<u>2º Projeto de ASA - Relatório</u>

<u>Introdução</u>

Os objetos do problema podem ser representados numa linguagem de programação como nós e arestas, sendo os fornecedores, as estações e o hipermercado representados como nós e os transportes como arcos. Podemos modelar o problema usando o conceito de rede de fluxos e, assim, utilizar algoritmos de cálculo de fluxos máximos para determinar a **capacidade máxima**, bem como **as estações de abastecimento e as ligações que devem ser aumentadas**.

Escolhemos implementar a nossa solução na linguagem C++.

<u>Descrição da solução</u>

Excetuando as operações de *input*/*output*, a nossa solução divide-se em 3 partes:

- Criação de uma rede de fluxos para representar a rede de fornecimento;
- Percorrer a rede de fluxo, utilizando a nossa implementação do algoritmo **FIFO Push-Relabel**, para calcular a sua capacidade máxima;
- Realização de uma pesquisa DFS sobre a rede residual, começando no hipermercado, para descobrir os elementos da rede a aumentar.

O programa começa por ler os *inputs* pedidos e guardar os valores lidos em variáveis e estruturas auxiliares, que serão usadas na construção da rede. Em particular, criámos uma estrutura **InputEdge**, que serve apenas para ler os *inputs* das arestas (origem, destino e capacidade).

É depois chamada a função **makeGraph**, que constrói a rede de fluxos. A rede é constituída por:

- nós (**struct Node**) que contêm um identificador (**num**), uma lista de arestas adjacentes (**adj**), o excesso de fluxo (**e**), a altura (**h**), e um booleano (**visited**) que é usado na DFS;
- arestas (**struct Edge**), que contêm referências para os nós origem e destino (**src**, **dst**), e os fluxos e capacidades em ambos os sentidos (**f**, **c**, **f_rev** e **c_rev**).

Sempre que uma aresta é adicionada (função **addEdge**), é colocada uma referência para a mesma na lista de adjacências de ambos os nós envolvidos. A construção é feita da seguinte forma:

- É criado um nó **0**, e são adicionadas arestas (**0**, **F**), com capacidades iguais às de cada fornecedor **F**;
- Cada estação de abastecimento **E** é convertida em dois nós, o nó **E**, e o nó **E** + **e** (onde **e** é o nº de estações de abastecimento);
- Para cada estação de abastecimento **E**, é adicionada uma aresta (**E, E** + **e**) com capacidade igual à da estação **E** (dada no input);
- As arestas do input (**struct InputEdge**) são ordenadas de forma a que arestas simétricas (ex.: (3, 5) e (5, 3)) fiquem em posições consecutivas;
- As arestas são finalmente adicionadas à rede, com o cuidado que, caso duas arestas consecutivas sejam simétricas, são juntas numa única **struct Edge**. Para além disso, arestas com origem em **E** são convertidas para arestas com origem em **E** + **e** antes de ser adicionadas.

Depois da criação da rede, é chamada a função principal, **fifoPushRelabel**. O algoritmo inerente a esta função consiste num **Push-Relabel** genérico, no qual o processo de seleção de nós a descarregar baseia-se numa fila de prioridades (**FIFO**).

Primeiro, as alturas dos nós e os fluxos da origem para os fornecedores são inicializados na função **initializePreflow**, e os fornecedores são adicionados à **FIFO**. Depois, repetidamente, é removido o 1º nó da **FIFO** e é chamada sobre ele a função **discharge**, que empurra (**push**) o excesso de fluxo num nó para os seus vizinhos até ficar sem excesso, chamando a função **relabel** se a sua altura for inferior ou igual à de todos os seus vizinhos. Sempre que um nó sem excesso recebe fluxo, é adicionado ao fim da **FIFO**. O algoritmo termina quando a **FIFO** estiver vazia.

