Informática para Ciências e Engenharias B

Trabalho Prático Nº 1 — 2017/18

1 Objectivo do Trabalho

Cansados das aulas de ICE, um grupo de alunos decidiu fazer um passeio pela Europa para desanuviar. Infelizmente, descobriram que mesmo para isso um pouco de programação pode ser útil. Neste trabalho irão criar algumas funções para calcular a distância total de um percurso por cidades europeias.

O módulo `tp1_data.py` tem uma variável `cities` com uma lista de nomes de cidades europeias e uma variável `distances` com uma lista de listas de distâncias entre cidades. A ordem das cidades é a mesma em todas estas listas. Tal como vimos para as strings (aula teórica 5), os objectos do tipo lista também implementam o método `index`, que devolve o índice de um elemento na lista. Como a tabela das distâncias segue a mesma ordem da lista com os nomes, se quisermos saber a distância entre duas cidades basta obter os índices das cidades e consultar a tabela. Notem que a tabela é simétrica:

```
In : from tp1_data import cities, distances
In : cities.index('Lisbon')
Out: 16

In : cities.index('Madrid')
Out: 20

In : distances[16][20]
Out: 638

In : distances[20][16]
Out: 638
```

Assim, os alunos têm toda a informação de que necessitam para planear o seu itinerário. Falta agora uma forma de a processar adequadamente.

O objectivo deste trabalho será conceber, implementar e testar algumas funções que facilitem a tarefa de planear esta viagem, de forma a visitar as cidades desejadas minimizando o tempo de viagem, os custos em combustível e as emissões de carbono. Para permitir os testes automáticos, todas as funções implementadas terão de ter a assinatura compatível com o que está indicado no enunciado: o nome exactamente igual ao indicado no enunciado, os parâmetros pela mesma ordem e o valor devolvido pela função ser exactamente o especificado. Além disso, todas as funções pedidas no enunciado devem ser implementadas num ficheiro de nome `tp1.py`. Tem de ter exactamente este nome (em minúsculas).

À parte dessas restrições, quaisquer outras funções auxiliares que não venham mencionadas no enunciado e que queiram implementar podem ficar nesse ficheiro ou noutros módulos, conforme preferirem. Podem também usar quaisquer elementos válidos da linguagem Python e qualquer biblioteca que venha instalada na distribuição Anaconda recomendada. Por exemplo, a biblioteca Numpy disponibiliza objectos que simplificam as operações com matrizes e vectores e que são compatíveis com as listas para todos os efeitos neste trabalho. Por exemplo, pode implementar uma função em que seja pedido devolver uma lista como devolvendo um vector da biblioteca Numpy, se preferirem.

2 Cálculo das distâncias num trajecto

A primeira função a fazer deverá devolver as distâncias parciais, entre cidades, dado um trajecto definido pelos índices das cidades na tabela das distâncias. Deve ter a seguinte assinatura:

```
def partial_distances(distances,indexes):
```

O primeiro parâmetro recebe a lista de listas (a tabela) das distâncias e o segundo uma lista com os índices das cidades a visitar ao longo do trajecto. Esta função devolverá uma lista com as distâncias entre cidades consecutivas. Por exemplo, para saber as distâncias parciais do trajecto Lisboa → Madrid → Lisboa, usando a tabela `distances` do módulo `tp1_data`, devemos obter:

```
In : partial_distances(distances,[16,20,16])
Out: [638, 638]
```

porque o índice de `'Lisbon'` na lista `cities` é 16 e o de `'Madrid'` é 20. Notem que a lista das distâncias parciais tem menos um elemento do que a lista dos índices das cidades.

2.1 Dias de viagem

Outro factor importante para determinar o itinerário são os dias gastos a viajar entre cidades. Se a viagem for suficientemente curta, ainda dá tempo para passear nesse dia na cidade de destino. Se a distância for intermédia, perde-se o dia na viagem entre cidades. E se for grande, será necessário pernoitar no caminho e acaba por se perder dois dias a viajar entre essas cidades. Para ter isto em consideração, é melhor criar uma tabela (uma lista de listas) semelhante à das distâncias mas com os dias de viagem entre cidades. A assinatura dessa função deverá ser:

```
def travel_days(distances, short, max_drive):
```

O primeiro parâmetro recebe a tabela com as distâncias, o segundo a distância abaixo da qual a viagem é considerada curta e, por isso, conta como zero dias de custo, e o terceiro parâmetro recebe valor acima do qual a viagem é considerada longa demais para se fazer de seguida, exigindo pernoitar e por isso custando dois dias. As distâncias entre estes valores, inclusive, contam como um dia de viagem. A função deverá devolver uma tabela (uma lista de listas) com as mesmas dimensões da tabela das distâncias como se mostra abaixo:

```
In : travel_days(distances,200,800)
Out:
[
[0, 0, 2, 2, 1, 2, 1, 1, 1, 2, 2, 1, 2, 2, 1, 1, 2, 1, 1, 2, 2, 2, 2, 2, ...],
[0, 0, 2, 2, 1, 1, 0, 1, 1, 2, 2, 1, 2, 2, 1, 1, 2, 1, 1, 2, 2, 2, 2, 2, ...],
[2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, ...],
[2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1, 1, 1, 2, 2, ...],
...
[2, 1, 2, 2, 2, 0, 1, 1, 1, 2, 2, 1, 1, 1, 2, 1, 2, 2, 1, 1, 2, 1, 1, 1, ...]
]
```

