Biblioteca de grafos em python

Avaliação 2 - 2024.2

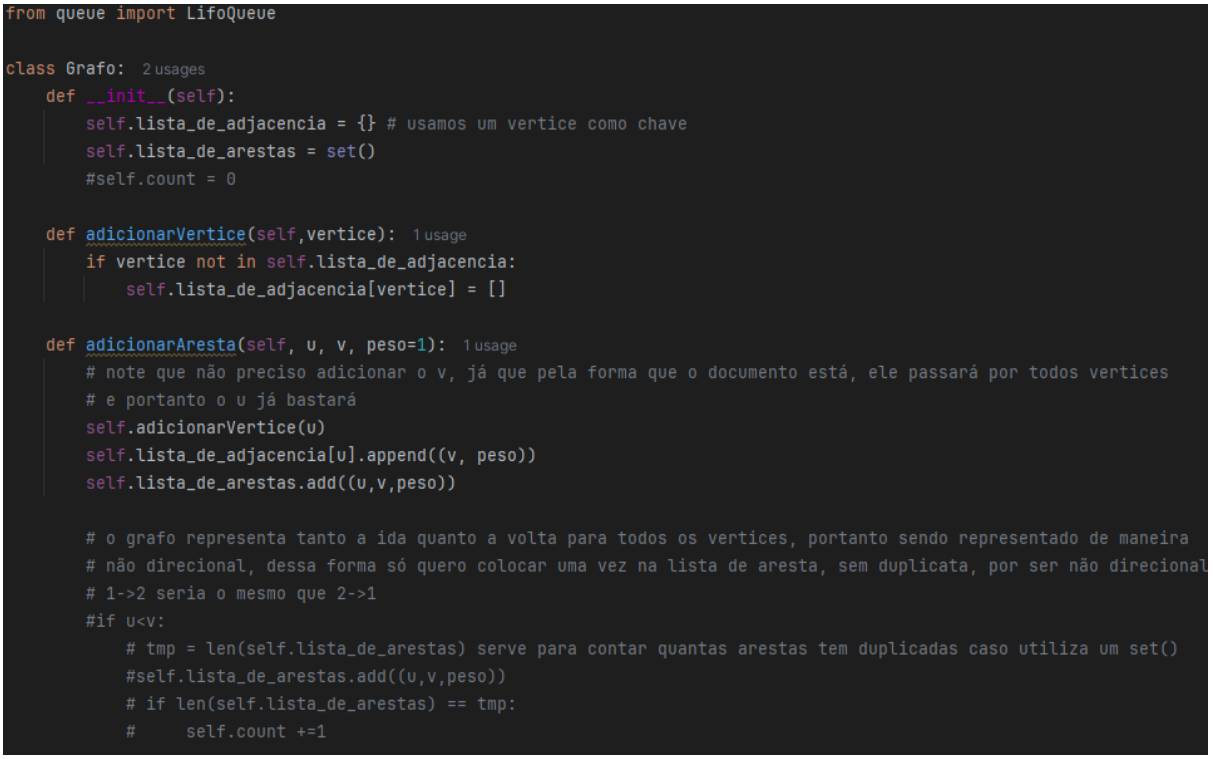
Guilherme Artur - 1500300
Emanuel da Silva Oliveira - 1647311

Primeiramente é necessário destacar algumas coisas:

- A representação do grafo, no documento, é feita com lista de adjacências, sendo formatado como um dicionário, onde a chave é o vértice, e o valor é uma lista de tuplas, onde o primeiro elemento da tupla é o vizinho, e o segundo elemento o peso da aresta que liga os dois. Ex: {1: [2,803]} -> vértice 1 tem como vizinho o vértice 2 e o peso da aresta é 803
- Não compreendi ao certo a separação de grafo não direcionado para dígrafo, isso por que o grafo dado para testes dá ao entender que é um grafo não direcionado, pois TODAS as arestas tem ida e volta entre os vértices conectados a ela. Portanto não sei ao certo, de que maneira considerar o dígrafo, qual direção eu escolheria? ou desconsidero isso e assumo que existem duas arestas(de ida e volta) entre dois vértices, sendo assim, ao computar como um dígrafo daria as 733.846 arestas, e ao computar como um grafo não direcionado(uma aresta para ida e volta) daria a metade disso 366923(considerando as duplicatas).
- **Portanto, minha escolha foi fazer o grafo não direcionado representado por 2 arestas para cada aresta não direcional(assim como está no documento), e o grafo direcionado sendo representado como 1 aresta, escolhendo a direção sempre por ordem de leitura do documento, ex: se tiver (1,2,803) irei descartar o (2,1,803).**
- Outro questão é, existem duplicatas no documento:
    - existem arestas que se repetem, mesmos vértices mesmo peso, se eu utilizo um set(que evita duplicatas) não são adicionadas todas arestas existentes
    - exemplos são arestas como: **(244450 244451 1663)** e **(264294 264296 132)**
    - **existem 3.746 arestas direcionadas duplicadas(1873 se considerar só uma aresta para ida e volta)**
    - **portanto são 730100 arestas direcionadas e 365050 se considerar só uma aresta para ida e volta**
- **TODOS OS TESTES ESTARÃO NO FINAL DO DOCUMENTO**

