Relatório de Programação para o Problema 3

Equipa:
ID de estudante: 2019219581 Nome: Tatiana Silva Almeida
ID de estudante: 2019220082 Nome: Sofia Santos Neves

1. Descrição do algoritmo e correção

O algoritmo começa por verificar se a pipeline é válida. Isso significa que não pode ter ciclos, a mesma tem que estar conectada, ter apenas um nó inicial e ter apenas um nó final. Para contornar estes problemas, fez-se o seguinte para a rede de operações:

1. **Acíclica -** Construiu-se um algoritmo de pesquisa em profundidade de modo a detetar se há ciclos. Para tal, usámos uma estrutura para ver se o nó já estava visitado, *visited*, e outra para ver se existe um ciclo, *stack_*. Percorremos os nós desde o nó inicial até ao nó final passando por cada um dos filhos. Se o nó não estivesse visitado, colocamos o *visited* desse nó a *true* e a *stack_* também. Seguidamente, percorremos todos os nós filho desse mesmo nó e verificamos se, para cada um dos filhos, o nó não está visitado e se está tudo bem com a sub-árvore dele(não encontrou ciclos) ou se a stack do nó filho está a true, pode acontecer um de quatro casos:
    a. **O nó não está visitado e a sub-árvore está mal (equivalente a ter ciclos para baixo) -** Retorna *true* (pipeline inválida - com ciclos).
    b. **Para o caso de a) não se verificar -** Verifica c) ou d).
    c. **O nó filho já tem a *stack_* a true -** Retorna *true* (pipeline inválida - com ciclos).
    d. **O nó filho não tem a stack_ a true -** Retorna *false* (nó está válido).

    Caso a função retorne *false* então é porque a pipeline está inválida. Se retornar *true* então a pipeline não é acíclica.
2. **Conectada -** Para verificar se a pipeline era conexa, usamos o vetor *visited* que construímos no algoritmo acima descrito, e verificamos se todos os nós tinham sido visitados, em caso negativo, a pipeline não é conexa e imprimimos INVALID.
3. **Ter apenas um nó inicial -** Verificámos todos os nós. Se houver algum nó que não tenha nós pais, então é nó inicial. Caso contrário, caso não haja nenhum nó sem pais ou se houver mais que um, então a pipeline é inválida.
4. **Ter apenas um nó final -** Verificámos todos os nós. Se houver algum nó que não tenha nós filho, então é nó final. Caso contrário, caso não exista nenhum nó sem filhos ou se houver mais que um, então a pipeline é inválida.

Se a pipeline for inválida, é impresso “INVALID”. Se não, então verifica-se a estatística e agimos de acordo. Existem quatro estatísticas possíveis:

- **0 -** É impresso “VALID”;

- **1 -** Consiste em imprimir o tempo mínimo necessário para correr a pipeline seguido da ordem de processamento dos nós (são impressos os nós por ordem crescente de número).

  Em primeiro lugar, fazemos o somatório dos tempos de todos os nós e imprimimos o mesmo.

  Em segundo lugar, vamos fazer a impressão de nós recorrendo a uma *priority_queue*. Para a construir, inicializamos a mesma com o first (nó inicial). Vamos ver, para cada filho desse mesmo pai, se todos os nós pais dele já foram processados. Se sim, então o nó filho é colocado na *pq* (priority queue) por ordem crescente de número do nó e, em seguida, retiramos o primeiro nó da *pq* e repetimos o processo até já não haver mais nós na *pq* (equivalente a terminar de imprimir todos os nós).

- **2 -** Para a construção deste algoritmo optamos por usar programação dinâmica. A nossa abordagem passou por, percorrermos todos os nós desde os pais até aos filhos.Para isso, deparamo-nos com três opções:
  1) **O filho já ter sido processado (o valor do filho na dp já se encontra diferente de 0)** - continua para o próximo filho;
  2) **O filho não ter sido processado** - chama-se a função recursivamente para esse filho;

  Depois de todos os filhos já terem sido processados existem 2 opções:
  1) **Ser o nó final -** Guarda na sua posição da dp o tempo que demora no seu processamento;
  2) **Não ser o nó final -** Neste caso, como não é o nó final, obrigatoriamente tem filhos. Logo, na sua posição da dp, vai guardar a soma do seu tempo de processamento com o maior valor guardado na posição da dp dos seus filhos.

- **3 -** Para a impressão da estatística 3 é necessário imprimir todos os nós *bottleneck* e para isso chamamos uma função para todos os nós da pipeline, pela mesma ordem que imprimimos os nós da estatística 1, com o objectivo de os testar.

  Para verificar se um dado nó é ou não um *bottleneck*, começamos por chamar, de forma recursiva, para todos os filhos do nó e consecutivamente para os filhos dos filhos, uma função. Nessa mesma função é apenas marcado o nó como visitado.

  Para além disso, fazemos exatamente a mesma coisa para os pais, e recursivamente para os pais dos pais, marcando no vetor quais os nós visitados.

  No final, caso todos os nós tenham sido visitados, significa que são ‘’família direta’’ do nó que está a ser testado, o que implica que não podem ser processados ao mesmo tempo que este. Sendo que não existe nenhuma outra operação que possa ser feita em simultâneo, consideramos o nó em estudo como sendo bottleneck.

Caso exista algum nó que não tenha sido visitado, então, esse mesmo nó e o nó que estamos a verificar se é bottleneck não são "família direta'', ou seja, podem ser executados em simultâneo. Logo, o nó que estamos a testar não é um bottleneck.

2. Estruturas de dados

Como estruturas de dados optámos por duas listas de adjacência.

- ➔ **Parents -** Guarda, em cada posição, os pais do nó respectivo;
- ➔ **Childs -** Guarda, em cada posição, os filhos do nó respectivo.

Estas foram as principais estruturas de dados que nos permitiram percorrer a pipeline desde a tarefa inicial até à final, mas também desde a final até à inicial.

Existem também algumas estruturas que nos auxiliaram no desenvolvimento de algumas estatísticas.

- ➔ **Visited -** Vetor de booleanos que guarda quais os nós já visitados quando precisamos de percorrer a árvore;
- ➔ **Stack_ -** Vetor de booleanos que nos auxiliou na validação da pipeline, para descobrir se existia um ciclo ou não e na estatística 3 (ambos explicados anteriormente);
- ➔ **Visited_childs -** Vetor de booleanos que guarda quais dos nós filhos já foram visitados. Auxilia na estatística 3;
- ➔ **Visited_parents -** Vetor de booleanos que guarda quais dos nós pais já foram visitados. Auxilia na estatística 3;
- ➔ **Pq -** priority_queue implementada para sabermos em qual ordem é que o algoritmo da estatística 3 deve de ser feito, e a ordem pela qual deve ser impresso o resultado da estatística 1;
- ➔ **Time_ -** Vetor de inteiros que em cada posição guarda o tempo que aquela operação demora a ser executada;
- ➔ **Dp -** Vetor de inteiros que nos auxilia na execução da estatística 2 (explicado anteriormente);
- ➔ **Degree -** Vetor de inteiros que nos auxilia na construção da priority_queue e que nos permite apenas introduzir os nós, quando todos os seus pais já se encontram processados.

4. Análise do algoritmo

Assumindo o pior dos casos a escolha da estatística 3 então, a complexidade temporal toma o valor de n^2 dado que para cada nó temos que testar os pais e filhos desse mesmo nó até atingir a raiz e as folhas, respectivamente.

A complexidade espacial toma o valor de n dado que as estruturas de dados utilizadas são sempre vetores com n+1 de tamanho.

5. Referências

[1] - https://en.cppreference.com/w/