

Concepção e Análise de Algoritmos

# Viagem Pela Europa

Parte 1

Ângela Cardoso, Felipe Galvão e Bruno Madeira



27 de Março de 2015

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>2</b>
<b>2</b>	<b>Restrições Consideradas</b>	<b>3</b>
<b>3</b>	<b>Formulação do Problema</b>	<b>5</b>
3.1	Dados de Entrada . . . . .	5
3.2	Dados de Trabalho . . . . .	5
3.3	Dados de Saída . . . . .	5
3.4	Funções Objetivo . . . . .	6
<b>4</b>	<b>Perspetivas de Solução</b>	<b>8</b>
4.1	Algoritmos de Força Bruta . . . . .	8
4.2	Algoritmo de Held-Karp . . . . .	9
4.3	Algoritmos Aproximados . . . . .	10
<b>5</b>	<b>Métricas de Avaliação</b>	<b>14</b>
<b>6</b>	<b>Principais Dificuldades</b>	<b>16</b>
<b>7</b>	<b>Conclusão</b>	<b>17</b>

# Capítulo 1

## Introdução

No âmbito da disciplina de Concepção e Análise de Algoritmos do Mestrado Integrado em Engenharia Informática e Computação, foi-nos proposta a resolução de um problema cuja formulação é frequentemente feita através da utilização de grafos.

O tema do nosso projeto é o planeamento de uma viagem pela Europa. Mais precisamente, o cliente pretende visitar várias cidades na Europa, deslocando-se de carro entre as cidades. Tem um tempo máximo predefinido para realizar a viagem, assim como uma classificação das cidades consoante o seu interesse, numa pontuação de 0 (pouco interessado) a 10 (muito interessado). Além disso, o cliente disponibiliza informação sobre o tempo de deslocação entre as cidades, assim como o tempo que demora a visitar cada uma delas. A solução a devolver ao cliente será uma lista ordenada de cidades a visitar, tendo em conta que começa e acaba na cidade onde reside.

Numa versão inicial do problema, o tempo para a viagem é limitado e, como tal, poderá não ser possível ir a todas as cidades, o objetivo é visitar as cidades mais interessantes. Numa outra versão, será calculado o tempo mínimo necessário para visitar todas as cidades, sendo devolvido o percurso mais eficiente.

Nesta primeira parte do projeto é apresentada uma formulação matemática do problema, que inclui as restrições assumidas. Serão ainda esboçadas possíveis resoluções, exatas e aproximadas, para as duas versões do problema, assim como propostas de análise do custo computacional dos algoritmos propostos.

# Capítulo 2

## Restrições Consideradas

Cada instância do nosso problema será representada por um grafo, em que os vértices representam as cidades a visitar e as arestas representam os percursos entre as cidades. Apesar de esta representação já contemplar uma série de aspetos do problema, existem ainda várias alternativas consoante os dados fornecidos e as restrições que sejam consideradas. A menos que algo seja dito explicitamente em contrário, à partida, o ideal será não impor restrições que limitem a aplicabilidade dos algoritmos desenvolvidos. Por outro lado, existem casos em que a limitação é recompensada pela eficiência dos algoritmos possíveis.

No caso do planeamento de uma viagem, há quatro questões que devem ser respondidas, por forma a identificar as eventuais restrições a considerar:

1. Dadas duas cidades, existe sempre um caminho que as une sem passar por nenhuma outra cidade da lista? Isto é, podemos considerar um grafo completo?
2. Dado um caminho que une as cidades A e B, existe alguma diferença entre ir de A para B ou de B para A? Isto é, teremos que considerar um grafo dirigido ou não?
3. No percurso devolvido é possível visitar a mesma cidade mais do que uma vez (além da cidade origem, claro)?
4. Podemos assumir que dadas três cidades A, B e C, o tempo de viagem de [A, C] nunca é maior que a soma dos tempos de [A, B] e [B, C]? Isto é, a desigualdade triangular é válida nos custos associados às arestas do grafo?

