Algoritmos em Grafo

Relatório Trabalho 2

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 2 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 o desenvolvimento da leitura dos pesos de um grafo em arquivo e a relação do menor caminho a ser percorrido dentro de um grafo que possua pesos ou não.

Grafos com Pesos

Como apresentado no Relatório 1, nossa função que faz a leitura de um arquivo com a representação de um grafo identifica se a aresta possui peso ou não e atribui este peso em um dicionário de chave/valor onde a chave é o conjunto de aresta e o valor o peso.

Por exemplo, se no arquivo com o grafo está representado a linha:

3 4 2

Será atribuído ao dicionário o seguinte conjunto:

'3-4': '2'

Caso não haja uma terceira coluna de dados no arquivo (que representa o peso) a função simplesmente não atribui nenhum valor ao dicionário *pesos*:

```
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()
```

Ao utilizar um dicionário (conjunto chave/valor) temos uma grande facilidade que este sistema permite que é o identificador único no formato de chave, assim, com uma única consulta efetuamos a chamada da chave para obter o valor dela.

Essa lista sendo criada no momento da leitura do grafo em arquivo facilita ao trabalharmos com o código que a usará, pois ela já existirá e não precisará ser processada

no momento do uso da função, não forçando o aumento de processamento da função executora.

Distância e Caminho Mínimo

Para descobrir o caminho mínimo e a distancia criamos a função *menor_caminho* que recebe como atributo o vertice *inicial* e **opcionalmente** você pode indicar o vertice *final*. Ambos devem ser indicados como uma string.

Caso não seja informado o vertice final a função imprimirá no console o caminho para percorrer todos os vértices quando não houver pesos, e quando houver pesos a função irá imprimir no console uma lista com todos os vértices e o menor custo para chegar em cada um deles.

```
def menor_caminho(self, inicio, fim=None):
    if len(self.pesos) > 0:
        self.dijkstra(inicio, fim)
    else:
        self.bfs_caminho(inicio, fim)
```

A função *menor_caminho* verifica se o grafo que a chamou possui informações no dicionário de pesos, caso possua irá chamar a função *dijkstra*, e caso contrário irá chamar a função *bfs_caminho*.

```
def dijkstra(inicio, fim=None):
    distancia = dict()
    anterior = dict()
    distancia[inicio] = 0
    Q = copy.deepcopy(self.arestas)
    def extract_min():
        min = None
        ret = None
        for key in distancia:
            if key in Q and ((distancia[key] < min) if min != None else True):
                min = distancia[key]
                ret = key
        if ret is not None:
            del Q[ret]
        return ret
    while Q:
        u = extract_min()
        for v in self.arestas[u]:
            par = str(u) + '-' + str(v)
            if par in self.pesos:
                alt = distancia[u] + int(self.pesos[par][0])
                if v not in distancia or alt < distancia[v]:
                    distancia[v] = alt
                    anterior[v] = u
    node_list = list()
    try:
```

```
            next = fim
            while True:
                node_list.insert(0,next)
                next = anterior[next]
        except:
            pass
        print('Relação vértice/menor custo possível: ' + str(distancia))
        if fim:
            print('Menor caminho: ' + str(node_list))

    def bfs_caminho(inicio, fim=None):
        q = []
        visitado = []
        arvore = {}
        caminho = []
        q.append(inicio)
        arvore[inicio] = 'raiz'
        visitado.append(inicio)
        while len(q) != 0:
            u = q.pop(0)
            for vertice in self.arestas[u]:
                if vertice not in visitado:
                    arvore[vertice] = u
                    visitado.append(vertice)
                    q.append(vertice)
                if fim:
                    if vertice == fim:
                        caminho.insert(0, vertice)
                        v = arvore.get(vertice)
                        while arvore.get(v) != 'raiz':
                            caminho.insert(0, v)
                            v = arvore.get(v)
                        caminho.insert(0, v)
                        print('Menor Caminho: ' + str(caminho))
                        return
        print('Caminho para todos os vértices: ' + str(visitado))
```