



universidade de aveiro
theoria poiesis praxis

Trabalho 1

Seminário de Matemática Aplicada

Cláudio Henriques
Mestrado em Matemática e Aplicações

21 de Novembro de 2018

Conteúdo

Lista de Figuras	3
1 Introdução	5
1.1 <i>Java</i>	5
1.2 Estruturas de Dados	6
1.3 Algoritmos	7
1.4 Módulos extra	8
2 Implementação	9
2.1 Grafo (<i>Graph.java</i>)	9
2.2 Vértice (<i>Vertex.java</i>)	14
2.3 Coloração dos vértices, versão 1	17
2.3.1 Algoritmo	20
2.4 Coloração dos vértices, versão 2	21
2.4.1 Algoritmo	21
2.5 Coloração dos vértices, versão 3	23
2.5.1 Algoritmo	24
2.6 <i>Depth First Search</i> (DFS)	26
2.6.1 Algoritmo	30
3 Resolução dos exercícios (Análise e resultados)	32
3.1 Exercício 1	32
3.1.1 Matriz H	32
3.1.2 Matriz J	33
3.1.3 Matriz M	34
3.1.4 Matriz I	35

3.1.5	Matriz F	37
3.2	Exercício 2	38
3.2.1	Matriz P	39
3.2.2	Matriz O	39
3.2.3	Matriz A	40
3.2.4	Matriz Q	41
3.2.5	Matriz M_{25}	42
3.2.6	Matriz M_{50}	43
3.3	Exercício 3	44
4	Conclusão	45
4.1	Considerações futuras	45

Lista de Figuras

2.1	Ler linhas do ficheiro	10
2.2	Definir sucessores e predecessores	11
2.3	Ordenar os vértices por ordem crescente de grau	13
2.4	Definir a cor do vértice	17
2.5	Obter a primeira cor disponível	18
2.6	Guardar as cores dos vizinhos	19
2.7	Colorização do grafo, versão 1	20
2.8	Colorização do grafo, versão 2	22
2.9	Guardar as cores dos vizinhos, versão 3	24
2.10	Colorização do grafo, versão 3	25
2.11	Obter o primeiro vértice não visitado	26
2.12	Verificação da existência de vértices não visitados	27
2.13	Verificação da existência de predecessores não visitados	29
2.14	Algoritmo <i>Depth First Search</i>	30
3.1	Grafo H, coloração 1	32
3.2	Grafo H, coloração 2	33
3.3	Grafo H, coloração 3	33
3.4	Grafo J, coloração 1	33
3.5	Grafo J, coloração 2	34
3.6	Grafo J, coloração 3	34
3.7	Grafo M, coloração 1	34
3.8	Grafo M, coloração 2	35
3.9	Grafo M, coloração 3	35
3.10	Grafo I, coloração 1	36

3.11 Grafo I, coloração 2	36
3.12 Grafo I, coloração 3	36
3.13 Grafo F, coloração 2	37
3.14 Grafo F, coloração 1	38
3.15 Grafo F, coloração 3	38
3.16 Grafo P, pesquisa em profundidade	39
3.17 Grafo O, pesquisa em profundidade	40
3.18 Grafo A, pesquisa em profundidade	41
3.19 Grafo Q, pesquisa em profundidade	42
3.20 Grafo M25, pesquisa em profundidade	42
3.21 Grafo M50, pesquisa em profundidade	43

Capítulo 1

Introdução

No presente relatório é apresentado o algoritmo desenvolvido para a coloração dos vértices de um determinado grafo, utilizando a sua matriz de adjacência para definir uma estrutura abstracta que o representa. É ainda apresentado um algoritmo de pesquisa em profundidade e uma proposta de resolução do exercício 3.

Por forma a auxiliar a construção dos algoritmos referidos anteriormente, foram desenvolvidas duas estruturas auxiliar para representar de forma abstrata um grafo e um vértice.

Estes algoritmos foram implementados recorrendo à linguagem de programação *Java*. Foi ainda desenvolvido um pequeno script em *R* para a visualização do grafo aplicando as colorações obtidas nos algoritmos. Para a resolução do exercício 3 foi utilizado o *software Octave*.

1.1 *Java*

A escolha da linguagem baseou-se essencialmente pelos recursos que esta oferece. Apesar de não ser tão rápida e consumir mais recursos computacionais quando comparada com o *Python* ou *Julia*, o *Java* oferece um bom suporte para o desenvolvimento de algoritmos sob o velho conhecido paradigma *Object-oriented programming (OOP)* (em português: Programação Orientada a Objetos (POO)).

A opção de implementar o trabalho em torno deste paradigma permite desenvolver *módulos* que possam ser usados futuramente noutros trabalhos relacionados

com esta temática. Um exemplo simples é poder usar a estrutura de dados abstrata criada para representar um grafo num outro trabalho/projeto em que seja necessário esta estrutura para desenvolver outro tipo de algoritmos. Assim permite a diminuição de código redundante (i.e. implementar várias vezes a mesma estrutura de dados sempre que necessitar de a utilizar em diferentes ambientes).

O facto de ter optado por esta linguagem permite também a adaptação das estruturas auxiliares criadas para auxiliar a construção dos algoritmos.

1.2 Estruturas de Dados

As estruturas de dados desenvolvidas ao longo do projeto foram:

- **Grafo** (*Graph.java*): representar um objeto do tipo grafo (recorrendo à sua matriz de adjacência). Os atributos desta estrutura são:
 - *id* (identificador único do grafo);
 - *order* (ordem do grafo);
 - *vertexList* (lista de vértices do grafo);Os métodos disponíveis são:
 - *getId ()* (retorna o (*id*) do grafo);
 - *getOrder ()* (retorna a ordem do grafo);
 - *getVertexList ()* (retorna a lista de vértices do grafo);
 - *isNeighbor (int k, int y)* (retorna *True* se o vértice *k* for vizinho de *y* ou *False* no caso contrário);
 - *getVertexByDegree ()* (retorna a lista de vértices ordenada de forma crescente tendo em conta grau de cada um);
 - *sortVertexByDegree ()* (ordena a lista de vértices do grafo de forma crescente tendo em conta grau de cada um);
 - *setDefaultColorVertexes ()* (define todos os vértices com a cor 0);
- **Vértice** (*Vertex.java*): A estrutura anterior instancia uma estrutura auxiliar para representar cada vértice como um objeto único. Desta forma um grafo é representado com um conjunto de vértices. (Lógicamente seria óbvio representar um grafo como um conjunto de vértices e arestas, contudo as arestas (neste caso) não são estritamente necessárias ao desenvolvimento dos algoritmos, recorrendo apenas a atributos no objeto vértice para identificar os seus vizinhos). Os atributos do objeto são:

- *id* (identificador único do vértice);
- *color* (cor do vértice);
- *order* (ordem do vértice);
- *visited* (estado de visita do vértice);
- *sucessorList* (lista de sucessores do vértice);
- *predecessorList* (lista de predecessores do vértice);

Os métodos disponíveis são:

- *getId ()* (retorna o id) do vértice);
- *setColor (int color)* (define a cor do vértice);
- *getColor ()* (retorna a cor do vértice);
- *setOrder (int order)* (define a ordem do vértice);
- *setVisited ()* (define o vértice como visitado);
- *isVisited ()* (retorna *True* se o vértice já foi visitado ou *False* no caso contrário);
- *setNeighbors (int k, int ps)* (define o vértice como vizinho do vértice k);
- *isNeighbor ()* (retorna *True* se o vértice é vizinho do vértice k ou *False* no caso contrário);
- *getNeighborList ()* (retorna a lista de vizinhos do vértice);
- *getSucessorList ()* (retorna a lista de sucessores do vértice);
- *getPredecessorList ()* (retorna a lista de predecessores do vértice);
- *getDegree ()* (retorna o grau do vértice);

1.3 Algoritmos

Os algoritmos desenvolvidos são:

- **Coloração dos vértices, versão 1** (*ColorizeVersion1.java*): Implementa o algoritmo apresentado nas aulas para a coloração dos vértices de um grafo, recorrendo à sua matriz de adjacência. Retorna a coloração obtida.
- **Coloração dos vértices, versão 2** (*ColorizeVersion2.java*): Implementado como uma extensão da versão 1 deste algoritmo, altera apenas a ordem pela qual percorre os vértices do grafo. Retorna a coloração obtida.
- **Coloração dos vértices, versão 3** (*ColorizeVersion3.java*): Organiza os vértices pela ordem de maior grau, sendo este o critério para os percorrer e atribuir uma cor. Retorna a coloração obtida.

- **Depth First Search (DFS)** (*DFS.java*): Implementa o algoritmo para percorrer os vértices. Retorna a lista de vértices e a ordem pela qual foram visitados.