Apesar de não ser dito de forma explícita, iremos assumir que o grafo é completo, porque não se trata de uma verdadeira restrição para este problema específico. De facto, se não existisse um caminho direto entre as cidade A e C e fosse sempre necessário passar por C, poderíamos considerar uma nova aresta entre A e C, cujo tempo de percurso é a soma dos tempos [A, B] e [B, C]. Na prática isto corresponderia a ir da cidade A para a cidade C passando pela cidade B, mas sem a visitar.

Tipicamente, dentro de localidades, existem diferenças entre ir do ponto A para o ponto B ou fazer o percurso inverso. Já neste caso, à partida não seria muito restritivo assumir que não é necessário considerar arestas dirigidas, porque os tempos de deslocação serão relativamente grandes e pequenas diferenças devidas à orientação poderiam ser desprezadas. No entanto, uma vez que também é necessário considerar o tempo de visita de cada cidade a que chegamos, iremos considerar arestas dirigidos, cujo custo incluirá, além do tempo de viagem, o tempo de visita da cidade de destino. Ora, uma vez que os grafos considerados serão dirigidos, podemos perfeitamente permitir que os tempos sejam diferentes consoante se vá num sentido ou noutro. Convém no entanto notar que, ao considerar grafos completos dirigidos, teremos o dobro das arestas, em relação à versão do problema não dirigido. Este crescimento do grafo poderá ter um impacto negativo na eficiência dos algoritmos desenvolvidos.

Dado que o objetivo da primeira versão do problema é visitar as cidades mais interessantes dentro do tempo limite, visitar duas vezes a mesma cidade não é nada aconselhável, será melhor reservar esse tempo para visitar outra cidade diferente. Isto poderá não ser tão claro se considerarmos que, estando na cidade A, regressar a uma cidade já visitada B e depois visitar a cidade C é mais rápido do que ir diretamente de A para C. Ora, isto está obviamente relacionado com a resposta à quarta questão. Além disso, está também relacionado com a resposta à primeira questão. Se ir de A para C passando por B é mais rápido, porque não considerar que o caminho de A para C é precisamente a junção dos caminhos de A para B e B para C? Posto desta forma, não parece muito restritivo considerar que a resposta à quarta questão é sim, donde resulta que não há vale a pena visitar duas vezes a mesma cidade. Por outro lado, como veremos mais adiante, em termos de complexidade dos algoritmos, não há vantagem em impor a desigualdade triangular. Já a passagem pelo mesmo vértice mais do que uma vez, conduz a um tipo de problema bastante diferente. Sendo assim, vamos assumir que cada cidade é visitada no máximo uma vez e que a desigualdade triangular poderá não se verificar.

Em suma, temos um grafo completo dirigido, do qual queremos extrair um percurso fechado que passa por cada vértice no máximo uma vez. Se passa por todos os vértices ou não, depende da versão do problema que estivermos a considerar.

# Capítulo 3

## Formulação do Problema

### 3.1 Dados de Entrada

- $N$  - número de cidades que o cliente gostaria de visitar, incluindo a cidade de partida;
- $T_{max}$  - tempo de que o cliente dispõe para a viagem na primeira versão do problema;
- $C[i]$ ,  $(1 \leq i \leq N)$  - vetor com o nome de cada cidade a visitar, por exemplo  $C[2] = Madrid$ ;
- $V[i]$ ,  $(1 \leq i \leq N)$  - vetor com o valor (entre 0 e 10) atribuído a cada cidade, por exemplo  $V[2] = 7$  representa a pontuação atribuída a *Madrid*;
- $T[i, j]$ ,  $(1 \leq i, j \leq N)$  - matriz com os tempos de viagem entre cada par de cidades, quando  $i = j$ ,  $T[i, i]$  é o tempo de visita da cidade  $i$ .

