UNIVERSIDADE FEDERAL DE VIÇOSA - *CAMPUS* FLORESTAL

Gabriel Vitor da Fonseca Miranda - mat.3857
Murillo Santhiago Souza Jacob - 4243
Pedro Augusto Maia Silva - mat.3878
Victória Caroline Silva Rodrigues - mat.3584

Trabalho Prático 2
Teoria e Modelo de Grafos

Trabalho prático da disciplina Teoria e modelos de grafos - CCF 331, do curso de Ciência da Computação da Universidade Federal de Viçosa - Campus Florestal.

Professor: Marcus Mendes

Florestal
2022

SUMÁRIO

1. INTRODUÇÃO

Este trabalho consiste em adicionar funcionalidades ao trabalho anteriormente realizado na modelagem e implementação de uma biblioteca com a finalidade manipular grafos não direcionados ponderados. As arestas do grafo terão valores reais associados para realizar as ponderações.

Esses dados serão fornecidos por meio de um arquivo de texto contendo o número de vértices e suas arestas com seus respectivos pesos, sendo que cada combinação de vértice inicial e vértice destino agora possui como característica o fato de ter o direcionamento, portando, no arquivo contendo a seguinte aresta “1 2”, indica que partindo do vértice 1 se chega ao vértice 2, mas o contrário não ocorre como ocorria anteriormente, mas apenas se for um grafo dirigido.

Além disso, deve-se projetar e implementar uma biblioteca que seja facilmente utilizada por outros programas, que seja capaz de representar grafos não-direcionados e algumas funcionalidades para grafos direcionados, além de fornecer um conjunto de algoritmos em grafos.

2. Desenvolvimento

Aqui, serão apresentados detalhes a respeito da implementação do código, de como foi pensado, implementado e aplicado as novas funcionalidades.

2.1 Implementação

A seguir será apresentado e explicado detalhadamente o funcionamento de cada uma das funções implementadas na biblioteca de grafos. Para melhor visualização, estes foram organizados em subtópicos.

2.1.1 Conjunto independente ou estável

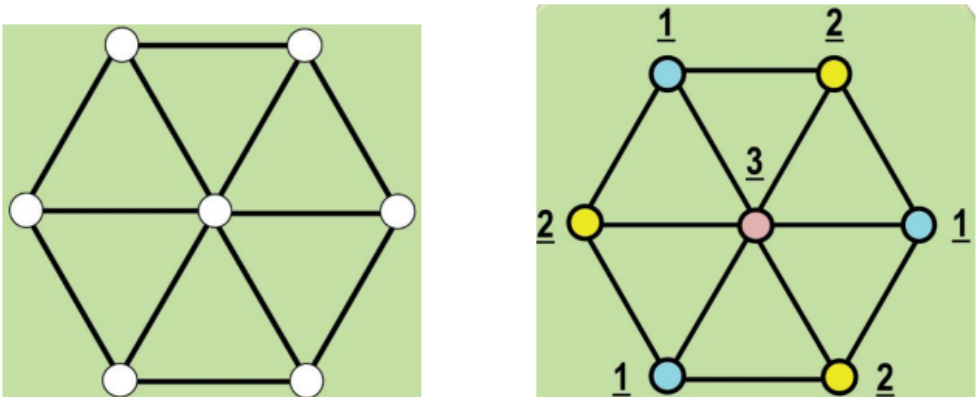
Um conjunto independente em um grafo não-dirigido é um conjunto de vértices dois a dois não adjacentes.Portanto, um conjunto S de vértices é independente se nenhuma aresta tem ambas as pontas em S. Os conjuntos independentes também são conhecidos como conjuntos estáveis.

Um conjunto independente em um grafo G é máximo se não existe outro maior em G.

Para acharmos o conjunto independente no nosso código, nós pegamos todos os vértices, e ordenamos a partir do vértice de maior grau. Após isso, percorremos esse vértice, e removemos ele e todos os vizinhos, e fazemos isso recursivamente até que não sobre nenhum vértice.