1.4 Módulos extra

Foi criado um módulo extra para o auxílio da construção de ficheiros **.txt*. Este mecanismo permite efetuar o *debug* dos algoritmos, ou seja, permite verificar a cada passo onde é que o algoritmo pode estar a falhar.

Estes ficheiros podem também ser utilizados para uma melhor compreensão dos algoritmos.

- **Ficheiro de logs** (*WriteLogFile.java*) Os ficheiros criados através deste módulo são:
 - *log_V1.txt*;
 - *log_V2.txt*;
 - *log_V3.txt*;
 - *log_dfs.txt*;

Capítulo 2

Implementação

Nesta secção é apresentada e explicada a implementação das estruturas de dados que foram desenvolvidas por forma a tornar os algoritmos mais eficientes.

Algumas linhas de código foram omitidas para simplificar a explicação, por exemplo, a implementação da criação do ficheiro auxiliar com a matriz de adjacência que vai ser lida pelo script *R* uma vez que este processo não é essencial para o desenvolvimento dos algoritmos. Foram também omitidas as linhas correspondentes à implementação dos ficheiros de *log*.

Todo o código está comentado e pode ser visto em github.com/cfchenr/sma.

2.1 Grafo (*Graph.java*)

Esta estrutura abstracta usa um *ArrayList* para representar o conjunto de vértices pertencentes ao objeto grafo (denominada por *vertexList*).

Cada grafo é identificado por um *id* e contém um atributo chamado *order* que representa a ordem do grafo.

```
1 private String id;  
2 private int order;  
3 private ArrayList<Vertex> vertexList;
```

Neste caso o *id* é o nome do ficheiro com a matriz de adjacência que é lida. Por exemplo, se construir um grafo recorrendo à matriz de adjacência do ficheiro *A.txt*, o *id* deste grafo é definido como A.

```
1 id = file.split("/") [1].split("\\.")[0];
```

À medida que cada linha do ficheiro com a matriz de adjacência é lida, é adicionado à lista de vértices um novo vértice (pois cada linha da matriz identifica um vértice do grafo). Estes são identificados por um número que é igual à ordem do grafo na iteração anterior.

Por exemplo, se ainda não foi lida nenhuma linha então o primeiro vértice que é criado é representado por 0. Na linha seguinte é criado um novo vértice, desta vez já é representado pelo número 1.

Consequentemente a ordem do grafo é o número de linhas da matriz de adjacência.

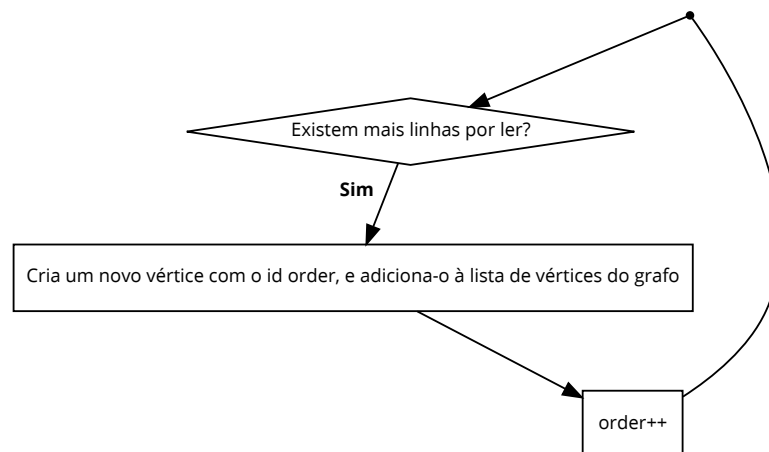


Figura 2.1: Ler linhas do ficheiro

```
1 while (scf.hasNextLine()) {
2     vertexList.add(new Vertex(order));
3     order++;
4 }
```

Seguidamente são lidas cada uma das entradas de cada linha da matriz para se definir as vizinhanças de cada vértice. Desta forma, a cada entrada da linha que é lida, se for 1 ou -1 então significa que o vértice que é representado pelo índice da

linha da matriz é vizinho do vértice representado pelo índice da coluna da entrada em questão.

Consequentemente, se for 1 então definimos que o vértice representado pelo índice da linha da matriz é sucessor do vértice representado pelo índice da coluna da entrada em questão. Lógicamente também definimos que o índice da coluna da entrada em questão é predecessor do vértice representado pelo índice da linha da matriz. (A distinção entre sucessor e predecessor é implementada na estrutura *Vertex.java*, através da distinção do segundo argumento que é passado na função. Esta anotação é explicanda mais à frente na secção 2.2).

Analogamente utilizamos a mesma metodologia no caso da entrada ser -1 .

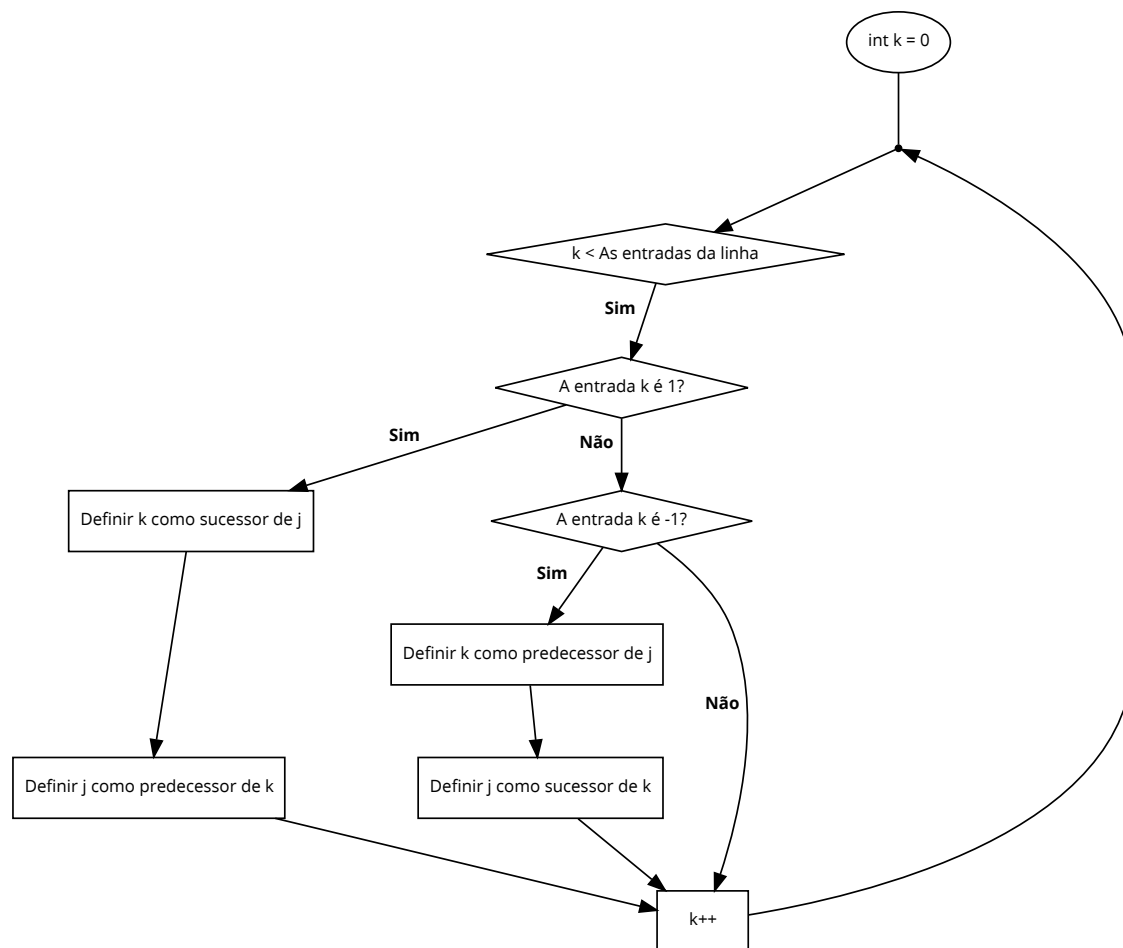


Figura 2.2: Definir sucessores e predecessores

```
1   for (int k = 0; k < line.length; k++) {  
2       if (line[k].equals("1")) {  
3           vertexList.get(j).setNeighbors(k, 1);  
4           vertexList.get(k).setNeighbors(j, -1);  
5       } else if (line[k].equals("-1")) {  
6           vertexList.get(j).setNeighbors(k, -1);  
7           vertexList.get(k).setNeighbors(j, 1);  
8       }  
9   }
```

O método *getId ()* retorna o *id* deste grafo.

```
1   return id;
```

O método *getOrder ()* retorna a ordem do grafo.

```
1   return order;
```

O método *getVertexList ()* retorna a lista com todos os vértices do grafo.

```
1   return vertexList;
```

O método *isNeighbor (int k, int y)* retorna um valor booleano para dar resposta à pergunta "k é vizinho de y?". Reparemos que k ser vizinho de y é a mesma coisa que y ser vizinho de k.

```
1   return (vertexList.get(k).isNeighbor(y) || vertexList.get(y).  
           isNeighbor(k));
```

O método *getVertexByDegree ()* retorna uma lista de vértices por ordem crescente do grau do vértice.

