2018/19


# SchoolBus: Transporte Escolar

Tema 5

Conceção e Análise de Algoritmos

Grupo H – Turma 2

Mestrado Integrado em Engenharia Informática e Computação


| | |
|---|---|
| Miguel Delgado Pinto | up201706156@fe.up.pt |
| Nuno Miguel Teixeira Cardoso | up201706162@fe.up.pt |
| Pedro Leite Galvão | up201700488@fe.up.pt |

# Índice

# 1. Introdução e Descrição Sucinta do Problema

Os autocarros escolares são uma forma segura e cómoda das crianças se deslocarem de casa para a escola e vice-versa. Permitem poupar tempo aos pais dos alunos e, devido a ser um transporte público, contribuem para a preservação do meio ambiente. Este tipo de serviço pode ser oferecido pela própria escola, ou por empresas especializadas.

Neste trabalho, pretende-se implementar um sistema que permita a gestão de transportes escolares por uma empresa, definindo a melhor rota para cada autocarro.

Divide-se o problema, então, em três iterações:

## 1.1 - Iteração 1: autocarro único de capacidade ilimitada para apenas uma escola

Inicialmente, considera-se apenas um autocarro escolar de capacidade de passageiros ilimitada. O objetivo principal neste momento é a definição do caminho mais curto que se inicia na garagem onde o transporte se encontra estacionado, passa por todas as habitações dos estudantes e termina o seu percurso na escola. Posteriormente, será também necessário definir a rota mais curta no sentido inverso.

De realçar que eventuais percalços, como obras e estreitamento das ruas podem tornar inacessíveis certas vias de circulação, impossibilitando o acesso à residência de certos estudantes. Torna-se, portanto, necessário e essencial avaliar a conectividade do grafo, de modo a identificar zonas de pouca acessibilidade.

## 1.2 - Iteração 2: vários autocarros de capacidade limitada para uma escola

A partir desta iteração é considerado o número máximo de estudantes que um autocarro pode transportar. Devido a esta nova condição terão de ser considerados dois fatores: custo por cada autocarro e distância total percorrida pelos autocarros. O objetivo passa por minimizar cada um destes parâmetros, dando prioridade à minimização do custo.

## NOTA - Iteração 1.5: vários autocarros de capacidade ilimitada para apenas uma escola

Inicialmente foi ponderada a utilização de uma iteração intermédia, entre a primeira e a segunda iterações referidas em cima, que considerava a existência de vários autocarros de capacidade ainda ilimitada, a fim de cobrir grandes áreas em tempos reduzidos. No entanto, para efeitos de simplificação e por se ter concluído

que a passagem de uma iteração para a outra não se justificava devido ao número mínimo de alterações verificadas, decidiu-se manter apenas as três iterações mencionadas.

# 2. Formalização do Problema

## 2.1 - Dados de Entrada

- $G = (V, E)$ - Grafo direcionado com vértices correspondendo a endereços de um mapa e arestas correspondendo a vias. Cada aresta $E_i$ deve ter um peso $w_i$ associado que corresponde a uma distância.

- $K$ - Conjunto de endereços (vértices) das crianças que utilizam o serviço.

- $S$ - Endereços de uma ou mais escolas.

- $Gar$ - Endereço da garagem.

- $O$ - Conjunto de vias que estão obstruídas.

- $N$ - Número de veículos disponíveis.

- $C$ - Capacidade dos veículos.

## 2.2 - Dados de Saída

- $R$ - Conjunto de rotas a serem traçadas por cada veículo. Cada rota será um conjunto ordenado de arestas e corresponde a exatamente um veículo.

- $Cost$ - Valor do custo total para a empresa de todas as rotas efetuadas por todos os autocarros.

- $D$ - Conjunto das distâncias a serem percorridas por cada veículo.

- $T$ - Conjunto dos tempos médios de viagem para cada veículo (em minutos).

### 2.3 - Restrições

- $\forall e \in E, \quad peso(e) \in \mathbb{R} \quad \wedge \quad peso(e) > 0$

Os pesos de cada aresta no grafo são números maiores que 0, não necessariamente inteiros.

- $C \in \mathbb{N}$

C é um número inteiro maior que 0.

