Algoritmos em Grafo

Relatório Trabalho 1

Prof. Glauco Amorim - 20 de junho de 2017

José Américo
Rafael Borges
Solon Oliveira

Introdução

Este relatório apresenta de forma detalhada as opções de desenvolvimento relacionados ao Trabalho Final 1 de Algoritmos em Grafo.

O Trabalho foi desenvolvido pelos alunos José Américo, Rafael Borges e Solon Oliveira.

Escolhemos utilizar Python como linguagem de programação para o desenvolvimento da biblioteca, apesar de ser uma linguagem não tão eficiente em desempenho ela oferece outros benefícios durante o desenvolvimento como não ser muito prolixa em sua sintaxe e podermos trabalhar com ela tanto de forma funcional como de forma orientada a objetos.

Neste relatório apresentaremos a estrutura básica que definimos para o objeto Grafo, a leitura de grafos em arquivo, as informações do grafo em arquivo, a possibilidade de representar os grafos tanto como matriz de adjacências ou como lista de adjacências, busca em largura e profundidade e a identificação de componentes conexos dentro do grafo.

Objeto Grafo

Apesar de diretamente ligados, a instanciação do objeto grafo e a leitura do arquivo que contém o grafo, acabamos por optar em utiliza essas funções separadamente.

Ao criar um novo grafo, teremos as seguintes atribuições no objeto:

```
class Grafo():
        n = 0
        m = 0
        d = 0
        arestas = {}
        pesos = {}
        distribuicao_graus = {}
```

Onde:

- n : indica o número total de vértices do grafo
- m : indica o número total de arestas do grafo
- d : indica o desvio médio de graus
- arestas: um dicionário com a relação das arestas do grafo
- pesos: um dicionário com a relação entre as arestas e peso (quando houver)
- distribuição_graus: um dicionário com a distribuição dos graus dos vértices

Leitura de um Grafo em Arquivo

Ao ter um objeto grafo, podemos chamar a função *ler_arquivo* que recebe um *arquivo* como atributo da função. Este arquivo deve ter um grafo conforme descrito no trabalho proposto onde a primeira linha do arquivo indica o número de vértices do grafo, e as linhas seguintes indicam a relação de arestas e o seu peso (que é opcional no arquivo).

Nosso desenvolvimento verifica o arquivo e identifica se o grafo presente no arquivo possui ou não peso e atribui todas as informações as propriedades do grafo.

```
    def ler_arquivo(arquivo):
        self.n = int(arquivo.readline())
        for line in arquivo:
            s = line.split()
            if len(s) == 3:
                u, v, p = s
                key = str(u) + '-' + str(v)
                if u not in self.arestas:
                    self.arestas[u] = [v]
                    self.pesos[key] = [p]
                    self.m = self.m + 1
                else:
```

```
                self.arestas[u].append(v)
                self.pesos[key] = [p]
                self.m = self.m + 1
            else:
                u, v = s
                if u not in self.arestas:
                    self.arestas[u] = [v]
                    self.m = self.m + 1
                else:
                    self.arestas[u].append(v)
                    self.m = self.m + 1
            if v not in self.arestas:
                self.arestas[v] = [u]
            else:
                self.arestas[v].append(u)
        self.d = 2 * self.m/self.n
        self._graus_empiricos()


    def _graus_empiricos():
        for m in self.arestas:
            grau = len(self.arestas[m])
            if grau not in self.distribuicao_graus:
                self.distribuicao_graus[grau] = 1
            else:
                self.distribuicao_graus[grau] = self.distribuicao_graus[grau] + 1
```

A função faz a leitura linha a linha do arquivo e vai adicionando as informações extraídas ao objeto grafo. A função tendo identificado o *n* e *m* do grafo efetua o cálculo do *d* (desvio médio) e chama a função privada *_graus_empiricos* para atribuir a distribuição dos graus ao objeto grafo.

Informações do Grafo em Arquivo

Ao ler o arquivo com o grafo o objeto grafo terá seus atributos devidamente preenchidos. Conseguimos gerar um arquivo com as informações detalhadas deste grafo utilizando a função *escrever_arquivo* que recebe um *arquivo* como atributo para indicar onde deve ser escrito a informação detalhada:

```
    def escrever_arquivo(arquivo):
        texto = '# n = ' + str(self.n) + '\n'
        texto += '# m = ' + str(self.m) + '\n'
        texto += '# d_medio = ' + str(self.d) + '\n'
        for m in self.distribuicao_graus:
            grau = self.distribuicao_graus[m]/self.n
            texto += str(m) + ' ' + str(grau) + '\n'
        arquivo.write(texto)
```

A função irá utilizar o dicionário de distribuição de graus do objeto para calcular a distribuição empírica dos graus dos vértices.

Representação de grafos

As funções *lista* e *matriz* ao serem chamadas pelo objeto grafo irão representar o mesmo em lista de adjacência e matriz de adjacência respectivamente.