Note-se que podemos referir-nos a cada cidade pelo seu índice, que é o mesmo no vetor  $C$ , no vetor  $V$  e na matriz  $T$ . Além disso, iremos assumir que a cidade 1 é a cidade de origem do cliente, que terá sempre que fazer parte do percurso, sendo  $V[1] = 0 = T[1, 1]$ .

### 3.2 Dados de Trabalho

Estes dados dependerão da versão do problema e do algoritmo usado para o resolver, pelo que serão apresentados numa próxima parte do projeto.

### 3.3 Dados de Saída

Para a primeira versão do problema, em que existe um tempo máximo para a viagem  $T_M$  e são consideradas as pontuações atribuídas às cidades, a solução final terá o formato seguinte:

- $M$  - número de cidades do percurso obtido;
- $I[i]$ ,  $(1 \leq i \leq M)$  - vetor com o índice de cada cidade do percurso, pela ordem em que é visitada, isto é, o percurso final será  $I[1] -> I[2] -> \dots -> I[M] -> I[1]$ ;
- $V_{max}$  - valor do percurso obtido, ou seja, é a soma das pontuações das cidades do vetor  $P$  e, numa solução exata, é a pontuação máxima possível dentro do limite de tempo  $T_{max}$ .

Para a segunda versão do problema, em que serão visitadas todas as cidades e queremos qual a ordem de visita e o tempo necessário, a solução final terá o formato seguinte:

- $T_F$  - tempo do percurso obtido, no caso de uma solução exata, é o menor tempo necessário para visitar todas as cidades;
- $I[i]$ ,  $(1 \leq i \leq N)$  - vetor com o índice de cada cidade do percurso, pela ordem em que é visitada, isto é, o percurso final será  $I[1] -> I[2] -> \dots -> I[N] -> I[1]$ .

### 3.4 Funções Objetivo

Mais uma vez, consoante a versão do problema, temos diferentes funções objetivo. Na primeira versão, queremos maximizar a soma de valores das cidades visitadas, garantindo que o tempo total do percurso não ultrapassa o tempo de viagem disponível. Isto é, assumindo que  $I[M + 1] = I[1] = 1$  é a cidade de origem, queremos obter um vetor  $I$  que cumpra as seguintes condições:

$$\mathcal{P}(I) : \sum_{1 \leq i \leq M} (T[I[i], I[i]] + T[I[i], I[i + 1]]) \leq T_{max}; \quad (3.4.1)$$

$$\begin{aligned} & \sum_{1 \leq i \leq M} V[I[i]] = V_{max} = \\ & \max \left\{ \sum_{1 \leq j \leq \bar{M}} V[J[j]] : |J| = \bar{M}, J[j] \in \{1, 2, \dots, N\}, \mathcal{P}(J) \text{ é válida} \right\}. \end{aligned} \quad (3.4.2)$$

Na segunda versão, queremos minimizar o tempo de visita a todas as cidades. Isto é, assumindo que  $I[N + 1] = I[1] = 1$  é a cidade de origem, queremos obter um vetor  $I$  que

cumpra a seguinte condição:

$$\begin{aligned} & \sum_{1 \leq i \leq N} (T[I[i], I[i]] + T[I[i], I[i+1]]) = T_F = \\ \min & \left\{ \sum_{1 \leq j \leq N} (T[J[j], J[j]] + T[J[j], J[j+1]]) : J[j] \in \{1, 2, \dots, N\} \right\}. \end{aligned} \quad (3.4.3)$$



# Capítulo 4

## Perspetivas de Solução

O problema de determinar o melhor percurso para visitar todas as cidades é conhecido como o Problema do Caixeiro Viajante (PCV) [1] e será assim que nos iremos referir a esta parte do trabalho. Relativamente à versão do problema em que existe um limite de tempo é conhecido como o Problema do Roteamento de Veículos (PRV) [2].

