

Ordenação Topológica com Grafos

Henrique Teixeira - 114588

Gabriel Silva - 113786

Índice

Índice.....	1
Introdução.....	1
TAD Graph.....	2
GraphFromFile.....	2
GraphCheckInvariants.....	2
GraphCopy.....	2
GraphRemoveEdge.....	2
Ordenação Topológica.....	3
Primeiro Algoritmo - Cópia do Grafo.....	3
Análise da complexidade do Pior Caso.....	3
Segundo Algoritmo - Array auxiliar.....	3
Análise da complexidade do Pior Caso.....	3
Terceiro Algoritmo - Manter o conjunto de candidatos.....	4
Análise da complexidade do Pior Caso.....	4
Análise dos resultados obtidos.....	5
Iterações.....	5
Comparação entre algoritmos.....	5
Conclusão.....	6

Introdução

O objetivo deste trabalho foi desenvolver e testar o Tipo Abstrato de Dados (TAD) “Graph”, destinado a representar e manipular grafos e grafos orientados, com ou sem pesos associados às suas arestas. Este TAD utiliza uma estrutura de dados baseada em listas de vértices e as suas respectivas listas de adjacências, implementadas através do tipo de dados genérico *SortedList*. A implementação do TAD “Graph” é essencial para a operação básica sobre grafos, além de ser a base para 3 algoritmos mais complexos implementados em módulos autónomos desenvolvidos. Os principais desafios deste trabalho incluem:

1. Completar o desenvolvimento do [TAD GRAPH](#), com foco nas funções com seus objetivos:
 - *GraphFromFile* - A leitura de informações de grafos a partir de arquivos de texto;
 - *GraphCopy* - A criação de cópias de grafos;
 - *GraphRemoveEdge* - A remoção de arestas em grafos orientado;
 - *GraphCheckInvariants* - A verificação de invariantes;
2. Desenvolver e testar três algoritmos específicos para determinar a ordem topológica de vértices em um grafo orientado, se possível. Estes algoritmos são:
 - 1º - Utiliza uma cópia do grafo orientado e nele realiza sucessivas eliminações de arcos emergentes de vértices sem arcos incidentes.
 - 2º - Não necessita da cópia do grafo, em vez disso usa um array auxiliar para sucessivamente procurar o próximo vértice a juntar à ordenação topológica.
 - 3º - Usa uma fila para manter o conjunto dos vértices que irão ser sucessivamente adicionados à ordenação topológica.

Além disso, caracterizamos também a complexidade algorítmica das nossas soluções de ordenação topológica.

TAD Graph

GraphFromFile

Esta função verifica se o grafo é *Weighted* ou não, de modo a fazer o preenchimento das arestas corretamente, para além disso, verifica também a existência de lacetes. Chegamos a esta solução:

```
int isDigraph;
int isWeighted;
int numVertices;
int numEdges;
Graph* g;
fscanf(f, "%d", &isDigraph);
fscanf(f, "%d", &isWeighted);
fscanf(f, "%d", &numVertices);
fscanf(f, "%d", &numEdges);
if (isWeighted == 0) {
    g = GraphCreate(numVertices, isDigraph, isWeighted);
    for (int i = 0; i < numEdges; i++) {
        int v, w;
        fscanf(f, "%d %d", &v, &w);
        if (v == w) {
            continue;
        }
        GraphAddEdge(g, v, w);
    }
}
```

No caso de o grafo ser *Weighted* é apenas mudado:

```
double weight;
fscanf(f, "%d %d %lf", &v, &w, &weight);
int intWeight = (int)(weight * 100);
GraphAddWeightedEdge(g, v, w, intWeight);
```

GraphCheckInvariants

Esta função verifica se o grafo é *Digraph* ou não, caso seja, é percorrida a lista de vértices e são calculadas as invariâncias de cada um, sendo depois comparadas aos seus parâmetros da *Struct*.

```
if (g->isDigraph){
    unsigned int inDegree = 0;
    unsigned int outDegree = 0;

    for (unsigned int i = 0; i < g->numVertices; i++) {
        ListMove(g->verticesList, i);
        struct _Vertex* vertex=ListGetCurrentItem(g->verticesList);
        if (vertex->id != i)
            return 0;
        inDegree += vertex->inDegree;
        outDegree += vertex->outDegree;
    }
    if (inDegree != outDegree)
        return 0;
    if (inDegree != g->numEdges)
        return 0;
    if (outDegree != g->numEdges)
        return 0;
}
```

Caso não seja é simplesmente retornado:

```
return (ListGetSize(g->verticesList) == g->numVertices) &&
(g->isComplete == 1 || g->isComplete == 0) &&
(g->isWeighted == 1 || g->isWeighted == 0) &&
(g->isDigraph == 1 || g->isDigraph == 0);
```

GraphCopy

Esta função verifica se o grafo é *Complete*, se sim:

```
if (g->isComplete){
    return GraphCreateComplete(g->numVertices, g->isDigraph);
}
```

