FACULDADE DE CIÊNCIAS E TECNOLOGIA DA UNIVERSIDADE DE COIMBRA
2016/2017

ALGORITMOS E ESTRUTURA DE DADOS

Rally Dakar

Projeto nº 2

Nuno Ferreira 2013153319

Aspetos teóricos relevantes

Grafos

Grafos são estruturas discretas compostas por vértices, ou nós e por arestas, ligando os nós. Vértice é parte fundamental do nosso grafo. Pode ter nome e informações adicionais acerca do problema a tratar. Aresta representa uma ligação entre dois nós do grafo. Peso é um atributo não obrigatório da aresta. Serve para representar o custo de ir de um vértice para outro.

Grafos completos

De forma a minimizar o número de soluções, o grafo de caminhos que vamos utilizar é sempre um grafo completo. Este é um grafo simples em que todos os vértices são adjacentes a todos os outros vértices. Um aspeto importantíssimo para este projeto é que o número de arestas de um grafo completo é dado por $\frac{n(n-1)}{2}$, sendo n o número de vértices do grafo. Como veremos mais á frente esta expressão será usada para calcular o número de cidades do grafo a partir de um ficheiro carregado para o programa.

Matriz de adjacência

Numa matriz de adjacência cada linha e cada coluna representa um vértice de um grafo. O valor que se encontra na intersecção de uma linha com uma coluna representa o peso da ligação entre os dois nós.

Estrutura de dados

Grafo

Cada nó do grafo possui a seguinte estrutura:

- Símbolo
- Dicionário com as ligações

O símbolo representa o nome do respetivo nó. O dicionário com as ligações associadas com cada vértice tem como chave o símbolo do respetivo nó e como chave o respetivo peso.

Escolhi este tipo de estrutura uma vez que possui complexidades temporais e espaciais relativamente aceitáveis e porque é uma estrutura que tem já tem funções estabelecidas que me permitem alterar a estrutura de forma a adicionar/remover/alterar mais rápida solucionando o problema mais rápido. Também escolhi esta estrutura uma vez que, segundo a minha pesquisa, é bastante usada na resolução de problemas relativamente a grafos.

Para juntar todos os nós e formar um grafo completo decidi usar uma lista simples.

Algoritmos usados para as tarefas

Símbolos usados

Cerca de 141052 símbolos estão disponíveis no ficheiro symbols.txt. Este ficheiro tem todas as letras e números disponíveis. Esses números são reordenados de forma a formarem *strings* que vão até 3 caracteres. Os símbolos foram criados e copiados através de um website.

Distâncias usadas entre cidades

Um aspeto importante sobre este projeto era o range das distâncias entre cidades. Após falar com o professor regente decidi optar por uma abordagem realista. Como tal verifiquei durante a minha pesquisa para este projeto que durante o último Rally Dakar a distância máxima nunca passava dos 1000 km. Como tal decidi que a distância mínima fixar-se-ia os 50km e a distância máxima nos 1000km.

Criação do grafo

Para a criação do grafo o algoritmo criado carrega todos símbolos disponíveis e guarda aqueles somente necessários, de acordo com o número de cidades que o utilizador introduziu. Por exemplo, se o utilizador pediu 20 cidades a lista vai ser constituída com os primeiros 20 símbolos do ficheiro. Posteriormente é criado um dicionário temporário que permite adicionar as ligações aos nós. O algoritmo coloca o primeiro símbolo e depois adiciona n-1 ligações. O valor destas ligações é dado pela função *random* e é guardado como valor no dicionário. Para a tarefa 1 é feita uma verificação adicional em que vai ás ligações já criadas e procura pela ligação. Se esta já existir copia o valor. É importante mencionar que esta verificação não é feita para todos os nós. Apenas aqueles que estão atrás do nó atual na lista de nós são verificados. Quando todas as arestas tiverem criadas é feito um *deepcopy* do dicionário para a estrutura do nó conjuntamente com o símbolo. Finalmente esse nó é adicionado á lista que representa o grafo.

Criação do caminho mais curto

De acordo com enunciado o caminho mais curto teria de passar por todas as cidades exatamente uma vez, exceto a cidade de origem. Este é um exemplo do clássico problema do caixeiro viajante (*travelling salesman problem*). Este problema permite representar o seguinte problema: "Dada uma lista de cidades e as distâncias entre elas, qual é o caminho mais curto possível que permita visitar cada cidade exatamente uma vez e retornar a cidade de origem?". Para este problema escolhi implementar um *greedy algorithm* designado de *nearest neighbour algorithm*. É um algoritmo bastante simples de implementar, daí o ter escolhido para este projeto. Estes são os passos do algoritmo:

1 - Começar na cidade origem;

2 - Encontrar a ligação mais curta que ligue o vértice atual e um vértice ainda não visitado V;

3 - O vértice atual passa a ser V;

4 - Marcar V como visitado;

5 - Se todos os vértices tiverem sido visitados adicionar o vértice de origem á lista e sair;

6 - Se não volta-se ao passo 2.