### 4.1 Algoritmos de Força Bruta

Este tipo de algoritmos geralmente são os mais simples de descrever e implementar. Costumam corresponder à primeira solução que ocorre à maior parte das pessoas que tentem resolver o mesmo problema. Para os nossos problemas (PCV e PRV), esta abordagem é mais uma vez a mais simples. No entanto, como já mencionamos tem um custo temporal de  $O(N!)$ , que o inutiliza ao fim de cerca de 20 cidades.

Concentremo-nos para já no Problema do Caixeiro Viajante.

A ideia deste algoritmo é enumerar praticamente todos os percursos, utilizando para isso uma árvore em que os nós serão os vértices e a sua sequência na árvore indica a ordem pela qual aparecem no percurso. Para cada nó, os seus filhos serão todos os vértices que ainda não tenham sido visitados até então. A árvore é percorrida em profundidade primeiro e só depois em largura. Assim, se a dada altura conseguirmos deduzir que uma sequência de nós é inviável (porque o seu tempo de percurso é superior ao de outra já analisada), descartamos todos as estratégias filhas dessa, retrocedendo a um ponto em que ainda haja estratégias viáveis não estudadas.

A este tipo de abordagem dá-se o nome de algoritmo de retrocesso, porque ao podar um determinado conjunto de soluções da árvore, regressamos a um ponto prévio para o qual ainda há possibilidades a estudar.

Apesar de o número total de estratégias ser  $N!$ , com o descartar precoce de algumas é possível reduzir o tempo de computação da solução. No entanto, ao guardar todas as

estratégias numa árvore, a complexidade espacial será também da ordem de  $N!$ . É possível melhorar isto utilizando outro algoritmo de força bruta sem recorrer a uma árvore. A ideia é para cada  $i$  entre 1 e  $N!$  gerar a  $i$ -ésima sequência (utilizando para isso uma resolução matemática), calcular o tempo de percurso dessa sequência e manter guardada apenas a melhor sequência encontrada até então. Com esta abordagem, o custo temporal será mesmo  $O(N!)$ , pois não existe a possibilidade de poda de soluções, no entanto, o custo espacial é  $O(N)$ , porque não mantemos registo de todas as sequências, apenas uma.

Para o caso do Problema de Roteamento de Veículos, isto é, quando temos um limite de tempo, o algoritmo de retrocesso é muito semelhante. No entanto, a cada instante, quando o tempo limite é atingido, não inspecionamos o resto daquele ramo da árvore. Por outro lado, não podemos descartar um ramo por ser mais demorado que outro visto antes, dado que o mais importante nesta versão do problema é maximizar o valor dos vértices visitados.

Para o algoritmo em que se gera cada sequência, já parece não haver tanta vantagem. De facto, interessa-nos inspecionar todas as subsequências de vértices do grafo, que ao todo são  $\sum_{1 \leq i \leq N} i!$ . Claro que para valores elevados de  $N$ , o que interessa é a parcela  $N!$ , isto é, a complexidade temporal continua a ser  $O(N!)$ , mas o tamanho da árvore será no máximo  $N!$  e o mais certo é que se possa podar uma parte significativa. Aliás, se não se conseguir podar nada de um ramo é porque existe alguma sequência com todos os vértices que cabe no tempo limite e isso seria imediatamente uma solução para o problema. Em termos de complexidade espacial, mantém-se a análise feita para o PCV, a cada instante apenas temos que guardar a melhor subsequência encontrada, que tem no máximo  $N$  vértices.

## 4.2 Algoritmo de Held-Karp

Um dos melhores algoritmos conhecidos para resolver o PCV é o algoritmo de Held-Karp [3]. Este é um algoritmo típico de programação dinâmica e assenta no facto de todo o subpercurso de um percurso de distância mínima ter também distância mínima. Sendo assim, a ideia é resolver o PCV para todos os subconjuntos de vértices, utilizando recursividade. Isto é, para cada subproblema, iremos utilizar as soluções ótimas dos problemas ainda mais pequenos do que ele. No final, à semelhança de outros problemas

de programação dinâmica (como o problema da mochila de que falamos na secção seguinte) obtém-se o percurso da seguinte forma:

- o primeiro vértice do percurso é o vértice obtido no passo final da recursão;
- o vértice seguinte obtém-se utilizando o subproblema com todos os vértices exceto aquele que já incluímos;
- repete-se o passo anterior até ter o percurso completo.

