcionalidades do programa. Este poderá **carregar mapas** que estão guardados em ficheiros e que serão traduzidos em grafos automaticamente, e poderá também **aplicar operações** que dizem respeito ao problema, com algumas variantes entre si.

Através de um **menu** o utilizador pode escolher que ação quer realizar, e no caso de aplicar operações, poderá selecionar de entre um grande leque de opções:

Carregar Mapa - permite ao utilizador carregar um grafo (onde poderá depois aplicar outras operações) graficamente (usando o software *GraphViewer*).

Verificar Conectividade - verifica se o grafo é uma boa representação de uma mapa real**:** um mapa real permite chegar a qualquer ponto dele a partir de outro.

Tempo de viagem - será possível verificar à priori se é possível realizar o transporte de todos os doentes para o seu respetivo hospital e, neste caso, saber-se-á também o tempo de viagem.

Pré processamento - através deste pré processamento serão eliminados vértices "redundantes" do grafo, tornado-o mais simples.

Caminho ótimo considerando uma ambulância de capacidade infinita - o utilizador pode especificar o *node* de partida da ambulância, os centros de saúde e o hospital. Ser-lhe-á retornado o caminho ótimo a ser realizado pela ambulância, de forma a percorrer todos os centros de saúde.

Caminho ótimo considerando várias ambulâncias de capacidade limitada - o utilizador pode especificar o *node* de partida da ambulância, os centros de saúde e o hospital. Ser-lhe-ão retornados os caminhos ótimos a serem realizados pelas várias ambulâncias, de forma a serem recolhidos todos os doentes.

Caminho ótimo considerando várias ambulâncias de capacidade limitada e tendo em conta os tipos de tratamento - o utilizador pode especificar o *node* de partida da ambulância, os centros de saúde e o hospital. Ser-lhe-ão retornados os caminhos ótimos a serem realizados pelas várias ambulâncias, de forma a serem recolhidos todos os doentes.

Após aplicadas estas operações, poderão observar-se alguns dados de saída, de modo a indicar ao utilizador como foram utilizados os recursos de entrada, como foi resolvido o problema, e quanto tempo de transporte é estimado pela solução.

É de notar que a aplicação garante o *input* válido do utilizador e que é possível visualizar qualquer grafo, antes ou depois de aplicadas operações, através do software *GraphViewer*.

4. Algoritmos Implementados

Foram analisados vários grafos, que representam mapas reais, e todos eles apresentam igual ordem de grandeza, podendo ser o grafo considerado esparso. Consequentemente, serão usadas listas ligadas para construir o grafo e o algoritmo para encontrar as distâncias mais curtas entre vértices não será o de *Floyd-Warshall*. Surgindo , assim, duas possibilidades: o uso do algoritmo de ***Dijkstra*** ou o ***A****.

1. *Depth First Search* (DFS)

Este algoritmo é imprescindível para o bom funcionamento do programa desenvolvido sendo usado para a verificação de um determinado caminho entre dois pontos existe.

O outro algoritmo que poderíamos ter usado em vez deste seria *Breadth First Search* (BFS). Ambos têm igual complexidade temporal, portanto, o fator decisivo foi o espacial. Apesar do BFS dar a solução ótima, o espaço usado seria superior ao do DFS, O(|V| + |E|), portanto o DFS é bastante melhor espacialmente, mas pode encontrar uma solução subótima. Notar que os vértices adjacentes são aqueles explorados primeiro. Como é usado para verificar a conectividade e se um caminho existe não há a preocupação de encontrar o caminho ótimo, sendo portanto a DFS preferível.

Complexidade Temporal: O(V + E)

Complexidade Espacial: O(V)

Pseudocódigo:

Algorithm 1 DFS

```
1:  procedure DFS(Vertex start)
2:      Solution ← []    //vertexes visited
3:      VertexS ← []     //vertex stack
4:      VertexS.push(start)
5:
6:      while not empty VertexS do
7:          V ← VertexS.pop()
8:
9:          if V not visited do
10:             mark V as visited
11:             Solution.push(V)
12:
13:         for each edge W of V.getAdjacentVertexes() do
14:             if W not visited do
15:                 VertexS.push(W)
16:
17:     return Solution
18:
```

2. Kosaraju

Para testar se um grafo é fortemente conexo usou-se o algoritmo *Kosaraju*. Este algoritmo usa a definição de um grafo transposto, decidindo resumir esse subproblema a outra classe que cujo o output é um grafo transposto do grafo que recebeu de input. Depois de abstrairmos este problema basta aplicar um DFS ao grafo inicial e verificar se todos são visitados, caso sejam, efetuar a transposta, e para esse grafo efetuar DFS e fazer a mesma verificação.

