**Instituto Superior Técnico**

**Análise e Síntese de Algoritmos**

Relatório – Projeto 2

**Eduardo Rodrigues** nº 178431; **Rui Matos** nº179100

1. **Introdução**

**1.1. O Problema**

O segundo projeto, previsto no programa da disciplina de Analise e Síntese de Algoritmos, **baseia-se no cálculo da perda mínima até uma determinada localidade**, que corresponde ao caminho de menor custo desde a sede da empresa Coelho e Caracol Lda até essa localidade, através de outras localidades, tendo em conta o valor de perda entre cada par de localidades (que corresponde ao custo menos a receita), logo, **quanto menor o valor maior será o ganho para a empresa**.

**1.2. Solução do Problema**

Em termos estruturais, esta sequência é simulada com **grafos**, dirigidos. **Cada localidade** corresponde a **um vértice**, e **cada trajeto entre duas localidades** corresponde a **uma aresta** do grafo. **A cada aresta fazemos corresponder o custo menos a receita**.

1. **Implementação**

**2.1. A Linguagem**

A linguagem escolhida para a realização deste projeto é C, por ser a linguagem com que estamos mais familiarizados pelo que, é assim mais fácil produzir um algoritmo eficiente.

**2.2. Estruturas**

Relativamente à implementação, a estrutura que optámos por usar para criar o **grafo**, são as **listas**. Cada grafo (estrutura *graph*) contém um vetor de **listas de adjacências**(*adj*) e um **vector de todos os arcos** (*edges*).

**2.2.1 – Listas de Adjacências*(adj)***

Cada elemento do vetor tem diversos conjuntos de nodes, e cada node possui um inteiro, que identifica o vértice correspondente a uma localidade, e um *link* para outro vértice com o qual também haja ligação.

**2.2.2 – Vector com todos os arcos(edges)**

Cada elemento do vector tem a informação correspondente a um arco.

Por exemplo: a uma aresta, corresponde, neste caso, o vértice do qual se parte***(from)*** para o vértice a que se chega**(*to)***, bem como o custo de percorrer o determinado caminho**(*cost)***.

**2.3. Exemplo**

Imaginemos um grafo G com 3 vértices todos com uma aresta.

Neste caso o vector *adj* do grafo G, teria a seguinte configuração:

G->adj[0] -> 1

15

-10

G->adj[1] -> 2

G->adj[2] -> 0

-10

G->edge[0] (from = 0 ; to = 1; cost = -10)

**Figura 1**

G->edge[1] (from = 1 ; to = 2; cost = -10)

G->edge[2] (from = 2 ; to = 0; cost = 15)

**2.4. Cálculo da perda mínima e Optimizações**

O cálculo da perda mínima, é feito pela função *BellmanFordandNegativeCycle* que numa primeira parte, funciona de forma muito semelhante ao algoritmo de Bellman-Ford, uma vez que determina, para cada vértice, **o valor do caminho mais curto**, começando no vértice correspondente à sede da empresa “Coelho e Caracol Lda”, que é indicado no input, tendo em conta o peso de cada aresta.

**2.4.1 – Algoritmo Original VS Optimizações**

O **algoritmo original** relaxa V-1 (em que V corresponde ao número de vértices) vezes todos os arcos. De seguida percorre uma segunda vez todos os arcos, e caso algum dos vértices sofra alterações é porque existe um ciclo negativo e esse vértice é identificado e o algoritmo termina.

Como **primeira optimização,** consideramos que caso não haja alterações a nenhum dos vértices, o que é verificado em cada iteração do ciclo *for* com um **“dirty bit”** então, podemos concluir que não irão existir mais alterações, podendo terminar a primeira parte do algoritmo.

Como **segunda optimização,** consideramos que no segundo ciclo for do algoritmo, onde se verifica a existência de vértices em ciclos negativos, em vez de terminar-mos o algoritmo adicionamos a identificação do respectivo vértice a uma queue e alteramos a sua distância para **INT\_MIN**.

Na segunda parte da função criada, existe um ciclo *while*, que funciona de forma semelhante a uma **BFS** que, iniciando num vértice anteriormente identificado que se encontra na queue, encontra todos os vértices que tenham ligação a esse, ou seja, que também estão no ciclo negativo.