É depois chamada a função **DFSVisit**, que percorre, marcando como **visited**, todos os nós acessíveis da rede residual (transposta), começando pelo Hipermercado. Isto implica que o corte {**S**, **V** \ **S**}, em que **S** = {**u**: **u** não foi visitado}, é o corte mínimo mais próximo do Hipermercado.

É depois criado um vetor de arestas do corte , ou seja {(**u**, **v**): apenas **u** ou apenas **v** foi visitado}, através da função **getCutEdges**. Deste modo, é então possível calcular os outputs:

- **fluxo máximo**: somar os fluxos das arestas com origem em 0;
- **postos a aumentar**: arestas do corte com **src** entre (**f** + **2**) e (**e** + **f** + **1**);
- **meios de transporte a aumentar**: arestas do corte que saem de fornecedores (**2** <= **src** <= (**f** + **1**)), ou de postos de abastecimento (**src** >= (**e + f + 2**)).

Análise Teórica

O número de nós **N** (**struct Node**) é dado por **f** + **2** × **e** + **2**. O número de arestas **M** (**struct Edge**), no pior caso (não havendo arestas simétricas no *input*), é dado por **f** +

e + **t**. Vamos então determinar a complexidade da nossa solução em função de **N** e **M**, usando a notação assintótica.

A leitura do *input* tem complexidade temporal O(**M**), pois consiste na leitura de três valores singulares, **f** fornecedores, **e** estações e **t** arestas. A apresentação do *output* tem também complexidade O(**M**) pois, como as arestas de corte incluem as estações e os transportes, no máximo haverá **M** (estações + transportes) a imprimir.

Quanto à criação da rede, a criação da estrutura que guarda os nós demora tempo O(**N**) e a adição dos fornecedores, estações e transportes em conjunto demoram tempo O(**M**) (<= O($\mathbf{N^2}$)), pois, como vimos anteriormente, **M** <= **f** + **e** + **t**. A ordenação das arestas demora O(**M** × **log M**) <= O($\mathbf{N^2}$ × **log** $\mathbf{N^2}$) = O(**2** × $\mathbf{N^2}$ × **log N**) = O($\mathbf{N^2}$ × **log N**). Logo, o tempo total é O(**N** + $\mathbf{N^2}$ + $\mathbf{N^2}$ × **log N**) = O($\mathbf{N^2}$ × **log N**).

A função **fifoPushRelabel** começa por chamar a função **initializePreflow** que, como percorre todos os nós, arestas, e as adjacências da **src**, tem complexidade temporal O(**N + M**) (<= O($\mathbf{N^2}$)). Para analisar a complexidade do ciclo principal, vamos determinar um limite superior para a complexidade das operações **relabel** e **push** (saturante e não-saturante):