2.1.2 Algoritmo DSATUR

O algoritmo DSATUR foi inicialmente proposto por Brélaz (1979) como uma heurística construtiva na qual os vértices são coloridos de maneira iterativa. Vale ressaltar que para esse algoritmo ser utilizado o menor número possível de cores, como ilustrado na figura 1. Além disso, foi introduzido pelo autor o conceito de grau de saturação de um vértice V é o número de cores distintas que estão associadas aos vértices vizinhos a v.



Fonte[1] Ilustração da configuração do grafo antes e depois da coloração mínima de G.

Para implementação do algoritmo DSATUR utilizou-se o pseudocódigo apresentado em sala de aula.

Para iniciar o algoritmo DSATUR utilizou-se a função *DSATUR()*, responsável por receber o grafo G como entrada. Logo em seguida, é criado o vetor de cores para armazenar as cores, inicialmente vazio, e o vetor chamado vetorGrauVertices para adicionar os vértices do grafo juntamente com seu grau vértice e seu grau de saturação, que será ordenado de forma decrescente. Além disso, obtém-se o vértice de maior grau do vértice, logo depois, o vetor V recebe todos os vértices de G, exceto o vértice de maior grau, como ilustrado na figura 2. Por fim, é colorido o vértice de maior grau, iniciando com a cor 1.

```python
def DSATUR(self):
    #cor é representada pelo valor 1,...,ordem

    cor = []
    vetorGrauVertices = []
    for i in range(self.ordem):
        vetorGrauVertices.append({"vertice": i + 1, "grau": self.getGrau(i + 1),"saturacao":0})

    N = sorted(vetorGrauVertices, key=lambda row: row['grau'], reverse=1)

    maiorGrau = N[0]["vertice"]
    V = [N[i]["vertice"] for i in range(1,len(N))] #lista de vertices sem o de maior grau

    cor.append({"vertice":maiorGrau,"cor":1}) #coloriu o vertice de maior grau
```

Figura 2. Função DSATUR e suas operações de inicialização.

A seguir, enquanto houver vértices no conjunto V, ou seja, conjunto de vértices não coloridos deve-se repetir os seguinte passos enumerados abaixo apresentado abaixo:

1. Obter o vértice de maior grau de saturação de todos os vizinhos do vértice de maior grau com interseção em V, utilizando a função *calculaDESAT()*, que é responsável por verificar qual é o vértice de maior grau de saturação. Caso aconteça empate será escolhido o vértice de maior grau.
2. O vértice de maior grau é colorido com a menor cor disponível através da função *selecionarMenorCor()*, que é responsável por encontrar a menor cor disponível entre as cores já utilizadas pelos seus vizinhos.
3. Retira-se o vértice de colorido do conjunto V.

Após a realização desses procedimentos, deve-se utilizar a função *selecionaMaiorCor(),* que é responsável por encontrar o número mínimo de cores distintas possíveis. A figura 3 apresenta a ordem da implementação descrita anteriormente.

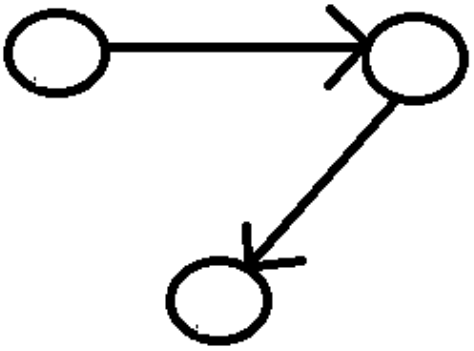
```python
while len(V) != 0:
    maiorGrau = self.calculaDESAT(maiorGrau,N,V)
    cor.append({"vertice":maiorGrau,"cor":selecionarMenorCor(cor,self.getVizinho(maiorGrau))})
    removeVerticeMaiorGrau(maiorGrau, V)

print(f"\n ******* O número de cores distintas deste grafo é :{selecionaMaiorCor(cor)} ********\n")
```

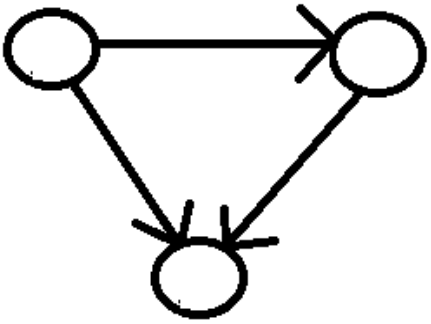
Figura 3. Função DSATUR e suas operações para encontrar o número cromático.

