# Segundo projeto de Análise e Síntese de Algoritmos 2015-2016
# Relatório

Ana Sofia Costa 81105 João Pedro Silvestre 80996

23 de Abril de 2016

## 1 Introdução

O presente relatório apresenta a análise ao problema, solução e respetivos testes referentes ao segundo projeto desenvolvido no ambito da UC de Análise e Síntese de Algoritmos do segundo semestre do ano escolar 2015/2016. A primeira secção descreve o problema proposto, que consiste em encontrar um ponto de encontro, entre varias localidades, para reunir empregados de várias filiais. A segunda secção reporta o modo de solucionar o problema proposto através de optimização algoritmica. A terceira secção apresenta a análise teórica que sustenta a solução descrita.

### 1.1 Problema

O problema propsto consiste em encontrar um ponto de encontro, entre varias localidades, para reunir empregados de várias filiais. Assim, o objetivo consiste na analise do caminho mais curto entre as diversas localidades que contém filiais, ou seja, encontrar um caminho que em que a perda total da empresa (custo menos a receita), considerando todas as filiais, seja minimo.

#### 1.2 Input

O input fornecido é constituido por um linha que contém o número de localidades (N), o número de filiais (F) e o número de ligações (E), todos separados por um espaço em branco. Uma segunda linha com F números que indica em que localidades as filiais se encontram. Por fim, uma lista com E linhas, em que cada linha indica três inteiros u, v e w. Cada linha descreve uma aresta de origem u com destino em v e peso w.

#### 1.3 Output

O output indica, na primeira linha, o número que identifica o ponto de encontro e a perda total, na segunda, F números inteiros (impressos pela mesma ordem do input) que indica a perda de cada uma das filiais. Caso não seja possivel encontrar um ponto de encontro, o output imprime a letra N.

## 2 Descrição da Solução

#### 2.1 Linguagem de Programação

Para implementar uma solução foi escolhida a linguagem C++. Das três possibilidades oferecidas, esta revelou-se mais adequada pois permite a gestão de memória e tempo. Permite também utilizar estruturas já implementadas em bibliotecas.

**2.2 Solução**
A partir do enunciado foi possível aferir que a solução se debatia sobre um grafo dirigido e tem peso, pois o deslocamento entre duas regiões tem um custo e um sentido, assim as localidades são representadas por vertices (V) e o caminho entre elas por arestas (E). Na procura do ponto de encontro identificou-se a necessidade de tratar as arestas negativas, portanto utilizamos o algoritmo de Johnson, aplicando o algoritmo de Bellman-Ford para transformar todos os vertices negativos em não negativos e permitir utilizar o algoritmo de Dijkstra no grafo transformado e encontrar o caminho mais curto.

**Dijkstra:** "assemelha-se ao BFS, mas é um algoritmo *greedy* que escolhe o caminho que lhe parece otímo no momento, desta forma construimos os melhores caminhos dos vértices alcançáveis pelo vértice inicial determinando todos os melhores caminhos intermediários."

**Bellman-Ford:** "algoritmo de busca de um caminho minimo. Embora demora a sua complexidade seja maior do que a de Dijkstra e permita resolver todos os problemas que Dijkstra resolve, este permite que haja arestas com peso negativo."

**2.3 Estrutura de Dados** Desta forma, considerou-se mais conveniente implementar uma estrutura semelhante à lista de adjacências, pois esta é vantajosa na análise de arestas e na sua adição. A estrutura implementada foi a seguinte:

- **typedef pair<int, int> edge** o primeiro inteiro do par corresponde ao vertice destino e o segundo corresponde ao peso da aresta.
- **typedef pair<int, int> set_item** utilizado apenas no algoritmo de Dijkstra, o primeiro inteiro representa o peso de um determinado vertice e o segundo o indice deste vertice no vetor _graph.
- **class Graph** que possui quatro inteiros (número total de vértices _vertexNum, o número de filiais _numFiliais, o número que identifica o ponto de encontro _min, a perda total _minCost) e quatro vetores: três vetores de inteiros: **vector< int >** _filiais (utilizado para armazenar os indices dos vertices que contem filiais), **vector< int >** _height (a altura de cada vertice), **vector< int >** _totalCost e **vector< vector<Edge> > _graph** (vetor de vetores de Edge, ou seja, par de dois inteiros ).

A razão de escolha das estruturas anteriormente mencionadas prende-se com o facto de ser possível navegar de forma praticamente imediata entre vértices e ocupar menos memória.(???)

**2.4 Algoritmo**
Com vista a resolver o problema proposto pela UC foi selecionado um algoritmo estudado nas aulas teóricas: Algoritmo de Johnson. Após uma revisão bibliográfica, foi possível concluir que este algoritmo era o melhor que se adaptava à resolução do problema. O algoritmo de Johnson tem como base dois algoritmos que também foram estudados nas aulas teóricas, o algoritmo de Bellman-Ford e o algoritmo de Dijkstra. Deste modo o algoritmo começa por aplicar Bellman-Ford, para que seja possível fazer a repesagem de arestas, de forma a que o grafo contenha apenas arestas de custo não negativo, requisito fulcral para aplicar o algoritmo de Dijkstra.
Com visa a resolver o problema proposto de forma eficiente é aplicado o algoritmo de Johnson, em que apenas é aplicado o algoritmo de Dijkstra com vértices fonte que contenham filiais. Sempre que Dijkstra é executado, é feita a repesagem dos arcos e somado o vetor que retorna desta função ao vetor _totalCost. No final deste loop for, é possível obter o vértice de ponto de encontro e a perda total. De seguida, aplica-se Dijkstra aos vértices com filiais para obter a perda mínima de cada um deles.

