Relatório do Problema de Programação 3

Equipa:

N.º Estudante: 2019216949 Nome: Duarte Emanuel Ramos Meneses
N.º Estudante: 2019213995 Nome: Patrícia Beatriz Silva Costa

1. Descrição do Algoritmo

O nosso algoritmo, à medida que lê as informações provenientes do *input*, vai vendo se o nó em questão tem ou não dependências (não ter dependências significa ser o inicial). Caso não as tenha e ainda não exista nenhum nó nessa condição, marcamos esse como sendo o nó inicial, para sabermos onde começa o grafo. Caso já tenhamos lido as informações de um nó sem dependências, significa que temos dois em condição de ser iniciais. Como essa é uma condição de rejeição do grafo, dizemos à partida que o grafo é inválido e terminamos a execução do algoritmo.

No caso de só existir um nó inicial, ao chegarmos ao final do *input*, o nosso algoritmo vai verificar se existe apenas um nó final. Para tal, vai percorrer todos e, para cada um, ver quantos nós dependem de si diretamente. Se existir mais do que um nó sem outros a dependerem de si, significa que existe mais que um nó em condição de ser final. Sendo que isto não pode acontecer, o nosso algoritmo deteta que o grafo é inválido e termina a execução.

Se no fim destas verificações o grafo ainda for válido, vamos fazer a última verificação para poder prosseguir para a próxima fase. Para um grafo ser válido, não pode ter ciclos. Deste modo, vamos, para cada nó, verificar o seu percurso em diante, numa espécie de procura em profundidade. Caso, no mesmo percurso, passemos por um nó anteriormente visitado, significa que voltamos atrás, ou seja, estamos perante um ciclo. Nesta situação, reportamos que o grafo é inválido e não prosseguimos com a execução do programa.

Chegando a esta fase, é sinal que o grafo é válido. Como tal, caso a opção lida no *input* tenha sido 0, o nosso algoritmo vai reportar que o grafo é válido.

Caso a opção tenha sido 1, vamos calcular o tempo necessário para percorrer todo o grafo e a respetiva ordem. Deste modo, começando no nó inicial, vamos percorrendo os nós que se seguem e adicionando a uma fila sempre que um nó for visitado por todos os seus antecessores diretos (significa que já pode ser executado). Como a ordem de execução, em caso de empate, deve ser realizada crescentemente, ordenamos a fila sempre que adicionamos um nó. Com isto, o nó no início da fila é sempre o próximo a ser executado e basta seguir a partir deste. Nesta opção, sendo que se pede o tempo necessário se as tarefas forem executadas uma de cada vez, este resultado será igual à soma dos tempos necessários a realizar todas as tarefas.

Já a opção 2 pede para indicar o tempo mínimo necessário para executar todas as tarefas caso se possam paralelizar operações. Deste modo, percorremos recursivamente o grafo. Indo do nó inicial para o final, chegando aqui, vamos em sentido inverso calculando o melhor caso para cada tarefa. Se

um nó tiver filhos, este ficará com a soma do seu tempo com o maior valor dos seus filhos. Isto fará com que chegando ao nó inicial tenhamos o melhor tempo possível.

Por último, a opção 3 pedia que indicássemos os *bottlenecks*, ou seja, os nós que não podem ser paralelizáveis. Fica óbvio que, para um nó ser *bottleneck*, a partir deste deve ser possível aceder a todos os outros, quer para trás quer para a frente. Se não se conseguir, é sinal que existe algum outro nó que pode ser paralelizável com este. Com este raciocínio em mente, decidimos, para cada nó, verificar todos os que se seguem até ao fim e os antecessores até ao início. Se nestes percursos passarmos por todos os nós exceto pelo próprio, significa que o nó não pode ser paralelizável, logo é *bottleneck*. Tal como na estatística da opção 1, apenas adicionamos o nó à fila de prioridade (para ser executado em ordem crescente) para ser analisado depois de todos os seus antecessores terem sido tratados para evitar repetições e retiramos sempre o que está no início.