2.1.3 Grafo dirigido acíclico ou cíclico

A funcionalidade implementada denominada internamente de “haveCicle” exige a dependência de uma fila e uma lista que armazena os vértices visitados. Para compreender o seu funcionamento, podemos observar o seguinte grafo:



Dado este grafo, notamos que o mesmo não possui ciclo, visto que se começado a percorrer o grafo a partir de qualquer um dos vértices o vértice inicial não é retomado. A diferença entre este algoritmo e o “haveCicle” para grafos não dirigidos pode ser analisado nesta representação abaixo:



Embora possa parecer que o grafo tenha ciclo, se começar a percorrer o grafo, pode ser notado que a repetição de um vértice para se percorrer o grafo por completo não ocorre. Isso porque embora o vértice que aponta para outros dois vértices não tem nenhuma linha de retomada para o mesmo, o que indica a ausência de um ciclo.

Para realizar a análise então é percorrido o grafo enquanto a fila de valores não é vazia, sendo inserido inicialmente o primeiro vértice do grafo por meio da primeira posição da fila que então é removido o seu valor, sendo este inserido na lista de visitados. Assim, é obtido os vizinhos do vértice e então percorrido o grafo vizinho por vizinho, inserindo na fila os seus vértices e retomando ao início da execução para obter o próximo vértice da fila até que todos os vértices tenham sido analisados.

Desta maneira se caso um dos vértices vizinhos obtidos por meio do vértice removido da fila já esteja na lista de visitados o algoritmo retorna “true”, indicando que este grafo possui ciclo. Caso contrário, se nunca entrar nesta condição é retornado “false”, indicando

que este grafo não possui ciclo, como podemos analisar na imagem abaixo o código construído:

```
def haveCicle(self):
    fila = []
    visitado = []
    fila.append(1)
    while len(fila) != 0:
        value = fila[0]
        fila.pop(0)
        visitado.append(value)
        aD = self.vertices[value - 1].getVizinhos()
        for item in aD:
            item = int(item)
            if item in visitado:
                return True
            fila.append(item)

    return False
```

Figura 4. Função “haveCicle” e suas operações

Esta funcionalidade é usada para determinar se em um dado grafo será realizada a ordenação topológica, visto que não é possível de se determinar a ordenação topológica em um grafo dirigido cíclico.

2.1.4 Ordenação Topológica

Se o grafo é um digrafo acíclico (DAG), a solução está contida na lista L (a solução não é única). Caso contrário, o grafo tem pelo menos um ciclo e, portanto, uma ordenação topológica é impossível.

Em teoria dos grafos, uma ordenação topológica de um digrafo acíclico (DAG) é uma ordem linear de seus nós em que cada nó vem antes de todos nós para os quais este tenha arestas de saída. Cada DAG tem uma ou mais ordenações topológicas.

Mais formalmente, define-se a relação acessibilidade R sobre os nós do DAG tal que xRy se e somente se existe um caminho dirigido de x para y. Então, R é uma ordem parcial, e uma ordenação topológica é uma extensão linear desta ordem parcial, isto é, uma ordem total compatível com a ordem parcial. O grafo mostrado a baixo tem muitas ordenações topológicas, incluindo:

- 7, 5, 3, 11, 8, 2, 9, 10 (visual esquerda-para-direita, de-cima-para-baixo)
- 3, 5, 7, 8, 11, 2, 9, 10 (vértice de menor número disponível primeiro)

- 3, 7, 8, 5, 11, 10, 2, 9
- 5, 7, 3, 8, 11, 10, 2, 9 (menor número de arestas primeiro)
- 7, 5, 11, 3, 10, 8, 9, 2 (vértice de maior número disponível primeiro)
- 7, 5, 11, 2, 3, 8, 9, 10

