Departamento de Engenharia Informática

AED

2016/2017

• U  C •

Relatório Projecto 2

Gonçalo Amaral 2015249122
Jorge Martins 20151199193
Cristóvão Gouveia 2015420420

Introdução

No decorrer da cadeira de AED como segundo projeto foi-nos proposto que pensássemos num algoritmo que resolve-se da maneira mais rápida o problema enunciado. O problema baseia-se numa edição do Rally Dakar que estaria sujeita a novas regras.

Novas regras essas que permitem que os pilotos pudessem escolher a sua própria rota com a condição de não poderem falhar nem passar duas vezes pela mesma cidade e que a cidade de partida seria também a cidade de chegada para todos os pilotos .

Para tal antes da edição do Rally Dakar era necessário criar uma rota que respeitasse as regras anteriormente faladas. Essas rotas seriam baseadas em mapas que podem ser representados como um grafo completo.

Descrição das tarefas

Tarefa 1.1

Nesta tarefa era pedido que criasse-mos um método baseado em grafos que permitisse representar e manipular um mapa. Para manipulação do grafo usamos o módulo networkx, pois continha todos os métodos necessários para uma fácil manipulação de grafos.

Tarefa 1.2

A tarefa 1.2 consiste em programar um gerador de mapas que assumia que a distância entre quaisquer duas cidades era sempre igual em ambos os sentidos. Este gerador está implementado no método generateMap da classe Graph. Também usamos o módulo numpy para gerar números inteiros aleatórios segundo uma distribuição uniforme, de maneira a obter valores o mais dispersos possível.

Tarefa 1.3

Aqui era pedido que criássemos um algoritmo que apresentasse a melhor solução possível para o nosso problema (caminho que implique percorrer menor distância respeitando as regras). Este algoritmo apenas seria o ponto de partida.

Porém, o grupo achou por bem criar dois algoritmos de modo a podermos comparar resultados . O primeiro algoritmo foi um brute force é um algoritmo pouco otimizado e lento, pois necessita de calcular todas os caminhos possíveis e as suas distâncias. Como o bruteforce não o nosso algoritmo final, não nos preocupamos com a sua implementação, usamos o módulo itertools para gerar todas as permutações possíveis. Partimos do princípio que este módulo teria os seus métodos bastantes otimizados, tendo assim uma sólida base de comparação. O segundo algoritmo de nome Dolly tenta ser uma otimização ao Bruteforce.

Tarefa 1.4

Nesta tarefa era nos pedido para testar a implementação feita várias vezes de maneira a saber qual o maior número de cidades que seria possível calcular em menos de 30 minutos. Os resultados estarão na secção de resultados neste documento.

Tarefa 1.5

Para finalizar a tarefa 1 era nos pedido para optimizar o nosso algoritmo e executar a parte 4 desta tarefa de novo.

Tarefa 2

Ao longo da leitura do enunciado retivemos que na tarefa 2 seria pedido o mesmo que na tarefa 1 apenas alterando uma propriedade do grafo, as arestas que unem dois vértices passariam a poder ter pesos diferentes ao contrário da tarefa anterior em que teriam obrigatoriamente o mesmo peso.
Para tal o método generateMap tem como argumento nWay, este argumento 1 e 2. Se for 1 significa que duas cidades têm sempre a mesma distância independente do sentido, caso o nWay for 2 significa que as duas cidades têm distâncias diferentes dependo do sentido.

Tarefa 3

O grupo achou bem optar por uma interface de consola pois seria mais rápido de programar.

Descrição algoritmo

Primeiramente, podemos representar todas as rotas válidas para o nosso problema como uma árvore onde cada nó seria uma cidade e na folha final que representaria a última cidade, teria por sua vez uma folha que seria a cidade inicial, completando assim o ciclo.
Partindo desta base, facilmente percebemos que um algoritmo de brute force iria percorrer toda a árvore e calcular as distâncias de todas as rotas. Tendo um número n de cidades teríamos (n-1)! caminhos possíveis.
O algoritmo criado tenta evitar chegar a todas as folhas da árvore. Imaginemos que a nossa árvore é um labirinto, que cada nó é uma intersecção no labirinto. Temos um indivíduo inicial, imaginemos que é uma ovelha. Esta ovelha tem a distância caminhada (walked), a distância para a próxima interseção (toWalk), os nós ja visitados (marks), os nós que faltam visitar (marksLeft) e para que nó se dirige (towards). Inicialmente temos uma ovelha no primeiro nó, esta ovelha irá clonar-se até existirem tantas ovelhas quantos os nós para onde se pode dirigir. Estas novas ovelhas clonadas teriam as mesmas propriedades que a ovelha inicial, alterando a toWalk para a distância do nó atual até ao nó para onde se dirige, e a towards para o nó para onde se dirige. De seguida iremos atualizar todas as ovelhas existentes. Pegamos na ovelha com a menor distância a caminhar, subtraímos esse valor à variável toWalk de todas as ovelhas no sistema e incrementamos esse valor à variável walked. Durante este processo de atualização sempre que uma ovelha fique com a variável toWalk a 0, adicionamos a identificação do nó presente na variável towards e removemos da variável marksLeft e alteramos a

