

SchoolBus: Transporte Escolar

Concepção e Análise de Algoritmos

Ano Letivo 2018/2019

Turma 7 Grupo B

Daniel Ferreira Brandão, up201705812, up201705812@fe.up.pt

Pedro Miguel Moás, up201705208, up201705208@fe.up.pt

Tiago Gonçalves da Silva, up201705985, up201705985@fe.up.pt

26 de abril de 2019

Índice

Tema do projeto	3
1. Veículo único e escola única	3
2. Múltiplos veículos e escola única	3
3. Múltiplos veículos e múltiplas escolas	3
Conectividade	4
Formalização do problema	5
Dados de Entrada	5
Dados de Saída	5
Restrições	6
Nos dados de entrada:	6
Nos dados de saída:	6
Função objetivo	6
Perspetiva de solução: Algoritmos utilizados	7
Algoritmo de Dijkstra	7
Algoritmo de Prim	9
Outros algoritmos	10
Perspetiva de solução: Aplicação de Algoritmos	11
Pré-Processamento	11
Problema do Caixeiro Viajante / <i>Travelling Salesman Problem</i>	11
TSP aplicado ao contexto do problema	15
Conectividade	17
Perspetiva de solução: Descrição	18
1. Veículo único e escola única	18
2. Múltiplos veículos e escola única	18
3. Múltiplos veículos e múltiplas escolas	19
Conectividade – implementação	20
Análise de complexidade	21
Complexidade Espacial	21
Complexidade Temporal Teórica	21
Complexidade Temporal Empírica	22
Estruturas de dados utilizadas	24
Casos de utilização	25
Conclusão	26
Principais dificuldades encontradas	26
Esforço dedicado por elemento	26

Tema do projeto

Neste trabalho, pretende-se implementar um sistema que permita a gestão de transportes escolares por uma empresa. Os seus veículos estão numa garagem, saindo logo de manhã em direção à escola, apanhando as crianças nas suas casas. No final do dia, efetuam o caminho inverso, deixando as crianças em suas casas.

A empresa terá um registo de todas as crianças (e respetivas moradas) que serão clientes do serviço, sempre escolhendo os caminhos que minimizem o tempo ou distância percorridos.

O problema será decomposto em três iterações:

1. Veículo único e escola única

Inicialmente, consideraremos que a empresa apenas possui um veículo. Por cada nova criança que for registada, dever-se-á atualizar o caminho da garagem até à escola. O objetivo será então calcular o melhor caminho que começa na garagem, passa por todas as casas das crianças uma única vez, terminando na escola.

2. Múltiplos veículos e escola única

Mais adiante, teremos em consideração a possibilidade de a empresa possuir vários veículos. As crianças registadas terão de ser agrupadas de modo a que seja possível balancear a utilização dos veículos e minimizar os caminhos percorridos, isto é, procurar o as rotas que levam a uma distância total percorrida mínima, com todos os veículos somados.

Como os veículos terão capacidade limitada, a partir desta iteração teremos de ter isso em conta, ou seja, se um autocarro já estiver cheio, será necessário arranjar outro.

3. Múltiplos veículos e múltiplas escolas

Por fim, será possível que a empresa deseje atender a escolas diferentes, sendo que os caminhos a executar terão de ser atualizados sempre que uma criança for acrescentada aos registos, com a sua morada e escola.

A função a minimizar continua a ser a mesma, mas teremos de ter em consideração que será possível vários veículos levarem crianças para as mesmas escolas.

Conectividade

Algumas vezes, obras nas vias públicas podem fazer com que certas zonas tornem-se inacessíveis, logo poderá ser importante avaliar a conectividade do grafo. Isto incluirá verificar se todas as casas são alcançáveis a partir da garagem, por exemplo, mas também identificar pontos de articulação para determinar se há alguma casa com baixa acessibilidade. Pode também ser interessante determinar se os pontos em questão correspondem ao mesmo componente fortemente conexo do grafo.

Formalização do problema

Dados de Entrada

- $G_i = (V, E)$ - Grafo dirigido pesado composto por
 - V - Conjunto de vértices, sendo que cada um representa um ponto da cidade. Estes guardam:
 - $Adj \subseteq E$: Conjunto de arestas que partem do vértice.
 - E - Conjunto de arestas, cada uma representando as estradas que ligam os vértices. Estes guardam:
 - **Weight**: Peso da aresta. Representa a distância entre os dois vértices ligados pela aresta.
 - $Dest \in V$: Vértice de destino da aresta.
- C_i - Lista de veículos (carros) que a empresa possui, $C(n)$ será o veículo na posição n da lista. Cada um terá associado:
 - **Capacity**: Número máximo de crianças que o veículo suporta
- $D \in V$ - Vértice que representa a garagem onde são guardados os veículos, isto é, o ponto de saída (de manhã) ou retorno (no fim do dia).
- $S \subseteq V$ - Conjunto de vértices correspondentes às escolas registadas.
- K - Lista de crianças registadas na empresa. $K(n)$ será a criança na posição n da lista. Um registo é composto por:
 - $H_k \in V$ - Vértice correspondente ao ponto de recolha da criança (ou seja, a sua casa)
 - $S_k \in S$ - Vértice correspondente à escola da criança.

Dados de Saída

- $G_f = (V, E)$ - O mesmo grafo dirigido fornecido como input.
- C_f - Lista de veículos (carros) usados pela empresa. $C(n)$ será o veículo na posição n da lista. Cada um terá associado:
 - **Capacity**: Número máximo de crianças que o veículo suporta
 - K - Lista de crianças que utilizarão este trajeto.
 - $P \subseteq V$ - Sequência ordenada de vértices a visitar no caminho de ida, $P_c(n)$ será o vértice número n , do veículo c .
 - $R \subseteq V$ - Sequência ordenada de vértices a visitar no caminho de regresso, $R_c(n)$ será o vértice número n , do veículo c .

Restrições

Nos dados de entrada:

- $\forall e \in E : \text{Weight}(e) > 0$, ou seja, as distâncias serão sempre positivas
- $\forall c \in Ci: \text{Capacity}(c) > 0$, ou seja, os carros têm capacidade positiva

Nos dados de saída:

- $G_f = G_i$, isto é, o grafo deverá permanecer igual ao grafo fornecido como input.
- $|C_i| \geq |C_f|$, isto é, não deverá haver mais carros utilizados do que disponíveis
- $\forall c \in C_f: \text{Capacity}(c) \leq |K(c)|$, ou seja, o número de crianças que usam este trajeto não pode ser superior à capacidade do autocarro.
- $\forall c \in C_f$:
 - $P_c(1) = D$, pois, no caminho de ida, o autocarro sai sempre da garagem.
 - $P_c(n) \in S$, pois, no caminho de ida, o autocarro termina sempre numa escola.
 - $R_c(1) = P_c(n)$, pois, no regresso, o autocarro sai sempre na escola onde terminou o caminho de ida.
 - $R_c(n) = D$, pois, no regresso, o autocarro termina sempre da garagem.
 - Em P_c , a casa de uma criança não pode aparecer **depois** da sua escola.
 - Em R_c , a casa de uma criança não pode aparecer **antes** da sua escola.

Função objetivo

Pretende-se minimizar a distância total percorrida na ida e no regresso, com todos os veículos somados. Ou seja, pretende-se minimizar as funções:

$$f = \sum_{c \in C} (\sum_{v \in P} \text{dist}(v_n, v_{n+1}) + \sum_{v \in R} \text{dist}(v_n, v_{n+1}))$$

$$g = |C_f|$$

Dar-se-á prioridade à minimização do número de veículos, ou seja, primeiro a função g , só depois a função f .

Nota: $\text{dist}(v_n, v_{n+1})$ - distância entre o vértice v , e o vértice v da iteração seguinte.

Perspetiva de solução: Algoritmos utilizados

A descrição da solução a ser implementada pode ser dividida em 3 partes, correspondendo cada uma delas às três iterações distintas já previamente descritas. Apesar de se tratarem de problemas distintos, são os três suficientemente semelhantes para possuírem similaridades no procedimento utilizado para o cálculo da solução.

Nas três iterações, será primeiro calculada a distância mínima entre os vários pontos de interesse do grafo, que, neste caso serão os pontos pelos quais o veículo tem de passar, passo a que chamaremos de “**pré-processamento**”.

Para além disso, as três iterações terão soluções similares, já que todas se tratam de instâncias do **Vehicle Routing Problem (VRP)**, uma generalização do **Problema do Caixeiro Viajante (Travelling Salesman Problem - TSP)**, partilhando o mesmo objetivo base, minimizar a distância percorrida pelo veículo ou, no caso das iterações 2 e 3, a soma das distâncias percorridas por todos os veículos. Por essa razão, também será necessária uma análise do TSP.

Antes de entrar em detalhes da implementação, dever-se-á ter uma noção sobre os 3 algoritmos aplicados em grafos, que poderão ser usados como base para a concepção da solução, **Algoritmo de Dijkstra**, **Floyd-Warshall** e **Algoritmo de Prim**.

Algoritmo de Dijkstra

O Algoritmo de Dijkstra é um método que pode ser utilizado para calcular o caminho mais curto entre um vértice de um grafo a todos os outros de um grafo.

O Algoritmo de Dijkstra é um exemplo de algoritmo ganancioso (greedy algorithm) e, portanto, segue uma heurística de fazer a escolha ótima local com o intuito de atingir um ponto ótimo global. Note-se que, ao contrário de alguns algoritmos que seguem a mesma heurística, o Algoritmo de Dijkstra obtém sempre a melhor solução.