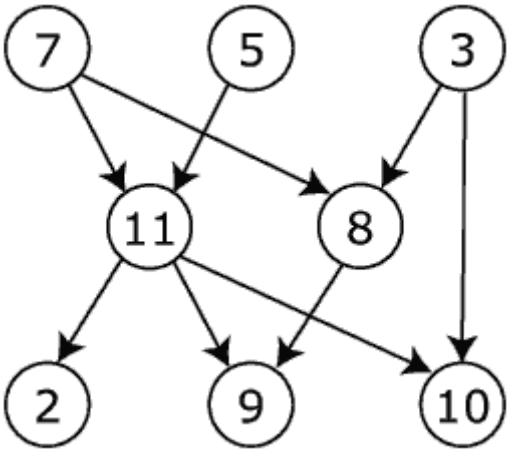


Figura 5. Grafo dirigido

Veja abaixo o algoritmo implementado pelo grupo:

```
def ordenacao_topologica(self):
    grafo1 = self.grafo #passando o grafo para um grafo Auxiliar
    R = [] #lista que terá a ordenação topológica
    N = [i for i in range(1, self.ordem + 1)] #lista N com o número de vértices
    for i in range(self.ordem):
        #Função que pega o vértice com menor quantidade de graus
        menor = self.pegarVerticeMenorGrau(grafo1, N)
        #adicionando o vertice com a menor quantidade de graus em R
        R.append(menor)
        #Removendo da lista N o vértice com menor grau
        removeDaLista(N,menor)
        #Removendo o vertice do grafo
        self.tirarMenorDag(grafo1, menor)
    #imprimindo a ordenação topológica
    print(R)
```

Figura 6. Ordenação Topológica

3. Conclusão

A seguir serão apresentados, respectivamente, os resultados encontrados a partir da execução das funções descritas na aba Implementação e as considerações finais a respeito deste trabalho prático.

3.1 Resultados

O código em questão foi testado através de grafos fornecidos dentro de sala de aula e obteve os resultados esperados. Na tabela 1, tem-se os resultados obtidos a partir da execução das funcionalidades presente no trabalho prático.

Funcionalidade	Entrada	Resultado
Cálculo do número cromático	5 1 2 1.2 2 5 2.3 3 5 8.4 3 4 0.3 4 5 4.6 1 5 0.1	2
Conjunto independente ou estável	5 1 2 1.2 2 5 2.3 3 5 8.4 3 4 0.3 4 5 4.6 1 5 0.1	[5] 1
Ordenação topológica	6 4 1 1 4 2 1 4 3 1 3 2 1 5 4 1 5 6 1 6 4 1	5, 6, 4, 1, 3,2

A seguir estão as representações dos grafos utilizados como entrada para realização dos testes apresentados anteriormente na tabela acima. Na figura 7 e 8, representação do grafo não dirigido utilizado no algoritmo de DSATUR e a heurística gulosa para determinar o conjunto independente, e o grafo dirigido utilizado na ordenação topológica respectivamente.

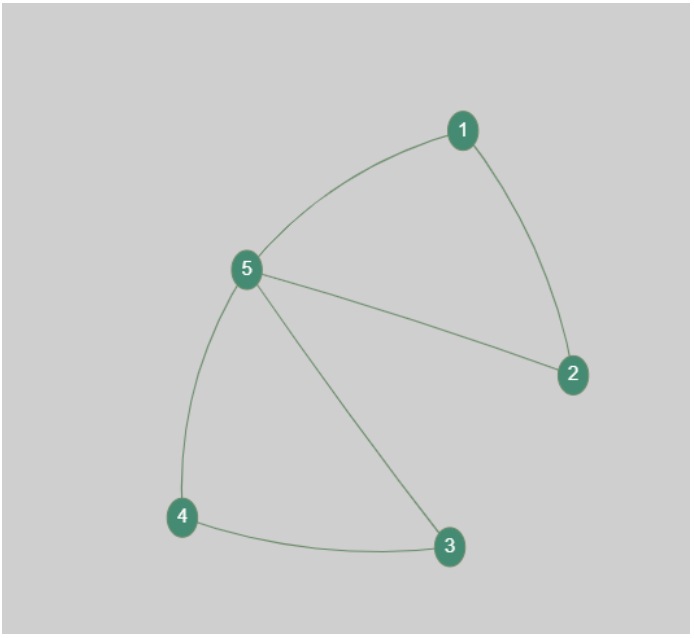


Figura 7. Grafo não dirigido utilizado no teste de execução no algoritmo DSATUR e heurística gulosa para determinar o conjunto independente.