- $Cost,\ D,\ T \in \mathbb{R} \quad \wedge \quad Cost,\ D,\ T \ >= \ 0$

Cost, D e T são números maiores ou iguais a 0, não necessariamente inteiros.

### 2.4 - Funções Objetivo

O algoritmo a ser implementado deve minimizar o número de autocarros utilizados e a soma das distâncias percorridas por cada um deles. Portanto, as funções objetivo em ordem de prioridade são definidas como:

- $f = |R|$
- $g = \sum_{r \in R} \sum_{e \in r} w(e)$

# 3. Estruturas de Dados

Quanto às estruturas de dados essenciais para o projeto (aquelas que representam um mapa e as casas e as estradas que o constituem), from desenvolvidas as classes **Graph**, **Vertex** e **Edge**, à semelhança das aulas práticas. No entanto, estas classes foram todas desenvolvidas de raiz, de modo a que se adequassem às necessidades do próprio projeto.

Para facilitar a gestão de toda a rede de alunos e escolas, criou-se a classe **Network**.

Para os algoritmo de cálculo de caminhos, decidiu-se criar uma classe para cada um, sendo estas: **DFS** (depth-first search), **BFS** (breadth-first search), **Dijkstra**, **NearestNeighbour**, **FloydWarshall**, **BranchAndBound** e **HeldKarp**. Tal permite a uma melhor organização do projeto, sendo mais fácil integrar vários destes algoritmos simultaneamente. Estes algoritmos serão explicados em mais detalhes no capítulo 4.

De seguida, segue a descrição detalhada das principais estruturas de dados:

### Classe Vertex (vértice)

Cada ponto de importância no mapa (seja este um edifício, uma casa de estudantes, uma escola ou uma garagem), é representado pela classe Vertex. Em cada vértice é guardado um ID único, as suas coordenadas X e Y relativamente ao mapa e o conjunto de arestas que saem de si.

### Classe Edge (aresta)

Representa uma ligação entre dois vértices, ou seja, uma estrada entre dois pontos de importância. As informações nela guardadas são: ID do vértice de Origem, ID do vértice de destino e o peso da ligação entre estes, que no caso do projeto simboliza o comprimento da estrada.

### Classe Graph (grafo)

Representa o mapa em si, que pode ser importado através dos ficheiros de texto disponibilizados pelos monitores da unidade curricular. Nesta classe são armazenados num vetor todos os vértices que o constituem.

Na leitura destes ficheiros, cada vértice é inserido no vetor com um ID sequencial (o primeiro vértice inserido tem ID 0, o segundo tem ID 1 e o terceiro ID 2, etc.), de modo a que o acesso a um vértice no vetor é de tempo constante, $O(1)$. Tal é verificado pois, uma vez sabido o ID do vértice, sabe-se a posição exata no vetor do grafo.

## Classe Network

Trata-se da classe principal do projeto, na qual se encontra a informação necessária para a gestão de toda a rede de alunos.

Nela é armazenada: o ID do vértice no qual se encontra a escola, o ID do vértice que representa a garagem e um vetor com todos os IDs das casas dos alunos.

É com a informação desta classe que serão executadas todos os algoritmos de determinação de caminhos.

# 4. Solução Implementada

O problema proposto é semelhante ao **Problema do Caixeiro Viajante** (ou, em inglês, *Travelling Salesman Problem*), diferenciando-se pelo fato de que os pontos inicial e final são determinados antes da resolução (estes pontos são a garagem e a escola) e pelo facto de que podem ser feitas múltiplas rotas para percorrer os vértices (utilizando múltiplos autocarros). Existem muitas possíveis soluções, mas nenhuma solução exata que possa ser encontrada em tempo polinomial.

## 4.1 - Análise da Conectividade

Em primeiro lugar, é necessário verificar a possibilidade de navegar entre dois locais. Tal pode ser feito a partir de uma pesquisa iniciada no vértice de partida. Poderão ser usados, então, os algoritmos de **Pesquisa em Profundidade** ou **Pesquisa em Largura.**

Algoritmo de Pesquisa em Profundidade (depth-first search)

Neste algoritmo, as arestas são exploradas a partir do vértice mais recentemente descoberto, explorando tanto quanto possível até ter de ocorrer *backtracking*.