Grafo não direcionado:

Funções básicas:



```python
from queue import LifoQueue

class Grafo:  2 usages
    def __init__(self):
        self.lista_de_adjacencia = {} # usamos um vertice como chave
        self.lista_de_arestas = set()
        #self.count = 0

    def adicionarVertice(self,vertice):  1 usage
        if vertice not in self.lista_de_adjacencia:
            self.lista_de_adjacencia[vertice] = []

    def adicionarAresta(self, u, v, peso=1):  1 usage
        # note que não preciso adicionar o v, já que pela forma que o documento está, ele passará por todos vertices
        # e portanto o u já bastará
        self.adicionarVertice(u)
        self.lista_de_adjacencia[u].append((v, peso))
        self.lista_de_arestas.add((u,v,peso))

        # o grafo representa tanto a ida quanto a volta para todos os vertices, portanto sendo representado de maneira
        # não direcional, dessa forma só quero colocar uma vez na lista de aresta, sem duplicata, por ser não direcional
        # 1->2 seria o mesmo que 2->1
        #if u<v:
            # tmp = len(self.lista_de_arestas) serve para contar quantas arestas tem duplicadas caso utiliza um set()
            #self.lista_de_arestas.add((u,v,peso))
            # if len(self.lista_de_arestas) == tmp:
            #     self.count +=1
```

Eu só preciso adicionar um vértice por vez pois por existir sempre a ida e a volta, a variável ‘u’ irá abordar todos os vértices do grafo. Do mesmo modo não preciso colocar ‘v’ recebendo ‘u’ como vizinho**([v].append((u,peso))**, pois chegará esse momento naturalmente, na verdade, se eu fizesse isso, ficaria tudo duplicado dentro do dicionário. Tenho também duas opções:

1. Ou represento uma aresta não direcional com duas arestas direcionais(ida e volta)
2. Ou represento uma aresta não direcional com apenas uma aresta

isso implica em algumas funções como bellman-ford, tendo que alterar algumas coisas(acrescentar checagem ou retirar)

Mas como especifiquei antes, irei representar como a primeira opção

```python
# retorna o numero de vertices do grafo
def n(self):
    return len(self.lista_de_adjacencia)

# retorna o numero de arestas no grafo
def m(self):  1 usage
    return len(self.lista_de_arestas)

#retorna a lista de vizinhos do vertice
def viz(self,vertice):  1 usage
    return self.lista_de_adjacencia[vertice]

# retorna o peso da aresta, a tupla seria (vertice,peso)
def w(self,vertice1,vertice2):  1 usage
    for tupla in self.lista_de_adjacencia[vertice1]:
        if tupla[0] == vertice2:
            return tupla[1] # peso
    return 0 # esse vertice1 nao tem esse vizinho

# retorna o grau do vertice
def d(self, vertice):  4 usages
    return len(self.lista_de_adjacencia[vertice])
```

funções bem básicas, apenas retornando informações das listas/dicionário

```python
# retorna o grau do vertice
def d(self, vertice):  4 usages
    return len(self.lista_de_adjacencia[vertice])

# retorna o menor grau do grafo
def mind(self):
    menor_grau = self.d(1) # pego o grau do vertice 1 para iniciar
    menor_vertice = 1
    for i in self.lista_de_adjacencia:
        grau = self.d(i) # utilizo a nossa outra funçao para calcular o grau
        if grau < menor_grau:
            menor_grau = grau
            menor_vertice = i
    return menor_vertice,menor_grau

# retorna o maior grau do grafo, quase igual a função do menor, mudando apenas a comparação
def maxd(self):
    maior_grau = 0
    menor_vertice = 1
    for i in self.lista_de_adjacencia:
        grau = self.d(i)
        if grau > maior_grau:
            maior_grau = grau
            menor_vertice = i
    return menor_vertice,maior_grau
```

Nas funções de menor grau e maior grau eu coloquei para retornar tanto o vértice de menor/maior grau, como o grau em si, são funções quase idênticas, só muda a forma de comparar os valores.