2. Estruturas de dados

Para implementar com sucesso o nosso algoritmo, decidimos utilizar uma estrutura representativa de cada operação que tem como atributos o tempo necessário para executar essa tarefa, o número de dependências diretas, um vetor com, exatamente, os nós de quem este depende diretamente, outro com os que o sucedem diretamente, o número de vezes que este nó já foi visitado (para se saber quando colocar o nó na fila de prioridade já acima explicada), o melhor tempo possível (para a estatística 2) e uma variável que indica quantos nós foram percorridos na estatística 3 a partir desse.

Para guardar as tarefas decidimos utilizar um *vector*. Já nas estatísticas 1 e 3, para conseguir calcular o devido por ordem crescente, usamos uma *priority queue* para que no início da fila esteja sempre o nó de menor *id*.

Utilizamos ainda outros vetores auxiliares como o que guarda o resultado e outros dois que ajudam a ver ciclos e na estatística 3 (vetor de visitados).

3. Correção

Para a nossa solução não ser ótima, tem de existir pelo menos uma solução mais ótima para calcular o pretendido.

Subestrutura ótima:

Assunção: A nossa solução é ótima.

Porém, existe outra solução mais ótima.

Negação: No caso de verificar se o grafo é válido, seria mais ótimo se conseguíssemos chegar a essa conclusão mais rapidamente. Sendo que precisamos de percorrer todos os nós para isso, é impossível melhorar.

Na estatística 1, o algoritmo seria mais ótimo se obtivéssemos um tempo inferior. Sendo que só se executa uma tarefa de uma vez, o resultado tem de ser igual à soma de todos os tempos, pelo que não dá para otimizar.

Quanto à estatística 2, seria mais ótimo se escolhêssemos os menores valores dos filhos. Se assim fosse, não haveria tempo suficiente para executar as tarefas de maior duração. Sendo assim, é impossível otimizar.

Já na estatística 3, o algoritmo seria mais ótimo se não fosse necessário percorrer todos os nós. Como essa é uma questão imprescindível para perceber se um nó pode ser paralelizável ou não, não existe uma solução mais ótima para isto.

Consequência: A nossa solução é ótima.

Contradição: Não existe nenhuma solução mais ótima que a nossa.

4. Análise do Algoritmo

Complexidade espacial: Sendo que no nosso algoritmo temos várias estruturas de tamanho igual ao número de tarefas (N), a complexidade espacial é:

$$N => O(N)$$

Complexidade temporal: No nosso algoritmo percorremos 1 vez as tarefas todas na leitura. Após termos as operações todas lidas e na estrutura devida, vamos verificar se tem mais que um nó final percorrendo todos os nós (N) e se tem ciclos percorrendo todos os nós recursivamente (2N). Com isto, o resto depende da opção escolhida. Se for a 0, ficamos por aqui, sendo a complexidade temporal dada por:

$$N + N + 2 * N => 4N => O(N)$$

Já se for a estatística 1, percorremos todos os nós e, para cada um, percorremos os seus sucessores diretos. No pior caso, a complexidade temporal é:

$$N + N + 2 * N + N * k => 4N + kN => O\big(N(4 + k)\big) => O(N),$$

com k igual ao número de sucessores de cada nó.

No caso da estatística 2, percorremos o grafo do início até ao fim e voltamos recursivamente até ao início. Deste modo:

$$N + N + 2 * N + 2 * N => 6N => O(N)$$

Por último, na estatística 3, para cada nó, percorremos todos os outros recursivamente, tanto os anteriores como os sucessores. Com isto:

$$N + N + 2 * N + 2 * N * N => 4N + 2N^2 => O(N^2)$$

5. Referências

- Material fornecido pelos docents [acedido em 22/04/2022];
- Trabalho prático 2.