

# Relatório 2º Projeto ASA 2020/2021

Grupo: al83

Aluno(s): Catarina Bento (93230) e Luís Freire D'Andrade (94179)

## Descrição do Problema e da Solução

Segundo o enunciado, o problema consiste, dados um programa (conjunto de processos) e um conjunto de processadores (neste caso, dois), em atribuir cada processo do programa a um processador, de forma a minimizar o custo total da execução do programa e evitar *overheads* computacionais. Na nossa solução, implementada em linguagem C++, o problema foi mapeado com recurso a uma rede de fluxos (grafo orientado, onde cada arco tem uma capacidade máxima e recebe um fluxo), onde: a fonte (nó de onde apenas sai fluxo) e o sumidouro (nó onde apenas entra fluxo) representam os dois processadores, e as capacidades máximas dos seus arcos os custos de execução dos processos nesse processador; enquanto que os restantes nós e arcos simbolizam os processos e os custos de comunicação entre eles, assumindo que estão a ser executados em processadores diferentes.

Tratando o problema como uma rede de fluxos, foi simples compreender que a sua solução teria de ser o corte de capacidade mínima (por outras palavras, o fluxo máximo), que separa os nós alcançáveis (processos executados pelo processador referente ao sumidouro) dos não alcançáveis (processos executados pelo processador referente à fonte). Para além disso, como sabemos que o fluxo máximo (custo de execução total do programa) é majorado ( $(\sum_{i \in PX} X_i + \sum_{i \in PY} Y_i) \in O(n) = O(V)$ , tal como enunciado na nota), partimos do princípio que teríamos de usar um algoritmo que dependesse deste dado para uma resolução eficiente do problema, nomeadamente, o algoritmo de Ford-Fulkerson ou as suas especializações (sendo que usámos o algoritmo de Ford-Fulkerson com recurso a uma BFS, isto é, o algoritmo de Edmonds-Karp<sup>1</sup>).

No código, para representar o grafo residual, foi utilizada uma classe “flowNetwork”. Esta classe contém uma variável inteira (com o número de nós) e um *array* de *arrays* de “edge”s (estrutura adicional que representa um arco, com variáveis inteiras para a capacidade máxima, para o fluxo e uma última para o fluxo reverso<sup>2</sup>) que retrata uma matriz de adjacências. Se, num caminho de aumento, se quiser aumentar o fluxo de um arco direcionado que liga  $u$  a  $v$  (assumindo que já foi verificado que não irá exceder a sua capacidade máxima), é necessário aceder à entrada  $[u][v]$  do *array* de *arrays* de “edge”s da classe “flowNetwork” e adicionar o fluxo à variável a que corresponde o fluxo, e retirar à que diz respeito o fluxo inverso.

(De notar que o código foi inspirado na solução apresentada [aqui](#)).



Figura 1 – Exemplo de um problema (onde  $x$  e  $y$  são processadores, e  $u$  e  $v$  processos)

<sup>1</sup> Complexidade do algoritmo passa a ser  $O(V \cdot E)$  em vez de  $O(V \cdot E^2)$ , uma vez que  $\min(|f|, V \cdot E) = |f| \in O(V)$

<sup>2</sup> É necessário incluir esta variável uma vez que existem arcos bidirecionados.

# Relatório 2º Projeto ASA 2020/2021

Grupo: al83

Aluno(s): Catarina Bento (93230) e Luís Freire D'Andrade (94179)

## Análise Teórica

Nesta análise teórica, considera-se  $V$  como o número de nós do grafo de input, e  $E^3$  como o número de arcos desse mesmo grafo.

- Leitura dos dados de entrada e construção do grafo: simples leitura do input, com um ciclo a depender linearmente de  $E$ . Logo,  $\Theta(E)$
- Execução do algoritmo Edmonds-Karp: ciclo *while* que, com o auxílio da BFS, enquanto existirem caminhos de aumento<sup>4</sup> (determinados pela BFS) que liguem a fonte ao sumidouro, determina o *bottleneck*<sup>5</sup> (maior fluxo que é possível aumentar) de cada caminho encontrado, e propaga esse fluxo (e soma a uma variável onde estão acumulados os vários *bottlenecks*).  $O(V \cdot E)^6$
- Apresentação dos dados.  $O(1)$

Complexidade global da solução:  $O(V \cdot E)$

## Avaliação Experimental dos Resultados

As experiências foram realizadas num computador com o Sistema Operativo Manjaro, processador Intel Core i5-10400F e 16GB de Ram.

Foram gerados 9 grafos de tamanho incremental. De seguida, foi cronometrado o tempo de execução do programa para cada um dos grafos gerados. Como resultado, foi originado o gráfico da Figura 2.

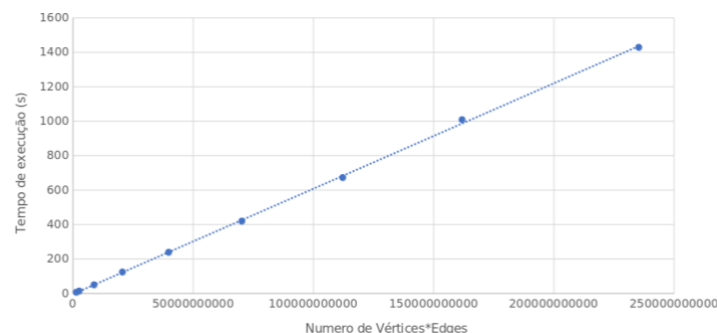


Figura 2 - Gráfico do Tempo de Execução em função de Vértices multiplicados pelos Arcos do grafo gerado

Sendo que a linha de tendência linear do gráfico (com o tempo de execução em função de  $V \cdot E$ ) se revela bastante próxima de todos os pontos, pode-se concluir que o gráfico gerado está concordante com análise teórica acima descrita, pois é possível observar que o tempo de execução do programa cresce linearmente com  $V \cdot E$ . Logo, a complexidade global da solução do algoritmo verifica-se,  $O(V \cdot E)$ .

<sup>3</sup> Nota:  $|V| = n + 2 \in O(n)$  e  $|E| = 2 \cdot n + 2 \cdot k \in O(n + k)$ , onde  $n$  e  $k$  são os parâmetros de entrada.

<sup>4</sup> No pior dos casos, existem  $V$  caminhos de aumento (em que o *bottleneck* de cada um é 1), uma vez que o fluxo é majorado em  $V$ .

<sup>5</sup> No pior dos casos, o caminho é formado por  $E$  arcos.

<sup>6</sup> Dado que, no pior dos casos, têm de ser determinados *bottlenecks* para  $V$  caminhos de aumento ( $V \cdot E$ ).