Optou-se pelo uso deste algoritmo devido à estrutura de dados que temos, mais concretamente, lista de adjacências, permitindo que se faça o algoritmo em tempo linear O(|V| + |E|).

Complexidade Temporal: O(V + E)

Complexidade Espacial: O(V + E)

Pseudocódigo:

Algorithm 1 Kosaraju

```
1:  procedure KOSARAJU(Graph G)
2:      for each vertex in G do
3:          vertex.visited ← false
4:      DFS(random vertex of G)
5:      for each vertex in G do
6:          if vertex.visited equals false
7:              return false
8:
9:      let GT = Transpose of G
10:     for each vertex in GT do
11:         vertex.visited ← false
12:     DFS(random vertex of GT)
13:     for each vertex in GT do
14:         if vertex.visited equals false
15:             return false
16:
17:     return true
18:
19: procedure DFS(Vertex vertex)
20:     if !vertex.visited
21:         vertex.visited ← true
22:
23:     for each edge in vertex do
24:         DFS(edge.destiny)
25:
26:
```

3. *A Star*

Este algoritmo é usado para o cálculo do tempo necessário para uma viagem entre dois pontos. O outro algoritmo que poderia ser usado seria o Dijkstra que, apesar de apresentar igual complexidade temporal, acaba por não ser preferível devido ao uso de uma heurística de aproximação, que tem em conta a distância euclidiana entre dois pontos do mapa. Ou seja, o *A Star*, escolhe o próximo vértice a explorar com base no custo até ao momento, e a distância euclidiana para os possíveis destinos.

Complexidade Temporal: O(E)

Complexidade Espacial: O((V + E)*log(V))

Pseudocódigo:

Algorithm 1 AStar

```
1:  procedure ASTAR(Vertex start, Vertex finish)
2:      PriorityQ ← []    //vertex priority queue
3:      PriorityQ.populate()
4:
5:      while not empty PriorityQ do
6:          V ← PriorityQ.pop()
7:
8:          if V = finish do
9:              Solution ← buildPath()
10:             break
11:
12:         if V.getAdjacentEdges() empty do
13:             PriorityQ.erase(V)
14:             continue
15:
16:         for each edge W of V.getAdjacentVertexes() do
17:             if weight(V) is greater than weight(W) - weight(W, finish) +
    weight(V, W) + weight(V, finish) then
18:                 weight(V) ← weight(W) − weight(W, finish) + weight(V, W) +
    weight(V, finish)
19:                 path(V) ← W
20:                 updateVertexOnPriorityQ(V)
21:
22:     return Solution
23:
```

4. *Nearest Neighbor*

Este algoritmo é um dos métodos heurísticos mais intuitivos para a resolução do problema do Caixeiro Viajante. O algoritmo segue um procedimento ganancioso muito simples. Em cada vértice é calculado o próximo pela procura do vértice mais próximo no conjunto de vértice ainda por visitar. Este algoritmo é importante no problema para descobrir a ordem pela qual devem ser visitados os centros de saúde.

Complexidade Temporal: O(V)

Complexidade Espacial: O(V)

Pseudocódigo:

Algorithm 1 Nearest Neighbor

```
1:  procedure NearestNeighbor(Graph G, Vertex s, Vertex f, Vertex[] to-
    Visit)
2:      let order = []
3:      CalcVisitOrder(G, s, f, toVisit, order)
4:
5:  procedure CalcVisitOrder(Graph G, Vertex s, Vertex f, Vertex[] to-
    Visit, Vertex[] order)
6:      if toVisit is empty do
7:          return
8:
9:      order.insert(s)
10:
11:     let u = s.findNearest (u is the nearest, unvisited, neighbor of s)
12:     toVisit.remove(u)
13:     CalcVisitOrder(G, s, f, toVisit, order)
```

5. *2-Opt*

Este algoritmo, ao contrário de todos discutidos previamente, parte de uma solução inicial (seja esta solução aleatória ou, idealmente, calculada com um algoritmo leve e rápido como a heurística de vizinho mais próximo) e, iterativamente, realiza modificações sobre essa solução de forma a tentar obter uma solução melhor. Este algoritmo é conhecido por ser de fácil implementação, rápido e por realizar uma otimização relativamente boa com pouco trabalho computacional.