Neste método é criada uma estrutura temporária para armazenar os vértices pela ordem pretendida.

Num primeiro momento, é obtido o grau máximo de todos os vértices. Após esta operação é adicionada à estrutura temporária os vértices que têm o grau igual ao grau máximo (os vértices são adicionados sempre ao início da lista, o que permite criar a lista por ordem crescente de grau dos vértices). Quando não houver mais vértices com o grau máximo então é reduzido o valor máximo numa unidade voltando a efetuar o processo anterior. Este ciclo termina quando o valor máximo for 0.

Por fim é retornada a estrutura temporária, sendo esta uma lista com os vértices organizados por ordem crescente do grau.

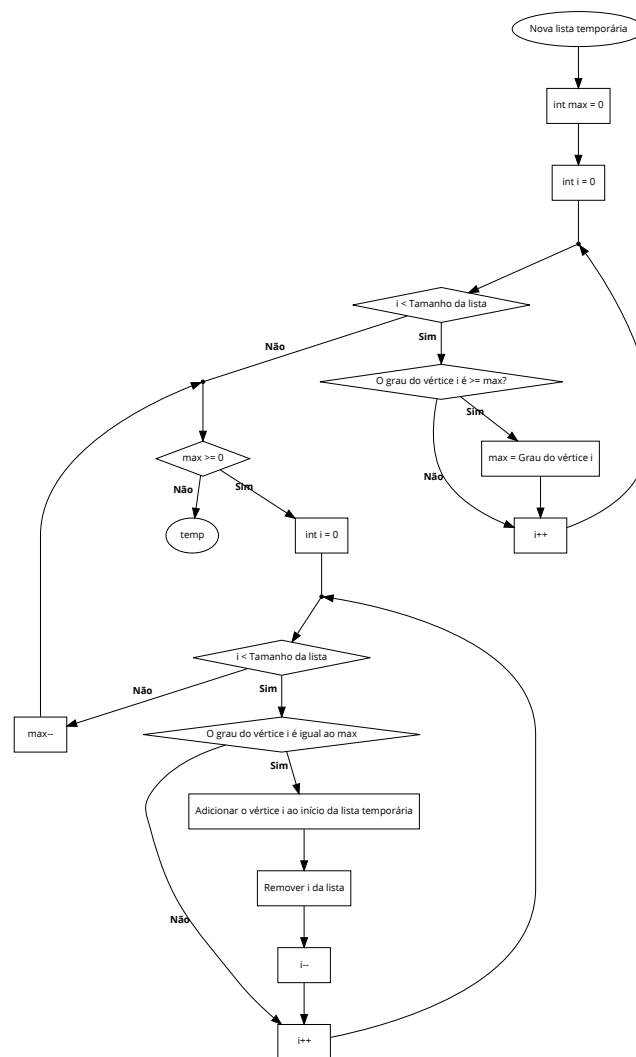


Figura 2.3: Ordenar os vértices por ordem crescente de grau

```
1      ArrayList<Vertex> temp = new ArrayList<Vertex>();
2      int max = 0;
3      for (int i = 0; i < vertexList.size(); i++)
4          if (vertexList.get(i).getDegree() >= max)
5              max = vertexList.get(i).getDegree();
6      while (max >= 0) {
7          for (int i = 0; i < vertexList.size(); i++) {
8              if (vertexList.get(i).getDegree() == max){
9                  temp.add(0, vertexList.get(i));
10                 vertexList.remove(i);
11                 i--;
12             }
13         }
14         max--;
15     }
16     return temp;
```

O método *sortVertexByDegree ()* organiza a lista de vértices deste grafo pela ordem do grau de cada vértice.

```
1      vertexList = getVertexByDegree();
```

O método *setDefaultColorVertexes ()* define a cor de cada vértice como 0 (cor inicial).

```
1      for (int i = 0; i < vertexList.size(); i++)
2          vertexList.get(i).setColor(0);
```

2.2 Vértice (*Vertex.java*)

Esta estrutura representa um vértice, identificado por um *id*, *color*, *order*, por um atributo que define se o vértice foi ou não visitado (*visited*), por uma lista de vértices sucessores (*sucessorList*) e uma lista de predecessores (*predecessorList*).

```
1 private int id , color , order ;  
2 private boolean visited ;  
3 private ArrayList<Integer> successorList , predecessorList ;
```

O método *getId ()* retorna o *id* do vértice.

```
1 return id ;
```

O método *setColor (int color)* define a cor do vértice com a cor *color*.

```
1 this.color = color ;
```

O método *getColor ()* retorna a cor do vértice.

```
1 return color ;
```

O método *setOrder (int order)* define a ordem do vértice com o número *order*.

```
1 this.order = order ;
```

O método *setVisited ()* define o vértice como visitado.

```
1 visited = true ;
```

O método *isVisited ()* retorna um booleano que indica se o vértice já foi ou não visitado.

```
1 return visited ;
```

O método *setNeighbors (int k, int ps)* define este vértice como vizinho do vértice identificado por *k*. Se *ps* for 1 então o vértice *k* é sucessor, caso contrário *k* é predecessor.


```
1  if (ps == 1) {  
2      if (!successorList.contains(k))  
3          successorList.add(k);  
4  }  
5  else if (ps == -1) {  
6      if (!predecessorList.contains(k))  
7          predecessorList.add(k);  
8  }
```

O método *isNeighbor* (*int k*) retorna um valor booleano que indica se o vértice é vizinho do vértice *k*.

```
1  return (successorList.contains(k) || predecessorList.contains(k))  
    ;
```

O método *getNeighborList* () retorna a lista de sucessores e predecessores do vértice.

```
1  Set<Integer> set = new HashSet<Integer>();  
2  set.addAll(successorList);  
3  set.addAll(predecessorList);  
4  return new ArrayList<Integer>(set);
```

O método *getSucessorList* () retorna a lista de sucessores do vértice.

```
1  return successorList;
```

O método *getPredecessorList* () retorna a lista de predecessores do vértice.

```
1  return predecessorList;
```

O método *getDegree* () retorna o grau do vértice. Este é igual ao número de vizinhos do vértice.

```
1 return getNeighborList().size();
```

2.3 Coloração dos vértices, versão 1

Esta implementação recebe um objeto do tipo grafo (*Graph.java*) e contém um conjunto de métodos que permite desenvolver um algoritmo de *fácil* leitura. Quando invocada, esta implementação guarda a lista dos vértices do grafo e define todos os vértices com a cor 0 (para garantir que o vértice não foi colorido anteriormente com outra cor distinta).

```
1 graph.setDefaultColorVertexes();  
2 vertexList = graph.getVertexList();
```

O método *setColorVertex (int id)* define a cor do vértice identificado pelo *id* com a primeira cor disponível (tendo em conta os seus vizinhos), com recurso ao método *getFirstColorAvailable ()*. No caso do *id* ser -1 significa que o vértice a colorir é o último vértice da lista. Após obter o vértice correspondente ao *id* obtém a primeira cor disponível (*color*) e atribui ao vértice a cor *color*.