Caso não o seja, percorre as listas de vértices e suas respectivas arestas e acrescenta cada uma ao grafo.

```
Graph* g2=GraphCreate(g->numVertices, g->isDigraph, ->isWeighted);
for (unsigned int i = 0; i < g->numVertices; i++){
    ListMove(g->verticesList, i);
    struct _Vertex* v = ListGetCurrentItem(g->verticesList);
    List* edges = v->edgesList;
    ListMoveToHead(edges);
    for (unsigned int j = 0; j < ListGetSize(edges);
        ListMoveToNext(edges), j++){
        struct _Edge* e = ListGetCurrentItem(edges);
        if (g->isWeighted)
            GraphAddWeightedEdge(g2, i, e->adjVertex, e->weight);
        else
            GraphAddEdge(g2, i, e->adjVertex);
    }
}
```

GraphRemoveEdge

Esta função começa por encontrar o vértice escolhido e obter as suas arestas,

```
List* edges;
int count = 0;
// ir para o vértice v
ListMove(g->verticesList, v);
struct _Vertex* vertex_v =
ListGetCurrentItem(g->verticesList);
edges = vertex_v->edgesList;
```

Após isso, vai para o início da lista de arestas, percorrendo-a até encontrar a aresta pretendida,

```
ListMoveToHead(edges);
// percorrer a lista de arestas
for (unsigned int i = 0; i < ListGetSize(edges);
    ListMoveToNext(edges), i++){
    struct _Edge* e = ListGetCurrentItem(edges);
    if (e->adjVertex == w) {
        // remove da lista e decrementa
        ListRemoveCurrent(edges);
        vertex_v->outDegree --;
        count++;
        break;
    }
}
```

No caso da aresta não existir é retornado 0,

```
if (count == 0) return 0;
```

Para atualizar o vértice W, caso o grafo não seja orientado, o processo anterior é repetido apenas sem decrementar o *outDegree*, sendo este feito abaixo,

```
if (g->isDigraph)
    vertex_w->inDegree -= count;
else
    vertex_w->outDegree -= count;
g->numEdges -= count;
g->isComplete = 0;
```

Todas estas funções terminam com `assert(GraphCheckInvariants(g))`; e um valor de retorno adequado

Order

A ordenação
topológica
vai haver p

Primeira

Este primeiro
os seus vért

```

count++;
while (nVert:
O(n)
    for (unsigned
O(n)
        if (Graph
            // Remov
            if (Graph
                unsigned
                unsigned
                for (
                    Graph
                }
                topoSort
                topoSort
                break;
            }
        }
        if (count ==
            break;
    }
    - -
O(n^2)

```

A resolução
caso o grafo
número de
A análise en

Segundo

Neste segu
cada vértice

```

for (unsigned
->O(n)
    topoSort->nu
Após isto, v
esteja mark
while (verific
->O(n)
    for (unsign
->O(n)
        if (topo
            topoSc
            // ins
            topoSc
O último pa
unsigned int
for (unsigned
->O(m)
    topoSort->nu
- - -

```

A lógica pa

A análise em

Terceiro Algoritmo - Manter o conjunto de candidatos

O terceiro algoritmo é implementado usando Filas (*Queues*). Repetimos inicialmente o passo do segundo algoritmo de fazer um *array* com todos os *inDegree* dos vértices, isto irá ser usado para criar uma fila com todos os vértices com *numIncomingEdges* == 0.

Análise da complexidade do Pior Caso

```

for (unsigned int v = 0; v < topoSort->numVertices; v++) {
    if (topoSort->numIncomingEdges[v] == 0)
        QueueEnqueue(queue, v);
}
-->O(n)

```

Enquanto esta fila não for vazia, retira-se o primeiro vértice dela e adiciona-se à sequência.

```

while ( !QueueIsEmpty(queue) ) {
    // retirar próximo vértice da fila
    unsigned int v = QueueDequeue(queue);
    // inserir v no topoSort
    topoSort->vertexSequence[verticesInSequence++] = v;
}
-->O(n)

```

Por último, para cada vértice adjacente ao vértice retirado da lista é decrementado o *outDegree*. Se isto significar que o *outDegree* de um vértice se vai tornar 0, este vértice irá ser adicionado à fila.

```

// para cada vértice w adjacente a v
unsigned int* adj = GraphGetAdjacentsTo(g, v);
for (unsigned int w = 1; w < GraphGetVertexOutDegree(g, v) + 1; w++) {
    // decrementar numEdgesPerVertex[w]
    topoSort->numIncomingEdges[adj[w]]--;
    // se numEdgesPer Vertex[w] == 0
    if (topoSort->numIncomingEdges[adj[w]] == 0)
        // inserir w na fila
        QueueEnqueue(queue, adj[w]);
}
}

```

--> Complexidade Total = $O(n+n+n+m) = O(n+m)$

NOTA: Todos estes algoritmos terminam com uma verificação sobre a validação do *sorting*:

```

if (verticesInSequence == nVertices) {topoSort->validResult = 1;}

```

A análise entre a complexidade obtida no estudo desta função e os valores práticos estão no [Gráfico 3](#).