Complexidade Espacial: O(T) : T -> tamanho de *order*

Pseudocódigo:

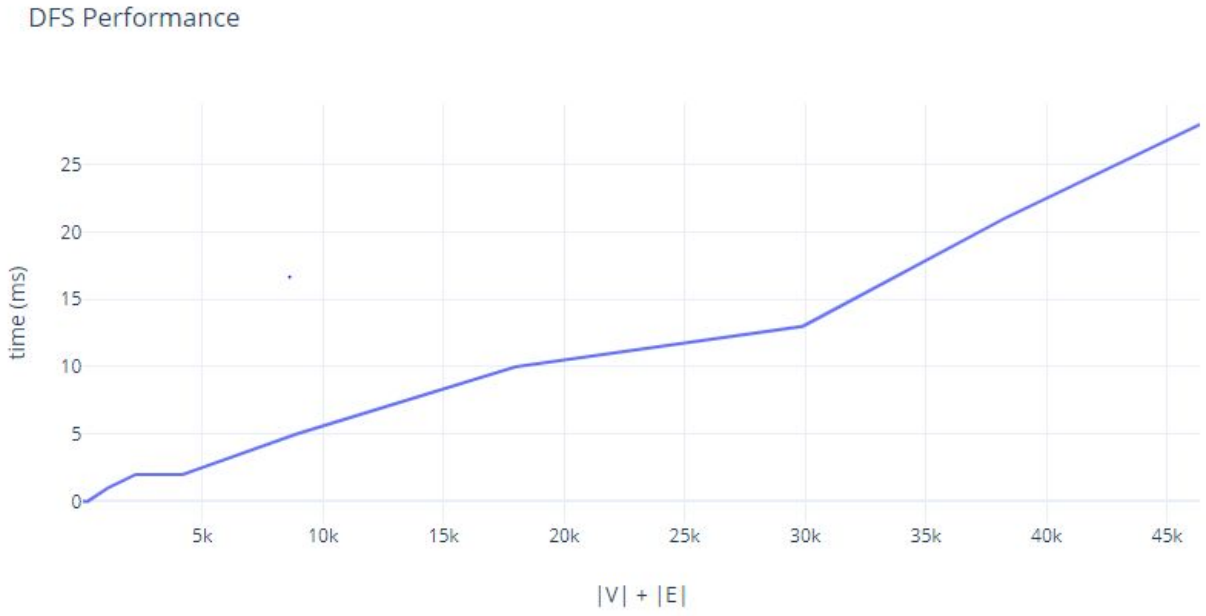
Algorithm 1 2Opt

```
1:  procedure 2OPT(Vertex[] order)
2:      let bestVisitOrder = order
3:      while visitOrderImproves do
4:          let bestCost = getTotalCost(bestVisitOrder)
5:          for i = 1 to numberOfNodesToBeSwapped - 1 do
6:              for j = i + 1 to numberOfNodesToBeSwapped do
7:                  let newVisitOrder = 2optSwap(bestVisitOrder, i, j)
8:                  let newCost = getTotalCost(newVisitOrder)
9:                  if newCost < bestCost
10:                     bestCost ← newCost
11:                     bestVisitOrder ← newVisitOrder
12:
13: procedure 2OPTSWAP(Vertex[] order, int i, int j)
14:     while i < j do
15:         swap(order[i], order[j])
16:         i++
17:         j++
18:     return order
```

5. Análise da complexidade dos Algoritmos

1. *Depth First Search* (DFS)

O algoritmo vai percorrer todos os vértices (acessíveis) do grafo (O(V)) e, para além disso, para cada vértice, percorre todas as suas arestas adjacentes (O(E)). Combinando ambos os trabalhos obtém-se uma complexidade temporal **O(|V| + |E|)**. Este resultado foi confirmado, realizando-se vários testes sobre grafos que permitiram construir o gráfico seguinte.