Para que o algoritmo possa executar é necessário ver cumpridos alguns requisitos relativos ao grafo sobre o qual o algoritmo vai operar:

- O grafo sobre o qual o algoritmo vai executar precisa de ser pesado, isto é, cada aresta terá um peso associado, que indica o custo entre os dois vértices que liga.
- Ao longo da execução do algoritmo, os vértices terão de guardar o custo mínimo do caminho desde o vértice inicial até ele próprio (∞ se não existir), como também o vértice que o precede nesse mesmo caminho.
- É preciso também utilizar uma estrutura de dados para guardar os vértices que se encontram “à espera” de ser processados. Para isto, usa-se uma fila de prioridade, onde os vértices com maior prioridade são os que têm menor distância ao vértice inicial, algo que caracteriza o algoritmo como ganancioso.

Deste modo, pode implementar-se o algoritmo em pseudocódigo:

<pre>DIJKSTRA(<i>G</i>, <i>s</i>): // <i>G</i>=(<i>V</i>,<i>E</i>), <i>s</i> ∈ <i>V</i> 1. for each <i>v</i> ∈ <i>V</i> do 2. <i>dist</i>(<i>v</i>) ← ∞ 3. <i>path</i>(<i>v</i>) ← nil 4. <i>dist</i>(<i>s</i>) ← 0 5. <i>Q</i> ← ∅ // min-priority queue 6. INSERT(<i>Q</i>, (<i>s</i>, 0)) // inserts <i>s</i> with key 0 7. while <i>Q</i> ≠ ∅ do 8. <i>v</i> ← EXTRACT-MIN(<i>Q</i>) // greedy 9. for each <i>w</i> ∈ <i>Adj</i>(<i>v</i>) do 10. if <i>dist</i>(<i>w</i>) > <i>dist</i>(<i>v</i>) + <i>weight</i>(<i>v</i>,<i>w</i>) then 11. <i>dist</i>(<i>w</i>) ← <i>dist</i>(<i>v</i>) + <i>weight</i>(<i>v</i>,<i>w</i>) 12. <i>path</i>(<i>w</i>) ← <i>v</i> 13. if <i>w</i> ∉ <i>Q</i> then // old <i>dist</i>(<i>w</i>) was ∞ 14. INSERT(<i>Q</i>, (<i>w</i>, <i>dist</i>(<i>w</i>))) 15. else 16. DECREASE-KEY(<i>Q</i>, (<i>w</i>, <i>dist</i>(<i>w</i>)))</pre>	Tempo de execução: $O((V + E) * \log V)$
--	--

Figura 1 - Implementação do Algoritmo de Dijkstra, em pseudocódigo

O algoritmo pode ser dividido em duas partes. Uma primeira parte onde é feita a preparação dos dados e uma segunda onde os caminhos mais curtos são efetivamente calculados.

A preparação dos dados consiste na atribuição de valores aos atributos de cada vértice. O valor da distância até ao vértice inicial é inicializado a ∞ e o apontador para o vértice anterior no caminho é inicializado com valor nulo. Para além disto, antes de se iniciar o cálculo de caminhos, o vértice inicial é adicionado à fila de prioridade e o seu valor de distância é alterado para zero.

Posto isto, pode ser iniciado o cálculo de caminhos que toma os seguintes passos:

1. Extrair um vértice (*v*) da fila de prioridade
2. Para cada vértice adjacente de *v* (*w*) verificar se o valor da distância de *v* somado com o peso da aresta que os liga é inferior ao valor de distância atual - se tal acontecer, então atualiza-se o valor da distância de *w* assim como a sua posição na fila de prioridade, ou adiciona-se caso este ainda não esteja presente (se o valor da distância anterior for ∞)
3. Se a fila de prioridade estiver vazia, terminar. Caso contrário, retornar a 1.

Quanto à complexidade temporal do algoritmo, esta pode ser obtida analisando os diferentes momentos que o compõem. A preparação de dados possui complexidade $O(|V|)$ pois todos os vértices serão processados. A extração e a inserção de um vértice da fila de prioridade é de complexidade $O(\log|V|)$ e, uma vez que no máximo estas operações serão feitas $|V|$ vezes, a complexidade total destas operações é de $O(|V| \log|V|)$. Por último, a atualização da posição de cada vértice na fila de prioridade tem complexidade $O(\log|V|)$ e como será realizada no máximo $|E|$ vezes a complexidade total da operação é de $O(|E| \log|V|)$.

Assim, o Algoritmo de Dijkstra determina o caminho mais curto entre o vértice inicial e todos os outros com complexidade temporal de $O(|V| + |V| \log|V| + |E| \log|V|)$, o que pode ser simplificado para $O((|V| + |E|) \log|V|)$.

Algoritmo de Prim

O Algoritmo de Prim é um algoritmo utilizado para encontrar a Árvore de Expansão Mínima (**Minimum Spanning Tree - MST**) de um grafo.

Uma **árvore de expansão** de um grafo é um subconjunto do mesmo, contendo as seguintes propriedades:

- Contém todos os vértices do grafo original
- Estende a todos os vértices
- É acíclica, ou seja, o grafo não contém nenhum nó que retorne para si mesmo.
- O número de arestas corresponde a $|V|-1$ do grafo original

Deste modo, uma MST corresponde à árvore de expansão que utiliza **arestas com custo total mínimo**. O mesmo grafo pode ter diferentes árvores de expansão mínimas.

O Algoritmo de Prim trata-se também de um algoritmo ganancioso (greedy). Em cada passo adiciona-se uma nova aresta, tendo o cuidado de garantir que as arestas já selecionadas são parte de uma mesma MST. Note-se, então, que o algoritmo funciona para qualquer grafo pesado, conexo e não dirigido.

Deste modo, pode implementar-se o algoritmo em pseudocódigo:

```
prim function (G, s) // G = (V, E), s ∈ V
  for each vertex v ∈ V
    cost[v] ← ∞
    path[v] ← null
  cost[s] ← 0
  Q ← ∅ // min-priority queue for vertices v with cost[v] as
  priority.
  Insert(Q, s)
  while Q ≠ ∅
    v ← Extract-Min(Q)
    for each w ∈ Adj(v)
      if cost[w] > weight (v, w)
        path[w] ← v
        cost[w] ← weight (v, w)
        Decrease-Key(Q, w)
```

Figura 2 - Implementação do Algoritmo de Prim, em pseudocódigo

O algoritmo pode ser informalmente descrito como executando as seguintes etapas:

1. Inicializar uma árvore com um único vértice, escolhido arbitrariamente no grafo
2. Das arestas que ligam a árvore aos vértices que ainda não estão na árvore, encontrar a aresta E com peso mínimo.
3. Adicionar E à árvore, assim como o seu vértice respetivo.
4. Terminar se todos os vértices pertencem à árvore, retornar a 2 caso contrário.

A complexidade temporal do algoritmo de Prim depende das estruturas de dados utilizadas para o gráfico e da ordenação das arestas por peso.

A implementação utilizando uma matriz ou lista de adjacências e procurando linearmente a aresta de peso mínimo terá complexidade $O(V^2)$. No entanto, esse tempo de execução pode ser melhorado utilizando uma fila de prioridade para localizar tais arestas.

Pode usar-se, então, a fila de prioridade para guardar todas as arestas do grafo, ordenadas por peso mínimo. Desta forma o algoritmo pode ser executado em tempo $O(|E| \log |E|)$. Para melhorar ligeiramente este resultado, é ainda possível guardar na fila os vértices em vez de arestas, organizando-os por menor peso de aresta que os conecta a qualquer nó na MST a ser construída.

Assim, com fila de prioridade, o algoritmo terá complexidade $O(|E| \log |V|)$.

Outros algoritmos

Para além do **Algoritmo de Dijkstra**, poderia ser útil utilizar também o **Algoritmo de Floyd-Warshall**, que calcula o caminho mínimo entre todos os pares de vértices, com complexidade $O(|V|^3)$. No entanto, como será explicado mais à frente, este fará muito mais trabalho do que o necessário, por isso foi deixado de parte.

Relativamente a algoritmos de cálculo de MST, o **Algoritmo de Kruskal** também seria uma opção, tendo um funcionamento e complexidade bastante semelhante ao **Algoritmo de Prim**.

Perspetiva de solução: Aplicação de Algoritmos

Pré-Processamento

Como já foi referido, o pré-processamento a efetuar é idêntico nas três soluções e é composto pelo cálculo da distância mínima entre os pontos de interesse do grafo. Estes pontos, nos problemas em questão, tratam-se da garagem, da escola e dos locais de recolha das crianças registadas no serviço de transporte e, portanto, são os pontos que terão de estar incluídos nos caminhos dos veículos.

Será então necessário um algoritmo que nos permita saber a distância entre quaisquer dois pontos de interesse do grafo, assim como o caminho entre eles. O **Algoritmo de Floyd-Warshall** seria uma opção, mas não só é demasiado dispendioso em termos de espaço (não nos interessa saber os caminhos mais curtos entre todos os pares de vértices), como mesmo em termos de tempo, quando comparado à execução repetida do **Algoritmo de Dijkstra**, no caso de grafos pouco densos ($|E| \cong |V|$), característica costume de grafos que representam redes de transporte.

Será, por isso, utilizado o algoritmo de Dijkstra. Por cada ponto de interesse, descobrir-se-á o caminho mais curto até todos os outros, guardando-se o caminho e a respetiva distância total numa matriz quadrada. Poderemos terminar cada iteração quando já se tiver calculado o caminho mínimo para todos os pontos de interesse.

Problema do Caixeiro Viajante / *Travelling Salesman Problem*

Como se pretende calcular o trajeto mínimo que passa num determinado conjunto de pontos de interesse, grande parte do problema pode ser visto como uma adaptação do problema do caixeiro viajante, em inglês TSP (*Travelling Salesman Problem*), no qual a partir de um grafo com as distâncias entre todos os vértices conhecidas, se pretende determinar qual o caminho que passa em todos os vértices uma única vez, retornando no fim ao vértice inicial, com custo total mínimo.

