AED - Projeto 2

Universidade de Aveiro

Orlando Marinheiro, Tomás Oliveira, Afonso Vieira

AED - Projeto 2

Dept. de Eletrónica, Telecomunicações e Informática
Universidade de Aveiro

afonso.vieira@ua.pt(112822), tomas.esteves.oliveira@gmail.com(113939), orlandomarinheiro@ua.pt(114060)

5 de janeiro de 2024

Conteúdo

1 Introdução **1**

2 Análise de Funções **2**
2.1 GraphTopoSortComputeV1 2
2.1.1 Descrição da Função 2
2.1.2 Métricas da Complexidade 2
2.1.3 Resultado dos Testes 3
2.2 GraphTopoSortComputeV2 4
2.2.1 Descrição da Função 4
2.2.2 Métricas da Complexidade 4
2.2.3 Resultado dos Testes 4
2.3 GraphTopoSortComputeV3 5
2.3.1 Descrição da Função 5
2.3.2 Métricas da Complexidade 5
2.3.3 Resultado dos Testes 5

3 Conclusão **6**

Capítulo 1

Introdução

A ordenação topológica é um método usado em grafos direcionados para organizar os vértices de tal forma que todos os arcos vão de vértices de menor para vértices de maior ordem. Existem diferentes algoritmos para alcançar essa ordenação em grafos direcionados.

Neste relatório, exploramos e comparamos três algoritmos diferentes para ordenação topológica:

- Algoritmo baseado em cópia do grafo.
- Algoritmo usando um array auxiliar.
- Algoritmo que faz uso de uma fila.

Capítulo 2

Análise de Funções

2.1 GraphTopoSortComputeV1

2.1.1 Descrição da Função

Este algoritmo opera criando uma cópia do grafo original. Ele efetua sucessivas remoções dos arcos emergentes de vértices que não possuem arcos incidentes. Isso continua até que todos os vértices sejam marcados e a ordenação topológica seja obtida.

2.1.2 Métricas da Complexidade

Quanto à complexidade de `GraphTopoSortComputeV1`, esta é dada por:

```
// iniciar a estrutura topoSort
GraphTopoSort* topoSort = _create(g);
```

Na linha de código é usada a função *_ Create*, que envolve a inicialização de estruturas de dados. Assim, a complexidade é dada por (onde n é o número de vértices do grafo):

$$O(f_1(n)) \tag{2.1}$$

```
// Criar uma cópia do grafo g
Graph* gCopy = GraphCopy(g);
```

Neste caso, *GraphCopy* é chamada, criando uma cópia do grafo. A complexidade desta operação é (onde m é o número de arestas no grafo):

$$O(f_2(n, m)) \tag{2.2}$$

Sendo *GraphCopy* um loop externo, este é executado n vezes. Dentro desse loop, há um loop interno que executa até n vezes para encontrar um vértice com grau de entrada zero. As operações dentro desses loops internos são, na pior das hipóteses, proporcionais ao número total de arestas m. Então, a complexidade do loop externo é:

$$O(n \cdot (n + m)) \tag{2.3}$$

Assim a complexidade total do algoritmo é:

$$O(f_1(n)) + O(f_2(n, m)) + O(n \cdot (n + m)) + O(f_3(n, m))$$

2.1.3 Resultado dos Testes

Grafos

Nome do Ficheiro	Time	Caltime	Somas	Memória
SWmediumEWD.txt	0.000550	0.000483	250	1005
SWtinyDAG.txt	0.000023	0.000022	56	214
SWtinyDG.txt	0.000019	0.000017	18	71

Grafos Orientados

Nome do Ficheiro	Time	Caltime	Somas	Memória
DAG_1.txt	0.000018	0.000022	32	118
DAG_2.txt	0.000012	0.000012	38	124
DAG_3.txt	0.000018	0.000023	36	122

2.2 GraphTopoSortComputeV2

2.2.1 Descrição da Função

Neste método, não é necessária uma cópia do grafo. Em vez disso, um array auxiliar (um dos campos do registro) é usado para procurar sucessivamente o próximo vértice a ser adicionado à ordenação topológica. Este algoritmo percorre o grafo e identifica os vértices sem arestas de entrada para adicioná-los à ordenação.