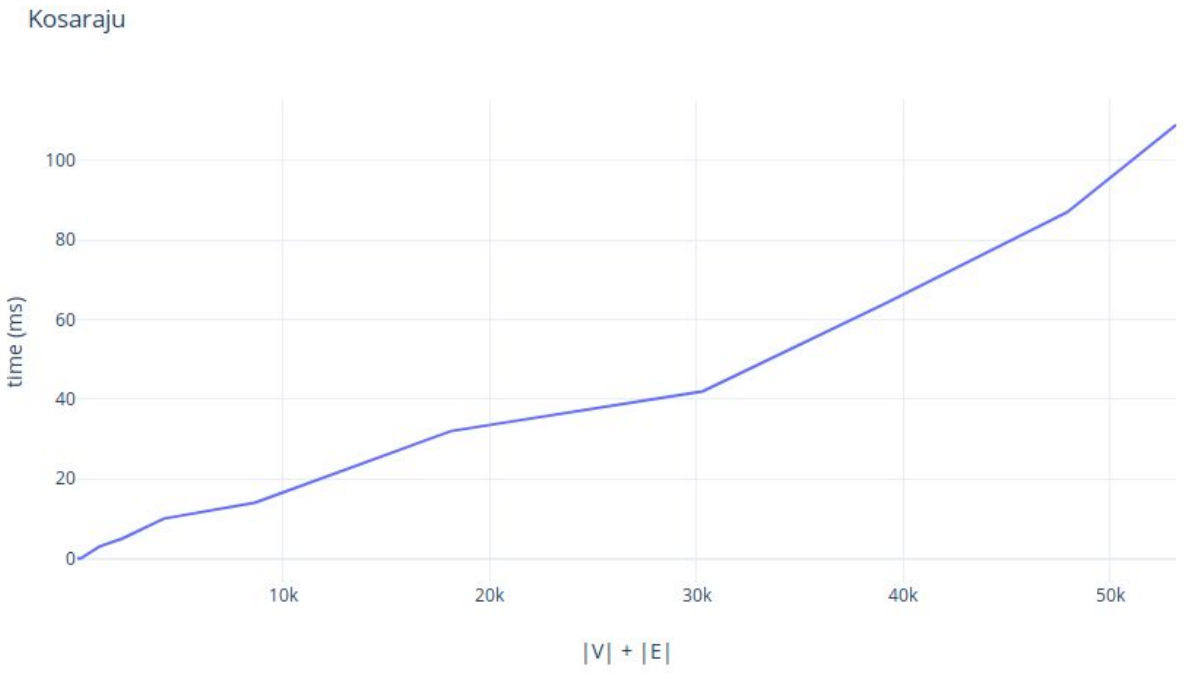


É de notar que foram também realizados testes sobre o algoritmo *Breadth First Search*, mas concluiu-se que ambos têm uma performance temporal aproximadamente igual através dos testes que estão representados no seguinte gráfico.

2. *Kosaraju*

Este algoritmo aplica um DFS ao grafo inicial cuja complexidade temporal já foi verificada (O(|V| + |E|)). Após isto, é feita a transposta do grafo (O(|V| + |E|)) e depois aplica-se de novo um DFS (O(|V| + |E|)). Somando os três componentes obtém-se uma complexidade O(3 * (|V| + |E|)) ≅ **O(|V| + |E|)**.

Esta relação linear foi verificada através de medições de tempo de execução, cujos resultados se apresenta no seguinte gráfico.