No entanto, deve-se notar que o TSP é um problema para o qual a solução ótima é bastante difícil de encontrar, apesar de ter sido muito estudado até hoje.

Podemos inicialmente pensar num modo **brute-force** de o resolver, isto é, verificar todas as combinações de caminhos possíveis entre os pontos de interesse, escolhendo por fim o caminho com custo mínimo. Tal algoritmo teria de testar os **$V!$** possíveis trajetos (sendo V o número de pontos de interesse), tendo complexidade **$O(n!)$** . Por isso, num contexto minimamente real, este algoritmo não será aceitável, pois **mesmo para um grafo relativamente pequeno** ($V = 20$, por exemplo), **o algoritmo já parecerá interminável**.

Outra opção mais eficiente utilizaria **programação dinâmica** para chegar à solução ótima. Ao longo do conjunto V de vértices $\{1, 2, 3, 4, \dots, n\}$, considere-se 1 como o ponto de partida e chegada, e determine-se o caminho de custo mínimo com cada vértice de S a aparecer exatamente uma vez. Poderia-se criar um termo $C(S, i)$, que será o custo do caminho de custo mínimo que visita cada vértice de S uma única vez, começando em 1 e terminando em i . Teríamos de calcular $C(S, i)$ para todos os subconjuntos de V de tamanho 2, de seguida de tamanho 3, e assim sucessivamente, desde que o vértice 1 esteja sempre presente em S . Este algoritmo teria $V \cdot 2^V$ subproblemas, cada um demorando tempo linear a resolver. Logo, a complexidade temporal e espacial deste algoritmo será $O(n^2 \cdot 2^n)$, o que **também não é aceitável**, mesmo que seja melhor.

O algoritmo com programação dinâmica seria a melhor hipótese para encontrar a

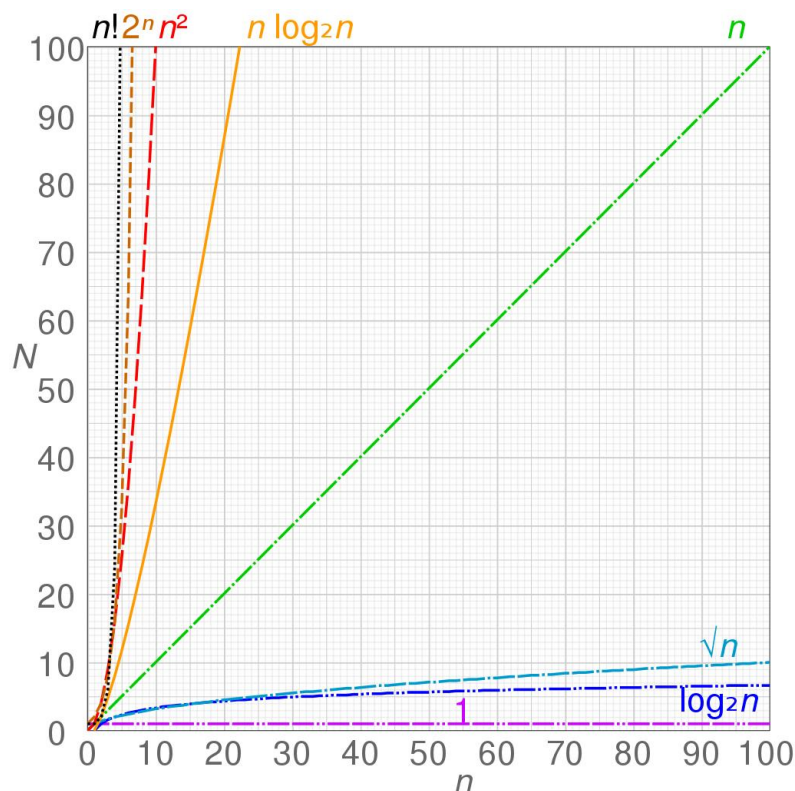


Figura 3 - Variação do tempo de execução (N) com o tamanho do input (n), para várias ordens de complexidade

solução ótima de um TSP. No entanto, como se viu, não apresenta uma complexidade temporal (ou espacial) aceitável. Por isso, a única opção será tentar encontrar um algoritmo que execute em tempo polinomial, mas que encontre uma solução que se aproxime ao máximo à solução ótima.

Uma hipótese para o algoritmo a usar, que será realmente utilizado, é utilizar o conceito de **Árvore de Expansão Mínima (Minimum Spanning Tree)**. Uma árvore de expansão mínima, como explicado acima, corresponde à árvore que liga todos os vértices de um grafo com custo total mínimo. Esta será determinada através do **Algoritmo de Prim**, com complexidade $O(|E| \log |V|)$.

O algoritmo usado para encontrar o caminho que passa em todos os pontos de um grafo exatamente uma vez será, então, o seguinte:

1. Seja 1 o vértice inicial/final do TSP
2. Construa-se a MST a partir de 1 utilizando o Algoritmo de Prim
3. Listem-se os vértices pela pré-ordem de visita da MST.
4. Adicione-se 1 ao fim da lista.

Com estes passos, obtém-se um conjunto de pontos que inicia e termina em 1, passando em todos os pontos do grafo exatamente uma vez.

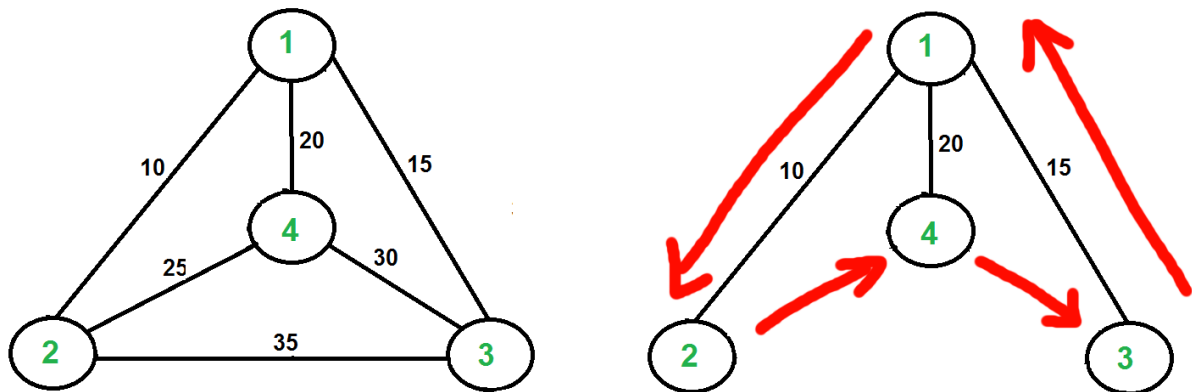


Figura 4 - Solução de TSP obtida (à direita) a partir da aplicação do algoritmo descrito num grafo (à esquerda)

Desigualdade triangular: O melhor caminho de i para j corresponde sempre a ir diretamente de i até j . Ou seja, para quaisquer vértices V_1, V_2, V_3 , $\text{dist}(V_1, V_2) + \text{dist}(V_2, V_3) \leq \text{dist}(V_1, V_3)$. Intuitivamente, conclui-se que um grafo que simula uma rede de transporte numa cidade satisfaz a desigualdade triangular na maior parte das situações.

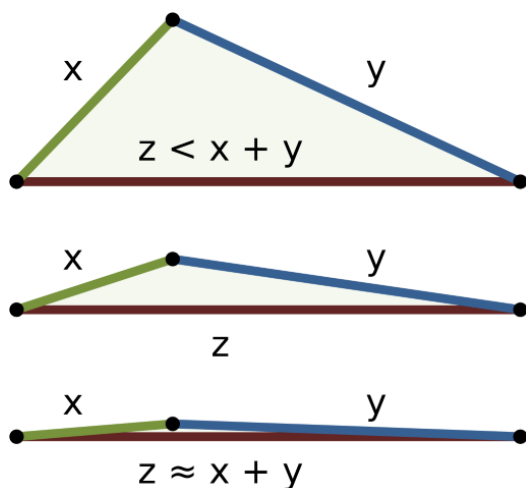


Figura 6 - A propriedade baseia-se no facto de que o maior lado de qualquer triângulo é maior do que a soma dos outros dois.

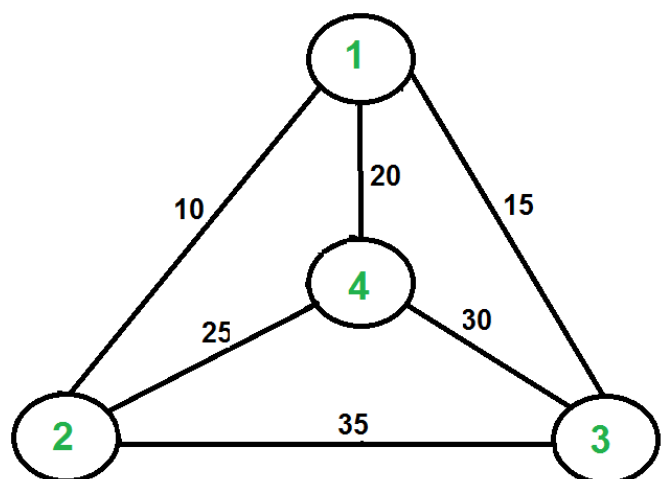


Figura 5 - Grafo que obedece desigualdade triangular

Pode provar-se que, se o problema verificar a desigualdade triangular, o caminho obtido nunca terá custo total maior do que o dobro do custo ótimo:

1. O custo do caminho ótimo de um problema TSP nunca é menor do que o custo de uma MST (por definição de MST).
2. O custo total da visita completa da MST é no máximo o dobro do seu custo (pois, no máximo, visitam-se duas vezes uma aresta)
3. A solução obtida pelo algoritmo indicado tem custo menor do que a visita completa da MST, pois em visita em pré-ordem, pelo menos duas arestas da visita completa serão substituídas por outra menor (exemplo: 2->1 + 1->4 será substituído por 2->4, que terá custo menor devido à desigualdade triangular)

Como o algoritmo tem custo total menor do que a visita completa, e como esta é no máximo o dobro do seu custo total, que será sempre menor ou igual ao custo do caminho ótimo, **conclui-se que o caminho obtido por este algoritmo nunca terá custo total maior do que o dobro do custo do caminho ótimo.**

Analisando a complexidade algorítmica, conclui-se que o passo mais demorado é a construção da MST, sendo os outros passos concluídos, no máximo, em tempo linear, $O(V)$. A complexidade temporal é então, $O(|V| + |E| \log|V|)$. Sendo $|E| \leq |V|^2$, pode-se simplificar como $O(|E| \log|V|)$, a mesma do algoritmo de Prim.

