universidade de aveiro
deti departamento de eletrónica, telecomunicações e informática

2º Trabalho - O TAD GRAPH

Algoritmos e Estruturas de Dados

Ano Letivo: 2023/2024

Prof. Joaquim João Estrela Ribeiro Silvestre Madeira

Prof. Mário Antunes

Trabalho realizado por:

Rodrigo Abreu, Nº113626

João Neto, Nº113482

Índice

Introdução

Este projeto visa desenvolver e testar o Tipo Abstrato de Dados (TAD) *Graph,* que permite a criação e manipulação de grafos. Estes grafos, são constituídos pela sua lista de vértices e a respetiva lista de adjacências, que são definidas através do tipo de dados genérico **SORTED-LIST**.

O ficheiro *Graph.c* fornece apenas operações básicas sobre grafos, enquanto que o ficheiro *GraphTopologicalSorting.c*, apresenta 3 funções com, respetivamente, 3 algoritmos de ordenação topológica dos vértices de um grafo passado como argumento à função, isto que só é possível implementar para digrafos acíclicos. Os algoritmos são:

- **Algoritmo 1 -** usa uma cópia do grafo orientado e efetua sucessivos apagamentos dos arcos emergentes de vértices que não tenham arcos incidentes.
- **Algoritmo 2 -** usa um array auxiliar (um dos campos do registo) com o inDegree de cada vértice para sucessivamente procurar o próximo vértice a juntar à ordenação topológica.
- **Algoritmo 3 -** usa uma fila para manter o conjunto dos vértices que irão ser sucessivamente adicionados à ordenação topológica.

Ao longo do relatório, vamos analisar a complexidade temporal de cada algoritmo e tirar conclusões acerca da eficiência computacional de cada um.

Algoritmos

Primeiro Algoritmo

Para o primeiro algoritmo, criamos uma cópia do grafo e depois enquanto for possível, selecionamos um vértice sem arestas incidentes, guardamos o seu ID (na sequência) e “apagamos” o vértice da cópia do grafo e as suas arestas incidentes.

Aqui como não tínhamos nenhuma função em *Graph.c* que permitisse remover o vértice, optámos por usar o **marked**, para marcar o vértice eliminado.

Este algoritmo demora mais, pois ele mexe num grafo diretamente, invocando GraphCopy e *GraphRemoveEdge,* o que torna o processo mais demorado.

Para o primeiro e segundo algoritmo o *Best Case* é o grafo ser um ciclo.

```
Digraph
Max Out-Degree = 1
Vertices = 10 | Edges = 10
 0 ->    1
 1 ->    2
 2 ->    3
 3 ->    4
 4 ->    5
 5 ->    6
 6 ->    7
 7 ->    8
 8 ->    9
 9 ->    0
---
```

Fig 1.1 - Exemplo grafo *Best Case*

```
FILE: MEUS_GRAFOS/bestcase10.txt
SORT: TopoSortV1
#           time         caltime          memops      iterations
       0.000010        0.000016              20              10
RESULT:  *** The topological sorting could not be computed!! ***
--------
FILE: MEUS_GRAFOS/bestcase10.txt
SORT: TopoSortV2
#           time         caltime          memops      iterations
       0.000001        0.000001              20              10
RESULT:  *** The topological sorting could not be computed!! ***
--------
FILE: MEUS_GRAFOS/bestcase10.txt
SORT: TopoSortV3
#           time         caltime          memops      iterations
       0.000001        0.000002              20              10
RESULT:  *** The topological sorting could not be computed!! ***
```

Fig 1.2 - Testes *Best Case*

Para o *Worst Case,* este verifica-se quando temos um grafo do género do seguinte, onde 1 um vértice tem como adjacentes todos os outros vértices, e esse vértice tem como adjacentes todos os outros vértices exceto o anterior, e assim sucessivamente até chegar ao último vértice que terá *outDegree* 0.

```
Digraph
Max Out-Degree = 9
Vertices = 10 | Edges = 45
 0 ->
 1 ->    0
 2 ->    0    1
 3 ->    0    1    2
 4 ->    0    1    2    3
 5 ->    0    1    2    3    4
 6 ->    0    1    2    3    4    5
 7 ->    0    1    2    3    4    5    6
 8 ->    0    1    2    3    4    5    6    7
 9 ->    0    1    2    3    4    5    6    7    8
```

Fig 1.3 - Exemplo grafo *Worst Case*

Pode ser calculado da seguinte forma:

$$n + \sum_{k=1}^{n-1} ((n-1)+(n-k)) \quad = \quad n + \sum_{k=1}^{n-1} 2n - \sum_{k=1}^{n-1} k + \sum_{k=1}^{n-1} 1 \quad =$$

$$n + 2n*(n+1) - \frac{(n-1)*((n-1)+1)}{2} - (n-1) \quad = \quad 2n^2 - 2n - \frac{n^2}{2} + \frac{n}{2} + 1$$

A ordem de complexidade é $O(n^2)$, sendo n, o número de vértices.

```
FILE: MEUS_GRAFOS/worstcase10.txt
SORT: TopoSortV1
#          time         caltime          memops      iterations
       0.000020        0.000030             130             136
RESULT: Topological Sorting - Vertex indices:
9 8 7 6 5 4 3 2 1 0
--------
FILE: MEUS_GRAFOS/worstcase10.txt
SORT: TopoSortV2
#          time         caltime          memops      iterations
       0.000003        0.000005             175             136
RESULT: Topological Sorting - Vertex indices:
9 8 7 6 5 4 3 2 1 0
--------
FILE: MEUS_GRAFOS/worstcase10.txt
SORT: TopoSortV3
#          time         caltime          memops      iterations
       0.000003        0.000004             130              55
RESULT: Topological Sorting - Vertex indices:
9 8 7 6 5 4 3 2 1 0
```

Fig 1.4 - Testes grafo *Best Case*

Segundo algoritmo

No segundo algoritmo, guardamos num array auxiliar (**numIncomingEdges**) o inDegree de cada vértice e depois enquanto for possível, selecionamos um vértice *v* cujo o inDegree seja igual a 0 e ainda não esteja marcado, guardamos o seu ID (na sequência) e marcámo-lo como pertencente à sequência, após isso para cada vértice *w* adjacente a *v*, subtraímos 1 no seu *inDegree* no array auxiliar.

