



universidade de aveiro
theoria poiesis praxis

Trabalho 1

Seminário de Matemática Aplicada

Cláudio Henriques
Mestrado em Matemática e Aplicações

19 de Novembro de 2018

Conteúdo

Lista de Figuras	3
1 Introdução	4
1.1 <i>Java</i>	4
1.2 Estruturas de Dados	5
1.3 Algoritmos	5
1.4 Módulos extra	5
2 Implementação	6
2.1 Grafo (<i>Graph.java</i>)	6
2.2 Vértice (<i>Vertex.java</i>)	10
2.3 Coloração dos vértices, versão 1	13
2.3.1 Algoritmo	16
2.4 Coloração dos vértices, versão 2	16
2.4.1 Algoritmo	17
2.5 Coloração dos vértices, versão 3	17
2.5.1 Algoritmo	18
2.6 <i>Depth First Search</i> (DFS)	19
2.6.1 Algoritmo	19
3 Exercício 1	20
3.1 Análise e Resultados	20
4 Exercício 2	24
4.1 Análise e Resultados	24

5	Exercício 3	25
5.1	Análise e Resultados	25
6	Conclusão	26
6.1	Considerações futuras	26

Lista de Figuras

3.1	Grafo I, coloração 1	21
3.2	Grafo I, coloração 2	21
3.3	Grafo I, coloração 3	21
3.4	Grafo F, coloração 2	22
3.5	Grafo F, coloração 1	23
3.6	Grafo F, coloração 3	23

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 para o representar. É ainda apresentado um algoritmo de pesquisa em profundidade.

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.

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 abstracta 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 da dados sempre que necessitar de a utilizar em ambientes diferentes).

1.2 Estruturas de Dados

As estruturas de dados desenvolvidas ao longo do trabalho são:

- **Grafo** (*Graph.java*): representar um objeto do tipo grafo (recorrendo à sua matriz de adjacência);
- **Vértice** (*Vertex.java*): A estrutura anterior instância 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*)

1.3 Algoritmos

- **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.
- **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.
- **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.
- **Depth First Search (DFS)** (*DFS.java*):

1.4 Módulos extra

- **Ficheiro de logs** (*WriteLogFile.java*) Para auxiliar na descrição dos algoritmos foi criado um sistema de *logs*, isto é, quando o algoritmos efetua alguma operação, esta é registada num ficheiro do tipo **.txt*, detalhando cada passo, por ordem cronológica.

Capítulo 2

Implementação

Nesta secção é apresentado e explicado 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 aqui

2.1 Grafo (*Graph.java*)

Esta estrutura abstracta usa um *ArrayList* para representar o conjunto de vértices pertencentes ao objeto (*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 foram lidas 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.

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

Seguidamente são lidos 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. Logicamente 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 .


```
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.

```
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, cor, ordem, por um atributo que define se o vértice foi ou não visitado, por uma lista de vértices sucessores e uma lista de predecessores.

```
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 *getOrder* () retorna a ordem do vértice.

```
1 return 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.

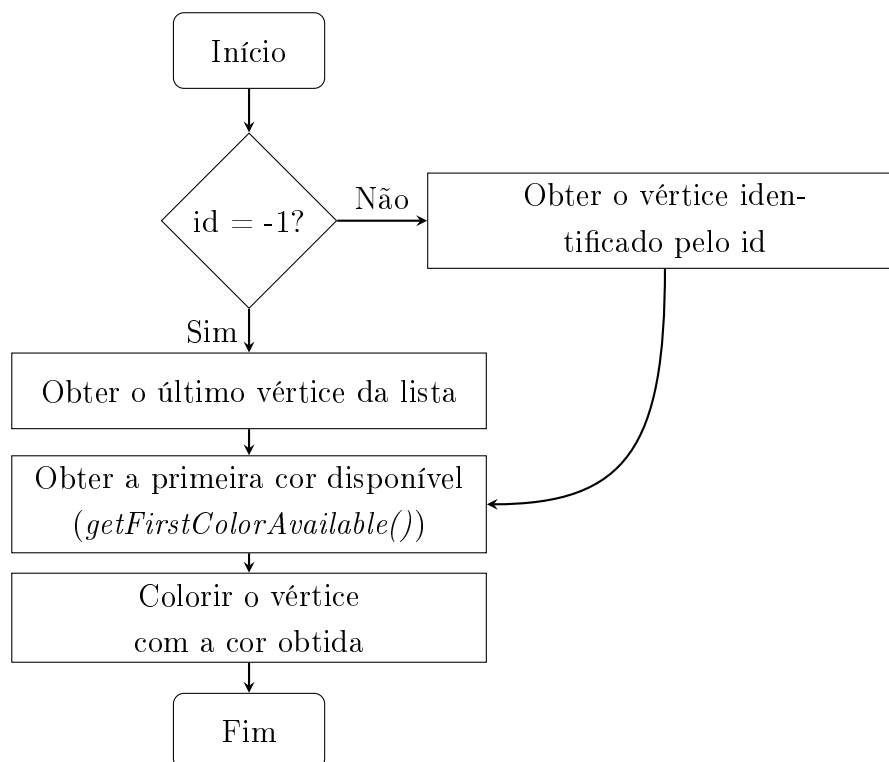
```
1     return getNeighborList().size();
```