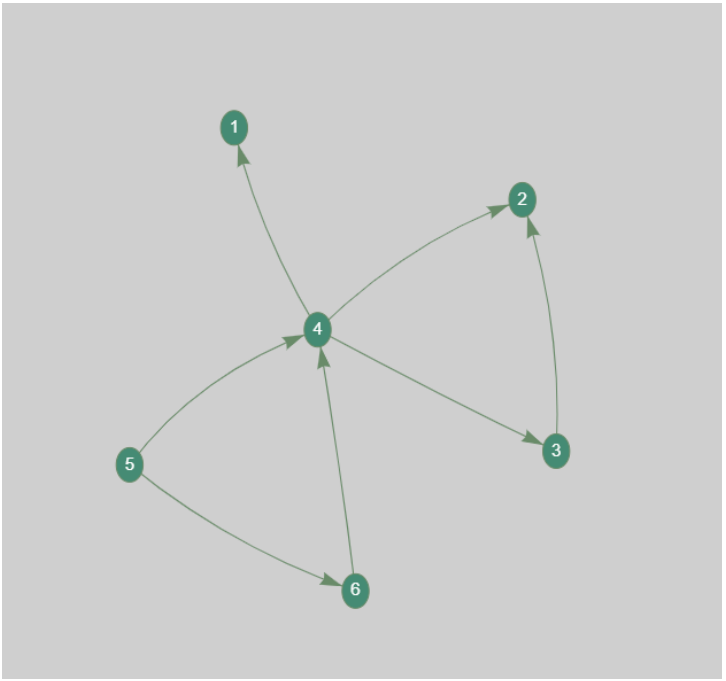


Figura 8. Grafo dirigido utilizado no algoritmo de ordenação topológica.

3.2 Considerações finais

Neste trabalho prático, foi explorado um tema de grande importância para o estudo da Teoria e modelos de grafos, que são os próprios grafos, que são amplamente usados em matemática, mas sobretudo em programação. Formalmente, um grafo é uma colecção de

vértices (V) e uma colecção de arcos (E) constituídos por pares de vértices. É uma estrutura usada para representar um modelo em que existem relações entre os objectos de uma certa coleção.

O objetivo principal deste trabalho prático foi adicionar mais funcionalidades na biblioteca de grafos que já tinha sido criada. As novas funcionalidades que foram adicionadas foram: Determinar o conjunto independente ou estável de um grafo por meio de uma heurística gulosa, determinar o número cromático de um grafo, utilizando o algoritmo DSATUR. E verificar se um grafo é acíclico ou não, e caso ele seja acíclico determinar sua ordenação topológica. Todas essas funcionalidades foram implementadas na nossa biblioteca de grafos.

Diante disso, alguns desafios foram encontrados durante o desenvolvimento desta tarefa, como encontrar uma solução para a ordenação topológica, verificar se um grafo dirigido era cíclico ou não, e entre outras funcionalidades. Porém o grupo teve um entendimento rápido para fazer estas tarefas.

Portanto, esta abordagem mais prática para implementar as novas funcionalidades da nossa biblioteca de grafos ajudou a entender melhor esse conteúdo da matéria e melhorou ainda mais os conhecimentos adquiridos em aula sobre os grafos, pois fez com que conceitos apresentados durante a disciplina fossem utilizados para a implementação desta estrutura.

4. Referências

[1]Marco C. Goldbarg; E. Goldbarg. Grafos: Conceitos, algoritmos e aplicações. Elsevier: campus, 2012

[2] Araujo, Miguel. Grafos 1º Parte. Revista Programar, 2007. Disponível em: Link. Acesso em: 12 de Mar. de 2022.

[3]Ordenação topológica. Wikipédia, 2018. Disponível em: Link. Acesso em: 15 de Mar. de 2022.

[4] Marie, Alane. Algoritmos Exatos Para o Problema Da Coloração de Grafos. Universidade Federal do Paraná. Disponível em: Link. Acesso em: 14 de Mar. de 2022.

[5] Carvalho, Lucas. PAAD GRAFOS. 2019. Disponível em: Link. Acesso em: 13 de Mar. de 2022.