Projeto 2 - AED

Análise da complexidade:
Algoritmos de Ordenação Topológica

Jorge Domingues nº113278

Guilherme Santos nº113893

---

1. Introdução

Este relatório consiste na realização de uma análise aprofundada de três funções essenciais: `GraphTopoSortComputeV1`, `GraphTopoSortComputeV2` e `GraphTopoSortComputeV3`. Todas as funções estão implementadas no ficheiro *GraphTopologicalSorting.c* e desempenham papéis cruciais na criação de uma ordenação topológica, à qual visa construir um grafo acíclico, de modo a garantir que todas as arestas apontem na mesma direção. A análise abordará a implementação destas funções assim como as suas respetivas complexidades temporais com o uso do contador `ITERATIONS` que é incrementado em todas as estruturas que nós considerámos relevantes para as diferenças de tempo de execução entre os vários algoritmos.

2. GraphTopoSortComputeV1

2.1. Algoritmos e complexidade

Nesta etapa, vamos abordar o algoritmo que utiliza como auxílio uma cópia do grafo orientado original. Este algoritmo realiza sucessivos apagamentos dos arcos emergentes de vértices que não têm arcos incidentes.

Para calcularmos a complexidade deste algoritmo, precisamos primeiro de ver a complexidade do algoritmo da criação da cópia do grafo orientado original, criado na função `GraphCopy` no ficheiro `Graph.c`, pois é uma fase que exige sempre algum processamento e tempo para o realizar. Primeiramente, o algoritmo `GraphCopy` copia todas as arestas do grafo original, ou seja, como cada vez que é copiada uma aresta, as iterações são incrementadas, o número das iterações no final de `GraphCopy` é simplesmente o número de *edges* do grafo.

Assim, o número de iterações de `GraphCopy` pode ser dado pela seguinte expressão, onde `V` é o número de vértices, `ei`, o número de *edges* de cada vértice `i`, `E` o número de *edges* do grafo e, sabendo que, a soma das arestas de cada vértice é igual ao número das arestas totais pois trata-se de um grafo orientado:

$$\sum_{i=0}^{V-1}\sum_{j=0}^{ei-1} 1 = \sum_{i=1}^{V}\sum_{j=1}^{ei} 1 = \sum_{i=1}^{V} ei = E$$

```c
Graph* GraphCopy(const Graph *g) {
  ...
  // create a new graph with the same characteristics
  Graph *copy = GraphCreate(g->numVertices, g->isDigraph, g->isWeighted);
  assert(copy != NULL);
  // copy the edges from the original graph to the new graph
  List *vertices = g->verticesList;
  ListMoveToHead(vertices);
  unsigned int i = 0;
  for (; i < g->numVertices; ListMoveToNext(vertices), i++)
  {
    ...
    unsigned int j = 0;
    for (; j < ListGetSize(edges); ListMoveToNext(edges), j++)
    {
      ITERATIONS++; // count iterations
      ...
    }
  }
  return copy;
}
```

Voltando á função `GraphTopoSortComputeV1`, o seu número de iterações vai ser dado pelo número calculado anteriormente adicionado ao número que é resultado do `while` e dos `for's`. Como o `while` obriga o `for` a ser executado `V` vezes por causa da variável `found`, mesmo que o `for` dê `break` quando encontra um vértice candidato, ele vai ter de passar por todos os vértices e, esta situação do `for` mais a obrigatoriedade que o `while` lhe implica, faz com que as iterações tenham uma soma cumulativa dos vértices e, como por cada vértice, são lhe removidas as arestas e, como em cada remoção é incrementada uma iteração, também estará presente no número final, o número total das arestas, como já explicado anteriormente. Assim, a complexidade temporal deste algoritmo é dado pela seguinte expressão:

$$E + \sum_{i=0}^{V-1} (i + \sum_{j=1}^{ei} 1) = E + \sum_{i=1}^{V} (i + ei) = E + \sum_{i=1}^{v} i + \sum_{i=1}^{v} ei = E + \frac{V(V+1)}{2} + E =$$

