Trabalho Final

Para o desenvolvimento do nosso trabalho, utilizamos a linguagem `python`, que tem um conjunto de bibliotecas que nos ajudaram bastante na resolução dos problemas propostos.

O trabalho foi desenvolvido com orientação a objetos, por isso, teremos três classes a serem analisadas: `main` , `grafo` e `menu`.

Neste relatório iremos descrever o que cada uma das classes listadas possuem e qual é a função de cada uma[1].

Main

1. Inicialmente foi criada uma `main` que basicamente chama as outras classes que farão todo o resto.

```
1  #! python3
2  # coding: utf-8
3
4
5  from Classes.grafo import Grafo
6  from Classes.menu import Menu
7
8  grafo = Grafo()
9  m = Menu()
10 m.mostra_menu(grafo)
```

Menu

1. A classe menu é uma classe que o próprio nome já diz, mostra o menu para o usuário.

```
1  import subprocess as sp
2  from PIL import Image
3
4
5  class Menu:
6      Classe menu
7
8      def __init__(self):
9          self.vertice = 0
10         self.opcao = 0
```

   Primeiramente importamos as libs `subprocess` e `PIL` que tem por finalidade (nesse código) limpar a tela a cada ENTER que o usuário der e mostrar a imagem gerada pelo método `grafo_visual()` da classe `Grafo` (será explicada mais adiante) respectivamente.

   Depois criamos a classe `Menu` e chamamos o construtor `def __init__(self)` que serve para iniciarmos os atributos globais da nossa classe.

[1] Algumas funções foram ocultadas para evitar que o relatório ficasse massivo.

2. Agora criamos o menu onde mostrará as opções para o usuário escolher:

```
    def mostra_menu(self, grafo):
        Mostrar o menu com todas as opcoes na tela

        while self.opcao != -1:
            grafo.mostra_dados(grafo.matriz)
            print('\t', end='')
            print('='*34)
            print('\t=', f'{Menu:^30}', '=')
            print('\t', end='')
            print('='*34)
            print(f'\t| {01 - Inserir vertice:<30}', '|')
            print(f'\t| {02 - Inserir aresta:<30}', '|')
            print(f'\t| {03 - Seu grafo é completo?:<30}', '|')
            print(f'\t| {04 - Gerar grafo complementar:<30}', '|')
            print(f'\t| {05 - Número de componentes:<30}', '|')
            print(f'\t| {06 - Seu grafo é uma árvore?:<30}', '|')
            print(f'\t| {07 - Busca em largura:<30}', '|')
            print(f'\t| {08 - Busca em profundidade:<30}', '|')
            print(f'\t| {09 - Algoritmo de Dijkstra:<30}', '|')
            print(f'\t| {10 - Gerar grafo aleatório:<30}', '|')
            print(f'\t| {11 - Visualizar grafo:<30}', '|')
            print(f'\t| {-1 - Sair:<30}', '|')
            print('\t', end='')
            print('='*34)
```

Feito isso um menu como este será mostrado na tela:

```
                ==================================
                =               Menu             =
                ==================================
                | 01 - Inserir vertice           |
                | 02 - Inserir aresta            |
                | 03 - Seu grafo é completo?     |
                | 04 - Gerar grafo complementar  |
                | 05 - Número de componentes     |
                | 06 - Seu grafo é uma árvore?   |
                | 07 - Busca em largura          |
                | 08 - Busca em profundidade     |
                | 09 - Algoritmo de Dijkstra     |
                | 10 - Gerar grafo aleatório     |
                | 11 - Visualizar grafo          |
                | -1 - Sair                      |
                ==================================
                Opção:
```

O usuário escolherá uma opção e os `if's` e `else's` farão o resto.

Grafo

1. Nosso coração do trabalho com certeza é esta classe. Nela está presente tudo o que precisa para funcionar perfeitamente.

```
1  from random import randint
2  import glob, os
3  import imageio
4  import pygraphviz as pgv
5
6
7  class Grafo:
8      def __init__(self):
9          self.vertices = []
10         self.matriz = [[0]*0 for i in range(0)]
11         self.predecessor = [None] * 0
12         self.time = 0
```

Primeiramente importamos várias libs que contribuíram bastante para o desenvolvimento deste projeto. Dentre elas a:

- `randint`: escolher números aleatórios para os pesos das arestas no método `gera_dijkstra()`.
- `glob e os`: são libs para comparação de `path names` e funcionalidades do sistema operacional, respectivamente.
- `imageio`: responsável pela criação dos `gifs` dos grafos.
- `pgv`: a cereja do bolo. Essa lib que nos proporcionou a criação visual dos grafos gerados.

