INSTITUTO SUPERIOR TÉCNICO

Análise e Síntese de Algoritmos - 2015/2016
Relatório 2º Projeto

Grupo 97

Carolina Inês Xavier – 81172 | Inês Leite – 81328

Introdução:

O presente relatório aborda a solução, respetivos testes e análise ao problema proposto como segundo projeto de Análise e Síntese de Algoritmos do 2ºSemestre do ano escolar 2015/2016.

O projeto consiste em descobrir qual é a melhor localidade para organizar o encontro de empregados da empresa Coelho e Caracol Lda., determinando qual a localidade em que a perda total da empresa, considerando todas as filiais, é minimizada.

Os empregados podem fazer entregas ao longo do percurso sendo que uma rota consiste numa sequência de localidades. A cada par de localidades está associado um valor de perda, que resulta de subtrair a receita ao custo.

Através de um *input* com o número de localidades, o número de filiais e o número de ligações seguido da identificação destas mesmas filiais, bem como uma lista de linhas que indica uma deslocação entre duas localidades e respectiva perda, o programa resultante tem de apresentar como output o número do ponto de encontro, a respetiva perda total e os valores de perda mínima de cada uma das filiais até ao ponto de encontro.

Abordagem inicial:

Primeiramente, tivemos de optar pela escolha da linguagem de programação na qual iríamos escrever o código. Optámos por C++ porque permite uma programação com base em classes, o que ao encontro das escolhas que iremos fazer na implementação.

Para conceber o programa criámos um mapeamento entre os conceitos:

- Rede de distribuição para grafo dirigido e pesado, pois no transporte de mercadoria, quando há um caminho entre duas localidades tem um sentido e um peso associados;
- Localidade para vértice do grafo;
- Arco entre dois vértices sempre que possa ser feito um transporte entre duas localidades;
- Peso de um arco para respetivo valor de perda.

Após uma pesquisa para a identificação de qual o melhor algoritmo a utilizar para a resolução deste problema, concluímos que o algoritmo a utilizar seria o algoritmo de Johnson,

tendo em consideração que este problema pode ser resolvido com um algoritmo de caminhos mais curtos entre todos os pares, assumindo que não existem ciclos de peso negativo.

Para tal, recorremos ao algoritmo Bellman-Ford (para encontrar, para cada um dos vértices do grafo, o menor peso*)*, seguido da função de repesagem e por fim o algoritmo de Dijkstra (para obter as distâncias mínimas das filias às localidades).

O tempo total do algoritmo neste caso é O(V(V +E) lg V).

Descrição da Solução:

Do *input* retiramos o número de vértice, o número de arestas e o número de filiais que o grafo vai ter.

Com o número de vértices, podemos inicializar todas as nossas estruturas de dados sendo uma delas um vetor de vetores que representa o grafo, em que cada uma das posições do vetor corresponde a um vértice do grafo e o vetor correspondente contém o vértice de destino bem como o peso desta ligação, que na nossa implementação são atributos da classe Edge – vetor adjacente.

Com o número de filiais sabemos quantos valores (que correspondem à identificação das filiais do grafo) temos de ler do *input,* e que vamos guardar num vetor, pela ordem de leitura, para facilitar, depois, a geração do *output*.

Com o número de arestas sabemos quantas linhas temos que ler do *input* que contém as ligações, que serão adicionadas ao grafo dirigido e pesado.

Depois de termos o grafo representado, de forma a identificar o ponto de encontro entre as várias filiais aplicamos, numa primeira fase, o algoritmo Bellman- Ford descrito nos seguintes passos:

- Inicialmente todas as alturas dos vértices do grafo são inicializadas a 0 e guardadas num vetor (substituindo assim a criação de um novo vértice origem de altura 0);

-Para todas as arestas (u,v) do grafo G fazemos:

-Se a soma altura [u] + peso (u,v) for menor que o valor de altura [v], então redefinimos uma nova altura [v] como altura [v] = altura [u] + peso (u,v);

Após obtermos as alturas mínimas de cada vértice, recorremos a função de repesagem de forma a calcular pesos positivos para cada aresta, permitindo assim aplicar o algoritmo Dijkstra:

-peso (u,v) = peso (u,v) + altura [u] – altura [v];

Podemos então aplicar o algoritmo Dijkstra:

O algoritmo de Dijkstra encontra o menor caminho entre quaisquer dois vértices do grafo, quando todos os arcos têm comprimento não-negativos. O algoritmo de Dijkstra utiliza um procedimento iterativo, determinando, na primeia iteração, o vértice mais próximo do vértice de

origem, na segunda iteração, o segundo vértice mais próxim, e assim sucessivamente, até que em alguma iteração o vértice pretendido seja atingido, utilizando o menor caminho.