A sua implementação em pseudocódigo é a seguinte:

**DFS(G):**

1. **for each** v ∈ V **do**:
2. visited(v) ← false
3. **for each** v ∈ V **do**:
4. **if** not visited(v) **then**:
5. VisitDFS(G, v)

**VisitDFS(G, v):**

1. visited(v) ← true
2. **for each** w ∈ Adj(v) **do**:
3. **if** not visited(v) **then**:
4. VisitDFS(G, w)

Este algoritmo possui uma complexidade temporal $O(|V| + |E|)$ e uma complexidade espacial $O(|V|)$.

Algoritmo de Pesquisa em Largura (breadth-first search)

Neste algoritmo, iniciando num vértice fonte, explora-se todos os vértices a que se podem chegar a partir do vértice original, só depois passando para outro vértice.

A sua implementação em pseudocódigo é a seguinte:

```
BFS(G, s):
1.  for each v ∈ V do:
2.      visited(v) ← false
3.  Q ← ∅                    //inicializa a fila de prioridade
4.  push(Q, s)
5.  visited(s) ← true
6.  while Q ≠ ∅ do:
7.      v ←pop(Q)
8.      for each w ∈ Adj(v) do:
9.          if not visited(w) then:
10.             push(Q, w)
11.             visited(w) ← true
```

À semelhança do algoritmo anterior, possui uma complexidade temporal $O(|V| + |E|)$ e uma complexidade espacial $O(|V|)$.

## 4.2 - Cálculo das distâncias entre vértices

Antes de começar a calcular as rotas dos autocarros, é essencial processar o grafo do mapa, de modo a obter as distâncias entre vértices de interesse (escola, garagem e casas dos estudantes).

Este processo divide-se, essencialmente, em duas abordagens: **algoritmo de Dijkstra** e **algoritmo de Floyd-Warshall**.

Algoritmo de Dijkstra

Este algoritmo, concebido pelo holandês *Edsger Dijkstra* em 1956 e publicado em 1959, permite obter o caminho com menor custo de um vértice para todos os outros num grafo pesado dirigido, em que os pesos de todas as arestas são não-negativos.

Este método possui uma complexidade temporal $O((|V| + |E|) \cdot log|V|)$, na obtenção do caminho mais curto relativamente a apenas um vértice.

No entanto, aplicado ao contexto do problema, em que é necessário passar por vértices específicos (casas dos estudantes), o algoritmo teria de ser aplicado sucessivamente, nos pontos de interesse. Neste caso, sendo $K$ o conjunto dos vértices com os endereços das crianças, o tempo de processamento é de ordem $O( (|V| + |E|) \cdot |log|V| \cdot |K| )$.

A sua implementação em pseudocódigo é a seguinte:

```
Dijkstra(G, s):
1.    for each v ∈ V do:
2.        dist(v) ← ∞
3.        path(v) ← nil
4.    dist(s) ← nil
5.    Q ← ∅                          //inicializa a fila de prioridade
6.    INSERT(Q, (s, 0))
7.    while Q ≠ ∅ do:
8.        v ← EXTRACT-MIN(Q)
9.        for each w ∈ Adj(v) do:
10.           if dist(w) > dist(v) + weight(v, w) then:
11.               dist(w) ← dist(v) + weight(v, w)
12.               path(w) ← v
13.               if w ∉ Q then:
14.                   INSERT(Q, (w, dist(w)))
15.               else:
16.                   DECREASE-KEY(Q, (w, dist(w))
```

## Algoritmo de Floyd-Warshall

O algoritmo Floyd-Warshall, publicado em 1962 por *Robert Floyd*, é um algoritmo de programação dinâmica que calcula o caminho mais curto entre todos os pares de vértices de um grafo pesado dirigido.

Este algoritmo retorna uma matriz contendo as distâncias mínimas entre todos os vértices no mapa e, se for conveniente, pode ser escrito de modo a armazenar cada um dos caminhos mais curtos. O tempo de processamento é, então, de ordem $O(|V|^3)$.