Comparado ao primeiro, embora dê o mesmo número de iterações, este demora menos tempo pois usa arrays auxiliares para controlar o *inDegree* dos vértices.

Para este algoritmo, o *Best Case* e o *Worst Case* são iguais ao primeiro algoritmo, mas este demonstra ser mais eficiente pois não efetua as especificações mencionadas no algoritmo anterior.

Terceiro algoritmo

No terceiro algoritmo, guardamos num array auxiliar (***numIncomingEdges***) o *inDegree* de cada vértice e criamos uma FILA vazia, e nessa FILA inserimos os vértices *v* cujo *inDegree* no array auxiliar for igual a 0. Depois, enquanto a FILA não for vazia, vamos buscar o próximo vértice *v* na FILA e retiramo-lo dela, guardamos o seu ID (na sequência) e depois para cada vértice *w* adjacente a *v*, subtraímos 1 no seu *inDegree* no array auxiliar, e se o *inDegree* do vértice *w* no array auxiliar for igual a 0 adicionamo-lo à FILA.

Ao analisar a complexidade deste algoritmo, reparámos que estávamos perante um cenário *Best Case* e *Worst Case.*

O *Best Case* verifica-se quando nenhum vértice do grafo tem *inDegree* igual a 0. Quando isto acontece o segundo loop não é executado, logo a complexidade deste algoritmo pode ser calculada da seguinte forma:

$$\sum_{i=1}^{v} 1 = v$$

A ordem de complexidade é $O(v)$, sendo v, o número de vértices.

O *Worst Case* é o mesmo dos algoritmos anteriores e neste caso, pode ser calculado da seguinte forma:

$$\sum_{i=1}^{v} 1 + \sum_{i=1}^{v} (v-i) = v + v^2 - \frac{v(v+1)}{2} = \frac{v+v^2}{2}$$

A ordem de complexidade é $O(v^2)$, sendo v, o número de vértices.

Gráficos

NºVértices	BestCase	WorstCase
2	2	4
10	10	136
20	20	571
30	30	1306
40	40	2341

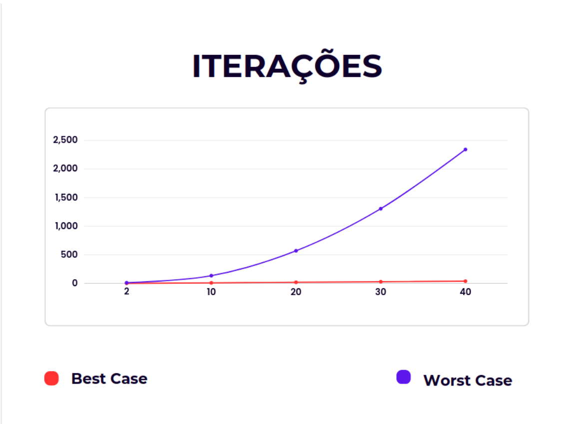

Fig 2.1 - Iterações para o *Best Case* e o *Worst Case* do primeiro algoritmo e do segundo algoritmo em função do número de vértices

NºVértices	BestCase	WorstCase
2	2	3
10	10	55
20	20	201
30	30	465
40	40	820

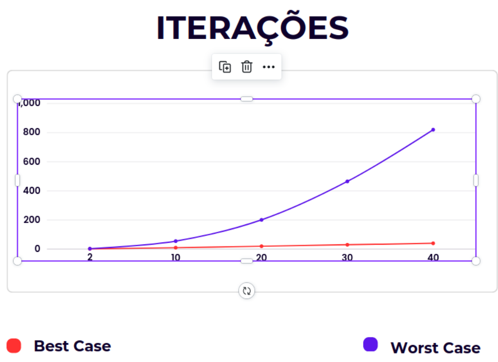

Fig 2.2 - Iterações para o *Best Case* e o *Worst Case* do terceiro algoritmo em função do número de vértices

NºVértices	AG1	AG2	AG3
2	0.000005	0.000001	0.000003
10	0.000037	0.000005	0.000004
20	0.000154	0.000010	0.000009
30	0.000450	0.000018	0.000016
40	0.000972	0.000057	0.000026

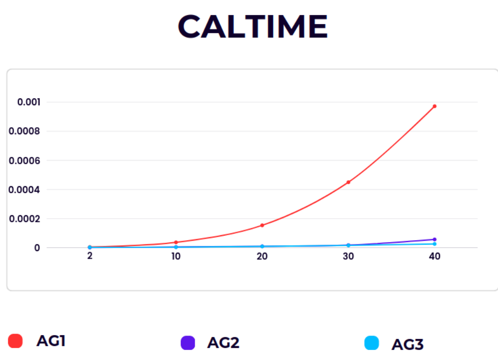

Fig 2.3 - Caltime para o *Worst Case* de todos os algoritmo em função do número de vértices

Conclusão

Depois de analisarmos formalmente os três algoritmos, verificámos que os resultados obtidos correspondiam com os resultados experimentais. E desta forma, podemos concluir com certeza, que o terceiro algoritmo é o mais eficiente.