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*.

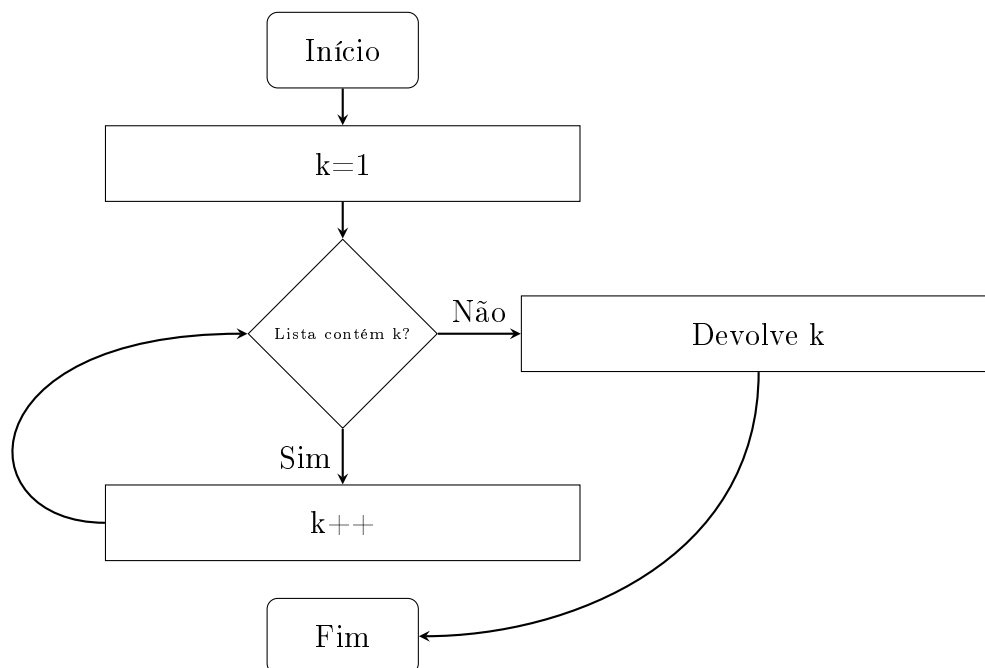


```

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.



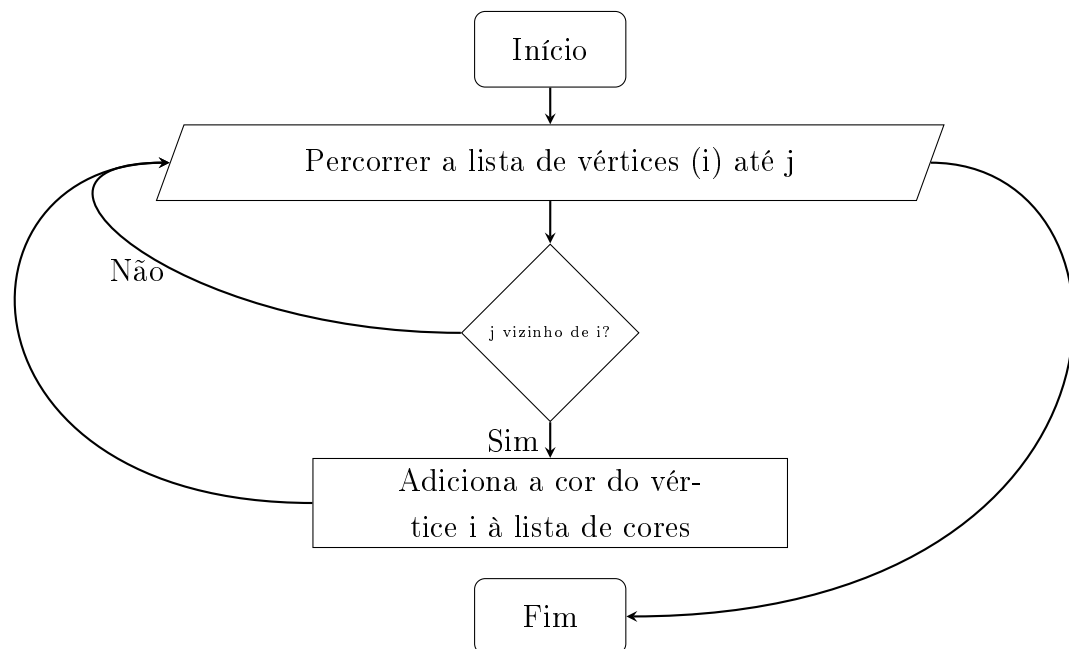
```