Uma vez que este algoritmo prevê a resolução de todos os subproblemas do PCV, à partida será possível deduzir dele uma solução também para o caso em que temos um tempo limitado para a viagem.

A grande vantagem deste algoritmo é ter uma complexidade temporal de  $O(N^2 2^N)$ , permitindo obter soluções exatas mais rapidamente do que os algoritmos de força bruta.

## 4.3 Algoritmos Aproximados

Novamente, vamos concentrar-nos primeiro no PCV. Uma abordagem simples para obter uma solução aproximada é utilizar um algoritmo ganancioso, em que a cada passo se visita a cidade mais próxima. Este algoritmo tem complexidade temporal  $O(N^2)$ , dado que para cada vértice se irá procurar de entre os vértices não visitados o mais próximo. Estudos mostram que dado um conjunto aleatório de vértices, este algoritmo em média obtém um percurso 25% mais demorado que a solução ótima. Se assumirmos que o grafo cumpre a desigualdade triangular, então o desempenho do algoritmo (em termos da qualidade da solução encontrada) é ainda melhor.

Em relação ao PRV, a heurística que propomos é baseada no problema da mochila. A capacidade da mochila será o tempo limite para fazer a viagem e cada cidade representará um objeto, com o valor correspondente ao interesse do cliente em visitá-la e um peso correspondente a uma métrica de tempo para a visitar que explicaremos de seguida.

O algoritmo funcionaria em 4 etapas que serão efetuadas por ordem:

1. Calcular o peso de cada vértice do grafo.
2. Utilizar a solução usual do problema da mochila para seleccionar um subconjunto de cidades a visitar.

3. Com um algoritmo aproximado, resolver o PCV para o subconjunto de cidades obtido.
4. Verificar se o percurso obtido ultrapassa o tempo limite para a viagem:
  - 4.1. Se sobrar tempo, tentar adicionais mais cidades, utilizando um algoritmo ganancioso em que a métrica seria a razão entre o valor da cidade e o tempo para a visitar.
  - 4.2. Se o percurso ultrapassar o tempo limite, utilizar a mesma métrica para retirar as cidades menos valiosas até obter um percurso válido.

O primeiro ponto tem uma complexidade temporal  $O(N^2)$ , dado que basta somar ao tempo de visita de cada cidade o maior tempo de percurso de uma aresta com destino nessa cidade, para obter o peso do vértice correspondente. Note-se que ao utilizar o custo da pior aresta que chega a cada vértice estamos a garantir que para o subconjunto de cidades escolhido no segundo passo é de facto possível encontrar um percurso que seja realizável dentro do tempo limite. No entanto, uma vez que iremos utilizar um algoritmo aproximado para encontrar o percurso, poderá ser necessário retirar cidades, daí considerarmos duas alternativas no passo 4.

Uma vez realizado o primeiro passo temos dois vetores  $V[i]$ ,  $1 \leq i \leq N$  com os valores das cidades (fornecido inicialmente) e  $P[i]$ ,  $1 \leq i \leq N$  com o peso de cada cidade. Também sabemos a capacidade da mochila  $T_{max}$ .

Assim, o segundo passo resolve-se da forma usual fazendo variar  $i$  entre 0 e  $N$  vamos calculando recursivamente dois vetores:

- $c[k]$ ,  $1 \leq k \leq T_{max}$ , contém o valor máximo que conseguimos acumular numa mochila de capacidade  $k$ , utilizando apenas os vértices de 1 a  $i$ ;
- $b[k]$ ,  $1 \leq k \leq T_{max}$ , contém o último item adicionado à mochila de capacidade  $k$  para obter  $c[k]$ , utilizando apenas os vértices de 1 a  $i$ .