Em alternativa, se preferirem, esta função pode devolver um objecto `ndarray` do Numpy. Seja como for, cada lista, ou cada linha da matriz, deverá corresponder a uma linha da tabela original, e a ordem das listas (ou linhas da matriz) deverão também corresponder à ordem da original.

2.2 Custo total

Há dois critérios importantes para os alunos decidirem um trajecto. Querem reduzir o impacto ambiental da sua viagem, minimizando os quilómetros totais do percurso. Além disso, não querem passar muitos dias de viagem, para poderem passar mais tempo nas cidades que querem visitar. Por isso, precisamos de uma função que nos diga o custo total de um itinerário, tanto em quilómetros como em dias de viagem entre cidades. Essa função deverá ter a assinatura como indicado abaixo, recebendo como argumentos a tabela das distâncias em quilómetros, a tabela dos dias de viagem criada pela função anterior e os índices das cidades, por ordem, que se quer visitar.

```
def total_cost(distances, days, indexes):
```

O resultado devolvido por esta função deve ser o número total de quilómetros e o número total de dias de viagem.

```
In : days = travel_days(distances,200,800)
In : total_cost(distances, days, [16,20,16])
Out: (1276, 2)
```

Note que já tem uma função que devolve o comprimento de cada troço da viagem, usando a matriz das distâncias. Deve reutilizar sempre as funções anteriores. Pense também se precisa, ou não, de outra para os dias. Para calcular o total, pode basear-se na aula teórica 6 ou usar a função `sum` como indicado nessa aula.

2.3 Custo total pelos nomes das cidades

A função anterior é pouco prática de usar porque exige saber os índices das cidades. Seria preferível conseguir obter o custo total da viagem usando os nomes das cidades. Para resolver esse problema, criamos primeiro uma função que devolva uma lista com os índices das cidades que queremos visitar. Essa função deverá receber como parâmetros a lista de referência com os nomes das cidades pela ordem que constam nas tabelas, como a lista em `cities`, e a lista dos nomes das cidades cujos índices se quer obter:

```
def city_indexes(reference, names):
```

Devolve então os índices das cidades indicadas. Se quisermos saber os índices para o trajecto Lisboa → Madrid → Lisboa, poderemos fazer:

```
In : city_indexes(cities,['Lisbon','Madrid','Lisbon'])
Out: [16, 20, 16]
```

Podemos agora fazer uma função que, recebendo a a lista de referência com os nomes das cidades, a tabela das distâncias, a tabela dos dias de viagem e uma lista com as cidades a visitar, nos devolve o custo total em quilómetros e dias. A assinatura deverá ser:

```
journey_cost(city_names, distances, days, visit):
```

e o resultado devolvido deve ser correspondente ao da função anterior, do custo total:

```
In : journey_cost(cities, distances, days, ['Lisbon','Madrid','Lisbon'])
Out: (1276, 2)

In : itinerary=  ['Lisbon', 'Madrid', 'Barcelona', 'Milan', 'Genoa', 'Rome', 'Nice',
                  'Lyon', 'Geneva', 'Cologne', 'Amsterdam', 'Paris', 'Berlin', 'Lisbon']
```

```
In : journey_cost(cities, distances, days, itinerary)
Out: (10199, 14)
```

Note que esta função deve ser implementada recorrendo às que já estão implementadas. Não deve repetir código desnecessariamente.

3 Encontrar o melhor caminho

Atenção: não implementar esta função desconta apenas 1 valor na avaliação, e esta parte do trabalho é mais exigente do que as anteriores. Por isso, considerem esta função como opcional, apenas para quem achou o resto demasiado fácil.

O problema genérico de encontrar o caminho mais curto, ou de menor custo, para visitar um número de destinos é conhecido como o problema do caixeiro viajante. Não se conhece (e provavelmente não existe) um algoritmo eficiente que garanta encontrar o melhor trajecto sem exigir um custo computacional que cresce exponencialmente com o número de destinos visitados [1].