$$= \frac{V^2}{2} + \frac{V}{2} + 2E \in O(V^2 + E)$$

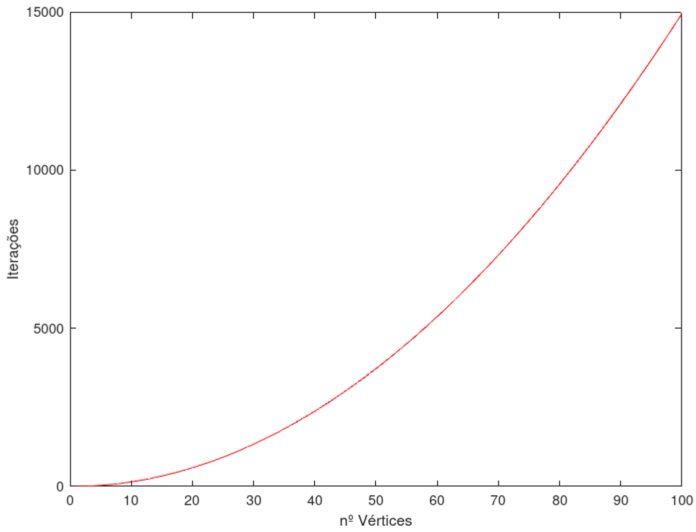
```
GraphTopoSort* GraphTopoSortComputeV1(Graph *g) {
  ...
  Graph *copy = GraphCopy(g); /// 1 - Create a copy of the graph
  unsigned int count = 0;
  int found; // aux variable to check if a vertex without incoming edges was found

  while (count < GraphGetNumVertices(copy))
  {
    found = 0; // reset found
    for (unsigned int i = 0; i < GraphGetNumVertices(copy); i++)
    {
      ITERATIONS++; // count iterations
      // find vertex without incoming edges
      if (GraphGetVertexInDegree(copy, i) == 0 && topoSort->marked[i] == 0)
      {
        found = 1; // found a vertex without incoming edges
        ...
        for (unsigned int j = 1; j <= adjacents[0]; j++)
        {
          ITERATIONS++; // count iterations
          GraphRemoveEdge(copy, i, adjacents[j]);
        }
        free(adjacents);
        break;
      }
    }
    // if no vertex without incoming edges was found, break the loop because it is a cycle
    if (found == 0) break;
  }
  ...
  GraphDestroy(&copy);
  return topoSort;
}
```