No final, o subconjunto de cidades ideal para a mochila de capacidade  $T_{max}$  é:  $b[T_{max}]$ ,  $b[T_{max} - P[b[T_{max}]]]$ , etc. Quanto à complexidade temporal deste passo, seria  $O(NT_{max})$ .

Alternativamente, para resolver este segundo passo, podemos utilizar um algoritmo de *branch and bound* [4, 5]. Para isso considera-se uma árvore cujos nós serão os estados sucessivos da mochila consoante se coloca lá cada um dos objetos:

- A raiz da árvore será o estado inicial da mochila, isto é, vazia.
- Esse nó terá dois filhos, consoante se colocar na mochila o primeiro objeto ou não.
- De facto, à exceção das folhas da árvore, cada nó terá dois filhos, consoante se coloca na mochila o objeto seguinte ou não.
- As folhas representam estados finais da mochila. Um percurso até uma folha, indica para cada objeto se foi lá colocado ou não.

Esta é a parte de *branch* do algoritmo. No entanto, não iremos calcular a árvore na sua totalidade, porque essa seria uma abordagem de força bruta, com um custo temporal de  $O(2^N)$ . É aqui que entra a parte de *bound* do algoritmo. Para o caso da mochila será um *upper bound*, ou seja, um majorante. Para cada estado da mochila (ou nó da árvore), iremos calcular o majorante do valor máximo que ainda é possível colocar na mochila:

- Começamos por calcular para cada objeto a razão entre o seu valor e o seu peso.
- Ordenamos os objetos por ordem decrescente da razão calculada.
- Enquanto houver espaço na mochila, acrescentamos ao majorante o valor de cada um dos objetos pela ordem anterior.
- Se sobrar espaço para parte de um objeto, mas não a sua totalidade, adicionamos ao majorante a parte correspondente do valor da cidade.

Este cálculo do majorante de um nó será utilizado para escolher a cada momento qual o nó que desejamos expandir: será aquele com maior majorante. Se numa dada expansão não for possível adicionar um objeto, por falta de espaço na mochila, paramos de expandir nesse sentido. O importante é escolher sempre o nó com maior majorante de entre todos aqueles que ainda não foram expandidos. No final teremos o melhor subconjunto de objetos a colocar na mochila.

Também é possível utilizar um algoritmo de *branch and bound* para calcular uma solução exata do PCV [5, 6]. Note-se ainda que poderíamos também utilizar uma heurística para resolver a parte da mochila, de forma a obter um algoritmo mais rápido.

O terceiro passo poderia ser resolvido com o algoritmo ganancioso descrito no início desta secção, ficando apenas por resolver o último ponto. Independentemente de ser

necessário, nesta última fase, acrescentar ou retirar cidades do percurso obtido, o processo é o mesmo:

- escolher cidade ideal (mais valiosa se for para acrescentar, menos valiosa se for para retirar);
- se for para acrescentar, escolher o melhor ponto do percurso (onde as viagens para a nova cidade são mais curtas), retirar uma aresta entre duas cidades do percurso e acrescentar a nova cidade e as respectivas arestas;
- se for para retirar, basta substituir a cidade e as suas arestas pela aresta que une as duas cidades do percurso às quais ela estava ligada.

Sendo assim, a complexidade temporal deste último passo é  $O(N^2)$ , tal como no terceiro passo. O que significa que ao todo a complexidade temporal do algoritmo é  $O(\max N^2, NT_{max})$ , ou seja, é pseudo-polinomial, comparando muito favoravelmente com a complexidade factorial das soluções exatas. Resta avaliar a qualidade da aproximação obtida. Para isso utilizaremos vários conjuntos de cidades nos quais testaremos o algoritmo exacto e o algoritmo aproximado, por forma a determinar quão pior é a solução aproximada.

