Mestrado Integrado em Engenharia Informática e Computação

SchoolBus: Transporte Escolar

Concepção e Análise de Algoritmos

Ano Letivo 2018/2019

Turma 7 Grupo 2

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

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

- Gi = (V, E) - Grafo dirigido pesado composto por
  - V - Conjunto de vértices, sendo que cada um representa um ponto da cidade. Estes guardam:
    - Adj ⊆ 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 ∈ V: Vértice de destino da aresta.
- Ci - 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 ∈ 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 ⊆ 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

- Gf = (V, E) - O mesmo grafo dirigido fornecido como input.
- Cf - 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 ⊆ V - Sequência ordenada de vértices a visitar no caminho de ida, $P_c(n)$ será o vértice número n, do veículo c.
  - R ⊆ 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:

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

Nos dados de saída:

- Gf = Gi, isto é, o grafo deverá permanecer igual ao grafo fornecido como input.
- |Ci| >= |Cf|, isto é, não deverá haver mais carros utilizados do que disponíveis
- $\forall c \in Cf$: Capacity(c) <= |K(c)|, ou seja, o número de crianças que usam este trajeto não pode ser superior à capacidade do autocarro.
- $\forall c \in Cf$:
    - $P_c(1) = D$, 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.
    - $R_c(1) = Pc(n)$, 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:

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

$$g = |Cf|$$

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

Nota: $dist(v_n, v_{n+1})$ - 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 oo 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 vértice 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:

```
DIJKSTRA(G, s):   // G=(V,E), s ∈ V
    for each v ∈ V do
        dist(v) ← ∞
        path(v) ← nil
    dist(s) ← 0
    Q ← ∅   // min-priority queue
    INSERT(Q, (s, 0)) // inserts s with key 0
    while Q ≠ ∅ do
        v ← EXTRACT-MIN(Q) // greedy
        for each w ∈ Adj(v) do
          if dist(w) > dist(v) + weight(v,w) then
            dist(w) ← dist(v)+ weight(v,w)
            path(w) ← v
            if w ∉ Q then  // old dist(w) was ∞
              INSERT(Q, (w, dist(w)))
            else
              DECREASE-KEY(Q, (w, dist(w)))
```

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, …, 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 **$V*2^V$** subproblemas, cada um demorando tempo linear a resolver. Logo, a complexidade temporal e espacial deste algoritmo será **$O(n^2*2^n)$, o que também não é aceitável**, mesmo que seja melhor.