2.2. `GraphTopoSortComputeV1` - Dados Experimentais

```
SORT: TopoSortV1
#           time          caltime       iterations
        0.000006         0.000010               35
RESULT: Topological Sorting - Vertex indices:
0 1 2 3 4
```

```
SORT: TopoSortV1
#           time          caltime       iterations
        0.000061         0.000102              590
RESULT: Topological Sorting - Vertex indices:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```



3. GraphTopoSortComputeV2

3.1. Algoritmos e complexidade

Para esta etapa, vamos abordar o algoritmo que utiliza como auxílio um `array` para sucessivamente procurar o próximo vértice a juntar à ordenação topológica. O cálculo da complexidade do segundo algoritmo é o mesmo do primeiro porém como não precisamos de usar uma cópia do grafo orientado original para nos auxiliar na resolução da implementação, apenas contabilizamos as iterações dentro das estruturas `while` e `for's`. Assim, a complexidade temporal deste algoritmo é dada pela seguinte expressão:

$$\sum_{i=0}^{V-1}(i+\sum_{j=1}^{ei}1)=\sum_{i=1}^{V}(i+ei)=\sum_{i=1}^{v}i+\sum_{i=1}^{v}ei=\frac{V(V+1)}{2}+E=\frac{V^2}{2}+\frac{V}{2}+E\in O(V^2+E)$$

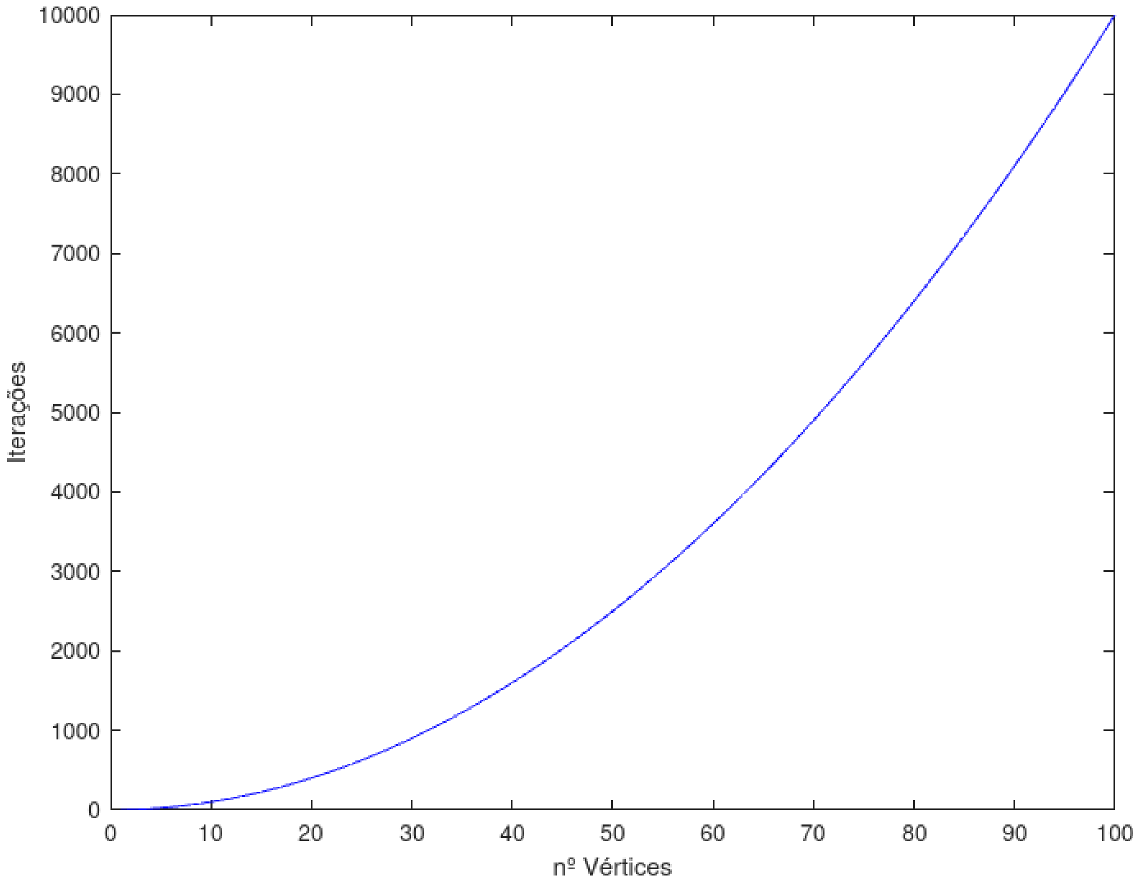
```
GraphTopoSort* GraphTopoSortComputeV2(Graph *g) {
  ...
  unsigned int count = 0;
  int found; // aux variable to check if a vertex without incoming edges was found

  while (count < GraphGetNumVertices(g))
  {
    found = 0; // reset found
    for (unsigned int i = 0; i < GraphGetNumVertices(g); i++)
    {
      ITERATIONS++; // count iterations
      // find vertex without incoming edges
      if (topoSort->numIncomingEdges[i] == 0 && topoSort->marked[i] == 0)
      {
        found = 1; // found a vertex without incoming edges
        ...
        for (unsigned int j = 1; j <= adjacents[0]; j++)
        {
          ITERATIONS++; // count iterations
          unsigned int w = adjacents[j]; // w is an adjacent of v
          topoSort->numIncomingEdges[w]--;  // remove edge (i,w)
        }
        free(adjacents);
        break;
      }
    }
    // if no vertex without incoming edges was found, break the loop because it is a cycle
    if (found == 0) break;
  }
  ...
  return topoSort;
}
```

3.2. `GraphTopoSortComputeV2` - Dados Experimentais

```
SORT: TopoSortV2
#            time          caltime      iterations
       0.000001         0.000002              25
RESULT: Topological Sorting - Vertex indices:
0 1 2 3 4
```

```
SORT: TopoSortV2
#            time          caltime      iterations
       0.000005         0.000009             400
RESULT: Topological Sorting - Vertex indices:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```



4. GraphTopoSortComputeV3

4.1. Algoritmos e complexidade

Este algoritmo aborda o problema de uma maneira diferente, usando uma `Queue`, o que altera a complexidade temporal significativamente. Nesta função, primeiramente, a `Queue` é inicializada com os vértices que não têm *incoming edges*, no entanto, esta inicialização acontece num `for` que depende do número de vértices, logo é contada para as iterações. Por fim, sempre que o `while` faz uma iteração, a mesma também conta devido á sua relevância e, caso hajam vértices adjacentes, existe outro `for`, onde as suas iterações não podem ser descartadas. No que toca á fórmula para calcular este número de iterações, difere significativamente das funções anteriores pois, por cada vértice candidato a iteração é incrementada uma única vez em de cumulativamente. Assim, a expressão para a complexidade temporal deste algoritmo é a seguinte:

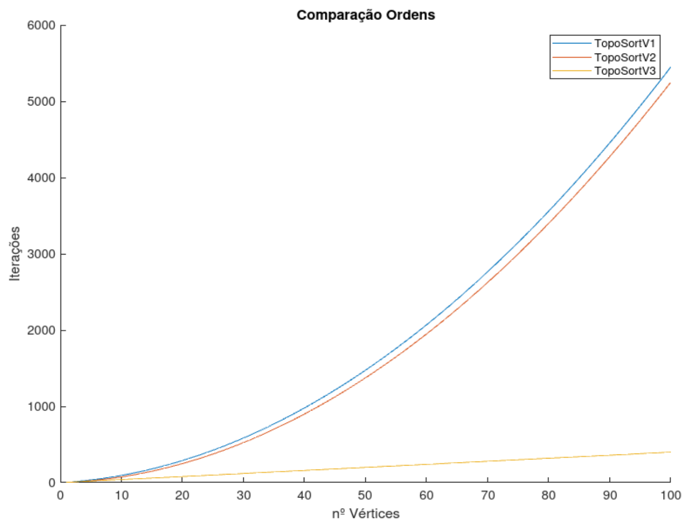
$$\sum_{i=0}^{V-1} 1 + \sum_{i=0}^{V-1} (1 + \sum_{j=1}^{ei} 1) = \sum_{i=1}^{V} 1 + \sum_{i=1}^{V} (1 + ei) = V + \sum_{i=1}^{v} 1 + \sum_{i=1}^{v} ei = 2V + E \in O(V + E)$$

```
GraphTopoSort GraphTopoSortComputeV3(Graph *g) {
  ...
  Queue *q = QueueCreate(GraphGetNumVertices(g));
  for (unsigned int i = 0; i < GraphGetNumVertices(g); i++) {
    ITERATIONS++; // count iterations
    if (topoSort->numIncomingEdges[i] == 0) QueueEnqueue(q, i);
  }
  unsigned int count = 0;
  while (!QueueIsEmpty(q)) {
    ITERATIONS++; // count iterations
    unsigned int v = QueueDequeue(q); // v is a vertex without incoming edges
    ...
    for (unsigned int j = 1; j <= adjacents[0]; j++) {
      ITERATIONS++; // count iterations
      unsigned int w = adjacents[j]; // w is an adjacent of v
      topoSort->numIncomingEdges[w]--;   // remove edge (v,w)
      if (topoSort->numIncomingEdges[w] == 0) QueueEnqueue(q, w);
    }
    free(adjacents);
  }
  QueueDestroy(&q);
  ...
  return topoSort;
}
```

4.2. `GraphTopoSortComputeV3` - Dados Experimentais

```
FILE: ./GRAFOS_ORIENTADOS/DAG_5.txt
SORT: TopoSortV3
#           time          caltime       iterations
        0.000002         0.000003               20
RESULT: Topological Sorting - Vertex indices:
0 1 2 3 4
--------
```

```
FILE: ./GRAFOS_ORIENTADOS/DAG_6.txt
SORT: TopoSortV3
#           time          caltime       iterations
        0.000007         0.000011              230
RESULT: Topological Sorting - Vertex indices:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
--------
```



5. Conclusão

Em suma, as implementações dos scripts `Graph.c` e `GraphTopologicalSorting.c` destacam-se pela sua otimização e eficiência devido ás diferentes abordagens para a resolução do problema em questão. A primeira versão de ordenação topológica apresenta uma ineficiência pois, para além de sucessivas procuras através do conjunto de vértices, demora tempo a fazer uma cópia do grafo original, sendo a versão menos otimizada. A segunda, difere da anterior unicamente no facto de usar uma *array* auxiliar, logo não precisa de usar uma cópia, no entanto, mantém as sucessivas procuras através do conjunto de vértices e, por fim, a terceira, por usar uma *queue* como forma de ir verificando a informação, é a versão mais otimizada e eficiente.