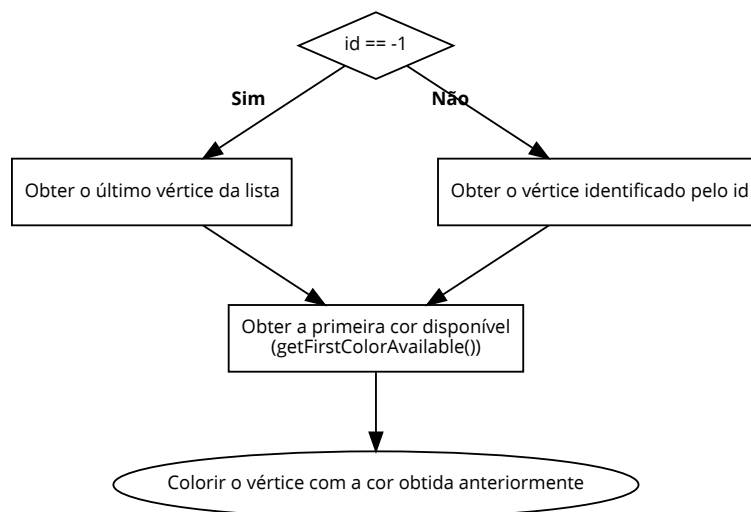


Figura 2.4: Definir a cor do vértice

```

1  Vertex vertex;
2  if (id == -1)
3      vertex = vertexList.get(vertexList.size()-1);
4  else
5      vertex = vertexList.get(id);
6  int color = getFirstColorAvailable();
7  vertex.setColor(color);

```

O método *getFirstColorAvailable()* analisa uma fila de cores e retorna a cor mínima disponível. Esta fila é preenchida no método *saveNeighborColors(int j)*. Inicialmente começa com a cor mínima $k = 1$ e verifica se esta cor existe na lista de cores. No caso de existir então k passa a ser igual a 2 e volta a verificar se 2 existe na lista de cores. Este processo é repetido até encontrar um k que não conste na lista de cores.

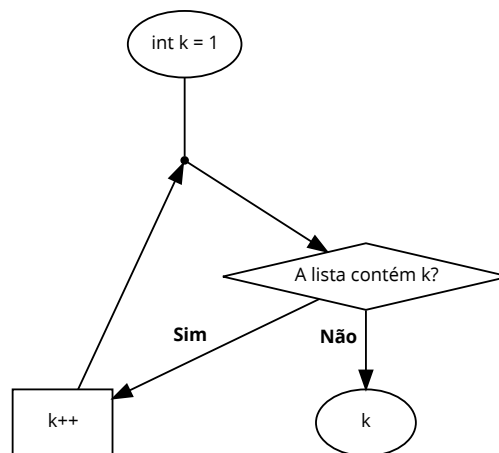


Figura 2.5: Obter a primeira cor disponível

```

1  int k = 1;
2  while(colors.contains(k))
3      k++;
4  return k;

```

O método *saveNeighborColors* (*int j*) guarda as cores dos vizinhos do vértice identificado por *j*. Nesta implementação, apenas verifica se os vértices anteriores são vizinhos, uma vez que como a ordem pela qual são percorridos os vértices começa no primeiro vértice até ao último, então aquando do vértice *j* ainda só foram coloridos os vértices anteriores. Quando encontrar um vértice vizinho, adiciona a cor deste à lista de cores.

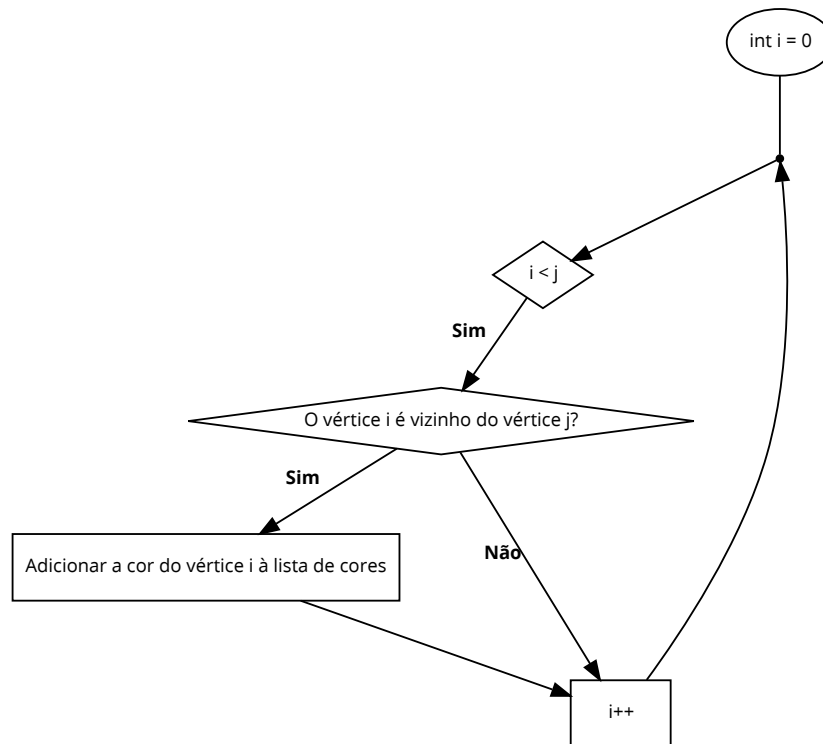


Figura 2.6: Guardar as cores dos vizinhos

```

1  colors = new LinkedList<Integer>();
2  for (int i = 0; i < j; i++)
3      if (vertexList.get(j).isNeighbor(vertexList.get(i).getId()))
4          colors.add(vertexList.get(i).getColor());
  
```

O método *getVertexList* () retorna a lista dos vértices do grafo.

```

1  return vertexList;
  
```

2.3.1 Algoritmo

Em suma, seja *v1* um objeto do tipo *ColorizeVersion1.java*, definimos a cor do vértice 0 (o primeiro vértice do grafo) com a primeira cor disponível (como é o primeiro vértice, a cor atribuída a este será 1). Após este processo, percorre-se todos os restantes vértices. A cada um destes, guarda-se a cor dos vértices anteriores que são seus vizinhos e seguidamente é definida a cor do vértice *j* com a cor mínima disponível (tendo em conta as cores obtidas no processo anterior). O algoritmo final terá o seguinte aspeto:

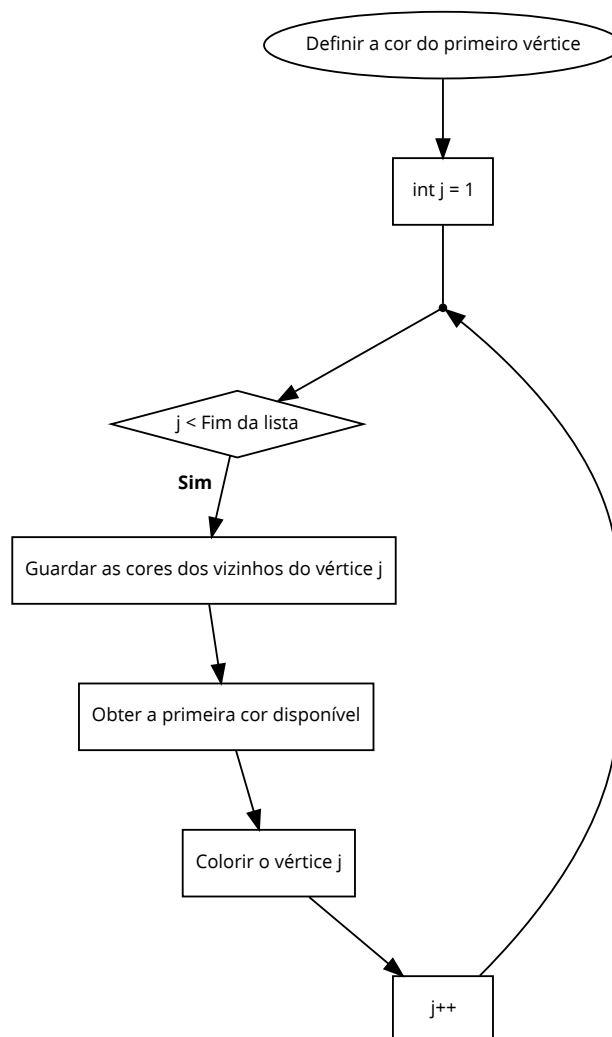


Figura 2.7: Colorização do grafo, versão 1

```
1 v1.setColorVertex(0);  
2 for (int j = 1; j < v1.getVertexList().size(); j++) {  
3     v1.saveNeighborColors(j);  
4     v1.setColorVertex(j);  
5 }
```

2.4 Coloração dos vértices, versão 2

Esta implementação, sendo uma extensão da anterior, é em grande parte igual à versão 1. A alteração que implementa é a ordem pela qual verifica os vértices vizinhos, ou seja, apenas verifica os vértices desde o fim até ao vértice j .

```
1 colors = new LinkedList<Integer>();  
2 for (int i = vertexList.size() - 1; i > j; i--)  
3     if (vertexList.get(j).isNeighbor(vertexList.get(i).getId()))  
4         colors.add(vertexList.get(i).getColor());
```

2.4.1 Algoritmo

Em suma, seja $v2$ um objeto do tipo *ColorizeVersion2.java*, definimos a cor do último vértice com a primeira cor disponível (como é o primeiro vértice a colorir, a cor atribuída a este será 1). Após este processo, percorre-se todos os restantes vértices, começando pelo fim (contrariamente à versão anterior). A cada um destes, guarda-se a cor dos seus vizinhos e seguidamente é definida a cor do vértice j com a cor mínima disponível (tendo em conta as cores dos seus vizinhos). O algoritmo final terá o seguinte aspeto:

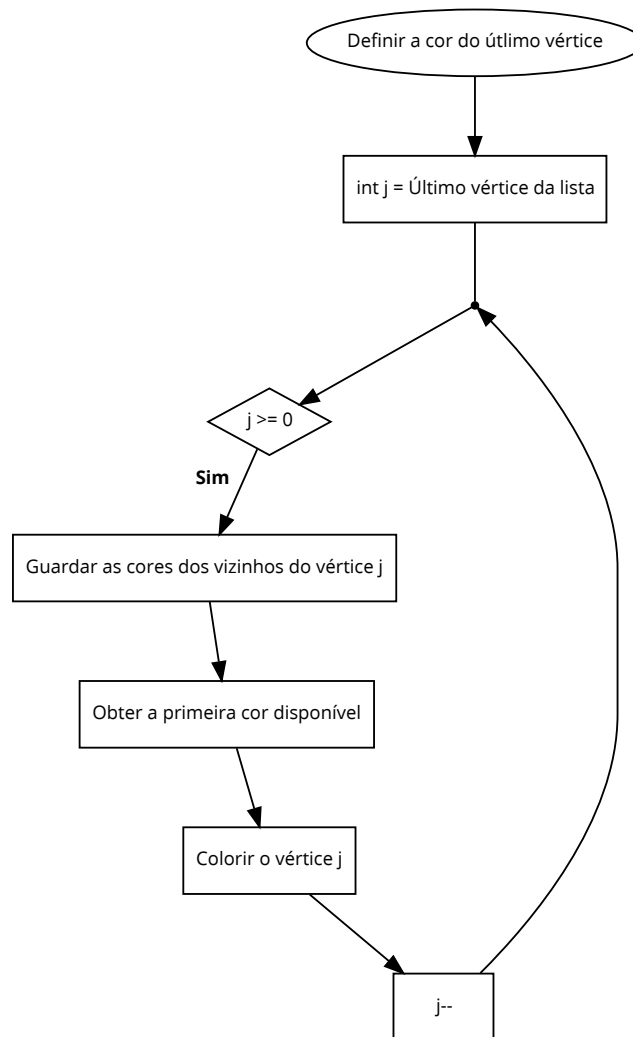


Figura 2.8: Colorização do grafo, versão 2

```
1  v2.setColorVertex(-1);  
2  for (int j = v2.getVertexList().size()-2; j >= 0; j--) {  
3      v2.saveNeighborColors(j);  
4      v2.setColorVertex(j);  
5  }
```

2.5 Coloração dos vértices, versão 3

Esta implementação, quando invocada, tem o mesmo comportamento da versão 1 e 2, contudo organiza os vértices pela ordem do seu grau. Uma outra diferença nesta implementação é que ao invés de usar uma lista de cores usa um dicionário em que a chave é o código da cor e o valor é 1 (0 no caso da cor não fazer parte da coloração dos vértices vizinhos de um determinado vértice, e neste caso não é adicionado ao dicionário de cores, 1 no caso de algum vizinho ter a cor identificada pelo código da cor).

```
1 colors = new Hashtable<Integer , Integer>();  
2 vertexByDegree = graph.getVertexByDegree();
```

No método *setColorVertex (int id)*, ao invés de se obter os vértices a partir da lista de vértices, o algoritmo recorre à lista dos vértices ordenados por grau. O restante código é igual ao implementado nas versões anteriores.

```
1 Vertex vertex;  
2 if (id == -1)  
3     vertex = vertexList.get(vertexByDegree.get(vertexByDegree.  
4         size()-1).getId());  
5 else  
6     vertex = vertexList.get(vertexByDegree.get(id).getId());  
7 int color = getFirstColorAvailable();  
   vertex.setColor(color);
```

O método *saveNeighborColors (int j)* tira proveito do facto de cada vértice ter associado uma lista de vértices vizinhos. Isto permite que o algoritmo não tenha de percorrer todos os vértices do grafo e verificar se algum é vizinho do vértice *j* e nesse caso adicionar a sua cor à tabela de cores. Neste caso, o algoritmo percorre apenas os vértices que fazem parte da lista de vizinhos do vértice *j* e adiciona a sua cor à tabela de cores. Contrariamente a uma lista, a tabela (sendo ela um dicionário) caso contenha a cor que está a adicionar simplesmente atualiza o seu valor (que neste caso fica exatamente igual). Por exemplo, considera-se uma tabela de cores que já contém a cor 2 e 5. Então os valores associados a 2 e 5 é 1 (como foi explicado em cima). Quando se adiciona novamente a cor 2 ou a cor 5, ao invés

de criar uma nova entrada nesta tabela de cores, simplesmente verifica se ela já existe antes de criar.

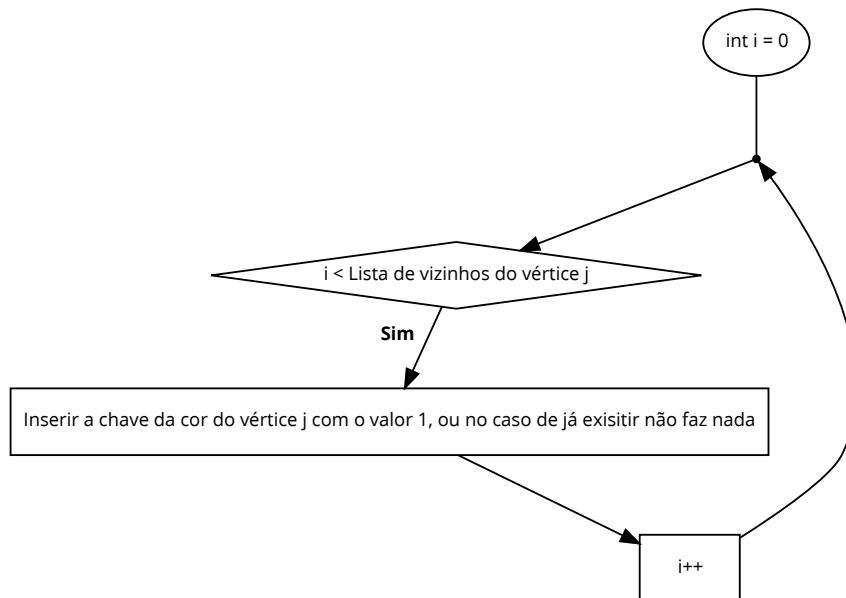


Figura 2.9: Guardar as cores dos vizinhos, versão 3

```

1  colors = new Hashtable<Integer , Integer>();
2  for (int i = 0; i < vertexList.get(vertexByDegree.get(j)).getId())
   .getNeighborList().size(); i++)
3    colors.put(vertexList.get(vertexList.get(vertexByDegree.get(j)
   ).getId()).getNeighborList().get(i)).getColor(), 1);
  
```

Comparando 2.6 com 2.9 vemos que existe alguma simplificação do processo.

2.5.1 Algoritmo

Em suma, a implementação deste algoritmo segue a mesma lógica da versão 2, pois uma vez que os vértices estão ordenados por ordem crescente de grau, então começa a percorrer do fim para o início, por forma a atribuir primeiramente as cores aos vértices com maior grau. Seja v3 um objeto do tipo *ColorizeVersion3.java*:

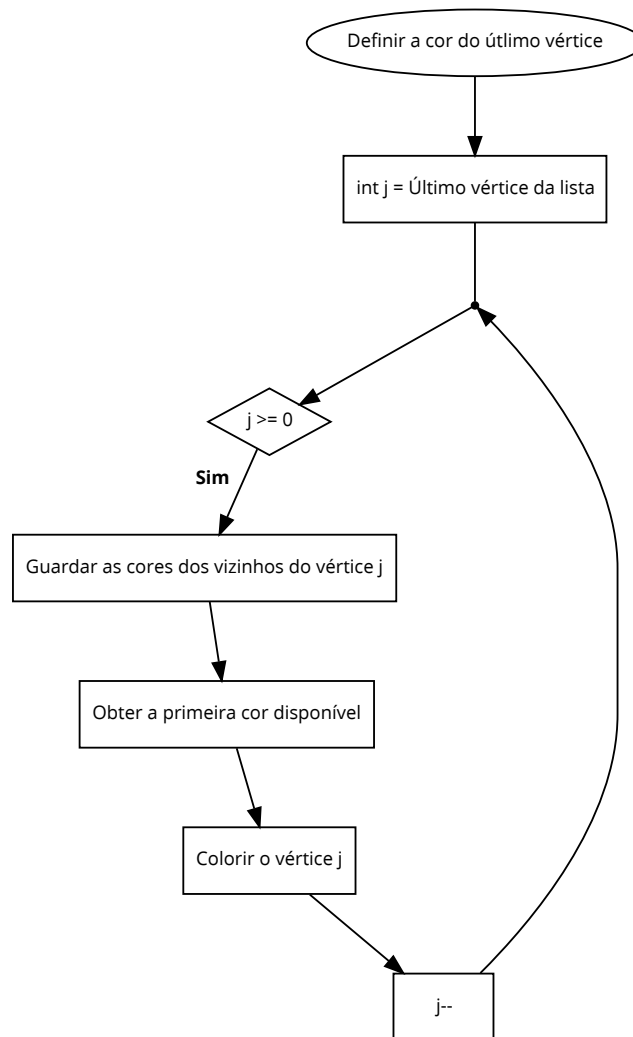


Figura 2.10: Colorização do grafo, versão 3

```

1  v3.setColorVertex(-1);
2  for (int j = v3.getVertexList().size()-2; j >= 0; j--) {
3      v3.saveNeighborColors(j);
4      v3.setColorVertex(j);
5  }
  