A sua implementação em pseudocódigo é a seguinte:

```
FloydWarshall(G):
1.    for i = 1 to |V| do:
2.        for j = 1 to |V| do:
```

```
3.              if exists edge from i to j then:
4.                  D[0, i, j] = W[i, j]
5.              else:
6.                  D[0, i, j] = ∞
7.      for k = 1 to |V| do:
8.          for i = 1 to |V| do:
9.              for j = 1 to |V| do:
10.                 D[k, i, j] = min( D[k-1, i, j] , D[k-1, i, k] + D[k-1, k, j] )
```

Comparando as duas hipóteses, conclui-se que a escolha de algoritmo depende da relação entre vértices e arestas:

- Caso $|E| \approx |V|$, ou seja, o número de arestas é aproximadamente igual ao número de vértices, simplificando que $|K| \approx |V|$, o algoritmo de *Dijkstra* assume uma ordem temporal $O(|V|^2 \cdot |log|V|)$.
- Caso $|E| \approx |V|^2$, ou seja, o número de arestas é aproximadamente igual ao quadrado do número de vértices, simplificando que $|K| \approx |V|$, o algoritmo de *Dijkstra* assume uma ordem temporal $O(|V|^3 \cdot |log|V|)$.

No primeiro caso, torna-se mais eficiente utilizar o algoritmo de *Dijkstra*, enquanto que no segundo caso, a melhor opção seria o algoritmo de *Floyd-Warshall*, pelo que este processamento apenas seria realizado uma vez para cada mapa.

Como tal, seria relevante utilizar uma condição para decidir, em função dos valores de $|V|$ e $|E|$, qual o melhor algoritmo a utilizar.

### 4.3 - Desenvolvimento do caminho entre dois vértices, passando por pontos de interesse

Numa situação na qual haja, por exemplo, apenas uma escola, um autocarro que parte de uma garagem e $|K|$ crianças a serem levadas de diferentes endereços, uma possível solução é testar qual seria a distância total que seria percorrida para cada possível permutação na ordem de visita dos endereços.

Trata-se de uma solução exata, mas simultaneamente *naive*, uma vez que o tempo de processamento seria proporcional a $O(|K|!)$, sendo, sem dúvida, um valor demasiado elevado.

Como tal, pretende-se desenvolver e aplicar métodos que conjuguem a exatidão (até um certo ponto) da solução e eficiência de tempo na sua obtenção. Serão analisados duas heurísticas, a fim de atingir esse objetivo. Como comparação, será também utilizado um algoritmo com solução exato, no entanto mais eficiente do que o método *brute force* mencionado nos parágrafos anteriores.

## Algoritmo de Bellman-Held-Karp

Uma possível solução exata para o problema é utilizar o algoritmo de *Bellman-Held-Karp*. Este algoritmo implicaria um tempo de processamento de ordem $O(|K|^2 2^{|K|})$, o que pode ser muito elevado, no entanto é melhor do que o tempo de ordem $O(|K|!)$ que seria obtido testando todas as possíveis permutações.

Trata-se de um algoritmo de programação dinâmica, baseando-se na divisão do problema em subproblemas mais simples. Parte-se do pressuposto de que se uma rota é solução do problema, então uma rota menor que esteja contida nesta é solução para um problema análogo ao anterior para um subconjunto dos vértices visitados.

Formalmente, seja $d_{ij}$ a menor distância percorrida entre $i$ e $j$ para qualquer par de vértices $i$ e $j$, seja $J \subseteq K$ um subconjunto qualquer dos vértices de $K$ e seja $D(J, c)$ a menor distância total possível para uma rota que passa por todos os vértices de $J$ e termina em $c$, sendo que $c \in J$.

Obtém-se a equação seguinte, observando que a solução do problema no subconjunto $J - c$ pode ser utilizada para calcular $D(J,\ c)$:

$$D(J,\ c)\ =\ min_{\,x \in (J-c)}\,(\ D(J-c,\ x)\ +\ d_{xc}\ )$$

De seguida, testa-se o resultado da expressão para cada possível valor de $x$ e utiliza-se a equação recursivamente para encontrar a solução ideal no conjunto $K$.