# Capítulo 5

## Métricas de Avaliação

O Problema do Caixeiro Viajante é um dos problemas de optimização mais estudados e é sabido que se trata de um problema  $NP$ -difícil [7, 8]. Informalmente, isto significa que é pelo menos tão difícil como os problemas mais difíceis em  $NP$ . Posto de outra forma, encontrar uma resolução exata para este problema em tempo polinomial seria equivalente a resolver todos os problemas em  $NP$  em tempo polinomial, ou seja, um tal algoritmo provaria que  $P = NP$ .

$P$  é a classe de todos os problemas que podem ser resolvidos em tempo polinomial. Já  $NP$ , vem da expressão em inglês para tempo polinomial não determinístico e contém todos os problemas para os quais é possível validar uma solução em tempo polinomial. Facilmente se conclui das definições que  $P \subseteq NP$ . No entanto há uma imensidão de problemas em  $NP$  para os quais não se conhece nenhum algoritmo polinomial, entre eles o PCV. A comunidade científica está maioritariamente convencida que tais algoritmos não existem, ou seja, que  $P \neq NP$ . No entanto, até hoje ninguém conseguiu provar uma coisa ou outra, sendo um dos problemas mais famosos da Ciência dos Computadores, cuja solução dá direito a 1 milhão de dólares.

A complexidade temporal da resolução do PVC por um algoritmo de força bruta é  $O(n!)$  num grafo com  $n$  vértices, dado que existem  $n!$  percursos possíveis. A complexidade espacial é constante, dado que a cada momento apenas é necessário guardar o percurso melhor até então e o seu custo. Existem algoritmos de programação dinâmica, nomeadamente o algoritmo de Held–Karp de que falamos, que calculam soluções exatas mais rapidamente, com um custo temporal de  $O(n^2 2^n)$ . No entanto, a complexidade espacial deste algoritmo é  $O(2^n n)$ , significativamente pior que um algoritmo de força bruta.

Devido ao elevado custo computacional do cálculo de soluções exatas, são conhecidos inúmeros algoritmos que providenciam soluções aproximadas. De facto, a procura de heurísticas para resolver o PVC é das áreas mais prolíferas da optimização. Por exemplo, se assumirmos que os tempos de viagem respeitam a desigualdade triangular, um algo-

ritmo ganancioso em que se escolha sempre a cidade não visitada mais próxima, obtém aproximações razoáveis da solução ótima com complexidade temporal  $O(n^2)$ .

Relativamente ao Problema do Roteamento de Veículos, tal como o PVC, este problema é NP-difícil, sendo o algoritmo de força bruta semelhante também ao que é utilizado para o PVC.

Nestes problemas o custo mais crítico é o temporal, isto é, com o crescimento do número de vértices, aquilo que impedirá a obtenção de uma solução exata será o tempo que esse cálculo demora e não a memória necessária. Assim, no que diz respeito às métricas de avaliação dos algoritmos que implementarmos, iremos concentrar os nossos esforços na avaliação da complexidade temporal dos mesmos. Para isso, faremos várias experiências com um número crescente de cidades. De seguida, para um conjunto de experiências com o mesmo número de vértices, utilizaremos a média dos tempos de computação como aproximação do tempo necessário para esse número de vértices. No final, para cada algoritmo, deverá ser possível obter um gráfico discreto cuja interpolação levará à sua função de complexidade temporal.

Tal como discutido anteriormente, no caso dos algoritmos aproximados será também necessário avaliar a qualidade da aproximação, além do tempo necessário para a calcular. O método será semelhante: utilizar vários conjuntos aleatórios, com várias dimensões, e calcular para cada amostra a diferença entre a solução exata e a solução aproximada.



# Capítulo 6

## Principais Dificuldades

A parte do problema com um tempo limite para fazer a viagem não parece ser muito conhecida. De facto, não chegamos a encontrar nenhum problema análogo ao nosso, mesmo a informação sobre o Problema de Roteamento de Veículos parece focar-se em problemas ligeiramente diferentes. Mais, este problema é pelo menos tão difícil como a versão sem tempo limite. De facto, se o tempo limite for suficientemente grande, passamos a calcular o melhor percurso para visitar todas as cidades, em vez de escolher um subconjunto de cidades a visitar. Por isso fomos obrigados a tratar os dois problemas em conjunto, o que complicou a tarefa.