Depois criamos a classe `Grafo` e chamamos o construtor `def __init__(self)` que serve para iniciarmos os atributos globais da nossa classe.

Métodos

Nós criamos vários métodos na classe Grafo. Detalharemos cada um a partir daqui.

Mostrar os Dados

Método simples que tem a função de mostrar a matriz (grafo) na tela.

```
1      def mostra_dados(self, matriz):
2          linha = coluna = len(matriz)
3          print()
4          for l in range(linha):
5              for c in range(coluna):
6                  print(f'\t{matriz[l][c]}', end='')
7              print()
```

Vértices Existentes

Retorna `True` se houver vértice na lista. A lista chama o método `count()` que retorna o número de ocorrências do valor passado por parâmetro. Se for igual a 0 significa que não há vértice no grafo.

```
1    def existe_vertice(self, v):
2        return self.vertices.count(v) != 0
```

Inserir Vértice

Função para adicionar vértice ao grafo. Cria-se uma matriz auxiliar adicionando uma coluna e uma linha a mais e recebe cada posição da matriz principal. Retorna True se não houver nenhum vértice igual

```
1    def inserir_vertice(self, v):
2        self.vertices.append(v)
3        linha = coluna = len(self.matriz)
4        matriz_aux = [[0]*(linha+1) for i in range(coluna+1)]
5
6        for l in range(linha):
7            for c in range(coluna):
8                matriz_aux[l][c] = self.matriz[l][c]
9        self.matriz = matriz_aux
```

Inserir Aresta

Função para adicionar arestas num par de vértices. É atribuído o valor 1 na posição da matriz determinada por inicio e fim.

```
1    def inserir_aresta(self, inicio, fim):
2        if not(self.existe_vertice(inicio)) or not(self.existe_vertice(
             fim)):
3            return False
4        if inicio == fim:
5            print('\n\t\tApenas grafos simples. Tente novamente')
6            return False
7
8        else:
9            self.matriz[self.vertices.index(inicio)][self.vertices.index(
                 fim)] = 1
10           self.matriz[self.vertices.index(fim)][self.vertices.index(
                 inicio)] = 1
11       return True
```

Checar se o Grafo é Completo

Checa se um grafo é completo ou não. Simplesmente verifica se há um 0 na metade triangular da matriz.

```
1    def checa_completo(self):
2        linha = coluna = len(self.matriz)
3
4        for l in range(linha):
5            for c in range(l+1, coluna):
6                if self.matriz[l][c] == 0:
7                    return False
8        return True
```

Grafo Complementar

Cria o grafo complementar da matriz. Percorre a matriz e verifica onde for 0, menos na diagonal principal, troca por 1, e onde for 1 troca por 0.

```
1    def complementar(self):
2        linha = coluna = len(self.matriz)
3        matriz_aux = [[0]*(linha) for i in range(coluna)]
4
5        for l in range(linha):
6            for c in range(coluna):
7                matriz_aux[l][c] = self.matriz[l][c]
8                if l != c and matriz_aux[l][c] == 1:
9                    matriz_aux[l][c] = 0
10               elif l != c:
11                   matriz_aux[l][c] = 1
12       print()
13       self.mostra_dados(matriz_aux)
```

Número de Componentes

Função para contar o numero de componentes do grafo. A variável componentes recebe o valor dos predecessores da função de busca em largura e retorna a quantidade de None's encontrados.

```
1    def num_componentes(self):
2        componentes, _, _ = self.dfs()
3
4        return componentes.count(None)
```