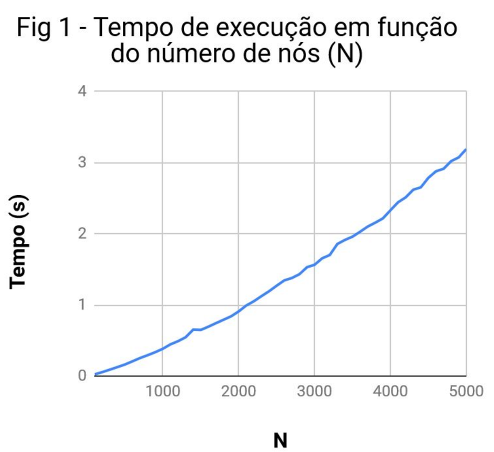
- **Relabels**: Como se verifica sempre que **h**[**u**] <= **2** × **N** - **1**, cada nó pode chamar a função **relabel** O(**N**) vezes, logo, o nº de **relabels** é O($\mathbf{N^2}$). A complexidade de um **relabel** é O(**N**), logo, a complexidade total é O($\mathbf{N^3}$).
- **Pushes saturantes**: Quando um **push** saturante é executado sobre uma aresta (**u**, **v**), só poderá ser executado sobre a aresta (**v**, **u**) quando **h**[**v**] aumentar pelo menos **2** unidades, e isto só pode ocorrer O(**N**) vezes (pelo mesmo argumento usado para a operação **relabel**). Logo, o nº total de **pushes** saturantes é O(**NM**) <= O($\mathbf{N^3}$), bem como a sua complexidade, pois um **push** tem duração constante.
- **Pushes não-saturantes**: Designe-se por **passagem 1**, a sequência de nós que estão no **FIFO** após chamar a função **initializePreflow** e, recursivamente, por **passagem n + 1**, os novos nós que surgiram na **FIFO** após realizar **discharge** sobre todos os nós da **passagem n**. Seja **H** = **max**{**h**[**u**]: **u** está na **FIFO**}. Se entre 2 passagens, o valor de **H** aumentar ou se mantiver, tem de ter havido pelo menos uma operação **relabel**, porque o nó da passagem atual cuja altura é **H** tem de ter recebido fluxo de um nó de altura superior a **H** (que não existia na passagem anterior). Como o nº de **relabels** é O($\mathbf{N^2}$), o nº de **passagens** em que **H** aumenta ou se mantém é também O($\mathbf{N^2}$). Como **H** começa a **0** e é sempre não-negativo, o nº de **passagens** em que **H** diminui é também O($\mathbf{N^2}$). Logo o nº total de **passagens** é O($\mathbf{N^2}$). Como cada **passagem** tem no máximo **N** - **2** nós e cada nó apenas pode realizar no máximo um **push** não-saturante por chamada da função **discharge** (o último que realizar), o nº total de **pushes** não-saturantes, bem como a sua complexidade total, é O($\mathbf{N^3}$).

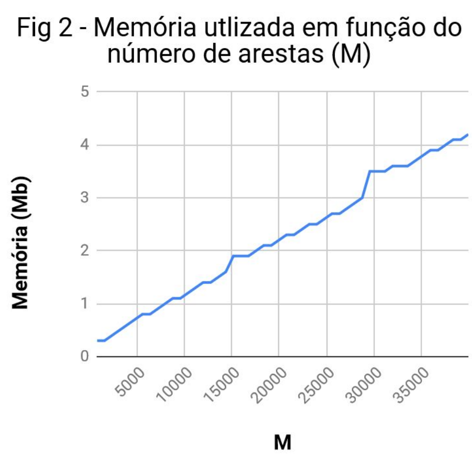
Assim, conclui-se que a função **fifoPushRelabel** tem complexidade O(**N^3**).
Finalmente, a DFS que determina o corte mínimo tem tempo O(**N + M**) <= O(**N^2**).
Em suma, a complexidade temporal do programa é O(**N^3**).

Em termos espaciais, o nosso programa guarda **N** nós e **M** arestas, ocupando espaço O(**N** + **M**). Mas como numa rede de fluxo **M** >= **N** - **1**, a complexidade destes dois componentes é O(**M**). Para além disto, cada nó tem uma lista de referências para arestas adjacentes. A soma dos comprimentos de todas as listas de adjacências é **2** * **M**, pois cada aresta é adjacente a dois nós. Logo, a complexidade destas listas é também O(**M**), coincidindo assim com a complexidade espacial total.

Avaliação experimental

Apresentamos aqui os resultados da avaliação da duração da execução e da memória utilizada pelo nosso programa com 50 *inputs* de 100 × **k** nós e 770 + 800 × (**k** - 1) arcos (com 1 <= **k** <= 50). Para cada um dos 50 *inputs*, realizámos 50 testes para calcular o tempo de execução, calculando a sua média, e um teste para calcular a memória utilizada pelo programa.





Os resultados experimentais apoiam a nossa análise teórica, visto que o gráfico da complexidade temporal é aproximadamente linear em **N** (O(**N**) <= O(**N^3**)) e o da complexidade espacial é linear com **M**, coincidindo com o limite superior determinado.

Referências:
Introduction to Algorithms, Third Edition
https://www.cs.princeton.edu/courses/archive/fall03/cs528/handouts/a%20new%20approach.pdf