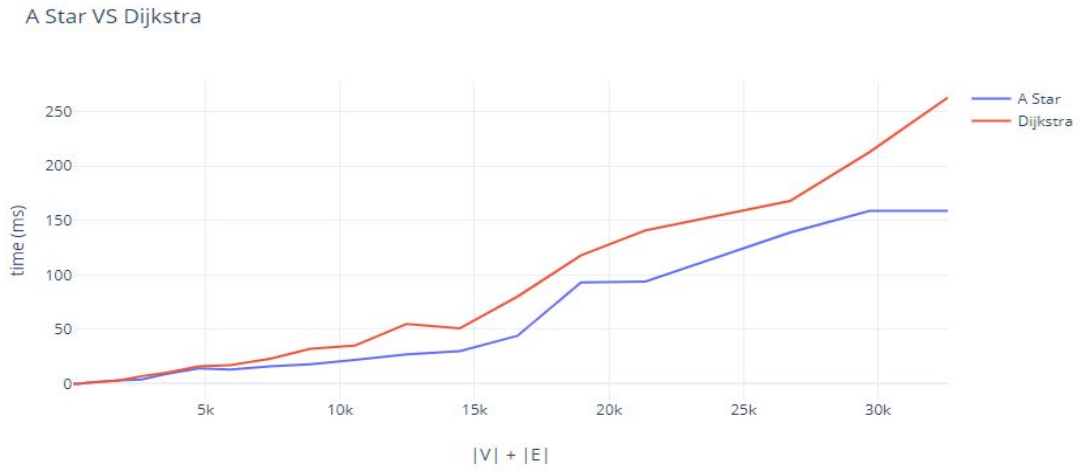


3. *A Star*

O *A Star* recorre ao uso de uma fila de prioridades de modo a melhorar a eficiência do algoritmo. Esta implica, em cada inserção, uma complexidade de O(log n) devido à reordenação dos seus elementos. O algoritmo em si, percorre um caminho de vértices (O(V)) e verifica cada um dos seus vértices adjacentes (O(E)), possivelmente inserindo-o na fila de prioridades (O(log n)), totalizando assim uma complexidade temporal de **O(|V|+|E|*log|V|)**.

Este resultado pode-se confirmar observando os dados seguintes, que representam a eficiência temporal do algoritmo.

Outro algoritmo que poderia ser utilizado ao invés do *A Star* seria o *Dijkstra*, que teoricamente tem a mesma complexidade que o *A Star.* Para tomar uma decisão, foram feitas medições também relativas à performance do *DIjkstra* que resultaram no gráfico seguinte.



Através da análise deste grafo conclui-se que o *A Star* tem uma performance melhor relativamente ao *Dijkstra*. Isto deve-se à heurística de otimização utilizada pelo algoritmo *A Star.*

6. Soluções Implementadas

Para se responder aos problemas apresentados por cada iteração, foram usados os algoritmos descritos anteriormente combinando-os e impondo algumas restrições, de modo a adaptá-los ao contexto do tema em questão.

1. Uma ambulância com capacidade infinita e vários centros de saúde

Nesta primeira iteração, uma ambulância vai buscar os vários doentes aos centros de saúde levando-os em seguida ao hospital existente. Algoritmos usados: Nearest neighbor (que por si só usa A* e DFS) e o two opt para melhorar a solução anteriormente encontrada. Assim garantimos que a ambulância percorre os vários centros de saúde, no menor tempo possível, uma vez que aqui já por si só é minimizado o número de ambulâncias, importando, então, minimizar o tempo a efetuar a rota.

2. Várias ambulâncias com capacidade limitada e vários centros de saúde

Na segunda iteração, são usadas o menor número de ambulâncias para ir buscar os vários doentes aos vários pontos de interesse, transportando-os para o hospital. Esta iteração vai usar os mesmo algoritmos que a anterior, mas agora será importante contabilizar o número e não apenas a ordem de visitas. Cada ambulância irá ter uma capacidade >= 1 e capacidade <= 10. Irá ter de ser atribuída a cada ambulância quais os centros de saúde a visitar, tendo em conta a capacidade e a melhor ordem de visita. Enquanto existirem pacientes a recolher e a capacidade duma ambulância ainda for maior que zero é realizado o algoritmo, sendo necessário ser verificado quando o centro de saúde deixa de um ponto a visitar, isto é, quando deixa de ter lá pacientes. Podem acontecer quatro situações no pré processamento da iteração:

(a) A ambulância fica cheia e o centro de saúde deixa de ser um ponto a visitar (fica vazio);

(b) a ambulância fica cheia, mas o centro de saúde continua com doentes (é necessário outra ambulância);

(c) a ambulância não fica cheia, mas o centro de saúde fica vazio, então, a mesma ambulância vai recolher ainda mais pacientes a outro centro de saúde até que fique cheia;

(d) a ambulância não fica cheia mas já foram todos os pacientes recolhidos - não há mais centros de saúde (acaba a iteração).