```

2.6 Depth First Search (DFS)

Esta implementação recebe um objeto do tipo grafo (*Graph.java*). Quando invocada, a lista dos vértices do grafo. Este objeto contém atributos como uma lista de vértices (*vertexList*), *orderIndex* que servirá para definir a ordem pela qual um vértice foi visitado (inicialmente contém o valor 0) e uma lista de vértices em espera para serem visitados (*stack*).

```
1 graph.sortVertexByDegree();
2 vertexList = graph.getVertexList();
```

O método *getFirstNonVisitedVertex()* verifica se existem algum vértice não visitado na lista. Caso se verifica então percorre os vértices contidos na lista e quando encontrar o primeiro vértice da lista não visitado, retorna-o.

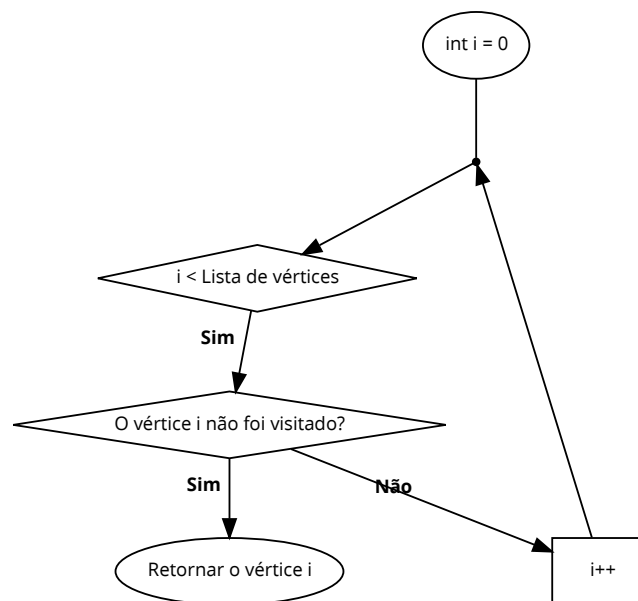


Figura 2.11: Obter o primeiro vértice não visitado

```
1 for (int i = 0; i < vertexList.size(); i++)
2     if (!vertexList.get(i).isVisited())
3         return vertexList.get(i);
4 return vertexList.get(0);
```

O método *haveNonVisitedVertexes ()* verifica se a lista contém vértices não visitados.

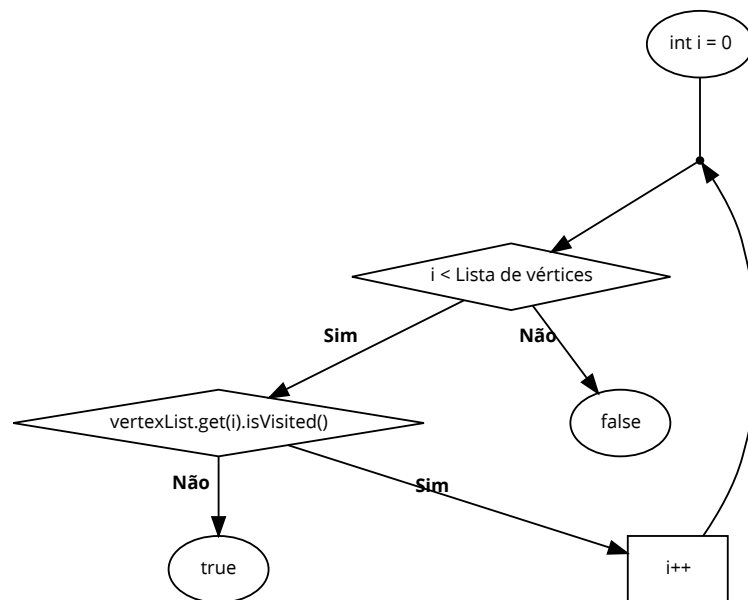


Figura 2.12: Verificação da existência de vértices não visitados

```

1  for (int i = 0; i < vertexList.size(); i++)
2      if (!vertexList.get(i).isVisited()) {
3          return true;
4      }
5  return false;
  
```

O método *setVisited (Vertex vertex)* recebe um vértice e define-o como visitado e atribui a ordem pela qual foi visitado.

```

1  vertex.setVisited();
2  setOrder(vertex);
  
```

O método *getVisited* (*Vertex vertex*) retorna um valor booleano para determinar se o vértice *vertex* já foi visitado anteriormente.

```
1   if ( vertex.isVisited() ) {  
2       return true;  
3   } else {  
4       return false;  
5   }
```

O método *setOrder* (*Vertex vertex*) define a ordem pela qual o vértice foi visitado e adiciona-o à *stack* dos vértices em espera.

```
1   vertex.setOrder(++orderIndex);  
2   addToStack( vertex );
```

O método *addToStack* (*Vertex vertex*) adiciona à lista de espera o vértice *vertex*.

```
1   stack.add( vertex );
```

O método *getStack* () retorna a lista com os vértices em espera.

```
1   return stack;
```

O método *getTopStack* () retorna o primeiro vértice na lista dos vértices em espera.

```
1   return stack.pop();
```

O método *getAllSucessors* (*Vertex vertex*) retorna um iterador dos sucessores do vértice *vertex*.

```
1   return vertex.getSucessorList().listIterator();
```

O método *getNextSucessor* (*Iterator<Integer> neighbors*) retorna o próximo sucessor (vértice) do vértice identificado por *neighbors*.

```
1 return vertexList.get(neighbors.next());
```

O método *haveNonVisitedPredecessors* (*Vertex vertex*) verifica se o vértice *vertex* tem predecessores visitados. Para tal, percorre a lista de predecessores do vértice *vertex* e quando encontrar (se encontrar) um vértice predecessor não visitado retorna *true*.

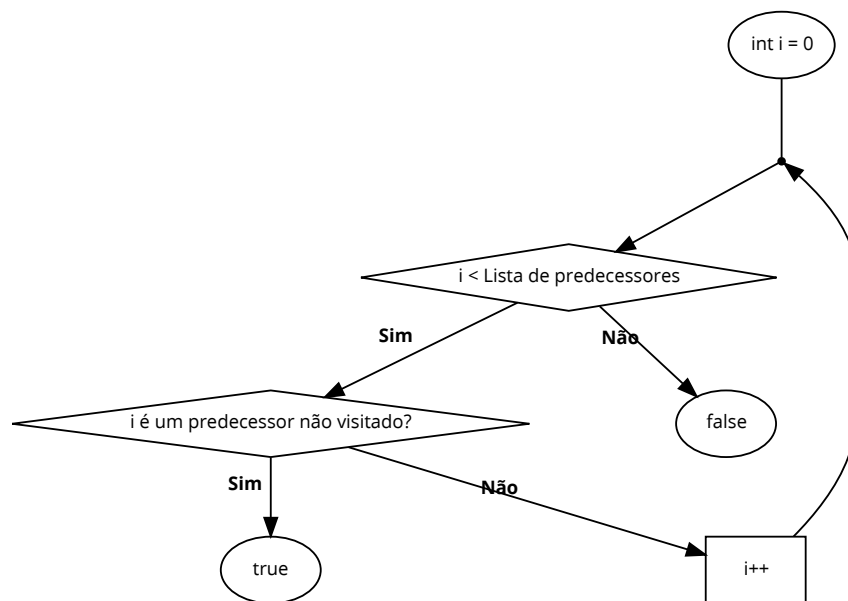


Figura 2.13: Verificação da existência de predecessores não visitados

```
1 for (int i = 0; i < vertex.getPredecessorList().size(); i++) {
2     if (!vertexList.get(vertex.getPredecessorList().get(i)).
3         isVisited())
4         return true;
5 }
return false;
```

2.6.1 Algoritmo

Em suma, a implementação final do algoritmo consiste na verificação da existência de algum vértice não visitado. No caso de existir obtém o primeiro vértice da lista que ainda não foi visitado, definindo-o seguidamente como visitado e adicionando-o à lista de espera. Após concluído este processo vão analisar se existem vértices na lista de espera, obtendo os vizinhos primeiro vértice disponível. Enquanto existirem vizinhos obtém o seu sucessor, verificando se já foi visitado e se contém predecessores não visitados. No caso de esta condição se verificar, define-o como visitado e volta à verificação da existência de mais vizinhos. Este processo repete-se enquanto houverem vértices não visitados ou existirem vértices na lista de espera.