Experimentalmente verificou-se que com frequência este algoritmo representa melhorias de 5% a 15% em relação aos que serão descritos a seguir. O tempo de processamento é curto para poucos vértices, mas cresce rapidamente conforme mais vértices são acrescentados. Para 13 vértices verificamos um tempo de processamento em torno de 72 segundos.

## Algoritmo do Vizinho Mais Próximo

Trata-se de uma heurística de fácil implementação, resolvendo em tempo aproximadamente linear. Consiste em, a partir do vértice inicial, tomar sempre o vértice mais próximo que ainda não foi visitado até atingir o vértice final.

Este algoritmo não encontra necessariamente a melhor solução para o problema, no entanto é mais rápido que qualquer outro algoritmo para soluções exatas.

A sua implementação em pseudocódigo é a seguinte:

**NearestNeighbour(G, C, current)**:

1. **for** v ∈ C **do**:
2. visited(v) = false
3. **while** ∃ v ∈ C: visited(v) = false **do**:
4. s = min_dist(current)
5. path(current) ← s
6. current = s

Verificou-se experimentalmente valores inferiores a um milissegundo para o tempo de processamento deste algoritmo até 20 vértices. Como a complexidade temporal é linear, é possível utilizá-lo em conjuntos com um grande número de vértices.

## Algoritmo Branch and Bound

O algoritmo de *Branch and Bound* foi apresentado inicialmente em 1960 e propunha-se a resolver problemas de programação discreta.

O método baseia-se em utilizar uma heurística para encontrar um limite inferior, que seria o melhor resultado possível, para cada ramificação. De seguida, compara-se esse valor com o melhor resultado obtido até o momento. Se o resultado da heurística for superior ao melhor resultado já obtido, não é válido seguir a partir desta ramificação, pois já se sabe que ela não irá melhorar o resultado. A cada vez que uma solução melhor é encontrada o limite é atualizado. Esta estratégia permite eliminar alternativas irrelevantes e leva a uma redução grande no tempo de resolução do problema.

É necessário definir um método para encontrar uma solução inicial para o problema, e uma heurística para determinar o limite inferior para cada ramificação. Para a solução inicial podemos utilizar o algoritmo do vizinho mais próximo, descrito anteriormente. A heurística a ser utilizada deve determinar sempre, para cada ramificação do problema, um valor menor ou igual ao caminho mais curto a partir dela. Idealmente a heurística retorna o maior valor possível, o mais próximo possível da distância percorrida no caminho mais curto, pois com a heurística, tendo um valor mais alto, é possível eliminar um número maior de ramificações e assim poupar mais tempo de processamento.

Uma heurística já utilizada para o problema em questão e que retorna bons resultados experimentais é a heurística da árvore de expansão mínima. Consiste em calcular a árvore de expansão mínima para os vértices que ainda precisam ser

percorridos e somar o valor de suas arestas. O valor obtido a partir disto é menor ou igual a qualquer possível solução a partir da ramificação em análise.

É válido ressaltar que é preferível priorizar o caminho mais favorável, ou seja, o caminho cujo vértice seguinte tiver o menor custo. A cada iteração haverá uma verificação da aresta de menor custo ainda não verificada e, seguindo essa análise, o vértice indicará o caminho a percorrer. Assim é provável que soluções melhores sejam encontradas mais rapidamente, possibilitando eliminar um número maior de ramificações.

O algoritmo Branch and Bound pode ser utilizado para obter soluções exatas para o problema, mas resultados experimentais indicam que muitas vezes a solução ideal é encontrada em apenas uma parte relativamente pequena do seu tempo de processamento, e que soluções próximas do ideal são encontradas muito mais rapidamente, por isso é possível utilizá-lo com tempo ou número de iterações limitados para obter resultados aproximados. Em concreto, neste projeto, a ideia seria recorrer a este algoritmo como forma de optimizar uma primeira análise executada pelo Algoritmo do Vizinho Mais Próximo. O Algoritmo do Vizinho Mais Próximo assegura rapidez, enquanto que o de Branch and Bound assegura eficiência e melhoramento.

Decidiu-se definir um tempo máximo de processamento de 30 segundos. No entanto, dependendo do número de vértices este acaba antes do tempo imposto. Para números de vértice pequenos, apresenta resultados semelhantes aos do algoritmo do Vizinho mais Próximo (apesar de serem ligeiramente melhores).

