Análise e Síntese de Algoritmos

Relatório do 1º Projeto

Introdução

Com este projeto foi introduzido o problema da ordenação cronológica de fotografias em que as datas são desconhecidas, sabendo apenas que dadas duas fotografias u e w, com u diferente de w, u é mais velho do que w se a aresta a é direcionada de u para w.

O conjunto de fotografias e as suas relações é representado por um grafo conexo dirigido (a relação entre as fotografias é direcional) no qual um vértice corresponde a uma fotografia e uma aresta corresponde à relação cronológica entre duas fotografias. O problema traduz-se em ordenar por ordem cronológica, e no caso do grafo ordem topológica, o conjunto de fotografias introduzido.

Neste problema é fornecido um input com informação acerca do número de vértices V e de arestas E do grafo (primeiro par do input). Os restantes E pares correspondem às arestas dos vértices V, sendo o vértice da esquerda respetivo à (mais antigo) e o vértice da direita respetivo ao destino (mais recente).

Descrição da Solução

A solução do problema foi concebida em linguagem de programação C, onde na implementação do grafo foram utilizadas listas de adjacência ligadas por nós: para cada vértice do grafo guardam-se os vértices com quem ele partilha uma aresta.

O índice de cada lista de adjacência representa o vértice de origem. Por exemplo, a lista com índice zero de um grafo representa todas as ligações que o vértice 1 tem com outros vértices, sendo este o vértice fonte da aresta dirigida. Esta estrutura permite adicionar dinamicamente arestas ao grafo, tendo sempre em atenção a direção das mesmas.

O algoritmo basicamente pesquisa em profundidade, percorrendo cada vértice do grafo, de modo a gerar no fim uma lista com o grafo ordenado topologicamente.

Para tal foi implementada uma adaptação do algoritmo DFS, de modo a perceber se o grafo introduzido possui informação suficiente ou se a informação dada não é incoerente, isto é, se é possível ou não gerar uma ordem topológica final do grafo. Para tal consideramos dois critérios:

1. O grafo não pode conter *back egdges*, isto é, o grafo não pode ser cíclico;
2. A ordenação tem de ser linear, isto é, o vértice u é mais velho do que $u+1$ e assim sucessivamente, não podendo existir vértices sem arestas.

O primeiro critério é evidente, se o grafo for cíclico existe um vértice que é mais velho e ao mesmo tempo mais recente que outro, logo a informação dada é incoerente e nenhuma ordenação pode ser gerada.

O segundo critério é facilmente verificável através das ligações entre vértices, isto é, depois de gerada a ordenação topológica do grafo verifica-se se cada vértice da lista ordenada tem ligação com o seguinte.

Imagine-se uma ordenação topológica: 3 4 1 2 5. Para verificar o segundo critério basta ir-se à lista de adjacências correspondente ao vértice 3 e verificar se existe uma ligação com o vértice 4. O mesmo para os vértices 4 e 1, 1 e 2 e assim sucessivamente.

Análise Teórica

Pseudo-código do algoritmo implementado.

```
DFS(G)

1.    for each vertex u ∈ V[G]
2.          do color[u] ← white; p[u] = NIL
3.    time ← 1
4.    for each vertex u ∈ V[G]
5.          do if color[u] = white
6.                then DFS-Visit(u)
```

```
DFS-Visit(u)

1.    color[u] ← gray
2.    d[u] ← time
3.    time ← time + 1
4.    for each v ∈ Adj[u]
5.          do if color[v] = white
6.                then p[v] ← u
7.                     DFS-Visit(v)
8.             else if color[v] = grey
9.                then print(‘‘Incoerente’’)
10.                    exit
11.            else if color[v] = black and d[u] > f[v]
12.               then p[v] ← u
13.   color[u] ← black
14.   f[u] ← time
15.   time ← time + 1

VerificaInsuficiencia(G, topologicList)

1.    for each v ∈ topologicList
2.          do if VerificaUnicidade(G, v, v+1) = false
3.                then print(Insuficiente)
4.                     exit

VerificaUnicidade(G, fonte, destino)

1.    for each v ∈ Adj[fonte]
2.          do if v = destino
3.                return true
4.    return false
```

Em termos de complexidade, o algoritmo equipara-se ao algoritmo DFS, sendo a complexidade *O(V + E)*.

Para a inicialização é *O(V)* percorrendo os vetores dos tempos de descoberta, tempos de finalização, predecessores e cores, inicializando estes. Como para cada vértice *u* são examinados *|Adj[u]|* arestas, a complexidade aqui é *O(E)*.

Para a função *VerificaInsuficiencia* a complexidade é *O(V)* visto que o tamanho da lista ordenada topologicamente gerada pelo grafo tem o mesmo tamanho que o número de vértices do grafo. Para a função *VerificaUnicidade* a complexidade é *O(E)* pois é visitada a lista de adjacências do vértice.

Avaliação Experimental de Resultados

Inicialmente sujeitou-se o programa aos 4 inputs fornecidos pelo enunciado do projeto, tendo este passado a todos os testes: os outputs gerados foram iguais aos esperados.

De seguida submeteu-se o projeto no sistema mooshak, onde o projeto foi sujeito a 16 testes de avaliação e aos quais passou.

Finalmente utilizou-se uma ferramenta de profiling do Linux, o *gprof*, de modo a saber o número de chamadas das funções existentes. Para um input de 10 vértice e 9 arestas verificou-se uma complexidade de O(19) para a função *DFS* e para a *VerificaInsuficiência*. Para inputs de 100 e 1000 vértices verificou-se, respetivamente, O(199) e O(1999).

Utilizando o *valgrind* para verificar fugas de memória, este gerou o seguinte output, onde se conclui que 0 blocos de memória foram perdidos.

```
==31185== HEAP SUMMARY:
==31185==     in use at exit: 0 bytes in 0 blocks
==31185==   total heap usage: 15 allocs, 15 frees, 280 bytes
allocated
==31185==
==31185== All heap blocks were freed -- no leaks are possible
==31185==
==31185== For counts of detected and suppressed errors, rerun with:
-v
==31185== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
from 0)
```

Referências: Livro da cadeira, manuais do valgrind e do gprof.