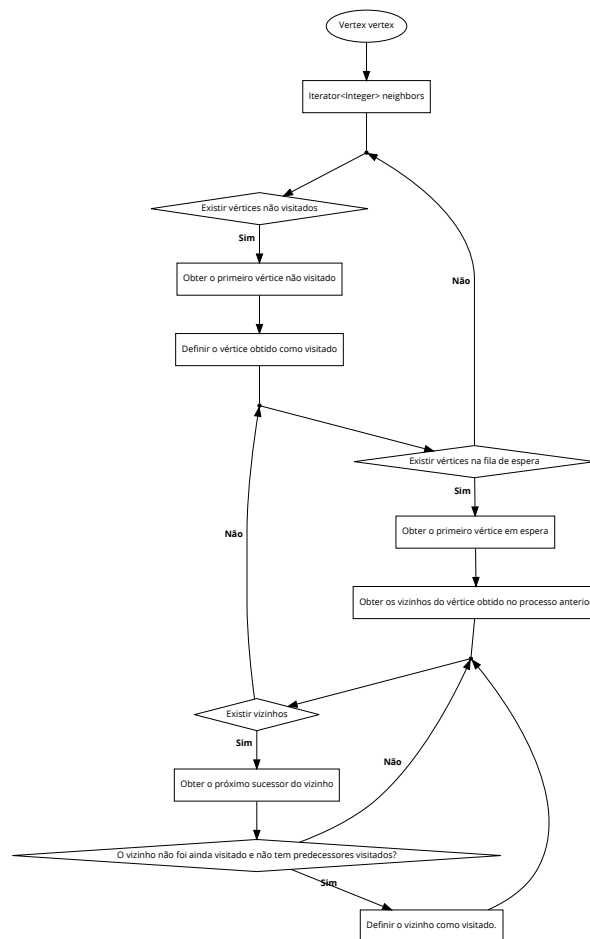


Figura 2.14: Algoritmo *Depth First Search*

```
1  Vertex vertex;  
2  Iterator<Integer> neighbors;  
3  while (dfs.haveNonVisitedVertexes()) {  
4      vertex = dfs.getFirstNonVisitedVertex();  
5      dfs.setVisited(vertex);  
6      while (dfs.getStack().size() != 0) {  
7          vertex = dfs.getTopStack();  
8          neighbors = dfs.getAllSucessors(vertex);  
9          while (neighbors.hasNext()) {  
10             Vertex n = dfs.getNextSucessor(neighbors);  
11             if (!dfs.getVisited(n) && !dfs.  
12                 haveNonVisitedPredecessors(n))  
13                 dfs.setVisited(n);  
14             }  
15         }  
16     }
```


Capítulo 3

Resolução dos exercícios (Análise e resultados)

3.1 Exercício 1

3.1.1 Matriz H

Consideremos a matriz H . Aplicamos o algoritmo *ColorizeVersion1.java* à matriz H .

A coloração obtida é a representada no seguinte grafo:

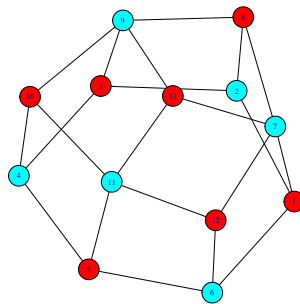


Figura 3.1: Grafo H , coloração 1

Aplicando a versão 2 e 3 do algoritmo de coloração dos vértices, obtemos os seguintes resultados:

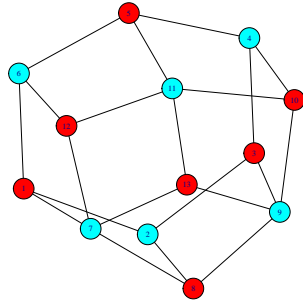


Figura 3.2: Grafo H, coloração 2

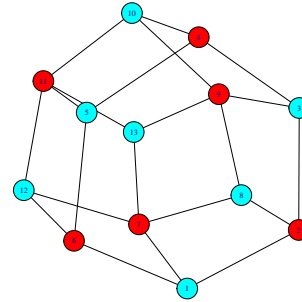


Figura 3.3: Grafo H, coloração 3

Analisando os grafos obtidos, podemos ver que neste caso o algoritmo 1, 2 e 3 retornam colorações iguais, a menos de uma alteração das cores.

3.1.2 Matriz J

Consideremos a matriz J . Aplicamos o algoritmo *ColorizeVersion1.java* à matriz J .

A coloração obtida é a representada no seguinte grafo:

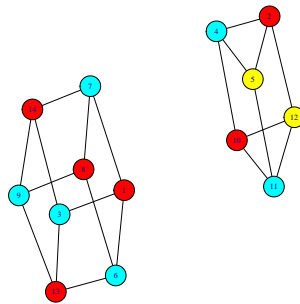


Figura 3.4: Grafo J, coloração 1

Aplicando a versão 2 e 3 do algoritmo de coloração dos vértices, obtemos os seguintes resultados:

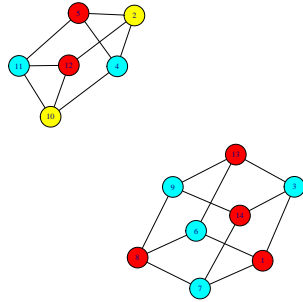


Figura 3.5: Grafo J, coloração 2

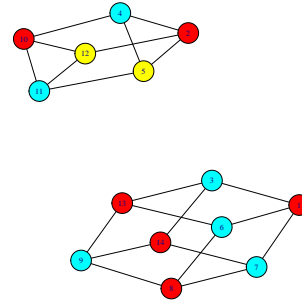


Figura 3.6: Grafo J, coloração 3

Analisando os grafos obtidos, podemos ver que neste caso o algoritmo 1, 2 e 3 retornam colorações equivalentes.

3.1.3 Matriz M

Consideremos a matriz M . Aplicamos o algoritmo *ColorizeVersion1.java* à matriz M .

A coloração obtida é a representada no seguinte grafo:

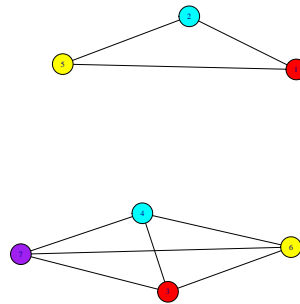


Figura 3.7: Grafo M, coloração 1

Aplicando a versão 2 e 3 do algoritmo de coloração dos vértices, obtemos os seguintes resultados:

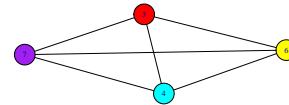
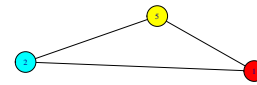
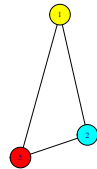
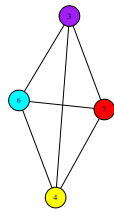


Figura 3.8: Grafo M, coloração 2

Figura 3.9: Grafo M, coloração 3

Analisando os grafos obtidos, podemos ver que neste caso o algoritmo 1, 2 e 3 retornam colorações equivalentes.

3.1.4 Matriz I

Seja I a matriz
$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}.$$
 Aplicamos o algoritmo *ColorizeVersion1.java* à matriz I .