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

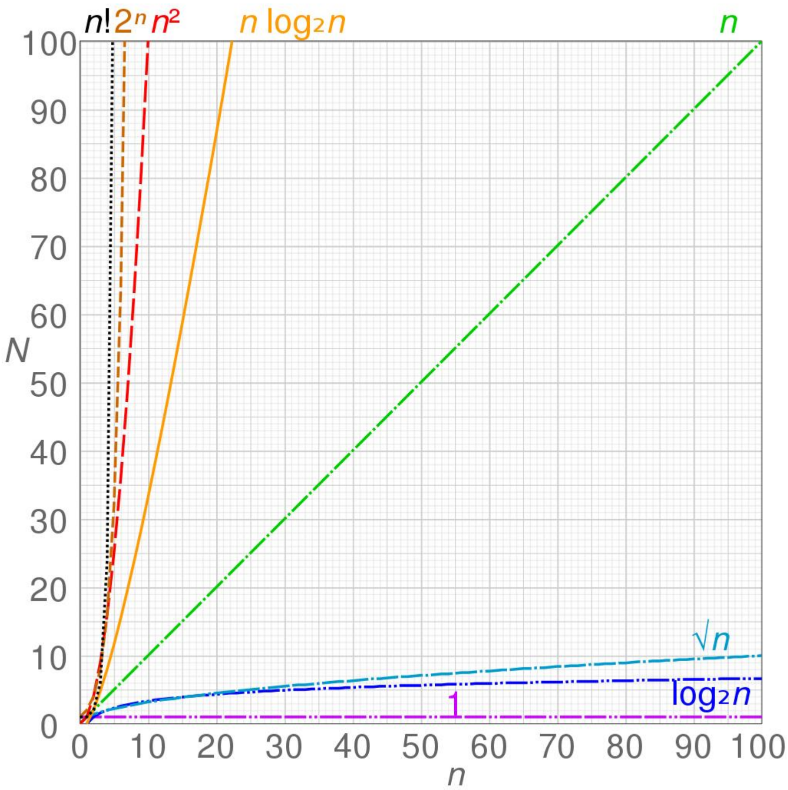

Figura 3 - Variação do tempo de execução (N) com o tamanho do input (n), para várias ordens de complexidade

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.

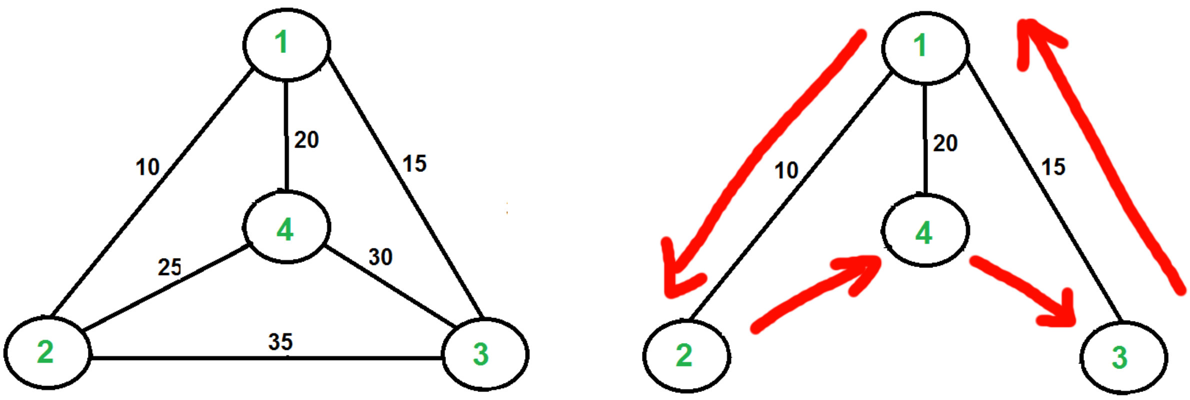

Figura 4 - Solução de TSP obtida (à direita) a partir da aplicação do algoritmo descrito num grafo (à esquerda)

<u>Desigualdade triangular:</u> 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, $dist(V_1, V_2) + dist(V_2, V_3) \leq dist(V_1, V_3)$. Intuitivamente, conclui-se que um grafo que simula uma rede de transporte numa cidade satisfaz a desigualdade triangular na maior parte das situações.

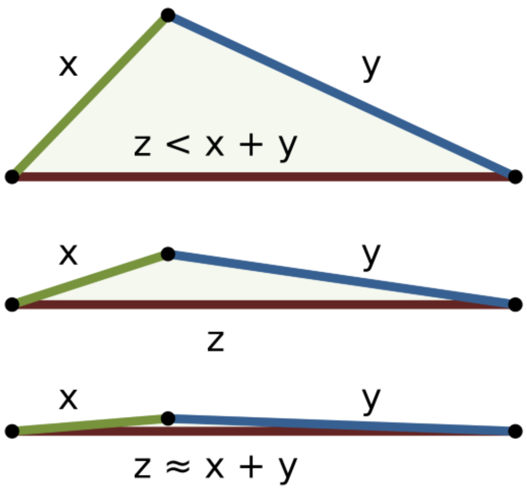

Figura 6 - A propriedade baseia-se no facto de que o maior lado de qualquer triângulo é maior do que a soma dos outros dois.