No entanto, é relativamente simples implementar o seguinte algoritmo:

- É nos dada uma lista de cidades a visitar. A primeira cidade da lista é a cidade de partida e também a de chegada, pelo que tem de estar no início e no fim do itinerário. As outras cidades devem ficar no itinerário de forma a minimizar o custo da viagem.
- O que queremos minimizar é principalmente os dias gastos a viajar entre cidades, para maximizar o tempo que temos para passear nas cidades. No entanto, em caso de empate, preferimos itinerários que minimizem a distância total percorrida, para poupar gasolina e poluir menos.
- Para tentar encontrar um itinerário melhor vamos testar cinquenta mil itinerários gerados aleatoriamente, com as cidades a visitar, mas sempre partindo da, e chegando à, cidade indicada em primeiro lugar.
- Se essa pesquisa aleatória permitiu reduzir o custo, quer em dias ou, em caso de empate, em distância total, tentamos novamente com mais cinquenta mil itinerários e vamos repetindo até já não haver melhorias entre dois lotes de cinquenta mil simulações.
- No final, a função deve devolver os nomes das cidades, por ordem, formando um itinerário. Note que o nome da última cidade deve ser igual ao primeiro, pois queremos voltar a casa no fim, mas nenhuma outra cidade deve ser repetida.

A assinatura deve ser como indicado abaixo, com o primeiro parâmetro sendo a lista de referência dos nomes de cidades, o segundo recebendo a tabela com as distâncias em quilómetros, o terceiro a tabela com os dias de viagem e o último a lista de cidades a visitar, sem nomes repetidos (assume-se que a primeira é a origem e destino final do percurso):

```
def best_trip(cities, distances,days, to_visit):
```

Uma forma prática de gerar os percursos aleatoriamente é usando a função `permutation` do módulo `numpy.random`[2]. Mas não se esqueçam de que a cidade de origem, e que também é o destino final do itinerário, tem de se manter sempre a mesma. Só as outras é que podem mudar de ordem.

[1] `https://en.wikipedia.org/wiki/Travelling_salesman_problem`

[2] Podem encontrá-la descrita aqui: `https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.permutation.html`

Notem também que ter várias variáveis a referir a mesma lista pode trazer problemas. Mas se precisarem de criar uma cópia de uma lista podem usar a função `list`, que devolve uma cópia de uma lista dada como argumento.

No final, depois de melhorar o itinerário até já não haver melhorias a cada lote de 50000 simulações, a função deverá devolver a lista ordenada dos nomes das cidades para o itinerário completo, incluindo o destino final de regresso à cidade de partida, bem como o comprimento total da viagem em quilómetros e o número de dias passados em viagem entre cidades. Esta função deverá usar as funções anteriores sempre que for necessário, e deve também ser decomposta em funções mais simples conforme seja adequado.

Como exemplo do resultado esperado, dando estas cidades numa ordem arbitrária, o melhor percurso encontrado foi de 6904, pela ordem indicada, estimando-se gastar 10 dias a viajar entre estas cidades.

```
In : to_visit = ['Lisbon', 'Madrid','Paris','Amsterdam','Barcelona',
                 'Berlin','Cologne','Nice','Lyon']
In : best_trip(cities, distances,days, to_visit)
Out: (['Lisbon', 'Madrid', 'Barcelona', 'Nice', 'Lyon', 'Paris', 'Cologne',
       'Berlin', 'Amsterdam', 'Lisbon'],
      6904, 10)
```

4 Critérios de Avaliação do Trabalho

De acordo com o Regulamento de Avaliação de Conhecimentos da FCT/UNL,[3] os estudantes directamente envolvidos numa fraude são liminarmente reprovados na disciplina. Em ICE, considera-se que um aluno que **dá** ou que **recebe** código num trabalho comete fraude. Os alunos que cometerem fraude num trabalho não obterão frequência.

Os trabalhos serão avaliados de acordo com os seguintes critérios.

- Utilização correta dos elementos básicos de Python.
- Implementação correcta das funções pedidas.
- Código legível, com nomes adequados e funções documentadas.

A maior parte da cotação total corresponde à implementação das funções pedidas nos pontos 2.1, 2.2 e 2.3. O estilo do código (nomes, organização e documentação) terá um peso pequeno, mas ainda significativo.

Não implementar função pedida no ponto 3 (encontrar o melhor caminho) desconta apenas um valor na nota final do trabalho. Dado o grau de dificuldade dessa função, recomendo que só tentem implementá-la depois de concluído tudo o resto e apenas se acharem o restante trabalho aborrecido por ser demasiado fácil.

Qualquer omissão ou incorrecção na implementação contribui para reduzir a nota, mas é sempre preferível entregar um trabalho incompleto ou com erros do que não entregar nada.

A nota final do trabalho prático estará sujeita a ajustes conforme a defesa de nota que cada aluno, individualmente, fará no dia do primeiro teste.

[3] Em http://www.fct.unl.pt/sites/default/files/documentos/estudante/informacao_academica/Reg_Aval.pdf

5 Testes, entrega e enunciado

Serão disponibilizados em breve na página da disciplina os módulos com dados e um script para testes locais que poderão usar para testar as vossas funções antes de submeter o trabalho.

Será também publicado um guião com as instruções detalhadas para a entrega.

Esse material, bem como este enunciado, poderão sofrer pequenas alterações caso for necessário esclarecer algumas dúvidas recorrentes. Por favor prestem atenção aos avisos caso haja actualizações e mantenham as vossas versões actualizadas.