Busca em Profundidade

Função retirada do conteúdo passado em sala.

```
        for u in self.vertices:
            if cor[u] in 'BRANCO':
                self.dfs_util(u, cor, d, f)

        self.time = 0

        return self.predecessor, d, f


    def dfs_util(self, u, cor, d, f):
        linha = coluna = len(self.matriz)
        cor[u] = 'CINZA'
        self.time += 1
        d[u] = self.time

        for v, i in enumerate(self.matriz[u]):
            if i == 1:
                if cor[v] in 'BRANCO':
                    self.predecessor[v] = u
                    self.dfs_util(v, cor, d, f)
        cor[u] = 'PRETO'
        self.time += 1
        f[u] = self.time

        return self.predecessor, d, f, cor


    def gera_dijkstra(self):
        linha = coluna = len(self.matriz)
        matriz_dijkstra = [[0]*(linha) for i in range(coluna)]
```

Checa se o Grafo é uma Árvore

Função para verificar se o grafo é uma arvore. Primeiro verifica se há apenas 1 componente, se não já retorna False e depois verifica se há ciclo.

```
    def checa_arvore(self):
        if self.num_componentes() > 1:
            return False
        linha = len(self.matriz)
        visitado = [False] * linha

        for i in range(linha):
            if visitado[i] == False:
                if self.isCicle(i, visitado, -1):
                    return False
        return True
```

Algoritimo de Dijkstra

Cria o grafo baseado em pesos das arestas. Percorre toda a matriz e verifica, onde for 1 o valor é trocado por um peso aleatório de 1 a 10.

```
        self.mostra_dados(matriz_dijkstra)
        linha = len(matriz_dijkstra)
        distancia = [999] * linha
        self.predecessor = [None] * linha
        distancia[inicio] = 0
        visitado = [False] * linha

        for i in range(linha):
            u = self.minDistancia(distancia, visitado)
            visitado[u] = True
            for v in range(linha):
                if matriz_dijkstra[u][v] > 0 and visitado[v] == False and
                    \
                    distancia[v] > distancia[u] + matriz_dijkstra[u][v]:
                    distancia[v] = distancia[u] + matriz_dijkstra[u][v]
                    self.predecessor[v] = u
        self.mostra_dijkstra(distancia, inicio, fim)


    def mostra_dijkstra(self, distancia, inicio, fim):
        Mostra o resultado de todo o algoritmo de Dijkstra.
```

Gera Grafo Aleatório

Função para gerar um grafo aleatório. Seta a matriz aleatória numa metade e depois espelha na outra.

```
            for j in range(i+1, tamanho):
                matriz[i][j] = randint(0,1)
                matriz[j][i] = matriz[i][j]

        linha = coluna = len(matriz)
        print()
        self.mostra_dados(matriz)


    def grafo_visual(self):
        interface_grafo = pgv.AGraph()
```

Visualizar Grafo

Função coringa do nosso trabalho. Conseguimos a partir da biblioteca pygraphviz criar um grafo e adicioná-lo a um arquivo de imagem para visualização. Primeiramente criamos as características do nosso grafo (resolução, cor de fundo, tipo de forma dos vértices, estilo e cor dos vértices) e depois adicionamos os vértices, as arestas, geramos o layout para poder esboçar o grafo.

Criamos um arquivo .dot que é o tipo de arquivo que se cria grafos com graphviz e então criamos a imagem para visualização, todos eles salvo no diretório Imagens na raiz do projeto.

```
        interface_grafo.graph_attr['bgcolor'] = 'mediumturquoise'
        interface_grafo.node_attr['shape']  = 'circle'
        interface_grafo.node_attr['style']  = 'filled'
        interface_grafo.node_attr['fillcolor']  = 'white'
        interface_grafo.node_attr['fontcolor']  = 'white'

        linha = coluna = len(self.matriz)

        if linha == 0:
            return False

        [interface_grafo.add_node(x, xlabel = f'V{x}') for x in range(
            linha)]

        for l in range(linha):
            for c in range(l+1, coluna):
                if self.matriz[l][c] == 1:
                    interface_grafo.add_edge(l, c)

```

```
19          interface_grafo.layout(prog='dot')
20          interface_grafo.write('/home/matheus/Documentos/Faculdade/5
                Periodo Computacao' +
21          '/Teoria dos Grafos/TrabalhoFinal/Imagens/grafo.dot')
22
23          interface_grafo.draw('/home/matheus/Documentos/Faculdade/5
                Periodo Computacao/' +
24          'Teoria dos Grafos/TrabalhoFinal/Imagens/grafo.png')
25
26          return interface_grafo
```

Busca em Largura

Com certeza essa foi a função mais impressionante que fizemos. Nela conseguimos (além executar o algoritmo de busca em largura perfeitamente) criar um algoritmo que gera gifs animados a partir do grafo que o próprio usuário montou. Ela faz todo o processo de coloração de vértices e arestas, além disso, mostra a distância necessária para chegar a cada vértice a partir do ponto de partida escolhido pelo usuário.