Este algoritmo, mesmo que seja muito mais eficiente do que qualquer outro que calcule a solução ótima, dará, no pior caso, um custo igual ao dobro do custo ótimo, em casos como o representado no diagrama abaixo:

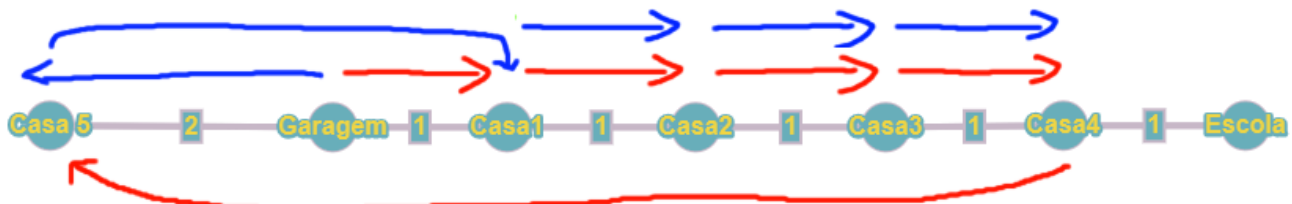


Figura 7 - Caminho obtido por aplicação do algoritmo (a vermelho) vs caminho ótimo (a azul)

A vermelho, observa-se o caminho obtido pelo algoritmo usado, e a azul tem-se o caminho que seria ótimo (algumas arestas foram escondidas para facilitar a visibilidade). Note-se que não só o caminho ótimo é muito mais eficiente, como também termina muito mais perto da escola. Embora estes casos extremos sejam bastante raros, teremos de procurar otimizar ao máximo o percurso obtido.

Para evitar tal problema, será utilizado também um outro algoritmo, que utiliza o princípio de **Nearest Insertion**.

Um algoritmo de *Nearest Insertion* utilizado para resolver um TSP tem, geralmente, a seguinte estrutura:

1. Escolher o vértice inicial V_1 , e o seu mais próximo V_2 , criando-se um trajeto parcial.
2. Selecionar um vértice V que ainda não esteja no trajeto
3. Descobrir qual a aresta (V_i, V_j) , que minimiza $\text{dist}(V_i, V) + \text{dist}(V, V_j) - \text{dist}(V_i, V_j)$, sendo V_i e V_j vértices pertencentes ao trajeto parcial.
4. Inserir V entre V_i e V_j .
5. Se todos os vértices do grafo original pertencem ao trajeto, terminar, senão, retornar a 2.

Tal algoritmo terá complexidade $O(|V|^2)$ ($|V|$ é comparado em média com $|V|/2$ arestas), sendo $|V|$ o número de vértices no grafo apenas com os pontos de interesse, o que seria bastante aceitável, pois o número de pontos de interesse seria bastante reduzido, comparado com o grafo original.

Para além dos mencionados haveria outras opções, tais como o **Algoritmo de Clarke-Wright** (direcionado para VRP - *Vehicle Routing Problem*, uma generalização do TSP que também se poderia ajustar ao problema), ou o **Algoritmo de Christofides**, que embora possa dar, em algumas situações, melhores resultados do que o algoritmo baseado em MST, é de mais complexa implementação.

TSP aplicado ao contexto do problema

Arranjado um algoritmo que forneça um trajeto que una todos os pontos de um grafo, podemos aplicá-lo ao contexto do problema.

O ponto inicial será a Garagem, onde se localizam os autocarros. De seguida, **utilizar-se-á o algoritmo baseado na MST**, para encontrar um caminho que começa na garagem, e que passa por todas as casas e escolas exatamente uma vez (repare-se que, numa perspetiva prática, não quer dizer que um autocarro não poderá passar duas vezes na mesma rua, mas como o algoritmo descrito recebe um grafo com caminhos entre todos os pares de vértices, passar duas vezes no mesmo não é preciso para obter o resultado).

No entanto, o caminho obtido não será a solução que o programa dará, mas sim a ordem de vértices utilizada para a implementação do algoritmo **Nearest Insertion**, com algumas restrições, explicadas a seguir.

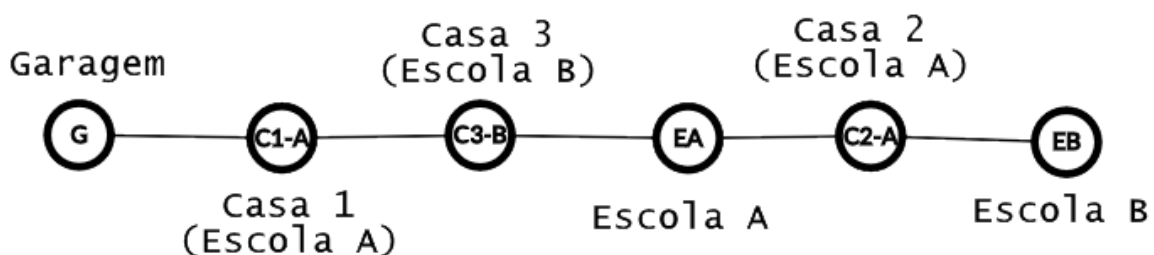


Figura 8 - MST (simplificada) para um conjunto simples de pontos de interesse

Acima mostra-se a árvore de expansão mínima para um certo grafo aplicado ao contexto do problema, com 2 escolas e 3 casas. Neste caso, aplicar-se-ia o algoritmo **Nearest Insertion** de forma normal, começando por C1-A, seguido de C3-B.

No entanto, chegando a EA (escola), a sua posição no trajeto não poderia ficar antes de qualquer casa associada à escola A, ou seja, o algoritmo consideraria entre deixá-la no fim da lista atual, ou entre C1-A e C3-B, sendo a primeira hipótese mais vantajosa.

Neste ponto, a lista do trajeto seria {G, C1-A, C3-B, EA}. Agora, para decidir onde colocar C2-A, como a escola respetiva já foi colocada no trajeto existe outra restrição: C2-A não pode aparecer depois de EA. Assim, as hipóteses serão: colocar C2-A entre G e C1-A, C1-A e C3-B, ou entre C3-B e EA.

Como se pode observar na imagem à direita, colocar C2-A entre C3-B e EA seria a escolha que levaria a um menor aumento da distância do trajeto (assumindo uma disposição linear dos pontos e distância unitária entre cada um).

Por fim, apenas falta escolher uma posição para o vértice EB, sendo a lista atual {G, C1-A, C3-B, C2-A, EA}.

Podemos colocar tal vértice no fim da lista, entre C2-A e EA, ou entre C3-B e C2-A (não antes, pois trata-se de uma Escola).

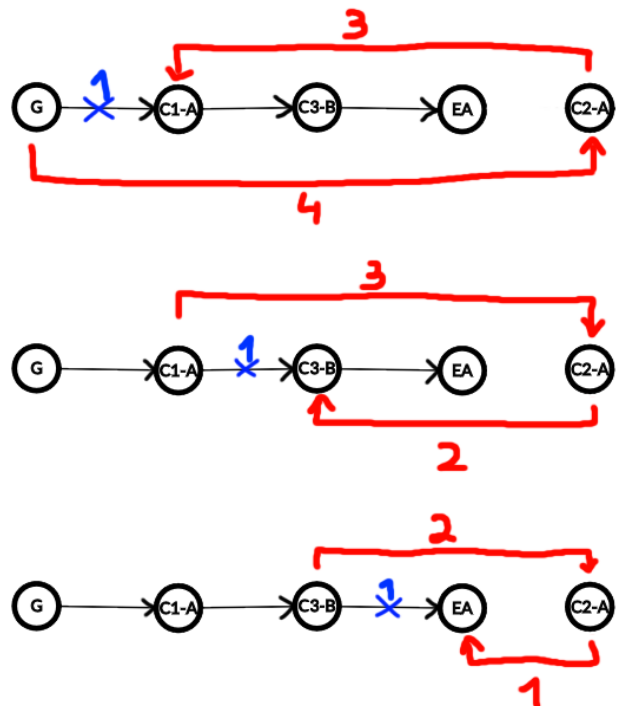


Figura 9 - Análise das 3 possibilidades de inserção de C2-A, sendo a última a que traz um prejuízo menor.

Pode facilmente confirmar-se que as 3 hipóteses dariam num aumento de 2 unidades ao trajeto, logo qualquer uma seria aceitável, terminando com o seguinte trajeto:



Figura 10 - Solução obtida para o problema, após aplicação do algoritmo descrito

Conectividade

Como foi explicado na descrição do tema, será interessante avaliar a conectividade do grafo fornecido como entrada.

Em primeiro lugar, é fundamental que haja sempre um caminho entre quaisquer dois pontos de interesse (garagem, casa ou escola). É necessário existir caminho entre ambas as direções, caso contrário, o trajeto será impossível de realizar, visto que o autocarro não só terá de levar as crianças á escola, mas também precisa de levá-las a casa.