Análise dos resultados obtidos

Para avaliar a complexidade dos algoritmos, comparamos vários elementos obtidos ao executar o nosso código e o valor por nós calculado teoricamente.

Iterações

A fim de ver a complexidade comparamos as iterações obtidas com os valores da complexidade teórica (obtida na explicação de cada algoritmo explicado acima).

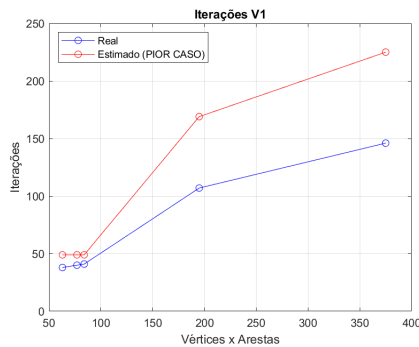


Gráfico 1

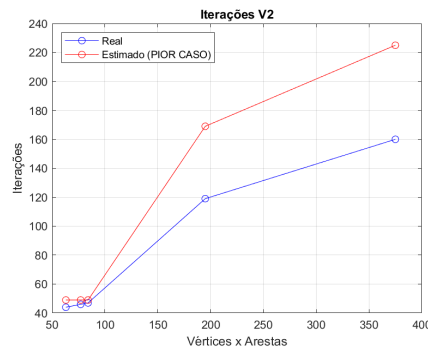


Gráfico 2

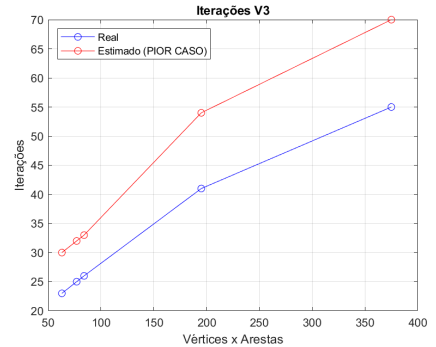
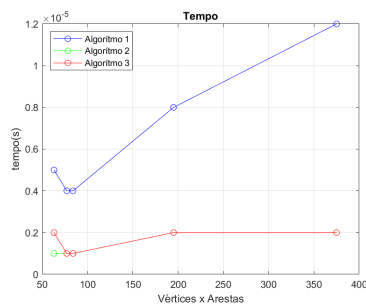


Gráfico 3

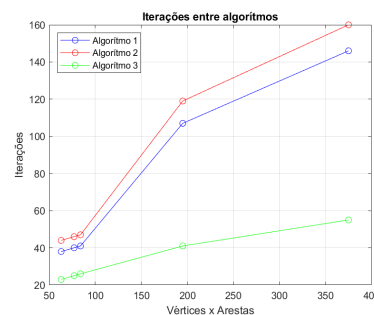
Podemos observar uma leve discrepância entre os [resultados teóricos](#) e os práticos. Isto acontece devido ao valor teórico calculado ter sido o pior caso. O pior caso é apenas obtido quando o grafo não é computável, o que não aconteceu nos nossos testes, visto que decidimos apresentar apenas resultados de dígrafos acíclicos.

Comparação entre algoritmos

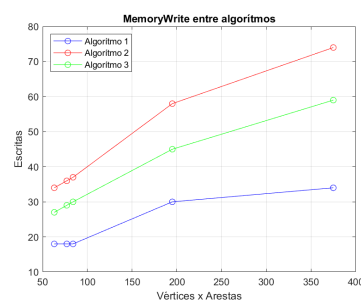
Decidimos também apresentar vários elementos de comparação entre os 3 algoritmos



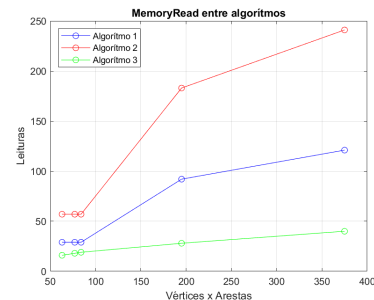
Tempo de execução



Iterações



Veze de modificação de *data structures*



Veze de acesso a *data structures*

Conclusão

Os resultados obtidos neste projeto vão de encontro com as expectativas iniciais. A análise feita demonstra que os algoritmos implementados se adequam aos requisitos de um DAG (Directed Acyclic Graph), respeitando a aciclicidade necessária para a ordenação topológica. Através dos testes e estudo realizados, verificamos que a complexidade dos algoritmos está alinhada com os valores teóricos, tendo a complexidade esperada.

Para ainda mais confirmar a construção do nosso código fizemos ainda múltiplas verificações com o *valgrind*, confirmando que não existe nenhuma fuga de memória no uso das funções e algoritmos por nós desenvolvidos. A mensagem para todos os grafos testados foi a mesma:

All heap blocks were freed -- no leaks are possible

Este projeto proporcionou um aprofundamento significativo no nosso entendimento dos conceitos de grafos e algoritmos em C, além de reforçar habilidades de programação e resolução de problemas. Além disso, contribuiu para melhorar nosso trabalho em equipa e fortaleceu a nossa capacidade de solucionar problemas complexos através de algoritmos, um desafio fundamental neste projeto.