# 3 Análise Teórica

(Com o intuito de simplificar a compreensão, o vértice será designado por V e a ligação entre dois vértices E. )

Inicialmente, na criação do grafo o algoritmo recebe o número de vertices que sao necessários para efetuar a criação do grafo, inserir a aresta e o peso que liga dois vertices no vetor de arestas adjacentes. Estas operações são constantes O(1).

Após a criação da estrutura e armazenamento das arestas adjacentes é chamada a **função Johnson** que chama a função **Bellman-Ford**. Bellman-Ford vai percorrer todos os vertices, iniciando o peso de todos os vertices a 0 (semalhante a estar a adicionar um vertice fonte ligado a todos os vertices do grafo com arestas de custo 0)

$$v \in V \tag{1}$$

caso o peso do vertice seja superior ao peso da sua aresta que lhe advém mais o peso do vertice adjacente. Em seguida será efetuada a repesagem das arestas do grafo. O algoritmo descrito acima é executado uma vez por cada vertice, a sua complexidade é:

$$\theta\left(V(V+E)\right) = \theta\left(V^2 + VE + (V+E))\right) = \theta\left(EV^2\right) \tag{2}$$

Em seguida é efetuado o algoritmo de Dijkstra para cada uma das filiais. Cada iteração do algoritmo de Dijkstra é

$$\theta\left((V+E)log(V)\right) \tag{3}$$

Como o Dijkstra é feito F vezes, fica:

$$\theta\left(F(V+E)log(V)\right) \tag{4}$$

A complexidade da criação do grafo, da execução do algoritmo e retorno do output corresponde à seguinte soma:

$$\theta\left(EV^2\right) + \theta\left(F(V+E)log(V)\right) \tag{5}$$

# 4 Avaliação Experimental

Neste secção é feita um analise dos resultados obtido de forma a comprovar a complexidade do algoritmo implementado. Nesta análise é executada uma série de testes, gerados por um programa que cria inputs e é medido a duração de execução do código realizado.

Os inputs gerados para os testes seguintes são aleatórios, havendo variação no número de filiais. Cada grafo tem 1000 vertices, de densidade média e foi apenas aumentando o número de filiais.(ver Figura1)

Nos testes seguintes, foram testados 2 casos contrarios, o primeiro foi gerado inputs, com o minimo de ligações entre localidades . No segundo caso é gerado inputs com o máximo de ligações entre localidades.

$$\frac{V(V-1)}{2}(arestas, aproximadamente) \tag{6}$$

Para estas ultimas experiências utilizamos grafos esparsos, ou seja, os vertices tinham poucos vertices adjacente excepto o inicial e final. Em relação aos grafos densos são grafos com maior quantidade de arestas possível e onde há uma grande quantidade de ciclos.

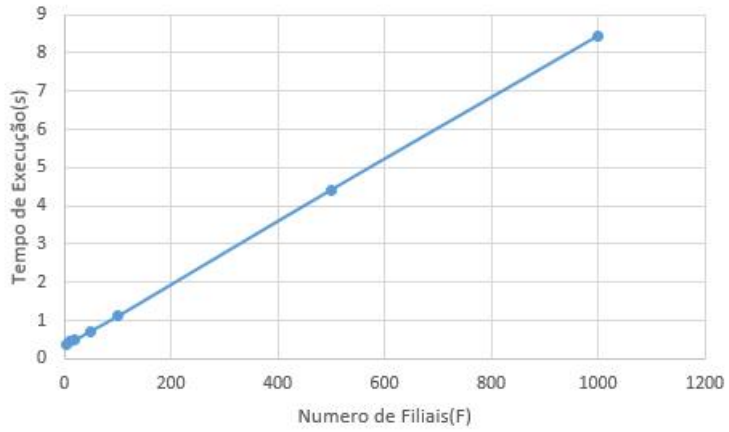


Figura 1: Gráfico da avaliação experimental

| Tipo de Grafo/Nº de Vértices | 10 | 50 | 100 | 500 | 1000 | 5000 |
|---|---|---|---|---|---|---|
| Esparso | 0.002(s) | 0.003(s) | 0.008(s) | 0.049(s) | 0.242(s) | 5.609(s) |
| Denso | 0.002(s) | 0.003(s) | 0.011(s) | 0.252(s) | 2.449(s) | 33.22(s) |

Figura 2: Tabela da avaliação experimental

# 5 Conclusão

Dos resultados apresentados, verificou-se que o algoritmo é parabolico. Ou seja, quando o número arestas e vertices aumenta, o tempo de execução é proporcionalmente crescente. Em relação ao algoritmo de Dijkstra, quando o número de filiais aumenta, aumenta também o tempo de execução.

# Referências

[1] Cormen, T.; Leiserson, C.; Rivest, R.; Stein, C. (2009). Introduction to algorithms. 3rd edition.