Estas duas funções possuem como **atributo opcional** um *arquivo* para escrita. Caso seja omitido este atributo ao ser chamada a função, a mesma irá imprimir o grafo da forma desejada no console, caso seja informado um arquivo como atributo o grafo será escrito no arquivo na forma indicada.

```
def lista(arquivo=None):
    texto = ''
    for u in self.arestas:
        texto += u + ' '
        for v in self.arestas.get(u):
            texto += '->' + v + ' '
        texto += '\n'
    if arquivo:
        arquivo.write(texto)
    else:
        print(texto)

def matriz(arquivo=None):
    texto = ''
    texto += '  '
    for u in self.arestas:
        texto += u + ' '
    texto += '\n'
    for u in self.arestas:
        texto += u + ' '
        for v in self.arestas:
            if v in self.arestas[u]:
                texto += '1' + ' '
            else:
                texto += '0' + ' '
        texto += '\n'
    if arquivo:
        arquivo.write(texto)
    else:
        print(texto)
```

Busca em Largura e Busca em Profundidade

Desenvolvemos as funções *bfs_arquivo* e *dfs_arquivo* que recebem um *arquivo* como atributo e o vértice de *início* por onde se deseja começar a busca. O início deve ser informado como texto (string).

Essa função irá chamar a respectiva busca, BFS ou DFS, e irá encontrar o pai de cada vértice e o seu nível na árvore, em seguida chamar a função privada *_texto_imprimir_arvore*

que irá retornar o texto que será escrito no arquivo, primeiramente a relação de vértice e o seu pai em seguida o vértice e o seu nível na árvore.

Nas duas funções de busca adicionamos um **contador de tempo** (em segundos) que irá imprimir no console o tempo de processamento executado pela busca que irá variar conforme o tamanho do grafo.

```
def _texto_imprimir_arvore(arvore, pais):
    texto = ''
    for chave in pais:
        texto += 'Vertice ' + str(chave) + ': pai = ' + str(pais[chave]) + '\n'
    texto += '\n'
    for chave in arvore:
        texto += 'Vertice ' + str(chave) + ': nível = ' + str(arvore[chave]) + '\n'
    return texto
```

Busca em Largura - BFS

```
    def bfs_arquivo(self, arquivo, inicio):
        pais, arvore = self.bfs(inicio)
        texto = _texto_imprimir_arvore(arvore, pais)
        arquivo.write(texto)

    def bfs(self, inicio):
        tempo1 = datetime.now()
        q = []
        visitado = []
        pais = {}
        arvore = {}
        count = 0
        q.append(inicio)
        arvore[inicio] = count
        visitado.append(inicio)
        while len(q) != 0:
            u = q.pop(0)
            count += 1
            for vertice in self.arestas[u]:
                if vertice not in visitado:
                    pais.update({vertice: u})
                    arvore[vertice] = count
                    visitado.append(vertice)
                    q.append(vertice)
        tempo2 = datetime.now()
        tempo = tempo2 - tempo1
        print(str(tempo.total_seconds()) + 's')
        return pais, arvore
```

Busca em Profundidade - DFS

```
    def dfs_arquivo(self, arquivo, inicio):
        pais, arvore = self.dfs(inicio)
        texto = _texto_imprimir_arvore(arvore, pais)
        arquivo.write(texto)

    def dfs(self, inicio):
        tempo1 = datetime.now()
        visitado = []
```

```
        arvore = {}
        pais = {}
        count = -1
        arvore[inicio] = count
        pai = inicio
        for vertice in self.arestas:
            if vertice not in visitado:
                if vertice in self.arestas.get(pai):
                    pais.update({vertice: pai})
                self._dfs_visita(vertice, visitado, count, arvore, pais)
        tempo2 = datetime.now()
        tempo = tempo2 - tempo1
        print(str(tempo.total_seconds()) + 's')
        return pais, arvore

    def _dfs_visita(self, vertice, visitado, count, arvore, pais):
        visitado.append(vertice)
        pai = vertice
        count += 1
        arvore[vertice] = count
        for vertice_aux in self.arestas[vertice]:
            if vertice_aux not in visitado:
                pais.update({vertice_aux: pai})
                self._dfs_visita(vertice_aux, visitado, count, arvore, pais)
```

Componentes Conexos

A função *conexo* que recebe o atributo *inicio* (em forma de string) irá chamar a busca em profundidade, e através da arvore gerada identifica todos os componentes conexos e imprime no console:

- O número de componentes conexos;
- O detalhe de cada componente: seu tamanho e os vértices que o compõe;
- Lista os componentes em ordem decrescente de tamanho.

```
    def conexo(inicio):
        pais, arvore = self.dfs(inicio)
        grafos = []
        grafo = []
        for vertice in arvore:
            if arvore[vertice] == 0:
                if len(grafo) > 0:
                    grafos.append(grafo)
                grafo = []
                grafo.append(vertice)
            else:
                grafo.append(vertice)
        grafos.append(grafo)
        grafos_aux = sorted(grafos, key=len, reverse=True)
        print('Número de componentes conexas deste grafo: ' + str(len(grafos)))
        for lista in grafos_aux:
            print(str(len(lista)) + ' vértices: ' + str(lista))
        print('Grafos Conexos em ordem decrescente:')
        print(grafos_aux)
```