variável towards para None. Na próxima iteração seguinte iremos clonar todas as ovelhas que não se dirigem para nó nenhum e de seguida atualizar todas as ovelhas. Para além disso quando uma ovelha não tiver mais nós para onde ir, iremos dirigir essa ovelha para o nó inicialmente. Quando uma das ovelhas chegar ao nó inicial, termina o algoritmo e obtemos a ovelha que percorreu a menor distância.

Vantagens

O nosso algoritmo brilha especialmente nos casos em que há caminhos muito mais curtos, pois o programa termina antes que os caminhos restantes se ramifiquem, apenas calculando um número de caminhos menor que (n-1)!. Iremos ver isto comprovado nos testes onde o número de cidades é superior.

Desvantagens

A clara desvantagem é que se o tamanho dos caminhos for muito próximo, teremos o mesmo número de caminhos calculados que o brute force, adicionando a computação extra requerida.

Resultados

Criamos vários mapas e para cada um deles executamos uma vez cada algoritmo e obtivemos os seguintes valores médios.

Versão 1 - Tarefa 1

Dolly	Bruteforce	Cidades
0.00095	0.00013	2
0.00280	0.00011	3
0.00990	0.00016	4
0.02900	0.00026	5
0.12000	0.00110	6
0.85000	0.01200	7
2.2	0.05900	8
16.7	0.49000	9
38.5	4.93	10
131.9	54.3	11
RAM insuficiente	RAM insuficiente	12

Versão 2 - Tarefa 1

Dolly	Bruteforce	Cidades
0.00013	0.00011	2
0.00018	0.00016	3
0.00066	0.00025	4
0.00190	0.00054	5
0.00760	0.00160	6
0.02600	0.01500	7
0.23000	0.08600	8
0.82000	0.79000	9
15.0	7.9	10
4.9	87.4	11
170.2	RAM insuficiente	12
>1800	RAM insuficiente	13

Versão 1 - Tarefa 2

Dolly	Bruteforce	Cidades
0.00042	0.00012	2
0.00130	0.00009	3
0.00640	0.00019	4
0.02300	0.00035	5
0.12000	0.00120	6
0.65000	0.00930	7
2.39	0.05500	8
3.29	1.46	9
53.6	4.89	10
103.7	55.5	11
RAM insuficiente	RAM insuficiente	12

Versão 2 - Tarefa 2

Dolly	Bruteforce	Cidades
0.00028	0.00010	2
0.00025	0.00011	3
0.00055	0.00017	4
0.00130	0.00048	5
0.00550	0.00160	6
0.05500	0.01500	7
0.07300	0.08500	8
0.77000	0.79000	9
5.85	7.80	10
37.5	87.5	11
272.6	RAM insuficiente	12
>1800	RAM insuficiente	23

Conclusões

Podemos concluir que os resultados batem certo com o que esperávamos, mesmo ignorando a falta de RAM, reparamos que quanto mais cidades houver melhor será a eficiência do nosso algoritmo pois a probabilidade de haver caminhos muito longos aumenta.
Achamos que conseguimos atingir as metas propostas, em todas as tarefas.

Notas finais

Antes da implementação do menu, todos os resultados estavam a ser consistentes, quer no cálculo do caminho mais curto lendo de ficheiros, quer através de mapas criados on the fly. Reparámos que na tarefa 1 na versão 1 com 10 cidades, este cálculo não era correto, mas após guardar o grafo criado e ler o ficheiro o resultado estaria consistentemente correto. Este erro também ocorreu na tarefa 2 toda.
Se analisarmos os resultados podemos constatar que há algumas variâncias visto que o bruteforce é sempre o mesmo. Estas variâncias devem-se, naturalmente ao estado de utilização do computador no dado momento. Mas ao nível de comparação entre o bruteforce e o algoritmo está equilibrado pois os testes foram consecutivos.

Todo o programa foi feito na versão 3.5.2 do python usando os módulos não default itertools, networkx e numpy.