Algoritmos e Estruturas de Dados
2º Projeto

Universidade de Aveiro

Diogo Carvalho, Tiago Garcia

Algoritmos e Estruturas de Dados
2º Projeto

Dept. de Eletrónica, Telecomunicações e Informática
Universidade de Aveiro

Diogo Carvalho, Tiago Garcia
(113221) diogo.tav.carvalho@ua.pt (114184) tiago.rgarcia@ua.pt

January 5, 2024

Contents

1 Introdução **1**
1.1 Contextualização . . . 1
1.2 Estruturas de Dados . . . 1

2 Análise Formal **2**
2.1 Algoritmo 1 . . . 2
2.1.1 Análise Temporal . . . 2
2.1.2 Análise Espacial . . . 2
2.2 Algoritmo 2 . . . 3
2.2.1 Análise Temporal . . . 3
2.2.2 Análise Espacial . . . 3
2.3 Algoritmo 3 . . . 3
2.3.1 Análise Temporal . . . 3
2.3.2 Análise Espacial . . . 3

3 Análise Experimental **4**
3.1 Método de Experimentação . . . 4
3.2 Resultados . . . 4
3.3 Análise dos Resultados . . . 4

4 Conclusões **5**

Chapter 1

Introdução

1.1 Contextualização

Neste trabalho será desenvolvido um programa que trabalha com grafos. O programa conseguirá criar ficheiros, quer a partir de funções, quer a partir de um ficheiro de texto. Permitirá ainda realizar a ordenação topológica de um grafo recorrendo a um de três algoritmos implementados, sendo eles:

- 2.1 através da cópia do grafo e remoção de vértices sem arestas de entrada;
- 2.2 utilizando uma lista auxiliar com os vértices marcados;
- 2.3 recorrendo a uma fila de prioridade (first in, first out).

Para cada um destes algoritmos será efetuada a respetiva análise de complexidade. Será ainda feita uma análise comparativa entre os três algoritmos, para determinar qual o mais eficiente.

1.2 Estruturas de Dados

Os grafos são representados com a seguinte estrutura:

- int isDigraph - indica se o grafo é direcionado ou não;
- int isComplete - indica se o grafo é completo ou não;
- int isWeighted - indica se o grafo tem pesos nas arestas ou não;
- unsigned int numVertices - número de vértices do grafo;
- unsigned int numEdges - número de arestas do grafo;
- List *verticesList - lista de vértices do grafo;

Os vértices são representados com a seguinte estrutura:

- unsigned int id - identificador do vértice;
- unsigned int inDegree - grau de entrada do vértice;
- unsigned int outDegree - grau de saída do vértice;
- List *edgesList - lista de arestas do vértice;

As arestas são representadas com a seguinte estrutura:

- unsigned int adjVertex - identificador do vértice de destino;
- double weight - peso da aresta;

As listas estão implementadas como listas ordenadas, sendo que a ordenação é feita por ordem crescente do identificador do vértice de destino da aresta. A lista está implementada no ficheiro *SortedList.c* e *SortedList.h*.

Chapter 2

Análise Formal

2.1 Algoritmo 1

2.1.1 Análise Temporal

Esta função tem os seguintes elementos de complexidade temporal:

1. **_create:** Esta função inicializa um array de tamanho V, logo a sua complexidade é $O(V)$;
2. **GraphCopy:** A chamada desta função tem de visitar todos os vértices e arestas do grafo, logo a sua complexidade é $O(V + E)$;
3. **Loop principal, procura de vértices e remoção de arestas:** Estes loops têm de percorrer todos os vértices do grafo, logo a sua complexidade é $O(V)$ (cada um), sendo que a remoção de arestas é $O(E)$ já que tem de percorrer todas as arestas do grafo. Isto resulta numa complexidade de $O(V^2 + E)$ para o loop principal para o pior caso e $O(V + E)$ para o melhor caso (quando não tem de percorrer os vértices todos na procura);
4. **GraphDestroy:** A chama desta função tem de visitar todos os vértices e arestas do grafo, logo a sua complexidade é $O(V + E)$;

Assim, a complexidade total da função é $O(V^2 + E)$ para o pior caso e $O(V + E)$ para o melhor caso.

2.1.2 Análise Espacial

Esta função tem os seguintes elementos de complexidade espacial:

1. **_create:** Esta função aloca memória para a estrutura da ordenação topológica, memória essa que escala consoante o número de vértices do grafo, logo a sua complexidade é $O(V)$;
2. **GraphCopy:** Esta função cria um grafo, cujo custo de memória depende do número de vértices e arestas do grafo original, logo a sua complexidade é $O(V + E)$;
3. **GraphGetAdjacentsTo:** Esta função aloca memória para a lista de vértices adjacentes, cujo custo de memória depende do número de vértices adjacentes, logo a sua complexidade é $O(V)$;

Assim, a complexidade total da função é $O(V + E)$.

2.2 Algoritmo 2

2.2.1 Análise Temporal

Esta função tem os seguintes elementos de complexidade temporal:

- **_create:** Esta função inicializa um array de tamanho V, logo a sua complexidade é $O(V)$;
- **Main loop:** Este loop percorre pelo menos uma vez todos os vértices do grafo, logo a sua complexidade é $O(V)$. Dentro dele, é feita uma Depth-First Search a partir do primeiro vértice não marcado, logo a sua complexidade é $O(V + E)$.

