ANÁLISE E SÍNTESE DE ALGORITMOS
RELATÓRIO 2º PROJECTO

TRABALHO REALIZADO POR:

Diogo Freitas, 81586
Francisco Mira, 80888
Grupo 105

Introdução

O 2º projecto da cadeira Análise e Síntese de Algoritmos consiste na identificação da melhor localidade para todas as filiais de uma empresa se encontrarem de forma a que a perda total da empresa seja a mínima possível, i.e., determinar um ponto de encontro para o qual a perda de todas as filiais seja o menor possível em termos de perda para a empresa.

O input contém a informação necessária sobre o número de localidades a ser consideradas, o número de filiais existentes nessas localidades e o número de ligações entre as ditas localidades. O input é recebido sob a forma de L + 2 linhas, sendo que L corresponde ao número de ligações entre localidades. Na linha inicial é recebido o número de localidades, o número de filiais e o número de ligações totais da rede, separados por um espaço. Na segunda linha são recebidos os identificadores das localidades onde se encontram filiais sediadas, também separados por um espaço. Nas linhas seguintes é recebida, por linha, uma ligação entre duas localidades com o número identificador de cada uma separados por um espaço e a determinada perda que essa ligação envolve.

O output é constituido por duas linhas, sendo que na primeira linha se encontra o número identificador da melhor localidade para todas as filiais de um empresa se encontrarem de forma a que a perda total da empresa seja a mínima possível e a respectiva perda total, separados por um espaço. Na segunda linha é retornada as perdas correspondentes ao caminho entre as filiais e a localidade determinada, separadas por um espaço.

Descrição da Solução

Linguagem de Programação

Como linguagem de programação para a implementação da solução, foi escolhida a linguagem C++ devido a um maior à vontade com a linguagem por parte dos elementos do grupo e à existência de bibliotecas predefinidas que iriam auxiliar na implementação da estrutura do código.

Estruturas de Dados

Para a implementação do algoritmo foram utilizadas quatro estruturas de dados: uma *struct* representativa de uma ligação entre localidades e respectiva perda, um vector representativo da rede de ligações entre localidades, no qual é inserido um vector por localidade representativo das ligações que esta possui para outras localidades na rede e as suas respectivas perdas (*structs* referidas acima) e vectores

e uma pilha auxiliares à execução do algoritmo, nos quais são inseridos números inteiros.

struct {int ; int ;} estrutura de dados com dois inteiros.

std::vector um vector dinâmico de memória contígua.

std::set uma pilha.

std::vector< std::vector<int> > um vector de vectores de inteiros.

Solução

Como solução para o problema apresentado no enunciado do projecto, organizámos os utilizadores num grafo dirigido e pesado, pois o problema diz respeito caminhos cuja perda/custo tem que ser mínima, logo o grafo necessita de ser pesado. Este grafo representa a rede de localidades contempladas para serem um ponto de encontro para as filiais da empresa, em que cada ligação entre localidades é representada por uma aresta entre vértices do grafo. Foi identificado que o ponto de encontro para as filiais corresponde à localidade para a qual a soma dos custos do menor caminho entre elas é menor.

Algoritmo

O programa começa por ler o input, definindo o número de vértices, filiais e arcos existentes no grafo. De seguida, cria o grafo e todos os vectores auxiliares necessários para a execução do Algoritmo de Johnson. No passo seguinte, o grafo é tratado pelo Algoritmo de Johnson que determina quais os caminhos mais curtos entre as filiais e todas as localidades, escolhe a localidade com os requisitos explicados acima para ser o ponto de encontro e calcula a perda total da empresa e de cada filial no caminho para este e ,por fim, o programa fornece o output desejado.

A nossa abordagem tem como base o seguinte procedimento:

I. Inferir a informação sobre o número total de localidades, número de filiais, número de ligações entre elas e as próprias ligações e criar o grafo correspondente à informação dada.
II. Passagem pela função *johnsonAlgorithm()* onde está implementado o Algoritmo de Johnson que engloba:
    i. Passagem pela função *reweightGraph()* que utiliza o Algoritmo Bellman-Ford, que está implementado na função *bellmanFordAlgorithm()* para encontrar os pesos actualizados de cada vértice, e recalcula os custos de cada aresta entre vértices.
    ii. Passagem pela função *dijkstraAlgorithm()* pela primeira vez, que procura os caminhos mais curtos entre os vértices correspondentes às filiais e todas as localidades, calculando o ponto de encontro ideal e o custo total de todas as filiais para este.
    iii. Passagem pela função *dijkstraAlgorithm()* pela segunda vez, que procura de novo os caminhos mais curtos entre os vértices correspondentes às filiais e todas as localidades, calculando desta vez o custo específico para cada filial até ao ponto de encontro.
    iv. Impressão dos dados produzidos pelos passos anteriores que correspondem ao output pretendido.