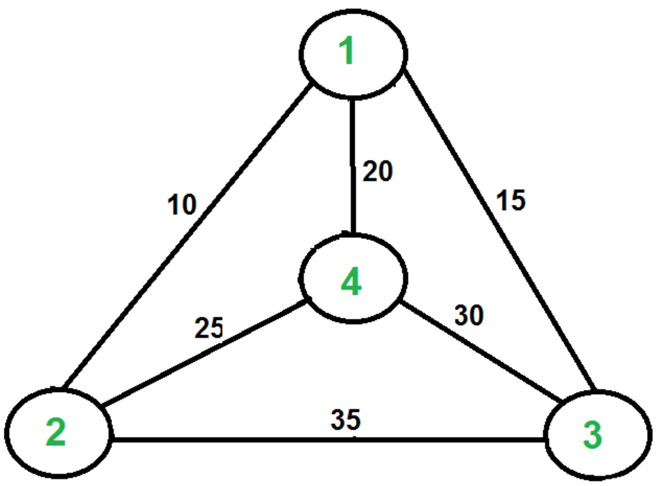

Figura 5 - Grafo que obedece desigualdade triangular

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 $|E| <= |V|^2$, 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:

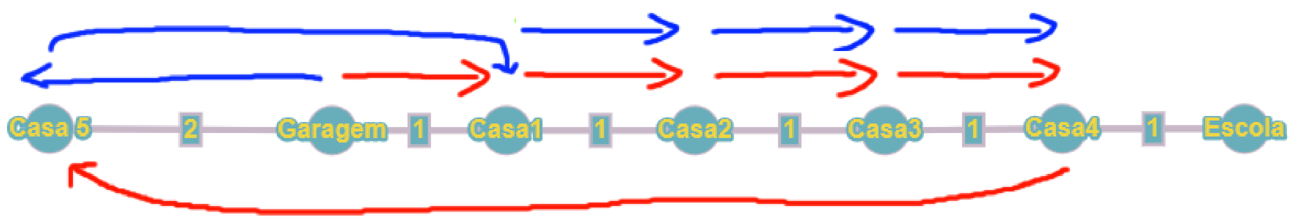

Figura 7 - Caminho obtido por aplicação do algoritmo (a vermelho) vs caminho ótimo (a azul)

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 dist(V_i, V) + dist(V, V_j) - 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||E|), o que seria pouco eficiente, mas mesmo assim aceitável quando comparado com os algoritmos exatos.

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.

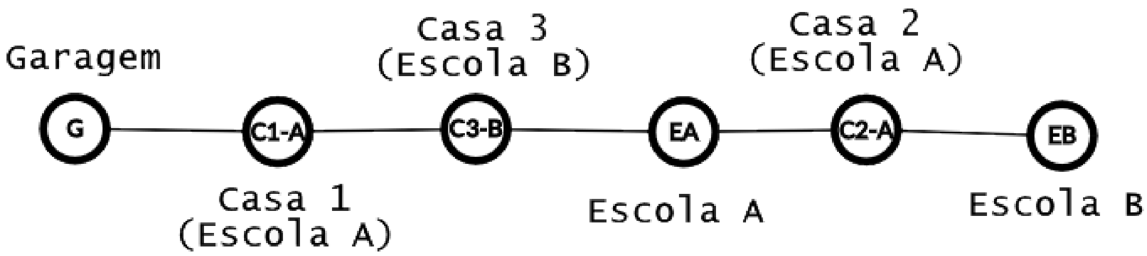

Figura 8 - MST (simplificada) para um conjunto simples de pontos de interesse

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).