### 4.4 - Partição do conjunto dos endereços no uso de vários autocarros

No caso do uso de vários autocarros para a mesma escolha, o primeiro autocarro percorrerá o trajeto até ficar sem lugares disponíveis. Quando tal acontecer, desloca-se para a escola (destino final). De seguida, o autocarro seguinte continuará, apanhando os alunos restantes, que não tiveram lugar no autocarro anterior. Este processo repete-se até que todos os alunos sejam recolhidos.

# 5. Casos de Utilização

O programa que nos propomos implementar pressupõe o cumprimento das seguintes funcionalidades:

### 1 - Menu Inicial e carregamento de mapas

O programa inicia-se com um menu principal que possibilita carregar três tipos de mapas: um mapa de tamanho reduzido (Fafe), um mapa de tamanho médio (Gondomar) ou um mapa de tamanho elevado (Porto).

### 2 - Menu do grafo

Numa fase seguinte, após carregamento do mapa e preparação do grafo respetivo, o programa apresenta um menu de 6 opções:

#### 2.1 - Visualização do grafo completo

O programa fornece a possibilidade de visualizar graficamente o grafo completo, com todos os seus nós e arestas. É utilizada a API Graph Viewer, disponibilizada pelos docentes da unidade curricular. De forma a acompanhar a aplicação ao grafo das funcionalidades do programa, a cor dos vértices do grafo é alterado segundo a seguinte codificação de cores:
Amarelo - vértice normal;
Azul - vértice visitado por determinado algoritmo (conjunto dos vértices azuis corresponde ao caminho de menor distância a percorrer pelos autocarros escolares);
Preto - vértice correspondente à localização da garagem;
Vermelho - vértice correspondendo à localização de uma escola;
Verde - vértice correspondendo à localização de uma casa de um aluno.

#### 2.2 - Ir para o menu da rede da escola

Esta opção permite alcançar um menu de gestão da rede da escola. Este menu tem 7 opções:

##### 2.2.1 e 2.2.5 - Inserção e remoção de habitações correspondentes a locais de paragem

De cada vez que uma nova família subscreve o serviço, será inserida uma nova habitação no sistema. Inversamente, de cada vez que uma família prescinde do serviço, a sua habitação será removida.

#### 2.2.2. - Adição de uma escola

De cada vez que uma nova escola subscreve o serviço, esta será inserida no sistema.

O utilizador deverá inserir o número identificador do vértice ao qual pretende atribuir a categoria de escola e apenas poderá executar esta operação caso tenha anteriormente adicionado uma garagem ao sistema.

#### 2.2.3. - Adição da localização da garagem

O utilizador pode definir qual o vértice que corresponde à garagem da qual partirão os autocarros, inserindo o número identificador do vértice.

#### 2.2.4. - Definir a capacidade do autocarro

Possibilidade do utilizador definir qual a capacidade do autocarro. Todos os autocarros passam a ter esta capacidade. Esta funcionalidade limita cada iteração do algoritmo de Nearest Neighbour, pois obriga à sua interrupção quando a capacidade é atingida.

#### 2.2.6. - Mostrar uma lista dos ids dos edifícios

O utilizador pode visualizar todos os números identificadores das habitações dos alunos, da escola e da garagem.

#### 2.2.7. - Regressar ao menu anterior

Possibilidade de voltar ao menu do grafo.

### 2.3 - Ir para o menu do cálculo dos algoritmos implementado

Esta opção permite alcançar um menu que possibilita ao utilizador calcular os caminhos de distância mínima. Este menu tem 7 opções:

#### 2.3.1. - Cálculo do algoritmo de Nearest Neighbour

#### 2.3.2. - Cálculo do algoritmo de Branch and Bound

#### 2.3.3. - Cálculo do algoritmo de Bellman-Held-Karp

#### 2.3.4. - Atualização da matriz de caminhos

#### 2.3.5. - Regresso ao menu do grafo

**2.4. - Ver estatísticas do grafo**

Mostra o número total de vértices e arestas.