Na implementação do algoritmo, a função recebe um vértice filial e calcula a menor distância entre esse vértice e todos os outros vértices localidade.

Cada filial é colocada num set (listaQ) de pares <peso do vértice, vértice> (com peso 0), o que permite ordenar automaticamente os vértices pelo menor peso. E existe também um vetor (caminho_minimo) que guarda as distâncias da filial a cada uma das localidades, estas distâncias são iniciadas a infinito.

Enquanto a fila não estiver vazia inicia-se um ciclo, sendo que em cada iteração:

- Retira-se o primeiro elemento da lista e actualiza-se o vetor com as distâncias mínimas finais para esse valor.

- Cada um dos vértices adjacentes ao vértice retirado é inserido de novo no set, ou atualizado, com a soma do peso do vértice retirado e o peso de aresta entre eles, se o seu peso atual no vetor caminho_minimo for maior que esta soma. Este valor é também introduzido na posição do vetor caminho_minimo correspondente ao vértice adjacente.

Quando a fila está vazia o algoritmo termina e, invertendo o processo de repesagem, temos assim os caminhos mínimos de uma filial para cada uma das localidades (_distanciaLocalidades).

Aplicamos então o algoritmo Dijkstra para cada uma das filiais e vamos somando as distâncias calculadas ao vetor _distanciaLocalidades.

No fim da aplicação do algoritmo, a localidade com distância menor (perda total menor) é o ponto de encontro, e o valor existente na posição do vetor _distanciaLocalidades correspondente a essa localidade é a respectiva perda total.

Para calcularmos as perdas mínimas de cada filial voltamos a aplicar o algoritmo Dijkstra sobre cada uma das filiais, sendo que o valor retornado pela função dijkstra corresponde à sua perda mínima.

Análise Teórica:

Admitindo que V é o número de vértices e E é o número de arestas do grafo dirigido e pesado:

O ciclo de inicialização das estruturas de dados tem uma complexidade O (V).

O ciclo que cria as ligações entre vértices do grafo tem complexidade O (E).

O algoritmo Bellman-Ford é executado em O (VE) pois temos dois ciclos encadeados, sendo que o externo é executado V vezes e o interno E vezes.

O algoritmo Dijkstra é executado em O (V (V +E) lg V). A inicialização deste algoritmo ocorre em O (V); o ciclo while é executado V vezes; todas as arestas do grafo são visitadas; Fila de prioridade: O (lg V).

Na geração do output, o ciclo em que calculamos o ponto de encontro tem complexidade O(V) e a função que devolve os valores de perda mínima de cada uma das filiais até ao ponto de encontro tem complexidade O(V).

Concluímos assim que a nossa solução tem uma complexidade O(V(V +E) lg V) o que faz dela uma solução eficiente.

Avaliação Experimental dos Resultados:

Submetemos o nosso programa aos testes dados na página da cadeira, aos quais passou com sucesso. Na tabela é possível observar o número de vértices do grafo (correspondente ao número de pessoas), o número de arestas do grafo (correspondente ao número de ligações), o número de filiais e o tempo de execução (obtido com o comando Unix time).

Testes:	Número de vértices	Número de arestas	Número de filiais	Tempo:
in1	5	8	2	< 0,002
in2	6	11	2	~ 0,002
in3	6	11	3	~ 0,002
in4	5	6	3	~ 0,002
T01	5	6	3	< 0,002
T02	2	2	2	< 0,001
T03	100	800	10	< 0,008
T04	100	1000	15	< 0,015
T05	250	5000	100	< 0,040
T06	300	5000	10	< 0,050

Como se pode ver pelos resultados obtidos, o tempo de execução aumenta à medida que o número de vértices e arestas também aumenta, o que era esperado tendo em conta a complexidade do algoritmo O(V(V +E) lg V).

Referências:

https://en.wikipedia.org/wiki/Johnson%27s_algorithm

http://www.geeksforgeeks.org/johnsons-algorithm/

https://alg12.wikischolars.columbia.edu/file/view/ALLPAIRS.pdf