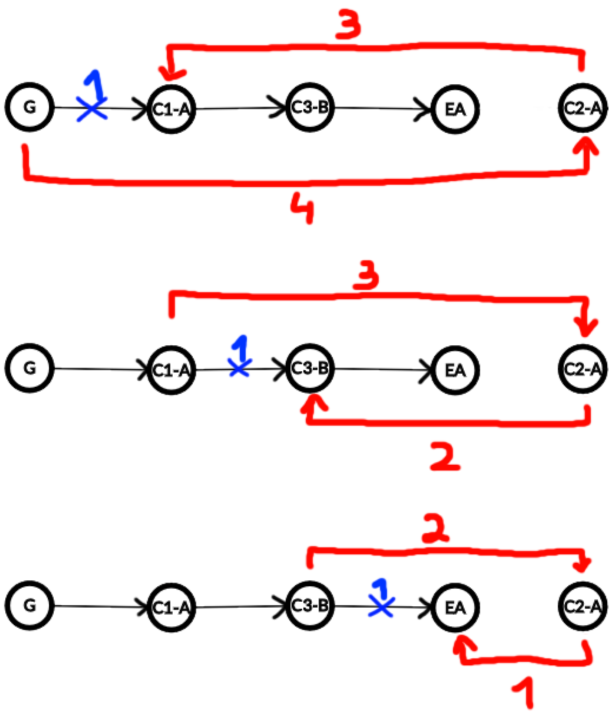

Figura 9 - Análise das 3 possibilidades de inserção de C2-A, sendo a última a que traz um prejuízo menor.

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:

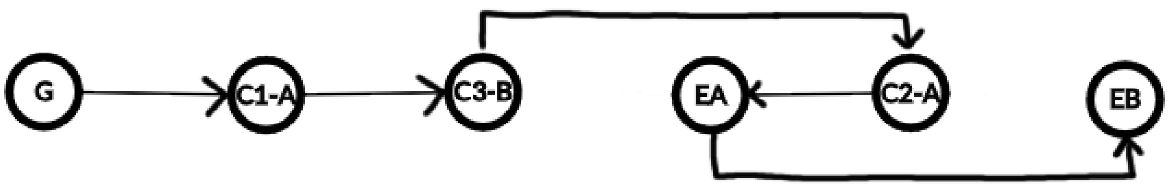

Figura 10 - Solução obtida para o problema, após aplicação do algoritmo descrito

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), caso contrário poderá haver trajetos impossíveis de realizar. 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 DFS 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. Realizar DFS no grafo a partir do último vértice visitado em 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.

Se o grafo for fortemente conexo, é importante determinar a existência de pontos de articulação, isto é, de vértices que, quando removidos, tornam o grafo não fortemente conexo. É informação crucial, pois poderão haver casas com baixa acessibilidade, casos em que obras nas vias públicas impossibilitarão alguns caminhos. Poder-se-á utilizar o seguinte algoritmo:

Para todos os vértices V do grafo:

1. Remover V.
2. Aplicar o algoritmo descrito anteriormente para determinar se o grafo é fortemente conexo. V é um ponto de articulação se, e só se, o grafo deixar de ser fortemente conexo.
3. Voltar a colocar V no grafo.

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).

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.

Complexidade

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

Complexidade Temporal

O pré-processamento terá complexidade temporal $O(|P|\ (|V| + |E|)\ \log(|V|))$, já que o algoritmo de Dijkstra é repetido $|P|$ vezes. De seguida, o algoritmo MST será da complexidade $O(|E|\log(|V|))$. Por fim, a terceira parte demorará sempre $O(|V||E|)$, 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 $O(|V||E|)$.

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

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

O que se pode simplificar para:

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

Complexidade Espacial

A etapa que consumirá mais espaço será o pré-processamento, que cria uma matriz $|P|x|P|$, 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:

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

Casos de utilização

O programa a realizar dará ao utilizador a oportunidade de, a partir de um mapa "real":

- Escolher um local para a garagem dos autocarros
- Adicionar, remover, alterar e listar registos sobre:
  - Escolas
  - Crianças (e suas escolas)
  - Autocarros (e suas capacidades)
- Obter o caminho mais curto entre quaisquer dois pontos de interesse (garagem, escola ou casa)
- 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.

Conclusão

Em geral, após esta análise detalhada do problema e posterior investigação de possíveis soluções, foi possível compreender melhor o *Travelling Salesman Problem* e as suas aplicações, assim como alguns modos de o 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.

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
- Perspetiva de solução: Descrição - Complexidade

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

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