**2.5. - Dar reset à rede da escola**

Apaga todos os dados relativos à escola, à garagem e às habitações dos alunos.

**2.6. - Volta ao menu de carregamento do mapa**

## 3 - Sair do programa

# 6. Conclusão

Foram alcançados todos os objetivos principais para a segunda entrega do trabalho prático:

- Descrição do tema implementado.
- Identificação e formalização do problema, em termos de dados de entrada, dados de saída, restrições e função objetivo.
- Solução implementada, com identificação das técnicas de concepção e descrição dos principais algoritmos implementados.
- Identificação dos casos de utilização suportados, e respectivas funcionalidades implementadas.
- Discussão sobre estruturas de dados utilizadas;

Todos os membros participaram de modo igualitário no desenvolvimento do projeto, tendo sido realizadas reuniões em pessoa, de modo a discutir em grupo todas as ideias.

Quanto às dificuldades enfrentadas, achamos a complexidade do trabalho um bocado elevada. No entanto, encontramo-nos satisfeitos com o que conseguimos fazer, dado o grande número e diversidade de algoritmos implementados. Foi também difícil a gestão do tempo, simultaneamente com outros projetos das restantes unidades curriculares.

É de salientar que a ajuda dos monitores da unidade curricular foi, sem dúvida, uma mais-valia, tendo esclarecido dúvidas importantes que definiram o rumo do trabalho.

# 7. Bibliografia


Michael Held and Richard M. Karp (1962). A dynamic programming approach to sequencing problems'. *Journal for the Society for Industrial and Applied Mathematics* 1:10.

David Rydeheard. Acessado dia 24 de Abril de 2019 em: http://www.cs.man.ac.uk/~david/algorithms/graphs.pdf

Fatma A. Karkory, Ali A. Abudalmola (2013). Implementation of Heuristics for Solving Travelling Salesman Problem Using Nearest Neighbour and Minimum Spanning Tree Algorithms. World Academy of Science, Engineering and Technology International Journal of Computer and Information Engineering Vol:7, No:10

Isabel Droste (2017). Algorithms for the travelling salesman problem. Disponível em: https://dspace.library.uu.nl/bitstream/handle/1874/366424/BachelorthesisIsabelDroste.pdf?sequence=2&isAllowed=y

Joaquim Meireles (28.29 JAN 2010). Heuristics Study For The Traveling Salesman Problem. Disponível em:
https://paginas.fe.up.pt/~dsie10/presentations/session%201/Heuristics%20Study%20for%20the%20Travelling%20Salesman%20Problem.pdf

(04/04/2019). Floyd-Warshall algorithm. Disponível em:
https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

Abdul Bari (13/04/2018). Traveling Salesman Problem - Branch and Bound. Vídeo disponível em:
https://www.youtube.com/watch?v=1FEP_sNb62k

(19/04/2019). Nearest neighbour algorithm. Disponível em:
https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm

Anurag Rai**.** Traveling Salesman Problem using Branch And Bound. Acessado dia 28/04/2019 em:
https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/

(26/02/2019). Algoritmo de Dijkstra. Disponível em:
https://pt.wikipedia.org/wiki/Algoritmo_de_Dijkstra

Gonçalo Leão, Francisco Queirós e Eduardo Leite (2016). Projeto de Concepção e Análise de Algoritmos: EcoPonto: recolha de lixo seletiva (tema 4) - Parte 1.

eEL,CSA Dept,IISc,Bangalore. Optimal Solution for TSP using Branch and Bound. Acessado dia 28/04/2019 em:
https://web.archive.org/web/20160520165234/http://lcm.csa.iisc.ernet.in/dsa/node187.html

(19/09/2016). Branch and bound. Disponível em:
https://pt.wikipedia.org/wiki/Branch_and_bound

Maria Antónia Carravilla e José Fernando Oliveira (2012). Programação Inteira: Resolução por Branch and Bound. Disponível em:
https://paginas.fe.up.pt/~mac/ensino/docs/OR/ProgramacaoInteira.pdf

T Cormen; C Leiserson; R Rivest (2009). C Stein. Introduction to Algorithms. Cambridge, MA: MIT Press. [PDF] .

Apontamentos das aulas teóricas.