Assim, a complexidade total da função é $O(V + E)$.

2.2.2 Análise Espacial

Esta função tem os seguintes elementos de complexidade espacial:

- **_create:** Esta função aloca memória cujo custo depende do número de vértices do grafo, logo a sua complexidade é $O(V)$;
- **GraphGetAdjacentsTo:** Esta função aloca memória para a lista de vértices adjacentes, cujo custo de memória depende do número de vértices adjacentes, logo a sua complexidade é $O(V)$;

Assim, a complexidade total da função é $O(V)$.

2.3 Algoritmo 3

2.3.1 Análise Temporal

Esta função tem os seguintes elementos de complexidade temporal:

- **_create:** Esta função inicializa um array de tamanho V, logo a sua complexidade é $O(V)$;
- **Queue initialization:** Esta função cria uma fila e adiciona todos os vértices sem arestas de entrada, logo a sua complexidade é $O(V)$ pois tem de percorrer todos os vértices do grafo;
- **Main loop:** Este loop corre enquanto a fila não estiver vazia cuja complexidade é $O(V)$. Dentro dele, é feita a atualização das arestas de entrada dos vértices adjacentes ao vértice que está a ser processado, logo a cuja complexidade é $O(E)$. Por fim, é feita a remoção do vértice da fila, cuja complexidade é $O(1)$. Isto resulta numa complexidade de $O(V + E)$ para o loop principal;

Assim, a complexidade total da função é $O(V + E)$.

2.3.2 Análise Espacial

Esta função tem os seguintes elementos de complexidade espacial:

- **_create:** Esta função aloca memória cujo custo depende do número de vértices do grafo, logo a sua complexidade é $O(V)$;
- **QueueCreate:** Esta função aloca memória para a fila, cujo custo de memória depende do número de vértices do grafo sendo que no pior caso a sua complexidade é $O(V)$;
- **GraphGetAdjacentsTo:** Esta função aloca memória para a lista de vértices adjacentes, cujo custo de memória depende do número de vértices adjacentes, logo a sua complexidade é $O(V)$;

Assim, a complexidade total da função é $O(V)$.

Chapter 3

Análise Experimental

3.1 Método de Experimentação

Os resultados foram obtidos a partir da utilização do módulo de instrumentação fornecido pelos professores. Este módulo permite medir o tempo de execução de uma função, bem como o número de instruções executadas.

Foi utilizado um computador com as seguintes especificações:

- **CPU:** 11th Gen Intel i7-1165G7 (8) @ 4.700GHz;
- **GPU:** Intel TigerLake-LP GT2 [Iris Xe Graphics];
- **RAM:** 16GB;
- **Kernel:** 6.6.9-zen1-1-zen (Arch Linux x86_64);

3.2 Resultados

Para a experimentação foram usados 5 grafos diferentes as seguintes dimensões:

- **Grafo 1:** 7 vértices e 9 arestas;
- **Grafo 2:** 7 vértices e 12 arestas;
- **Grafo 3:** 7 vértices e 11 arestas;
- **Grafo 4:** 13 vértices e 15 arestas;
- **Grafo 5:** 15 vértices e 25 arestas;

Grafo 1		**Grafo 2**		**Grafo 3**		**Grafo 4**		**Grafo 5**	
Versão	**Caltime**	**Versão**	**Caltime**	**Versão**	**Caltime**	**Versão**	**Caltime**	**Versão**	**Caltime**
V1	0.010 ms	V1	0.006 ms	V1	0.007 ms	V1	0.011 ms	V1	0.014 ms
V2	0.001 ms	V2	0.002 ms	V2	0.002 ms	V2	0.002 ms	V2	0.003 ms
V3	0.001 ms	V3	0.002 ms	V3	0.001 ms	V3	0.002 ms	V3	0.002 ms

Table 3.1: Resultados obtidos

3.3 Análise dos Resultados

Como se pode verificar pela tabela anterior, a versão 1 é a mais lenta, isto era esperado visto que esta versão precisa de criar uma cópia do grafo para chegar ao resultado. Por mais que as versões 2 e 3 sejam bastante semelhantes, a versão 3 é ligeiramente mais otimizada, o que se nota especialmente se escalarmos o número de vértices e arestas.

Chapter 4

Conclusões

Em conclusão, o trabalho desenvolvido permitiu-nos aprofundar os nossos na linguagem C, bem como aprofundar os nossos conhecimentos sobre as estruturas de dados *SortedList* e *Graph*.

A implementação dos algoritmos de *Topological Sort* permitiu-nos aprofundar os nossos conhecimentos sobre os 3 algoritmos implementados, bem como aprofundar os nossos conhecimentos sobre a estrutura de dados *Graph*.

Permitiu-nos ainda desenvolver a nossa capacidade de análise de otimização de código, bem como a importância de escolher o algoritmo mais adequado para cada situação.

Colaboração dos autores

Ambos os autores contribuíram para a realização deste trabalho, para diferentes partes do mesmo, embora com um nível de contribuição diferente. Consideramos que a percentagem de contribuição justa é de 35% para o Diogo Carvalho e 65% para o Tiago Garcia.