Devido ao pré-processamento feito (não há caminho entre A e B se a distância entre A e B for ∞), facilmente se fazem essas verificações.

De seguida, pode também ser interessante verificar se o grafo é fortemente conexo ou não. Não é crucial que haja um caminho entre todos os pontos do grafo, mas é importante saber se a rede está bem preparada. O algoritmo para o verificar é simples:

1. Colocar todos os vértices como não visitados.
2. Realizar BFS no grafo a partir de um vértice. Se houver algum vértice não visitado, o grafo não é fortemente conexo, terminando.
3. Inverter a direção de todas as arestas.
4. Colocar todos os vértices como não visitados.
5. Repetir passo 2. O grafo será fortemente conexo se e só se todos os vértices tiverem sido visitados.

Este algoritmo tem complexidade temporal $O(|V| + |E|)$, a mesma complexidade da DFS, já que todos os vértices e arestas são visitados seguidamente.

Estando todos os pontos de interesse (POIs) fortemente conectados entre si, é importante determinar a existência de pontos de articulação entre eles, isto é, de vértices que, quando removidos, façam com que os pontos de interesse não estejam fortemente conectados. É informação crucial, pois poderá haver casas com baixa acessibilidade, casos em se houver obras nas vias públicas num desses pontos impossibilitarão o trajeto do autocarro.

O algoritmo para verificar a existência desses pontos é o seguinte:

1. Para todos os vértices V do grafo:
 - a) Remover V .
 - b) Colocar todos os vértices como não visitados.
 - c) Realizar BFS no grafo a partir de um vértice. Se houver algum POI não visitado, adicionar V aos Pontos de Articulação (caso ainda não tenha sido adicionado)
 - d) Voltar a colocar V no grafo.
2. Inverter a direção de todas as arestas
3. Repetir o passo 1.

Sendo necessário fazer uma BFS por cada vértice, na qual se verifica se todos os pontos de interesse (**não todos os vértices**) têm caminho entre si, tal algoritmo terá complexidade $O(|V| (|V| + |E| + |P|))$, simplificável para $O(|V| (|V| + |E|))$, já que $|P|$ será geralmente muito reduzido, quando comparado com $|V|$ ou $|E|$.

Perspetiva de solução: Descrição

Para as três iterações, a ordem de colocação dos vértices por *Nearest Insertion* no trajeto de retorno irá utilizar os vértices de interesse pela ordem inversa do trajeto de ida. Mesmo que por vezes haja exceções, o comprimento do caminho mínimo de A para B, é geralmente semelhante ao comprimento do caminho mínimo de B para A, não justificando, à partida, recalcular a MST, quando o critério usado para posicionar o vértice será, realmente, o de *Nearest Insertion*. Para além disso, como o trajeto de ida já coloca as crianças antes das suas escolas, a aplicação das restrições impostas será facilitada, como será descrito.

1. Veículo único e escola única

Numa primeira iteração, após o pré-processamento, teremos um grafo constituído pela escola, garagem e todas as casas das crianças, a distância mínima entre cada ponto de interesse (∞ se não houver tal caminho), assim como os caminhos completos entre esses pontos.

Devido ao pré-processamento feito, podemos simplificar o grafo das cidades num grafo com apenas os pontos de interesse, e as distâncias mínimas entre estes. A partir daí, o procedimento a tomar já foi descrito anteriormente, sendo simplificado pelo facto de não haver mais do que uma escola, ficando nesta secção apenas um breve resumo.

Em primeiro lugar, correr-se-ia o algoritmo baseado em MST, obtendo-se uma lista ordenadas de vértices a escolher. De seguida, efetuar-se-ia ***Nearest Insertion*** pela ordem ditada na lista, com a única restrição de que o vértice da escola terá de ficar na última posição (e a garagem na primeira). Para o cálculo do trajeto de retorno, aplicar-se-ia o mesmo algoritmo, mas desta vez forçando o início na escola, e o fim na garagem.

2. Múltiplos veículos e escola única

Numa segunda iteração, poderemos ter vários veículos em circulação, cada um com a sua capacidade máxima. Para minimizar o número de veículos usados, mas também o espaço desperdiçado (veículos maiores são mais dispendiosos), começa-se por escolher do veículo maior até ao mais pequeno, até sobrar espaço. Nesse caso, tenta-se escolher o mais pequeno que suporta todas as crianças restantes (Note-se que os autocarros ainda não foram atribuídas às crianças, apenas foi usado o número total para efeitos de minimização de número de veículos).

Assim, para que os custos de transporte fossem minimizados foi necessário obter um método que calculasse quais os veículos a ser utilizados de forma a que o seu número fosse mínimo. Atendendo a esta característica constata-se facilmente que o algoritmo a ser concebido se trata de um exemplo de um ***greedy algorithm*** e pode ser implementado tal como o pseudo-código a seguir apresentado o demonstra.

```

getVehicles (n, V) // n = number of kids, V = available vehicles
    sort(V)
    reverse(V)
    v ← V.begin()
    while n > 0 & V.size > 0 do
        if n > v.capacity then
            n - v.capacity
            v -> U
            V.erase(v)
            v ← V.begin()
        else
            v ← next(V)
            if v == V.end() || n > v.capacity then
                v ← retrieve(V)
                n - v.capacity
                v -> U
                V.erase(v)

```

Figura 11 - Algoritmo Greedy para escolha de veículos (Pseudocódigo)

Seguidamente, o passo a tomar será aplicar os algoritmos usados na parte 1. para cada autocarro ordenadamente. Por cada veículo (exceto o último) encontrar-se-á um trajeto que passa por N (capacidade) casas, terminando na escola. Os veículos seguintes aplicarão os tais algoritmos mas apenas considerando as crianças que não foram ainda recolhidas.

O último veículo apenas terá em conta as restantes crianças, logo procederá exatamente como se tratasse de um problema da parte 1.

O caminho de retorno passará pelas mesmas casas, mas desta vez a aplicação das restrições é mais simples, pois as crianças já estão posicionadas depois das suas escolas, podendo-se começar as comparações de custos a partir do vértice da escola respetiva.

3. Múltiplos veículos e múltiplas escolas

Numa terceira iteração, poderá ainda ter-se em conta a possibilidade de a empresa atender a mais do que uma escola, sendo necessário otimizar a utilização de veículos e escolha de trajetos, de modo que cada criança vá ter à respetiva escola.

O modo de escolha dos veículos mantém-se inalterado, assim como os algoritmos utilizados. Porém, como já foi extensamente descrito, terá de se ter em conta as restrições impostas. Isto é, no trajeto final, as crianças têm de aparecer antes das respetivas escolas. A solução para cumprir esta restrição já foi explicada na secção anterior, ou seja, esse problema está resolvido.

Deste modo, por cada trajeto / veículo calculado, vai ser reduzido o número de crianças restantes, procedendo o último autocarro do mesmo modo que na parte 2.

Como descrito na parte anterior, o caminho de retorno será calculado de modo bastante similar, sendo apenas necessário repetir a última parte do algoritmo (*Nearest Insertion*), e a aplicação das restrições facilitada.

Conectividade – implementação

Foi implementado o algoritmo descrito anteriormente para verificar se o grafo utilizado (baseado no mapa da cidade do porto) era fortemente conexo, de modo a averiguar se a rede rodoviária está bem preparada.

Porém, os grafos fornecidos apresentavam inconsistências que o tornavam sempre não conexo, tornando o algoritmo um pouco desnecessário para tais grafos. Por isso, e como o facto de os grafos não serem fortemente conexos não ser um fator importante para o cálculo do trajeto, um algoritmo para cálculo de pontos de articulação geral não seja necessário.

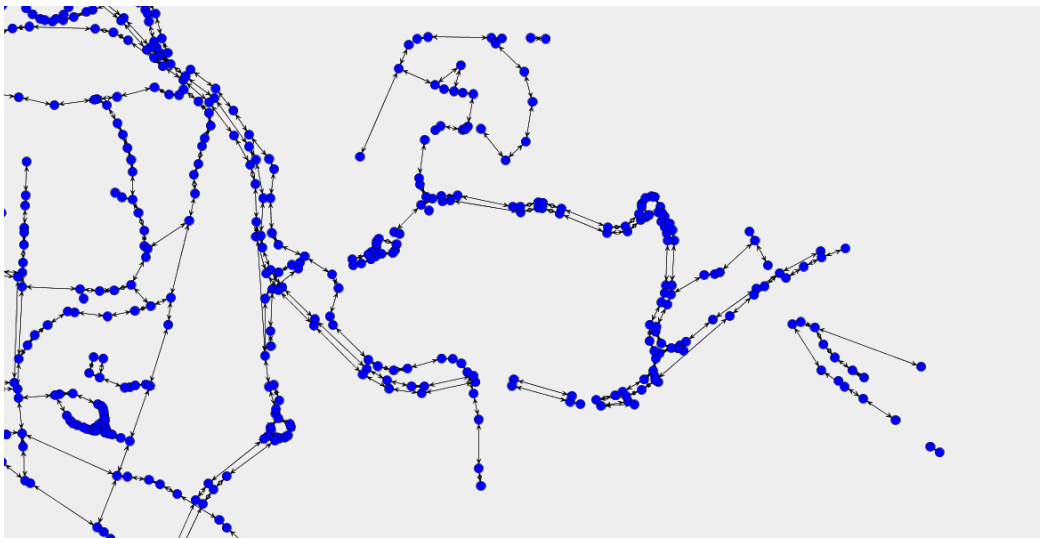


Figura 12 – Exemplo de zonas não acessíveis do grafo

Contudo, a análise da conectividade entre pontos de interesse tornou-se uma prioridade, pois não estando fortemente ligados entre si, o trajeto do(s) autocarro(s) tornar-se-ia impossível. Sendo assim, tanto esse algoritmo como o algoritmo para calcular os pontos de articulação entre PoIs foram implementados. Este último permitiu-nos encontrar alguns pontos de interesse com baixa acessibilidade.