Utilizamos o algoritmo padrão mostrado em sala para executar a busca em largura. Já para criar os gifs utilizamos o suporte das libs já listadas anteriormente (pgv, glob, os, PIL e imageio).

```
1       def busca_largura(self, indice):
2           if not(self.existe_vertice(indice)):
3               return False
4
5           linha = len(self.matriz)
6           f = 10
7
8           # Criando grafo para executar a busa em largura
9           grafoV = self.grafo_visual()
10          node = grafoV.get_node(indice)
11          node.attr['fillcolor'] = 'gray'
12
13          for i in range(linha):
14              node = grafoV.get_node(i)
15              if i == indice:
16                  node.attr['label'] = '0'
17              else:
18                  node.attr['label'] = '   '
19              node.attr['fontcolor'] = 'crimson'
20
21          filename = f'grafo{f}.png'
22          grafoV.layout(prog='dot')
23
```

```
24          grafoV.draw('/home/matheus/Documentos/Faculdade/5   Periodo
                Computacao/' +
25          'Teoria dos Grafos/TrabalhoFinal/Imagens/' + filename)
26
27          cor = ['BRANCO'] * linha
28          distancia = [999] * linha
29          self.predecessor = [None] * linha
30
31          cor[indice] = 'CINZA'
32          distancia[indice] = 0
33          self.predecessor[indice] = None
34          fila = []
35          fila.append(indice)
36
37          while len(fila):
38              u = fila.pop(0)
39              node = grafoV.get_node(u)
40              f += 1
41
42              for v, i in enumerate(self.matriz[u]):
43                  if i == 1:
44                      if cor[v] in 'BRANCO':
45                          cor[v] = 'CINZA'
46                          distancia[v] = distancia[u] + 1
47                          self.predecessor[v] = u
48                          fila.append(v)
49
50                          f += 1
51                          node = grafoV.get_node(v)
52                          edge = grafoV.get_edge(u, v)
53                          node.attr['fillcolor'] = 'gray'
54                          node.attr['label'] = f'{distancia[v]}'
55                          edge.attr['color'] = 'crimson'
56
57                          filename = f'grafo{f}.png'
58                          grafoV.layout(prog='dot')
59
60                          grafoV.draw('/home/matheus/Documentos/Faculdade/5
                                Periodo Computacao/' +
61                          'Teoria dos Grafos/TrabalhoFinal/Imagens/' +
                                filename)
62
63              f += 1
64              cor[u] = 'PRETO'
65              node = grafoV.get_node(u)
66              node.attr['fillcolor'] = 'black'
67
68              filename = f'grafo{f}.png'
```

```
            grafoV.layout(prog='dot')

            grafoV.draw('/home/matheus/Documentos/Faculdade/5   Periodo
                Computacao/' +
            'Teoria dos Grafos/TrabalhoFinal/Imagens/' + filename)


        print('\n\tPredecessores:\n')
        for i in range(linha):
            print(f'\tV{i}', end='')
        print()
        for pi in self.predecessor:
            print(f'\t{pi}', end='')
        print('\n')

        path = ('/home/matheus/Documentos/Faculdade/5   Periodo
            Computacao/' +
        'Teoria dos Grafos/TrabalhoFinal/Imagens/')

        files = [f for f in glob.glob(path + *.png)]

        files.sort()
        images = []
        kargs = { 'duration': 2 }
        for filename in files:
            images.append(imageio.imread(filename))
            os.remove(filename)

        imageio.mimsave('/home/matheus/Documentos/Faculdade/5   Periodo
            Computacao/' +
            'Teoria dos Grafos/TrabalhoFinal/Imagens/grafo.gif', images,
                'GIF', **kargs)

        return True
```

Conclusão

Esse trabalho foi desafiador. Aprendemos coisas que nem sabíamos que existiam e que eram possíveis sem ser tão complexo assim. As bibliotecas usadas foram de enorme ajuda e sem elas não seria possível fazer nem 1/3 do que pretendíamos.

Referências

StackOverflow.com