A cada iteração do algoritmo vai-se adicionando o custo da ligação entre vértices ao custo total.

Este algoritmo lê estes valores a partir de uma matriz de adjacência que é criada no início deste algoritmo. Como todos os grafos neste problema são completos o facto de existirem elementos da matriz vazios em grafos não completos não se aplica aqui.

O custo das ligações é representado por um inteiro, enquanto que os vértices visitados são adicionados a uma lista simples. No final esta lista é imprimida em conjunto com o custo total da viagem.

Criação de um grafo a partir de um ficheiro

O programa apresentado permite ao utilizador carregar um ficheiro e criar um grafo a partir deste. Como tal, o número de cidades é obtido a partir da expressão $\frac{n(n-1)}{2} = x$, onde x é dado pelo número de linhas do ficheiro. Com o número de cidades obtido o grafo é lido linha a linha de forma a obter todas as ligações.

O algoritmo calcula que existem $n(n-1)$ ligações no ficheiro. Como tal o ficheiro vai lendo $n-1$ vezes para cada nó.

Guardar um grafo num ficheiro

No final do algoritmo de criação do grafo é perguntado ao utilizador se pretende guardar o grafo atual num ficheiro. De acordo com enunciado o nome do ficheiro é criado da seguinte forma: tarefa_x_n.txt, onde x representa o número da tarefa e n o número de cidades. O programa permite ainda ao utilizador guardar mapas que sejam da mesma tarefa e com o mesmo número de cidades. Neste caso adiciona caracteres ao fim do nome do ficheiro. Exemplo: se já existir um ficheiro com o nome "tarefa_1_5.txt" e o utilizador queira guardar um ficheiro com as mesmas características o programa guarda o ficheiro na mesma localização, mas com o nome *"tarefa_1_5(1).txt"*. Se este já existir o nome fica *"tarefa_1_5(2).txt"* e assim sucessivamente. A verificação da existência dos ficheiros é feita recorrendo ao módulo os do *python*.

Otimizações feitas aos algoritmos

Estruturas de dados

Foram feitas algumas alterações à estrutura de dados que tinha inicialmente. No inicio do desenvolvimento do projeto era uma estrutura baseada em lista, onde as ligações e o seu custo eram guardadas em listas e depois todos estes elementos eram guardados noutra lista. Esta resolução era bastante complexa e confusa pelo que os resultados apresentados eram bastante insatisfatórios.

Algoritmos usados

Criação do grafo

Inicialmente, para a tarefa 1, era verificado para todos os nós se existia alguma ligação anterior o que não era necessário. Este algoritmo implicava uma pesquisa pela estrutura criada até ao momento o que era bastante dispendioso. Em conjunto com as alterações indicadas no ponto anterior foi possível obter uma melhoria significativa em termos de performance.

Criação do caminho mais curto

Desde o inicio da resolução do problema apresentado que o *nearest neighbour algorithm* foi a solução escolhida á variante do problema do caixeiro viajante. A diferença para a solução atual é que antes era guardado o valor dos nós visitados, em vez do seu símbolo. Esta solução apresentada tinha dois problemas: não resolvia o problema na sua totalidade uma vez que falhava e não resolvia o problema para a tarefa 2 onde as distâncias entre cidades seriam sempre diferentes. Como tal passou-se a guardar o símbolo que representa o nó e passou-se a ter uma solução generalizada para as duas tarefas.

Casos de teste utilizados

Como forma de verificar os algoritmos anteriores foram criados 3 mapas diferentes para cada tarefa, num total de 6 diferentes mapas. O primeiro possui 4 cidades, o segundo 7 e finalmente o terceiro tem 10 cidades. Estes mapas foram verificados manualmente para verificar se as distâncias entre cidades respeitavam as regras impostas mediante a respetiva tarefa.