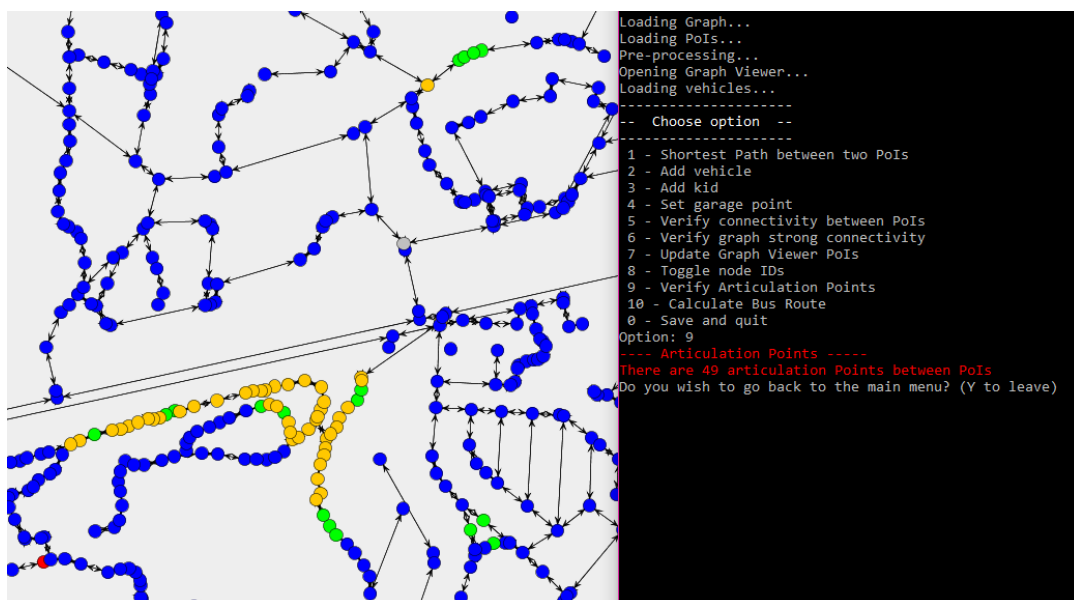


Figura 13 - Visualização de pontos de articulação (Amarelo) entre PoIs (Verde / Vermelho)

Análise de complexidade

Assuma-se que $P = S \cup \{D\} \cup (\cup H_k)$, isto é, o conjunto de todos os pontos de interesse (Escolas, Garagem e Casas).

Complexidade Espacial

A etapa que consumirá mais espaço será o pré-processamento, que cria uma matriz $|P| \times |P|$, para guardar as distâncias. As outras etapas apenas necessitarão de espaço que varia linearmente com o número de pontos de interesse.

Deste modo, a complexidade espacial do cálculo do trajeto será:

$$O(|P|^2)$$

Complexidade Temporal Teórica

Como o algoritmo de Dijkstra será repetido $|P|$ vezes, o pré-processamento terá complexidade temporal:

$$O(|P| (|V| + |E|) \log(|V|))$$

De seguida, o algoritmo MST será da complexidade $O(|E| \log(|V|))$. Por fim, a terceira parte demorará sempre $O(|P|^2)$, já que as restrições impostas ao nível da ordem escolhida não alteram a complexidade temporal do algoritmo. O cálculo do caminho de retorno repetirá essa terceira parte, em $O(|P|^2)$

Deste modo, a complexidade total do cálculo do trajeto será:

$$O(|E| \log(|V|) + |P|^2)$$

Como foi descrito anteriormente, o cálculo de pontos de articulação entre pontos de interesse terá complexidade:

$$O(|V| (|V| + |E|))$$

Complexidade Temporal Empírica

Após correr o programa, será interessante comparar os valores previstos para a complexidade com os valores de execução. Para isso fez-se crescer separadamente o valor de $|P|$ e o valor de $|V|$.

Para o pré-processamento, obtiveram-se os seguintes tempos de execução, primeiro para $|P| = 45$, com $|V|$ variável, e separadamente para $|V| = 10000$, com $|P|$ variável.

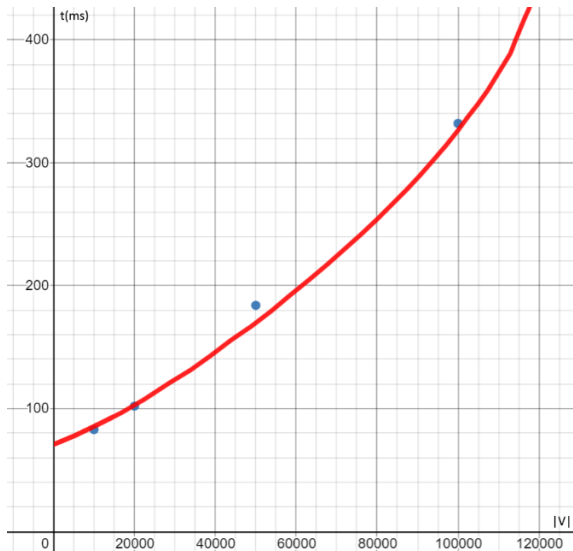


Figura 15 - Crescimento do tempo de execução do pré-processamento, variando $|V|$

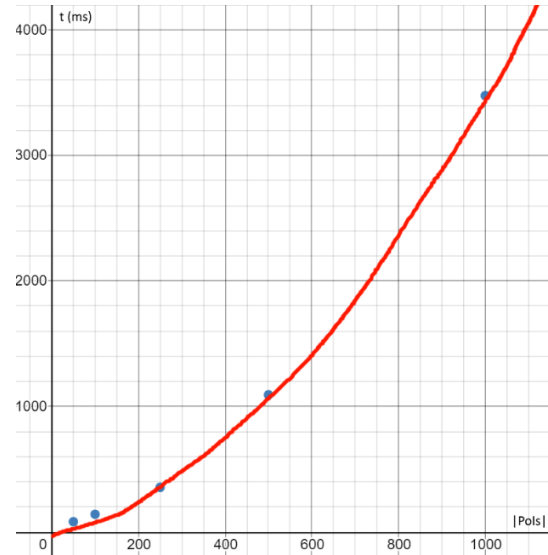


Figura 14 - Crescimento do tempo de execução do pré-processamento, variando $|P|$

Para o cálculo do trajeto, obtiveram-se os seguintes tempos de execução, primeiro para $|P| = 45$, com $|V|$ variável, e separadamente para $|V| = 10000$, com $|P|$ variável

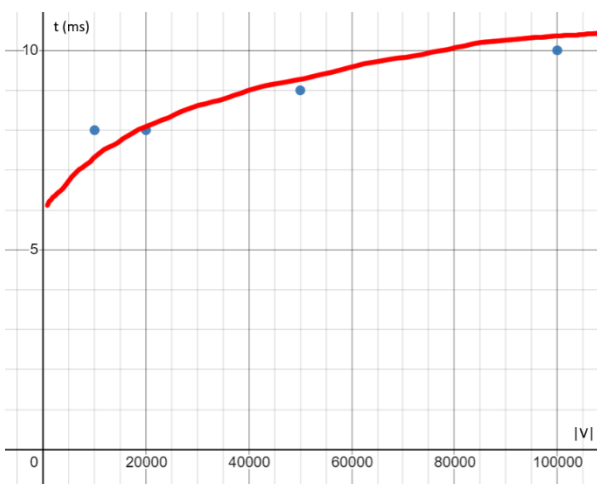


Figura 17 - Crescimento do tempo de execução do cálculo do trajeto, variando $|V|$

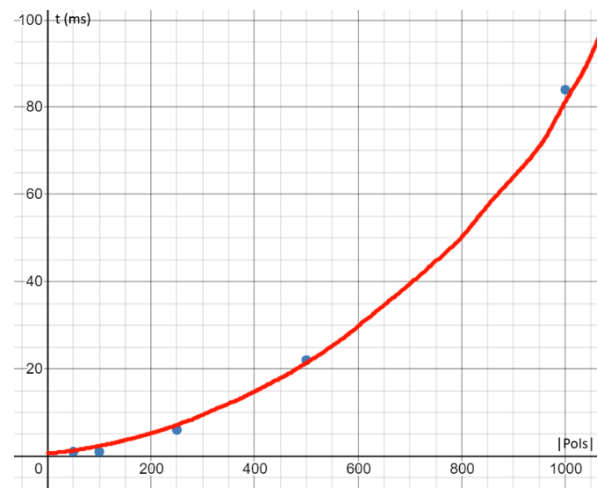


Figura 16 - Crescimento do tempo de execução do cálculo do trajeto, variando $|P|$

Note-se que, embora a complexidade empírica concorde com a teórica para o cálculo do trajeto, no pré-processamento chega a parecer quadrática e não linear, como teria sido previsto. Isso deve-se ao facto de que, após o cálculo de um caminho de dijkstra, seja necessário procurar o vértice no grafo por cada ponto de interesse, sendo essa parte feita ao mesmo tempo do que o pré-processamento para simplificar a implementação.