Como **terceira optimização,** começamos por verificar **as adjacências dos vértices** que estavam identificados na queue, alterando a dist[u] para INT\_MIN e adicionando esse vértice à queue para que a sua adjacência seja verificada. Mas, se ao percorrer a adjacência de um vértice, encontrar-mos vértices para os quais a sua **dist[u] == INT\_MIN** já não adicionamos esse vértice à queue, **uma vez que ou já foi processado, ou já está na queue para ser processado**.

\*- seja u um vértice genérico.

A perda mínima até um determinado vértice é guardada no vetor *dist*. No final, o vetor *dist* corresponde ao output esperado do algoritmo, da seguinte forma: **INT\_MIN=** Vértice em ciclo negativo; **INT\_MAX=** Vértice com o qual não existe ligação; **valor\*=** Custo mínimo de chegar a um determinado vértice.

Valor\*= valor real, positivo ou negativo.

**2.4.2 – Estruturas e Escolhas**

A escolha da ***queue* em array**, apesar de alocar mais memória, é simplesmente para tornar o algoritmo mais simples e evitar o número de comparações que haveria, caso tivéssemos usado uma lista ligada.

1. **Análise do Algoritmo**

Para analisar este algoritmo, utilizam-se os seguintes parâmetros: **V**, número de localidades (vértices), **e E**, número de trajetos entre duas localidades (arestas).

**3.1. Memória**

Quanto à memória virtual utilizada pelo nosso algoritmo, as estruturas que devemos considerar são os array *dist* (V inteiros), o grafo (estrutura, de adjacências e arcos )(V+E), e a queue (V+V).

No final da utilização de cada estrutura, a memória alocada é libertada, sendo assim, existem leaks de memória.

**3.2. Situações limite**

Quanto a situações limite, o melhor caso seria **zero vértices e zero arestas**, em que o tempo de execução seria igualmente zero, e o pior caso, seria um **número muito grande de vértices e de arestas**, pois quanto maior estes forem, maior é o tempo de execução do algoritmo.

**3.3. Complexidade**

Para estudar a complexidade deste algoritmo, analisa-se a complexidade da função *main*. Na leitura do input, faz-se a leitura dos valores iniciais, a inicialização do grafo, que corresponde **a O(V+E)**. No cálculo da perda mínima, que recorre ao algoritmo de Bellman-Ford, a complexidade de tempo pode ser representada pela expressão **O(VE)**. Quanto ao ultimo ciclo da função, no pior caso, ou seja, se todos os vértices estiverem num ciclo negativo, a complexidade é **O(V+E)**. No final, a libertação de memória expressa-se por **O(V+E)**, que corresponde à libertação de todas as ligações no grafo.

**3.4. Desempenho**

Relativamente ao tempo de execução, **o algoritmo comporta-se como O(VE).** Para provar isso, realizaram-se experimentalmente vários testes, de modo a calcular o tempo de execução com tamanhos de input diferentes (superiores a 104), através do comando *time*, e obtiveram-se os seguintes gráficos:

Como podemos observar, a linha de tendência (a tracejado) tem um declive muito pequeno. Uma vez que a componente quadrática da função está muito perto do zero. Desta forma prova-se que o limite assimptótico superior da função é **O(VE).**

**3.5. Notas**

Verificamos, claramente, que **a presença de ciclos negativos também aumenta, bastante significativamente,** o tempo de execução do programa, como já era de esperar. Uma vez que a componente quadrática é 10^2 vezes maior com ciclos, continuando muito baixa, 9\*10^(-11). A componente linear também aumenta 10 vezes.

1. **Conclusão**

O objetivo do trabalho era o cálculo da perda mínima nas rotas de distribuição da empresa Coelho e Caracol Lda. Para tal, usamos **um grafo dirigido** e percorremos todo o grafo, ao estilo do algoritmo de Bellman-Ford e uma BFS. O programa foi desenvolvido na linguagem C, onde usamos duas estruturas principais para o efeito, Graph (lista de adjacências e arcos) e QUEUE. Toda a memória alocada foi libertada. Por fim, concluímos que a **complexidade do algoritmo é O(VE)**, algo que consideramos eficiente e indicado para a situação. É assim um algoritmo que corre até mesmo numa máquina com baixas “specs” e produz um bom resultado.