Por outro lado, a formulação/escolha de algoritmos concretos e o modelo a utilizar são dependentes um do outro, o que gerou uma maior incerteza inicial sobre como abordar a resolução do problema. Logo à partida, a escolha do tipo de grafo a usar gerou algumas confusões. Havia várias hipóteses e cada uma delas necessitava de ser tratada de diferente forma, fazendo o uso de algoritmos diferentes. Ao ponderar possíveis resoluções, pequenas ideias que intuitivamente pareciam corretas eram frequentemente falaciosas. Além disso, tentar conciliar diferentes técnicas para chegar a um algoritmo relativamente eficiente parecia uma tarefa impossível.

Apesar deste atrito inicial, existe bastante informação disponível sobre problemas ligados a grafos e desde cedo tivemos conhecimento do Problema do Caixeiro Viajante, que nos levou a escolher um grafo completo e consequentemente simplificar a análise do problema. A ajuda dos docentes também se verificou valiosa na formulação de soluções.

# Capítulo 7

## Conclusão

Esta fase inicial de pesquisa e formulação de soluções permitiu aprofundar o conhecimento relativo a problemas similares e possíveis resoluções. A equipa foi exposta a alguns dos algoritmos a ser implementados na próxima fase do trabalho prático e já tem uma noção geral de como prosseguir.

Ficamos também com maior sensibilidade face à escolha dos tipos de dados a usar, nomeadamente no que diz respeito às restrições consideradas e até que ponto ajudam a encontrar algoritmos mais eficientes.

Será certamente gratificante passar tudo isto à prática numa próxima fase do trabalho, embora a quantidade de algoritmos diferentes que estamos a considerar faça esta tarefa parecer hercúlea. Eventualmente será necessário optar por apenas alguns destes caminhos, consoante a dificuldade de implementação e a promessa de eficiência de cada um.

# Bibliografia

- [1] Wikipedia, “Travelling salesman problem,” 2015, [Online; accessed 25-March-2015]. [Online]. Available: [http://en.wikipedia.org/wiki/Travelling\\_salesman\\_problem](http://en.wikipedia.org/wiki/Travelling_salesman_problem)
- [2] —, “Vehicle routing problem,” 2015, [Online; accessed 25-March-2015]. [Online]. Available: [http://en.wikipedia.org/wiki/Vehicle\\_routing\\_problem](http://en.wikipedia.org/wiki/Vehicle_routing_problem)
- [3] M. Held and R. M. Karp, “A dynamic programming approach to sequencing problems,” *J. Soc. Indust. Appl. Math.*, vol. 10, pp. 196–210, 1962.
- [4] S. Saurabh, “0/1 knapsack problem using branch and bound method,” 2013, [Online; accessed 26-March-2015]. [Online]. Available: <https://youtu.be/slayHO7gKEQ>;<https://youtu.be/R6BQ3gBrfjQ>
- [5] Y. Narahari, “Data structures and algorithms,” 2000, [Online; accessed 26-March-2015]. [Online]. Available: <http://lcm.csa.iisc.ernet.in/dsa/index.html>
- [6] S. Saurabh, “Branch and bound (travelling salesman problem),” 2013, [Online; accessed 26-March-2015]. [Online]. Available: [https://youtu.be/FJkT\\_dRjX94](https://youtu.be/FJkT_dRjX94);<https://youtu.be/KzWC-t1y8Ac>
- [7] M. R. Garey and D. S. Johnson, *Computers and intractability*. W. H. Freeman and Co., San Francisco, Calif., 1979, a guide to the theory of NP-completeness, A Series of Books in the Mathematical Sciences.
- [8] C. H. Papadimitriou, *Computational complexity*. Addison-Wesley Publishing Company, Reading, MA, 1994.