Na nossa solução, a parte que merece mais relevância é o funcionamento do Algoritmo de Johnson na função *johnsonAlgorithm()*. A nossa implementação do algoritmo tem por base a utilização de quatro vectores, uma pilha e duas variáveis auxiliares, respectivamente, de nome *compBranches*, que é utilizado para guardar os identificadores correspondentes às localidades nas quais filiais estão sediadas; *vWeight*, que é utilizado para guardar a informação sobre o peso de cada vértice após a execução da função *bellmanFordAlgorithm()*; *d*, que é utilizado para guardar a informação sobre o custo dos caminhos mais curtos entre uma determinada filial e todas as localidades e, por fim, *dijkstraResult*, que é utilizado para guardar a informação sobre o a soma do custo dos caminhos mais curto entre todas as filiais e todas as localidades, relativamente à pilha, *stack*, esta é utilizada para guardar a ordem de pesquisa dos vértices na execução da função *dijkstraAlgorithm()*, relativamente às variáveis, são elas: *PE*, identificador correspondente à localidade determinada como ponto de encontro e *totalCost*, soma do custo dos caminhos entre todas as filiais e o ponto de encontro.

O nosso algoritmo corre uma vez, o que engloba uma vez a função *reweightGarph()* e 2N vezes a função *dijkstraAlgorithm(), sendo N o número de filiais*. No interior da função é primeiramente adicionado um vértice com ligações para todos os outros com custo zero para auxiliar a função *reweightGraph()*. Após a adição do referido vértice é corrida a função de repesagem dos arcos *reweightGraph()* que recorre à função *bellmanFordAlgorithm()* para determinar os pesos dos vértices do grafo percorrendo-o com dois ciclos for, ao ter executado esta função percorre de novo o grafo com dois ciclos for para fazer o cálculo de repesagem dos arcos. De seguida, já de volta à função *johnsonAlgorithm()*, é corrida a função *dijkstraAlgorithm()* para todas as filiais através de um ciclo for que as percorre. Na funçáo *dijkstraAlgorithm()* com o auxílio da pilha *stack* que contem a ordem de pesquisa dos vértices encontramos os caminhos mais curtos dos vértices das filiais até todos os vértices percorrendo todos os arcos existentes no grafo e anotando o custo que estes têm no caminho. Para cada iteração do ciclo for que corre a função *dijkstraAlgorithm()*, os custos até cada vértice são "desrepesados" e somados ao vector *dijkstraResult*, no qual, no fim do ciclo, estarão os custos mínimos totais da empresa para cada vértice. Assim podemos determinar a primeira linha do output, o vértice correspondente ao ponto de encontro e o custo do caminho de todas as filiais para lá que são impressos após verificação da existência de um ponto de encontro. Caso exista, como referido acima, o identificador desse vértice é impresso bem como o custo total. De seguida, para calcular os custots específicos de cada filial corremos a função *dijkstraAlgorithm()* de novo para todas as filiais e vamos imprimindo o custo dessa fililal até ao ponto de encontro como pretendido.

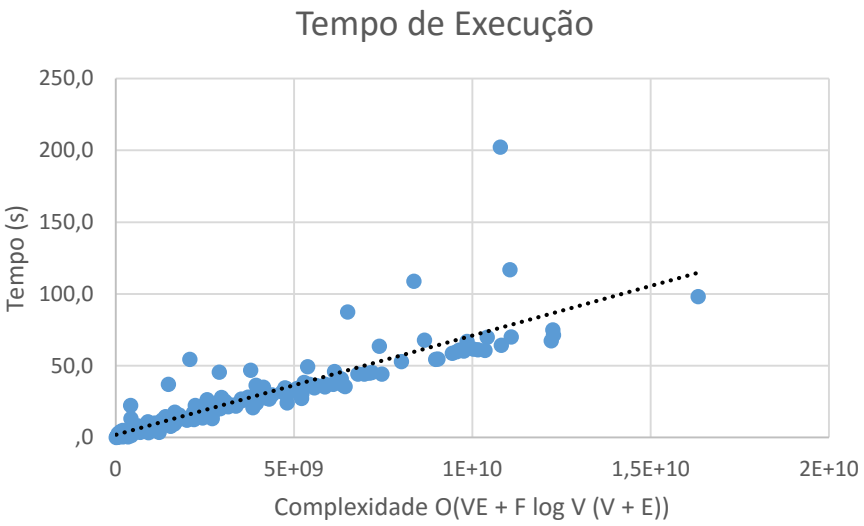
Análise Teórica

O nosso programa é executado em tempo linear ao tamanho do problema. Sendo assim, a complexidade do nosso programa é:

- Função *bellmanFordAlgorithm()* - O(VE), sendo V o número de vértices e E o número de ligações
- Função *dijkstraAlgorithm()* - O((V + E) log V)
- Funções de inicialização dos vectores auxiliares - O(V)

- Função *johnsonAlgorithm()* - O(VE + F log V (V + E)) sendo F o número de filiais

Análise Experimental



O nosso projecto passa a todos os 16 testes fornecidos.
Para verificar a linearidade do tempo de execução do nosso programa, criámos 200 grafos com um número de vértices aleatório entre 2 e 5000 e testámos o tempo de execução deste para cada um deles, obtendo os resultados representados no gráfico acima.

Pesquisa e suporte para a elaboração do nosso algoritmo:

https://gist.github.com/ashleyholman/6793360