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.



```
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:

```
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 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:

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

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 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.

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

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*:

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

2.6 *Depth First Search* (DFS)

2.6.1 Algoritmo

Capítulo 3

Exercício 1

3.1 Análise e Resultados

Seja I a matriz
$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$
 e seja $graph$ um objeto do tipo *Graph.java*

construído com recurso à matriz I .

Implementamos o objeto *ColorizeVersion1.java* usando o grafo $graph$.

A coloração obtida é a representada no seguinte grafo:

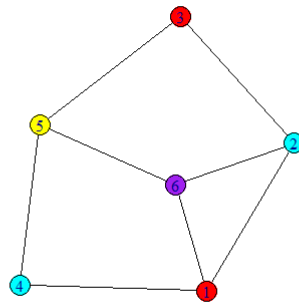


Figura 3.1: Grafo I, coloração 1

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

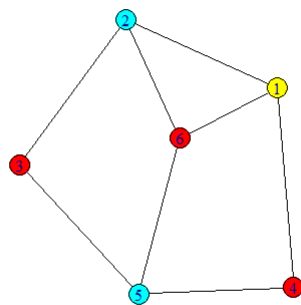


Figura 3.2: Grafo I, coloração 2

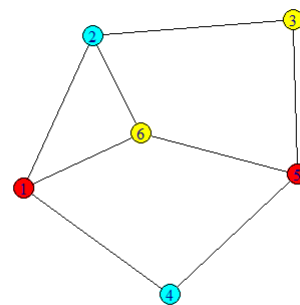


Figura 3.3: Grafo I, coloração 3

Analisando os resultados, 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.

Seja F a matriz

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

e seja $graph$ um objeto do

tipo *Graph.java* construído com recurso à matriz F .

Implementamos o objeto *ColorizeVersion2.java* usando o grafo $graph$.

A coloração obtida é a representada no seguinte grafo:

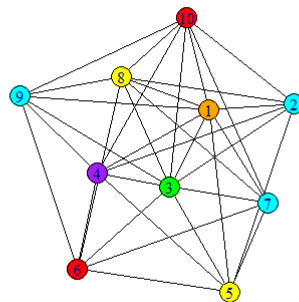


Figura 3.4: Grafo F, coloração 2

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

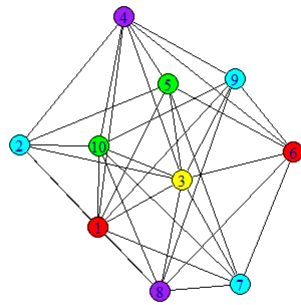


Figura 3.5: Grafo F, coloração 1

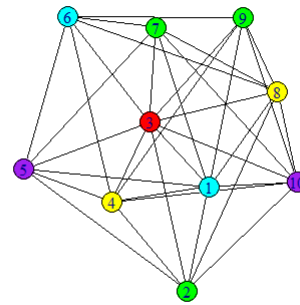


Figura 3.6: Grafo F, coloração 3

Analisando os resultados, 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.

Capítulo 4

Exercício 2

4.1 Análise e Resultados

Capítulo 5

Exercício 3

5.1 Análise e Resultados

Capítulo 6

Conclusão

6.1 Considerações futuras