Posteriormente foi também verificado o caminho mais curto de cada um destes mapas. Todos estes mapas seguem em conjunto com este relatório. **Todos estes três mapas passaram no teste.**

Tarefa_1_4.txt

	e	b	l	t
e	-	404	765	340
b	404	-	491	379
l	765	491	-	345
t	340	379	345	-

Tarefa_1_7.txt

	e	b	l	t	w	W	p
e	-	181	897	934	691	938	983
b	181	-	518	836	420	295	613
l	897	518	-	50	965	853	336
t	934	836	50	-	964	218	686
w	691	420	965	964	-	933	825
W	938	295	853	218	933	-	228
p	983	613	336	686	825	228	-

Tarefa_1_10.txt

	e	b	l	t	w	W	p	O	P	V
e	-	562	54	686	96	702	50	391	425	957
b	562	-	82	428	336	440	920	463	397	641
l	54	82	-	254	126	891	791	770	916	132
t	686	428	254	-	202	63	844	274	72	515
w	96	336	126	202	-	896	743	771	97	442
W	702	440	891	63	896	-	948	966	978	471
p	50	920	791	844	743	948	-	891	479	198
O	391	463	770	274	779	966	891	-	972	696
P	425	397	916	72	97	978	479	972	-	246
V	957	641	132	515	442	471	198	696	246	-

Tarefa_2_4.txt

	e	b	l	t
e	-	638	900	327
b	575	-	764	609
l	937	986	-	51
t	332	380	560	-

Tarefa_2_7.txt

	e	b	l	t	w	W	p
e	-	335	793	990	958	319	531
b	369	-	726	904	130	832	842
l	590	854	-	463	613	192	219
t	834	869	912	-	56	925	318
w	879	158	905	603	-	494	655
W	246	334	486	188	976	-	675
p	888	130	226	98	206	127	-

Tarefa_2_10.txt

	e	b	l	t	w	W	p	O	P	V
e	-	881	587	588	624	356	221	50	592	775
b	54	-	817	692	477	848	129	147	342	399
l	114	873	-	333	550	179	964	439	813	849
t	55	869	62	-	300	229	61	671	348	899
w	775	542	927	424	-	552	830	55	844	332
W	135	412	394	52	407	-	748	437	58	914
p	224	50	798	65	265	674	-	727	846	597
O	817	120	61	464	90	53	571	-	537	263
P	635	626	395	942	308	820	401	332	-	461
V	841	746	655	59	757	136	439	981	823	-

Resultados

Os mapas anteriormente descritos foram introduzidos como argumento no algoritmo para calcular o caminho mais curto. A seguir apresentam-se os resultados obtidos para estes mapas bem como os resultados para os mapas cujo tempo de execução do algoritmo não passou dos 30 minutos, conforme pedido no enunciado.

Mapas da Tarefa 1

	Tarefa_1_4.txt	Tarefa_1_7.txt	Tarefa_1_10.txt
Custo total	1580	2596	2591
Caminho obtido	e, t, l, b, e	e, b, W, t, l, p, w, e	e, p, V, l, b, w, P, t, W, V, O, e

Mapas da Tarefa 2

	Tarefa_2_4.txt	Tarefa_2_7.txt	Tarefa_2_10.txt
Custo total	2253	2554	2565
Caminho obtido	e, t, b, l, e	e, W, t, w, b, l, p, e	e, O, W, t, l, w, V, p, b, P, e

Mapa da Tarefa 1.4 e Tarefa 2.4

	Tarefa_1_25000.txt	Tarefa_2_20000.txt
Tempo Total	1799,95s	1799,84s

Conclusões finais

Olhando para as estruturas e algoritmos implementados é fácil perceber que a tarefa 2 apresenta melhores resultados, o que se traduz num maior número de cidades, uma vez que o algoritmo de criação do grafo não tem de ir á lista criada até ao momento procurar se existe uma ligação correspondente. Em teoria era isso que devia acontecer, mas isso não se verifica de acordo com os resultados apresentados. Fatores como o número de aplicações abertas no PC, a performance do PC e outros devem ser tidos em conta uma vez que afetam diretamente estes valores. Isto mesmo foi verificado: fechando todas as aplicações e desligando o WI-FI fez com que o número de cidades aumentasse. Relativamente ás estruturas usadas todas elas apresentam complexidades temporais/espaciais relativamente aceitáveis, exceto no algoritmo de criação do grafo para a tarefa 1 onde são usados dois ciclos for e para procurar por uma ligação existente recorreu-se á função *get* que possui uma complexidade temporal O(n). Pessoalmente penso que existe uma solução mais simples para este algoritmo no sentido da redução da complexidade temporal. Relativamente ao algoritmo escolhido para resolver o problema é debatível se o *nearest neighbour algorithm* é a melhor solução para o problema. O algoritmo cumpre o que é pedido para um número de cidades pequeno, mas para um número de cidades maior o algoritmo pode falhar. Poderia e deveria ter recorrido a outros algoritmos, como o algoritmo de Christofides, e verificar o seu tempo de resolução. O mesmo processo deveria ter sido feito para o algoritmo de ordenação da array temporária que permite ao algoritmo saber qual é o nó mais perto do nó atual.

Bibliografia

https://www.random.org/strings/
https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm
https://en.wikipedia.org/wiki/2016_Dakar_Rally#Stages
https://en.wikipedia.org/wiki/Travelling_salesman_problem
Slides da cadeira