Por fim, para cálculo de pontos de articulação, apenas se variou $|V|$:

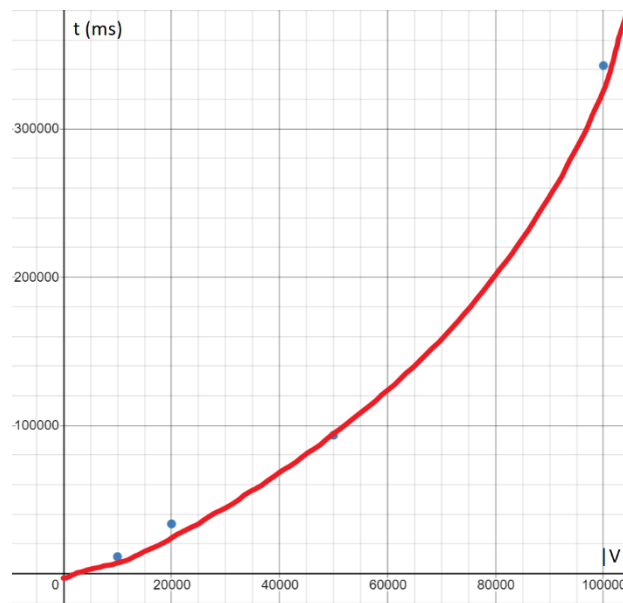


Figura 18 - Crescimento do tempo de execução do cálculo de pontos de articulação, variando $|V|$

Verifica-se que o crescimento da duração é quadrático, como previsto. Embora o algoritmo seja relativamente demorado, é crucial saber quais os pontos que, quando removidos, impediam o cálculo do trajeto dos autocarros.

Estruturas de dados utilizadas

Ao longo do desenvolvimento do projeto, foram utilizadas várias estruturas de dados que facilitaram a elaboração dos diferentes elementos da solução.

Assim, estão presentes nos vários componentes as seguintes estruturas de dados:

- Vector
- Priority Queue
- Queue
- Hash table

No que diz respeito a vetores, estes foram utilizados numa grande parte dos objetos concebidos devido à facilidade de uso e a todas as vantagens que traz: simplificam a inserção e remoção de elementos, e permitem acesso fácil ao número de elementos inseridos e aos próprios elementos, não só com iteradores, mas também a partir de índices.

Deste modo, atendendo a todas estas valências, vetores foram frequentemente preferidos a estruturas de dados como arrays e listas sempre que foi necessário armazenar as diferentes entidades do programa (vértices do grafo, veículos, pontos de interesse, etc.).

Quanto a filas, foram utilizadas filas de prioridade e filas "regulares" (First-In First-Out), estruturas necessárias para a conceção dos algoritmos de Dijkstra e de Prim e do algoritmo de Breadth First Search (BFS), respetivamente. Contudo, ao contrário do que aconteceu com as restantes estruturas de dados, foi usada a implementação de uma fila de prioridade mutável, que foi disponibilizada para as aulas, em detrimento do container priority queue da linguagem C++.

Por fim, foram utilizadas *Hash tables* pois, após uma análise cautelosa a todas as estruturas de dados presentes nos *containers* da linguagem C++, é esta a que mais simplifica o armazenamento e acesso ao caminho e distância entre cada par de POIs, funcionando como um array bi-dimensional, como será descrito abaixo.

No caso dos caminhos, foi utilizada uma *Hash table* tal que a *key* corresponde ao ID do vértice de origem e o *value* constitui uma outra *Hash table*, nesta a *key* trata-se do ID do vértice de destino e o *value* corresponde a um vetor que armazena a sequência de vértices a percorrer para completar o caminho mais curto entre os dois pontos.

Já no caso das distâncias foi utilizada uma *Hash table* que segue o mesmo princípio, no entanto, o vetor de vértices é substituído por um valor inteiro que quantifica a distância.

Casos de utilização

O programa permite ao utilizador a oportunidade de, a partir de um mapa “real”:

- Calcular o caminho mais curto entre quaisquer dois pontos de interesse (garagem, escola, casa)
- Adicionar, remover, alterar e listar registos sobre:
 - Autocarros (e suas capacidades)
 - Crianças (e suas escolas)
- Escolher um local para a garagem dos autocarros.
- Verificar a conectividade entre pontos de interesse (se há caminhos entre todos eles)
- Verificar se o grafo usado é fortemente conexo
- Procurar a existência de pontos de articulação entre pontos de interesse
- Obter o trajeto que cada autocarro deverá tomar para minimizar os custos e entregar todas as crianças às respetivas escolas, assim como o trajeto de retorno.

```
-- Choose option --
1 - Shortest Path between two PoIs
2 - Add vehicle
3 - Add kid
4 - Set garage point
5 - Verify connectivity between PoIs
6 - Verify graph strong connectivity
7 - Update Graph Viewer PoIs
8 - Toggle node IDs
9 - Verify Articulation Points
10 - Calculate Bus Route
0 - Save and quit
Option:
```