3. Vários tipos de ambulância com capacidade limitada

Vários tipos de ambulância com capacidade limitada, sendo necessário considerar vários tipos de doentes, sendo estes recolhidos por uma ambulância de um tipo específico respetivo, levando-o também para um hospital respetivo específico, isto é, que apresenta o tratamento para o seu tipo de problema. Para além disso, é necessário considerar o fator de tempo, relevante para os doentes. Cada doente irá ter associado um tempo que será o tempo máximo que a ambulância pode demorar a levá-lo ao hospital que trata a sua doença. A solução passa também por criar subgrafos que representem um determinado "tipo" - partida da ambulância de um determinado tipo, centros de saúde com doentes desse tipo e o hospital que trata esse tipo. Esta parte, requer o uso da iteração anterior, para cada um dos tipos. Depois disto, basta considerar as restrições temporais, adicionando à solução os algoritmos que permitem calcular o melhor tipo de decisão. Assim, é necessário implementar um algoritmo de decisão (pré processamento da iteração), mais especificamente, que calcule se a ambulância deverá levar os doentes ao hospital correspondente imediatamente ou se ainda poderá, caso tenha capacidade suficiente, recolher outro(s). A cada iteração podem acontecer várias situações: semelhantes às anteriores para cada subgrafo tipo, acrescentando a restrição de que a ambulância não se dirige para o hospital apenas caso fique cheia, mas também sempre que o tempo não permita ir a outro centra de saúde. Assim, as situações (a) ,(c) e (d) mantêm-se e a à (b) é acrescentada a restrição descrita.

7. Conclusão

Os objetivos propostos para a entrega do trabalho foram conseguidos de forma bem sucedida e grande parte dos temas abordados nas aulas foram aplicados, devido à abordagem bastante teórica do trabalho. Desta forma, não só se consolidou melhor a matéria como foi possível aprofundá-la e aplicá-la.

Implementaram-se correta e eficientemente vários algoritmos abordados no decorrer na unidade curricular (e outros que não foram abordados), desde alguns algoritmos mais simples como DFS (*Depth-first Search*) até alguns mais complexos como *A Star* e como o algoritmo heurístico *Nearest Neighbor* utilizado para resolver o problema, que é uma variante do *Problema do Caixeiro Viajante*.

Inerente aos vários algoritmos utilizados, estão várias estratégias de concessão de algoritmos abordados no início da unidade curricular como algoritmos gananciosos e backtracking.

Na decisão sobre quais os algoritmos a utilizar, serviram de apoio análises teóricas e empíricas de complexidades temporais e espaciais dos diferentes algoritmos que podem ajudar na conceção de um resultado, obtendo-se gráficos que permitem comparar performances temporais e , desta forma, escolher o melhor algoritmo possível.

Na escolha das estruturas de dados que permitem resolver o problema, teve-se sempre em conta a forma como estes podem alterar a eficiência das soluções implementadas, tentando assim garantir a melhor solução possível.

Embora se tenta tentado simplificar o problema proposto, através de uma abordagem *bottom-up*, dividindo o problema em iterações, enfrentaram-se várias dificuldades ao longo da implementação da solução, desde dúvidas sobre quais os melhores algoritmos a utilizar até ao modo de estruturação de dados e do código, de forma a poder obter os melhores resultados possíveis.

Assim, conclui-se também que o problema apresentado retrata uma área realmente difícil e complexa, onde todos os detalhes importam, devido à grande dimensão do problema. Para além disso, no tema em concreto pode-se mesmo dizer que o algoritmo implementado para o resolver é crucial na vida das pessoas, reforçando a importância da eficiência.

8. Bibliografia e Referências

- Apresentações das Aulas Teóricas de Conceção e Análise de Algoritmo 2018, da autoria da Professora Doutora Liliana Ferreira, Professor Doutor João Pascoal Faria e Professor Doutor Rosaldo Rossetti.
- *Dijkstra Algorithm* - https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
- *Floyd-Warshall Algorithm* - https://pt.wikipedia.org/wiki/Algoritmo_de_Floyd-Warshall
- *Travelling Salesman Problem* - https://en.wikipedia.org/wiki/Travelling_salesman_problem
- *Travelling Salesman Problem Visualization* - *https://youtu.be/SC5CX8drAtU*
- *Shortest Path Problem* - https://en.wikipedia.org/wiki/Shortest_path_problem
- *Nearest Neighbor Algorithm* - https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm
- *Opt-2 Algorithm* - https://en.wikipedia.org/wiki/2-opt
- *The Traveling Salesman Problem: A Case Study in Local Optimization* - http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.70.7639&rep=rep1&type=pdf
- T.H. Cormen, C. E. Leiserson, R. L. Rivest , C. Stein. Introduction to Algorithms, 3rd Edition. MIT Press, 2009
- Thomas H. Cormen... [et al.]; Introduction to algorithms. ISBN: 978-0-262-53305-8
- Steven S. Skiena; The algorithm design manual. ISBN: 0-387-94860-0
- Sedgewick, Robert; Algorithms in C++ Part 5: Graph Algorithms, 3/E, Addison-Wesley Professional, 2001. ISBN: 0201361183