A coloração obtida é a representada no seguinte grafo:

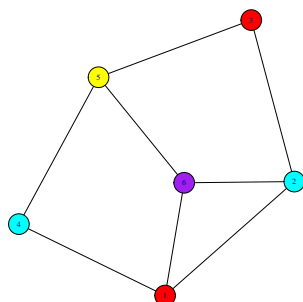


Figura 3.10: Grafo I, coloração 1

Aplicando a versão 2 e 3 do algoritmo de coloração dos vértices, obtemos os seguintes resultados:

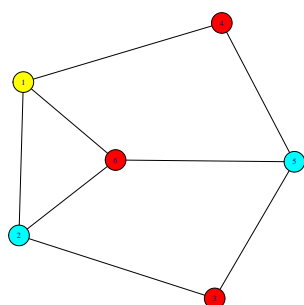


Figura 3.11: Grafo I, coloração 2

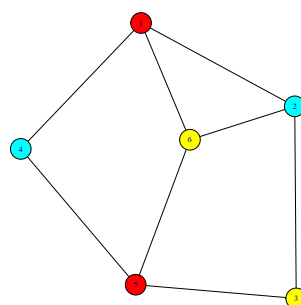


Figura 3.12: Grafo I, coloração 3

Analisando os grafos obtidos, podemos ver que neste caso o algoritmo 1 não consegue encontrar uma coloração mínima, ao contrário do que acontece com as versões 2 e 3.

Vamos agora analisar se a versão 3 consegue obter uma coloração mais baixa em algum caso, relativamente à versão 2.

3.1.5 Matriz F

Seja F a matriz

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}.$$

Aplicamos o algoritmo *ColorizeVersion2.java* à matriz F .

A coloração obtida é a representada no seguinte grafo:

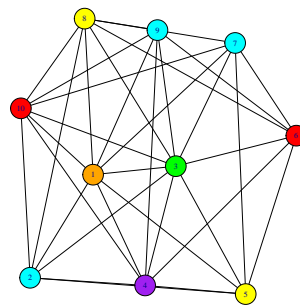


Figura 3.13: Grafo F , coloração 2

Aplicando a versão 1 e 3 do algoritmo de coloração dos vértices, obtemos os seguintes resultados:

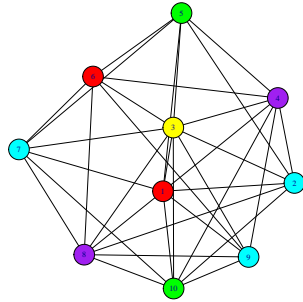


Figura 3.14: Grafo F, coloração 1

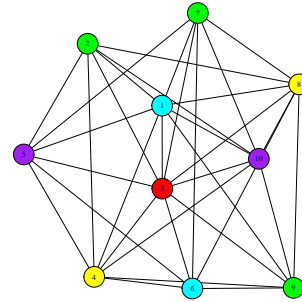


Figura 3.15: Grafo F, coloração 3

Analisando os grafos obtidos, podemos ver que neste caso o algoritmo 2 não consegue encontrar uma coloração mínima, ao contrário do que acontece com as versões 1 e 3.

3.2 Exercício 2

O algoritmo implementado é o representado em 2.14.

Para dar resposta à alínea b) foi utilizado o método *haveNonVisitedVertexes()* para verificar se na lista de vértices do grafo existe algum vértice não visitado.

O algoritmo só termina quando não existirem mais vértices não visitados: *while (dfs.haveNonVisitedVertexes())*.

Para a alínea c) foi introduzida uma verificação antes de marcar um vértice como visitado. A condição $(!dfs.getVisited(n) \ \&\& \ !dfs.haveNonVisitedPredecessors(n))$ verifica se o vértice em questão já foi visitado e se contém algum predecessor não visitado.

Nos grafos apresentados nesta secção, o número em cada vértice representa a ordem pela qual o vértice foi visitado.

3.2.1 Matriz P

Para o caso da matriz $P = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$, o resultado obtido é:

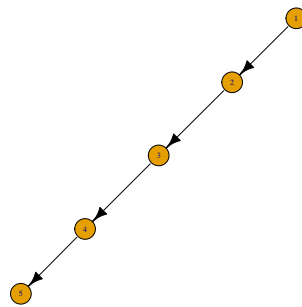


Figura 3.16: Grafo P, pesquisa em profundidade

Neste caso podemos ver que os vértices são visitados de forma sequencial (como era previsto), ou seja, analisando valor de cada vértice podemos ver que é sempre superior ao seu antecessor.

3.2.2 Matriz O

Para o caso da matriz $O = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & -1 \\ 0 & 0 & 0 \end{bmatrix}$, o resultado obtido é:

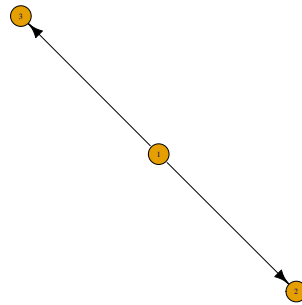


Figura 3.17: Grafo O, pesquisa em profundidade

À semelhança do exemplo anterior podemos ver que os vértices contém sempre um número superior ao seu antecessor.

3.2.3 Matriz A

Para o caso da matriz $A = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 1 \\ 0 & -1 & 0 & 0 & 0 & -1 & 1 \\ 0 & 0 & -1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & -1 & -1 & -1 & 0 \end{bmatrix}$, o resultado obtido

é:

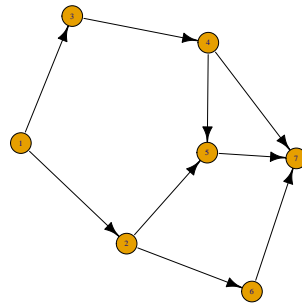


Figura 3.18: Grafo A, pesquisa em profundidade

Sendo este um caso mais complexo quando comparado com os exemplos anteriores podemos continuar a concluir que os vértices são sempre visitados após os seus antecessores.

3.2.4 Matriz Q

Para o caso da matriz $Q = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$, o resultado obtido é:

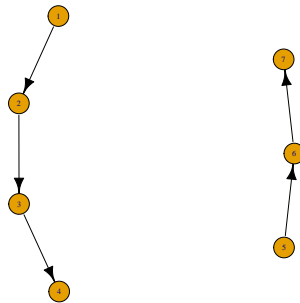


Figura 3.19: Grafo Q, pesquisa em profundidade

Neste caso temos um grafo não conexo, contudo todos os vértices contêm uma ordem pela qual foram visitados. Daqui podemos concluir que o algoritmo visita todos os vértices do grafo, sendo ele conexo ou não.

3.2.5 Matriz $M25$

Para o caso da matriz $M25$, o resultado obtido é:

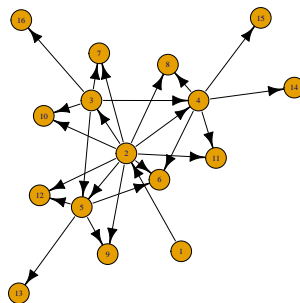


Figura 3.20: Grafo M25, pesquisa em profundidade

Apresento a baixo a lista dos vértices juntamente com a ordem pela qual foram visitados, de forma a poder visualizar melhor o resultado obtido uma vez que o grafo é mais complexo.

1	1	[1]	2	[2]	3	[3]	4	[4]	5	[5]	10	[6]
---	---	-----	---	-----	---	-----	---	-----	---	-----	----	-----

2	11	[7]	12	[8]	13	[9]	14	[10]	15	[11]	16	[12]
3	9	[13]	7	[14]	8	[15]	6	[16]				

sendo o primeiro valor o (id) do vértice e o segundo a ordem pela qual foi visitado (a lista está ordenada pela ordem crescente da visita dos vértices).

Analisando o ficheiro *log_dfs.txt* (em anexo) conseguimos ver o procedimento feito pelo algoritmo.

3.2.6 Matriz *M50*

Para o caso da matriz *M50*, o resultado obtido é:

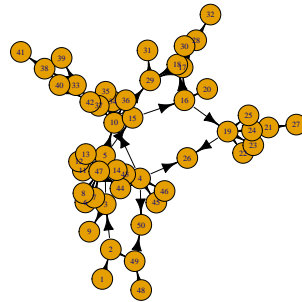


Figura 3.21: Grafo M50, pesquisa em profundidade

À semelhança do exemplo anterior, apresento a baixo a lista dos vértices juntamente com a ordem pela qual foram visitados.

1	1	[1]	2	[2]	3	[3]	4	[4]	5	[5]	7	[6]
2	10	[7]	13	[8]	29	[9]	6	[10]	9	[11]	12	[12]
3	14	[13]	15	[14]	18	[15]	19	[16]	20	[17]	21	[18]
4	24	[19]	37	[20]	25	[21]	38	[22]	39	[23]	40	[24]
5	41	[25]	43	[26]	42	[27]	22	[28]	23	[29]	36	[30]
6	35	[31]	34	[32]	26	[33]	44	[34]	46	[35]	47	[36]
7	48	[37]	27	[38]	45	[39]	28	[40]	49	[41]	50	[42]
8	8	[43]	11	[44]	30	[45]	31	[46]	32	[47]	16	[48]
9	17	[49]	33	[50]								

3.3 Exercício 3

Capítulo 4

Conclusão

4.1 Considerações futuras