Figura 19 - Menu principal da interface

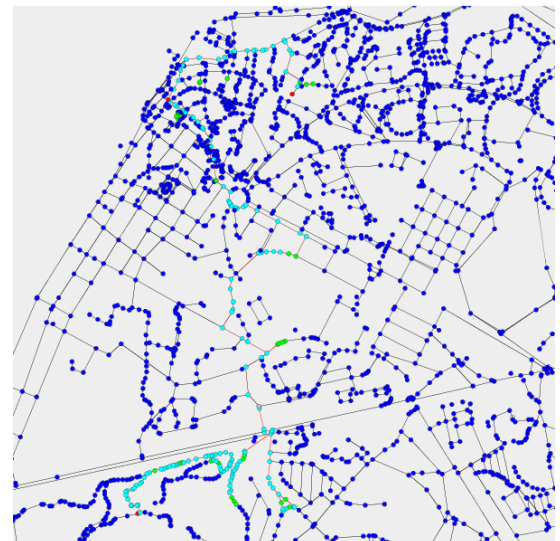


Figura 19 - Visualização do trajeto em modo gráfico

```
-- Vehicle ID: 4 (Capacity: 20) --
---- Path to school ----
312402720 497191395 497191379 1264279389 1264279493 1264279482 1264279403 1264279502 497317073 497311112 497301871 497301862 497301862 497301858 497311324 1105954942 497311009 497311023 497311039 497311073 497301850 497301854 497301858 497301862 497301866 497301871 497311112 497317073 1264279502 1264279403 1264279342 133267070 133267072 133267073 133267074 133267077 133267079 133267080 133267081 133267083 133267084 133267086 133267088 133267089 133267091 133267092 133267093 133267095 133267093 133267092 133267091 133267089 133267088 133267086 133267084 133267083 133267081 133267078 133267077 133267074 133267072 133267070 133267068 133267066 133267064 133267062 133267060 133267058 133267056 133267054 133267052 133267050 133267048 133267046 133267044 133267042 133267040 133267038 133267036 133267034 133267032 133267030 133267028 133267026 133267024 133267022 133267020 133267018 133267016 133267014 133267012 133267010 133267008 133267006 133267004 133267002 133267000 133266998 133266996 133266994 133266992 133266990 133266988 133266986 133266984 133266982 133266980 133266978 133266976 133266974 133266972 133266970 133266968 133266966 133266964 133266962 133266960 133266958 133266956 133266954 133266952 133266950 133266948 133266946 133266944 133266942 133266940 133266938 133266936 133266934 133266932 133266930 133266928 133266926 133266924 133266922 133266920 133266918 133266916 133266914 133266912 133266910 133266908 133266906 133266904 133266902 133266900 133266898 133266896 133266894 133266892 133266890 133266888 133266886 133266884 133266882 133266880 133266878 133266876 133266874 133266872 133266870 133266868 133266866 133266864 133266862 133266860 133266858 133266856 133266854 133266852 133266850 133266848 133266846 133266844 133266842 133266840 133266838 133266836 133266834 133266832 133266830 133266828 133266826 133266824 133266822 133266820 133266818 133266816 133266814 133266812 133266810 133266808 133266806 133266804 133266802 133266800 133266798 133266796 133266794 133266792 133266790 133266788 133266786 133266784 133266782 133266780 133266778 133266776 133266774 133266772 133266770 133266768 133266766 133266764 133266762 133266760 133266758 133266756 133266754 133266752 133266750 133266748 133266746 133266744 133266742 133266740 133266738 133266736 133266734 133266732 133266730 133266728 133266726 133266724 133266722 133266720 133266718 133266716 133266714 133266712 133266710 133266708 133266706 133266704 133266702 133266700 133266698 133266696 133266694 133266692 133266690 133266688 133266686 133266684 133266682 133266680 133266678 133266676 133266674 133266672 133266670 133266668 133266666 133266664 133266662 133266660 133266658 133266656 133266654 133266652 133266650 133266648 133266646 133266644 133266642 133266640 133266638 133266636 133266634 133266632 133266630 133266628 133266626 133266624 133266622 133266620 133266618 133266616 133266614 133266612 133266610 133266608 133266606 133266604 133266602 133266600 133266598 133266596 133266594 133266592 133266590 133266588 133266586 133266584 133266582 133266580 133266578 133266576 133266574 133266572 133266570 133266568 133266566 133266564 133266562 133266560 133266558 133266556 133266554 133266552 133266550 133266548 133266546 133266544 133266542 133266540 133266538 133266536 133266534 133266532 133266530 133266528 133266526 133266524 133266522 133266520 133266518 133266516 133266514 133266512 133266510 133266508 133266506 133266504 133266502 133266500 133266498 133266496 133266494 133266492 133266490 133266488 133266486 133266484 133266482 133266480 133266478 133266476 133266474 133266472 133266470 133266468 133266466 133266464 133266462 133266460 133266458 133266456 133266454 133266452 133266450 133266448 133266446 133266444 133266442 133266440 133266438 133266436 133266434 133266432 133266430 133266428 133266426 133266424 133266422 133266420 133266418 133266416 133266414 133266412 133266410 133266408 133266406 133266404 133266402 133266400 133266398 133266396 133266394 133266392 133266390 133266388 133266386 133266384 133266382 133266380 133266378 133266376 133266374 133266372 133266370 133266368 133266366 133266364 133266362 133266360 133266358 133266356 133266354 133266352 133266350 133266348 133266346 133266344 133266342 133266340 133266338 133266336 133266334 133266332 133266330 133266328 133266326 133266324 133266322 133266320 133266318 133266316 133266314 133266312 133266310 133266308 133266306 133266304 133266302 133266300 133266298 133266296 133266294 133266292 133266290 133266288 133266286 133266284 133266282 133266280 133266278 133266276 133266274 133266272 133266270 133266268 133266266 133266264 133266262 133266260 133266258 133266256 133266254 133266252 133266250 133266248 133266246 133266244 133266242 133266240 133266238 133266236 133266234 133266232 133266230 133266228 133266226 133266224 133266222 133266220 133266218 133266216 133266214 133266212 133266210 133266208 133266206 133266204 133266202 133266200 133266198 133266196 133266194 133266192 133266190 133266188 133266186 133266184 133266182 133266180 133266178 133266176 133266174 133266172 133266170 133266168 133266166 133266164 133266162 133266160 133266158 133266156 133266154 133266152 133266150 133266148 133266146 133266144 133266142 133266140 133266138 133266136 133266134 133266132 133266130 133266128 133266126 133266124 133266122 133266120 133266118 133266116 133266114 133266112 133266110 133266108 133266106 133266104 133266102 133266100 133266098 133266096 133266094 133266092 133266090 133266088 133266086 133266084 133266082 133266080 133266078 133266076 133266074 133266072 133266070 133266068 133266066 133266064 133266062 133266060 133266058 133266056 133266054 133266052 133266050 133266048 133266046 133266044 133266042 133266040 133266038 133266036 133266034 133266032 133266030 133266028 133266026 133266024 133266022 133266020 133266018 133266016 133266014 133266012 133266010 133266008 133266006 133266004 133266002 133265999 133265996 133265993 133265990 133265987 133265984 133265981 133265978 133265975 133265972 133265969 133265966 133265963 133265960 133265957 133265954 133265951 133265948 133265945 133265942 133265939 133265936 133265933 133265930 133265927 133265924 133265921 133265918 133265915 133265912 133265909 133265906 133265903 133265900 133265897 133265894 133265891 133265888 133265885 133265882 133265879 133265876 133265873 133265870 133265867 133265864 133265861 133265858 133265855 133265852 133265849 133265846 133265843 133265840 133265837 133265834 133265831 133265828 133265825 133265822 133265819 133265816 133265813 133265810 133265807 133265804 133265801 133265798 133265795 133265792 133265789 133265786 133265783 133265780 133265777 133265774 133265771 133265768 133265765 133265762 133265759 133265756 133265753 133265750 133265747 133265744 133265741 133265738 133265735 133265732 133265729 133265726 133265723 133265720 133265717 133265714 133265711 133265708 133265705 133265702 133265699 133265696 133265693 133265690 133265687 133265684 133265681 133265678 133265675 133265672 133265669 133265666 133265663 133265660 133265657 133265654 133265651 133265648 133265645 133265642 133265639 133265636 133265633 133265630 133265627 133265624 133265621 133265618 133265615 133265612 133265609 133265606 133265603 133265600 133265597 133265594 133265591 133265588 133265585 133265582 133265579 133265576 133265573 133265570 133265567 133265564 133265561 133265558 133265555 133265552 133265549 133265546 133265543 133265540 133265537 133265534 133265531 133265528 133265525 133265522 133265519 133265516 133265513 133265510 133265507 133265504 133265501 133265498 133265495 133265492 133265489 133265486 133265483 133265480 133265477 133265474 133265471 133265468 133265465 133265462 133265459 133265456 133265453 133265450 133265447 133265444 133265441 133265438 133265435 133265432 133265429 133265426 133265423 133265420 133265417 133265414 133265411 133265408 133265405 133265402 133265399 133265396 133265393 133265390 133265387 133265384 133265381 133265378 133265375 133265372 133265369 133265366 133265363 133265360 133265357 133265354 133265351 133265348 133265345 133265342 133265339 133265336 133265333 133265330 133265327 133265324 133265321 133265318 133265315 133265312 133265309 133265306 133265303 133265300 133265297 133265294 133265291 133265288 133265285 133265282 133265279 133265276 133265273 133265270 133265267 133265264 133265261 133265258 133265255 133265252 133265249 133265246 133265243 133265240 133265237 133265234 133265231 133265228 133265225 133265222 133265219 133265216 133265213 133265210 133265207 133265204 133265201 133265198 133265195 133265192 133265189 133265186 133265183 133265180 133265177 133265174 133265171 133265168 133265165 133265162 133265159 133265156 133265153 133265150 133265147 133265144 133265141 133265138 133265135 133265132 133265129 133265126 133265123 133265120 133265117 133265114 133265111 133265108 133265105 133265102 133265099 133265096 133265093 133265090 133265087 133265084 133265081 133265078 133265075 133265072 133265069 133265066 133265063 133265060 133265057 133265054 133265051 133265048 133265045 133265042 133265039 133265036 133265033 133265030 133265027 133265024 133265021 133265018 133265015 133265012 133265009 133265006 133265003 133264999 133264996 133264993 133264990 133264987 133264984 133264981 133264978 133264975 133264972 133264969 133264966 133264963 133264960 133264957 133264954 133264951 133264948 133264945 133264942 133264939 133264936 133264933 133264930 133264927 133264924 133264921 133264918 133264915 133264912 133264909 133264906 133264903 133264900 133264897 133264894 133264891 133264888 133264885 133264882 133264879 133264876 133264873 133264870 133264867 133264864 133264861 133264858 133264855 133264852 133264849 133264846 133264843 133264840 133264837 133264834 133264831 133264828 133264825 133264822 133264819 133264816 133264813 133264810 133264807 133264804 133264801 133264798 133264795 133264792 133264789 133264786 133264783 133264780 133264777 133264774 133264771 133264768 133264765 133264762 133264759 133264756 133264753 133264750 133264747 133264744 133264741 133264738 133264735 133264732 133264729 133264726 133264723 133264720 133264717 133264714 133264711 133264708 133264705 133264702 133264699 133264696 133264693 133264690 133264687 133264684 133264681 133264678 133264675 133264672 133264669 133264666 133264663 133264660 133264657 133264654 133264651 133264648 133264645 133264642 133264639 133264636 133264633 133264630 133264627 133264624 133264621 133264618 133264615 133264612 133264609 133264606 133264603 133264600 133264597 133264594 133264591 133264588 133264585 133264582 133264579 133264576 133264573 133264570 133264567 133264564 133264561 133264558 133264555 133264552 133264549 133264546 133264543 133264540 133264537 133264534 133264531 133264528 133264525 133264522 133264519 133264516 133264513 133264510 133264507 133264504 133264501 133264498 133264495 133264492 133264489 133264486 133264483 133264480 133264477 133264474 133264471 133264468 133264465 133264462 133264459 133264456 133264453 133264450 133264447 133264444 133264441 133264438 133264435 133264432 133264429 133264426 133264423 133264420 133264417 133264414 133264411 133264408 133264405 133264402 133264399 133264396 133264393 133264390 133264387 133264384 133264381 133264378 133264375 133264372 133264369 133264366 133264363 133264360 133264357 133264354 133264351 133264348 133264345 133264342 133264339 133264336 133264333 133264330 133264327 133264324 133264321 133264318 133264315 133264312 133264309 133264306 133264303 133264300 133264297 133264294 133264291 133264288 133264285 133264
```

Conclusão

Em geral, após esta análise detalhada do problema, posterior investigação de possíveis soluções, e finalmente a sua implementação, foi possível compreender melhor o *Travelling Salesman Problem* e as suas aplicações ao *Vehicle Routing Problem*, assim como alguns modos de os aproximar.

Principais dificuldades encontradas

Ao longo da formalização do problema não foram encontradas muitas dificuldades, já que a sua descrição foi bastante clara e o contexto é muito comum nos dias de hoje.

Porém, descobrir o modo de implementação da solução já foi uma tarefa mais complicada. Inicialmente, a intenção seria procurar algoritmos que dessem a solução ótima, contudo, após uma breve investigação concluiu-se que tais algoritmos não têm uma complexidade temporal razoável. A tarefa complicada foi, então, determinar um processo que permitisse minimizar os custos em tempo polinomial. Após escolhido o processo principal, poder-se-iam aplicar as otimizações necessárias.

Durante a implementação, não foram encontradas também muitas dificuldades, tendo em conta que a solução já havia sido pensada.

Esforço dedicado por elemento

Durante esta primeira fase de formalização e descrição da solução, a elaboração do relatório ficou, em geral, igualmente dividida pelos 3 elementos do grupo. A divisão de tarefas ficou do seguinte modo:

Daniel Brandão

- Tema do Projeto – 1ª iteração
- Algoritmos utilizados: Prim
- Casos de utilização
- Perspetiva de solução: Descrição - 1ª iteração
- Discussão sobre a conectividade do grafo
- Implementação de análise de conectividade
- Implementação de procura de pontos de articulação

Pedro Moás

- Tema do Projeto – 2ª iteração e Conectividade
- Formalização do problema
- Perspetiva de solução: Aplicação de algoritmos – TSP e Conectividade
- Perspetiva de solução: Descrição - 2ª iteração
- Análise de complexidade dos algoritmos (teórica e empírica)
- Implementação da interface
- Implementação de Nearest Insertion no cálculo do trajeto
- Implementação de Pré-processamento

Tiago Silva

- Estrutura geral do relatório
- Tema do Projeto – 3ª iteração
- Algoritmos utilizados: Dijkstra
- Perspetiva de solução: Aplicação de algoritmos – Pré-processamento
- Perspetiva de solução: Descrição - 3ª iteração
- Discussão sobre estruturas de dados utilizadas
- Implementação de MST no cálculo do trajeto
- Implementação de Algoritmo Greedy