ALGORITMOS E ESTRUTURAS DE DADOS III

DOCUMENTAÇÃO TP0

13 de Outubro de 2018

Matrícula: 2015079208
Universidade Federal de Minas Gerais

1 INTRODUÇÃO

O problema consiste em organizar um grafo de forma que suas dependências estejam ordenadas, isto é, um vértice que depende de outro sempre estará atrás desse vértice em uma lista de dependência.

Esse cenário pode ter três saídas diferentes:

1. O grafo é acíclico e possui apenas uma lista de dependência óbvia;
2. O grafo é cíclico, possui mais de uma lista de dependência possível mas é possível estimá-la;
3. O grafo é cíclico ou disperso e não é possível estimar uma lista de dependência.

2 MODELAGEM E SOLUÇÃO DO PROBLEMA

O problema mostrado na sessão anterior foi resolvido utilizando o método de ordenação topológica descrita no livro Introdução a algoritmos de Thomas Cormen.

Esse algoritmo usa um DFS para caminhar pelo grafo e um arranjo para armazenar quanto tempo levou para atingir cada um dos vértices. O que oferece como saída a lista de dependências necessária para a resolução do problema.

A seguir oferece-se uma visão detalhada sobre o algoritmo e os códigos acessórios para que o programa funcione como um todo.

Inicia-se um grafo que consiste em um dicionário de listas. Essa estrutura de dados foi escolhida em detrimento de uma simples lista de adjacência porque ela é capaz de lidar com entradas que não são uma lista sequencial de números.

5	5
2	1
3	2
4	3
4	2
5	4

Tabela 1: Exemplo entrada 1

5	5
22	1
3	22
14	3
14	22
5	4

Tabela 2: Exemplo entrada 2

Um grafo implementado como dicionário de listas é capaz de lidar com ambos exemplos, enquanto uma implementação usando um vetor de vetores não conseguiria lidar com o segundo caso já que o índice é implícito.

Além disso é necessário criar um set que contém a ordem das chaves do DFS, um set que será responsável por definir se um grafo é do tipo 1 ou 2, um dicionário que contém quais os vértices visitados e um dicionário que contém o tempo final que é uma das saídas de DFS.

No momento em que se obtém os dados insere-se os vértices no set de ordem, no grafo e no set que definirá os tipos. Usando o Exemplo entrada 1 como exemplo as estruturas podem ser visualizadas melhor da seguinte forma:

Vértice	Vizinhos	Vizinhos
2	1	
3	2	
4	3	2
5	4	

Tabela 3: Grafo

2	1	3	4	5

Tabela 4: Ordena

2	3	4	5

Tabela 5: Define o tipo

A seguir é necessário usar o set que contém todas as chaves para criar um vetor com as chaves ordenadas. Esse vetor é responsável por tratar os grafos do segundo tipo e será discutido mais tarde. O vetor criado é algo nas linhas de:

1	2	3	4	5

Tabela 6: Vetor de vértices ordenado

Só então usa-se o algoritmo de DFS, essa é a forma como o algoritmo é chamado no código:

```
int time = 0;
for (int vertex : vertices) {
    if (marked[vertex] != 2) {
        if (DFS(graph, vertex, marked, &time, finalTime) == 1) {
            fprintf(outputFile, "0 -1\n");
            return 0;
        }
    }
}
```

Esse trecho do código itera sobre os vértices do vetor mostrado acima na tabela 6. Depois checa-se se o vértice marcado é igual a 2 e também se o resultado da função DFS é igual a 1, qualquer um dos dois casos significa que esse é um grafo cíclico e o classifica como um problema da categoria 3.

```
int DFS(unordered_map<int, vector<int>>& graph, int key, unordered_map<int,
    int>& marked, int *time, unordered_map<int, int>& finalTime) {
    *time = *time + 1;
    marked[key] = 1;
    for (int neighbor : graph[key]) {
        if (marked[neighbor] == 0) {
            if (DFS(graph, neighbor, marked, time, finalTime)) return 1;
        }
        else if (marked[neighbor] == 1) {
            marked[neighbor] = 2;
            return 1;
        }
    }
    marked[key] = 2;
    *time = *time + 1;
    finalTime[key] = *time;
    return 0;
}
```

O modo como esse algoritmo itera sobre o grafo pode ser entendido facilmente se observa-se a tabela 3. O for do primeiro trecho de código itera sobre os vértices enquanto o for do trecho acima itera sobre os vizinhos do grafo. Caso o vizinho não tenha sido visitado ainda o algoritmo é chamado em si mesmo de forma recursiva, caso já tenha sido visitado é marcado como um nó PRETO adiciona-se 1 ao tempo e adiciona-se àquele tempo ao vetor que contém os tempos de cada vetor. Uma vez que o algoritmo tenha passado por todas as colunas de uma respectiva linha ele termina e então itera-se sobre outra linha.

Para definir o tipo de grafo usa-se o vetor de definição de tipos. Caso o número de elementos que exista no vetor seja menor do que o número total de vértices menos um significa que esse grafo é do tipo 2, caso contrário esse grafo é do tipo 1. Isso acontece porque caso haja mais de um vértice que não aponta para lugar nenhum então não é possível definir uma única lista de dependências.

Por fim, ordena-se o resultado de acordo com o vetores de tempos finais obtidos por meio do algoritmo e imprime-se os resultados.

Vértices	Tempo
100	0.006
200	0.008
300	0.01
400	0.014
500	0.018
600	0.27
700	0.29
800	0.04
900	0.044
1000	0.054

3 ANÁLISE TEÓRICA DO CUSTO ASSINTÓTICO

De forma geral o algoritmo não passa de uma complexidade O(n) sendo n ou o número de vértices ou o número de arestas. A maior complexidade do programa se encontra no algoritmo que checa a ordenação topológica: $\theta(V+E)$, onde V é o número de vértices e E o número de arestas.

4 ANÁLISE DE EXPERIMENTOS

A máquina usada tem as seguintes configurações:
Memória: 16 GB 1600 MHz DDR3
Processador: 2,6 GHz Intel Core i7

5 CONCLUSÃO

De forma geral tentei tomar certos cuidados para otimizar a velocidade com que o programa roda tendo em vista que passei estruturas de dados como referência quando possível. Também tentei comentar e deixar o código o mais claro possível, usando nomes descritivos de variáveis e estruturas lógicas não obscuras.

Na documentação do trabalho estava especificado que era esperado