Relatório do Problema 3

Team:

2019215917 Inês Marçal

2019219433 Noémia Gonçalves

1. Algorithm description

Ao mesmo tempo que é lido o input verifica-se qual nó é o inicial, ou seja, o nó que não apresenta dependências. Caso exista mais do que um nó inicial nos dados de input, ou seja, mais do que um nó sem dependências, o grafo é considerado inválido.

Os nós são percorridos de acordo com uma ordem topológica ordenada numericamente sendo para isto utilizado um vetor para simular uma *priority queue*. Procede-se em simultâneo à verificação dos nós *bottlenecks*. Para tal, têm que estar sozinhos na queue e o seu *vetor parent* tem de ter o mesmo tamanho que o dos nós já visitados. A cada nó é verificado o número de nós filhos, caso este seja zero o nó é o final. Deste modo, caso exista mais do que um nó final, o grafo é considerado inválido.

Recorre-se à função *addParent(int p,int c)* com o objetivo de se inserir os nós contidos no v*ector parents* do nó p no v*etor parents* do nó c. Para se evitar ter complexidade quadrática nesta função, começa-se por marcar os nós que já se encontram no vetor parents do nó c. Posteriormente, percorre-se o *vetor parents* do nó p e adiciona-se ao vetor c os nós que ainda não se encontram no mesmo. No fim se o nó p não estiver marcado é também adicionado ao vetor. Através desta implementação conseguimos saber todos os nós que se precisou de percorrer para se chegar a um nó, não sendo necessário passar pelos anteriores.

Sempre que se visita um nó que seja filho do nó em questão é calculado o tempo que se demorou a chegar até aquele nó, sendo que o tempo mais longo de entre todos os possíveis caminhos é guardando de forma a obter o mínimo tempo que a *pipeline* demora a processar a tarefa, tendo esta disponível um número infinito de processadores. Ao mesmo tempo que se percorre os nós, o seu grau é diminuído. Caso este fique a zero, é adicionado à *queue*. Após esta passagem, o nó em questão é adicionado ao vetor *topologicalOrder*. Por sua vez, este permite não só que seja obtido a ordem topológica e numérica crescente pretendida como também o conjunto de nós que já foram visitados. Antes de se passar ao nó seguinte, o vetor que representa a queue é ordenado de forma a garantir a ordem pretendida.

É também incrementado um contador para que, quando não for possível visitar mais nós, seja indicado se todos os nós foram percorridos ou se existe algum ciclo.

2. Data structures

As estruturas utilizadas no algoritmo implementado foram as seguintes:

Struct node:

int indegree: grau interno do nó

int time: tempo que a tarefa demora a ser realizada

int best: maior tempo conseguido até ao respetivo nó

vector<int> parents: vetor com todos os nós percorridos antes de se chegar ao respetivo nó

vector<int> children: conjunto de nós que representa as arestas ligadas ao nó em questão

vector<int> bootleneck: vetor onde são guardados os *bootlenecks* encontrados

vector<int> topologicalOrder: vetor onde é guardada a ordem topológica dos nós passados

array<node,1001> nodes: array de nós onde são guardadas as struct nodes dos nós existentes

3.Correctness

Para otimizar a solução foram melhoradas as seguintes operações:

-Tempo mínimo que a pipeline demora a processar a tarefa, tendo esta disponível um número infinito de processadores: Uma pipeline pode ser agrupada em blocos que têm início em um *bottleneck* e vão até ao *bottleneck* seguinte. O tempo mínimo para processar todos os blocos de um conjunto foi definido como o tempo do caminho mais longo, uma vez que um *bottleneck* depende de todos os nós que o antecede e como tal só avança para o *bottleneck* seguinte após a conclusão do caminho mais longo entre os *bottlenecks*.
-Ordem topológica: Após a obtenção do conjunto de nós que já não têm dependências e podem ser ainda visitados, este é ordenado de forma a garantir uma ordem topológica e numérica crescente.
-Bottlenecks: Os *bottlenecks* são dependentes de todos os nós que já foram visitados e são comuns a todos os caminhos. Para o controlo das dependências é mantido um vetor para cada nó que contêm todos os caminhos possíveis para alcançar o nó em questão. Para melhorar a complexidade recorreu-se à função *addParent* referida anteriormente.

4. Algorithm Analysis

A complexidade temporal de percorrer os nós é O(V + E), onde E representa o número de arestas e V o número de vértices. Nesta passagem recorre-se à função *sort* cuja complexidade é O(N*logN), sendo que N representa o número de nós que se pode visitar. Por fim, utiliza-se a função *addParent(int p,int c)*, tal como referido anteriormente, cuja complexidade é O(k + j), sendo que k representa o número de elementos do *vetor parent* do nó com o índice do parâmetro c e j o número de elementos do *vetor parent* do nó com o índice do parâmetro p. Deste modo, a complexidade total é O((V + E) * (E * (k + j) + N log N)).

Ao ser utilizado um *array nodes* que contém N *structs node*, um *vetor topologicalOrder* que terá N elementos e por fim um *vetor bootleneck* que irá conter no máximo N elementos também, a complexidade espacial irá ser 3N. No entanto, como não se está a considerar os *vetores parent e children* que dependem da estrutura da *pipeline*, a complexidade espacial do algoritmo será sempre maior do que 3N.

5. References

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