2.2.2 Métricas da Complexidade

Quanto à complexidade de `GraphTopoSortComputeV2`, esta é dada por:

```
for (unsigned int v = 0; v < topoSort->numVertices; v++) {
  //registar num array auxiliar o InDegree de cada vértice ( já temos na estrutura numIncomingEdges)
  topoSort->numIncomingEdges[v] = GraphGetVertexInDegree(g, v);
```

$$O(n) \quad \text{onde } n \text{ é o número de vértices no grafo.} \tag{2.4}$$

A complexidade total do algoritmo é $O(n^2)$, onde n é o número de vértices do grafo. Este é o resultado da combinação do loop externo que executa n vezes para além do loop interno que executa até n vezes. Isso ocorre porque, no pior caso, para cada vértice, são verificados todos os outros vértices.

2.2.3 Resultado dos Testes

Grafos

Nome do Ficheiro	Time	Caltime	Somas	Memória
SWmediumEWD.txt	0.000010	0.000009	500	1252
SWtinyDAG.txt	0.000022	0.000020	119	331
SWtinyDG.txt	0.000013	0.000011	36	93

Grafos Orientados

Nome do Ficheiro	Time	Caltime	Somas	Memória
DAG_1.txt	0.000014	0.000017	44	139
DAG_2.txt	0.000012	0.000011	47	145
DAG_3.txt	0.000025	0.000031	46	143

2.3 GraphTopoSortComputeV3

2.3.1 Descrição da Função

Este algoritmo utiliza uma fila para manter o conjunto de vértices que serão sucessivamente adicionados à ordenação topológica. Ele começa identificando os vértices sem arestas de entrada e os coloca na fila. Então, continua removendo vértices da fila e adicionando à ordenação topológica, atualizando a fila conforme necessário.

2.3.2 Métricas da Complexidade

Quanto à complexidade de `GraphTopoSortComputeV3`, esta é dada por:
Complexidade do preenchimento do array `numIncomingEdges`:

$$O(n)$$

Posteriormente, a adição de vértices com in-degree zero à fila:

```
// Adiciona os vértices com in-degree zero à fila
for (unsigned int v = 0; v < topoSort->numVertices; v++)
{
    if (topoSort->numIncomingEdges[v] == 0) {
        QueueEnqueue(queue, (int)v);
```

$$O(n)$$

Já para o loop da fila (formado por um `while` e um `for` iterados) temos:

- Processamento dos vértices (incluindo a decrementação do in-degree):

$$O(m)$$

- Na adição de vértices com in-degree zero à fila, temos $O(1)$ para cada vértice removido da mesma (constante).

Assim, a complexidade total do algoritmo é (onde n é o número de vértices e m é o número total de arestas no grafo):

$$O(n+m)$$

2.3.3 Resultado dos Testes

Grafos

Nome do Ficheiro	Time	Caltime	Somas	Memória
SWmediumEWD.txt	0.000015	0.000013	250	754
SWtinyDAG.txt	0.000019	0.000018	43	153
SWtinyDG.txt	0.000012	0.000010	17	54

Grafos Orientados

Nome do Ficheiro	Time	Caltime	Somas	Memória
DAG_1.txt	0.000014	0.000017	25	87
DAG_2.txt	0.000007	0.000007	27	92
DAG_3.txt	0.000014	0.000017	26	90

Capítulo 3

Conclusão

Após explorar e comparar três algoritmos para ordenação topológica em grafos direcionados, destacamos diferenças significativas em abordagens e desempenho. O 1º algoritmo mostrou-se menos eficiente em termos de tempo e memória devido à sua complexidade. Enquanto isso, a estratégia com um array auxiliar revelou-se mais otimizada, alcançando resultados superiores, como se pode ver no teste realizado. Finalmente, o 3º método que foi o que mais se destacou dos três algoritmos obtendo ainda melhores tempos e resultados mais eficientes

Concluímos que a escolha do algoritmo de ordenação topológica depende das características específicas do problema, como o tamanho do grafo e a necessidade de desempenho. Cada método tem pontos fortes e fracos, sendo essencial considerar não apenas a precisão do resultado, mas também a eficiência computacional. Este estudo deu-nos a conhecer estratégias diversas, ressaltando a importância de ponderar as exigências e restrições do contexto ao selecionar o algoritmo mais adequado.