DFS:

```python
def dfs(self, inicio):  2 usages
    visitados = set()  # conjunto para armazenar os vértices visitados
    pilha = LifoQueue()  # pilha que irá processar os vertices

    tempo_inicial = {}
    tempo_final = {}
    pai = {} # chave = cada vertice, contendo o valor pai
    tempo = 1  # contador de tempo

    # nossa pilha irá receber uma tupla, (vertice,indice do vizinho que ainda não foi visitado)
    # é necessário pois precisamos ter um controle em relação a quando um vertice foi de fato processado por completo
    # e assim podermos atribuir o tempo final corretamente
    # normalmente feito com cores, mas por ser uma pilha, e não uma recursão, achei mais facil assim
    pilha.put((inicio,0))
    visitados.add(inicio) # marca o vértice inicial como visitado
    tempo_inicial[inicio] = tempo
    tempo += 1
    pai[inicio] = None  # vértice inicial não tem pai
```

```
while not pilha.empty():
    vertice, indice_vizinho = pilha.get()  # pega o vértice e o indice do próximo vizinho a ser processado

    vizinhos = self.lista_de_adjacencia[vertice] # todos vizinhos do vértice

    if indice_vizinho < len(vizinhos): # checa se o indice atual do vértice é menor que o tamanho da lista
        vizinho, _ = vizinhos[indice_vizinho]
        pilha.put((vertice,indice_vizinho+1))

        # importante pois o vizinho já pode ter sido visitado, exemplo: pai do vértice já que o grafo é não direcional
        if vizinho not in visitados:
            visitados.add(vizinho)
            tempo_inicial[vizinho] = tempo
            tempo += 1
            # só queremos considerar o vértice antecessor ao caminho
            if vizinho not in pai:
                pai[vizinho] = vertice # vizinho recebe vértice antecessor a ele como pai

            pilha.put((vizinho,0))


    else: # se não for menor é por que o vértice já foi totalmente processador e colocar o tempo final
        tempo_final[vertice] = tempo
        tempo += 1

return tempo_inicial, tempo_final, pai
```

O DFS eu fiz de modo iterativo, com pilha, porém com algumas diferenças para conseguir retornar de maneira certa os tempos finais.

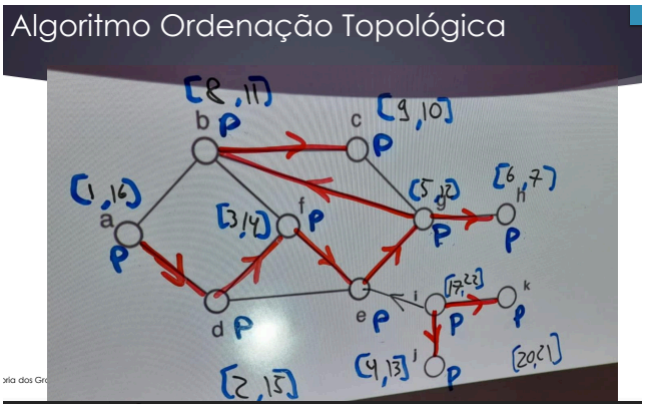
Para isso a pilha utiliza uma tupla, (vértice, índice do vizinho que ainda não foi visitado, isso acontece pois só posso tirar meu vértice da pilha quando eu tiver certeza que todos os vizinhos dele foram visitados e os vizinhos dos vizinhos também, etc… pois preciso contabilizar com exatidão o tempo de início e fim. claro que normalmente isso é feito com aquela representação das cores, mas preferi fazer assim no modo iterativo, se fosse recursivo(iria ser custoso demais pelo tamanho do grafo) eu teria feito com cores.

O pai apenas será o vértice antecessor ao atual, pois queremos formar apenas um caminho

Para fazer o teste se o tempo de início e final estava correto, utilizei esse exemplo de um dos slides do professor, e computei o grafo de exemplo, de forma que ele seguisse esse caminho e não outros que também seria possível a partir do DFS.

Teste complementar do DFS:

teste da função em si apenas no final do documento



Saída(a=1,b=2,etc…)

Escolhendo o caminho do exemplo acima(retirando as arestas que não formam o caminho):

Vértice	Inicio	Término
1	1	16
2	8	11
3	9	10
4	2	15
5	4	13
6	3	14
7	5	12
8	6	7

Sem escolher o caminho do exemplo acima(deixando o grafo todo):

Vértice	Inicio	Término
1	1	16
2	2	15
3	3	14
4	7	8
5	5	10
6	6	9
7	4	13
8	11	12

OBS: isso é apenas um teste para o tempo, óbvio que a escolha de caminho melhor se daria por outros métodos, é apenas pra validar que a computação do tempo está acontecendo de maneria ideal

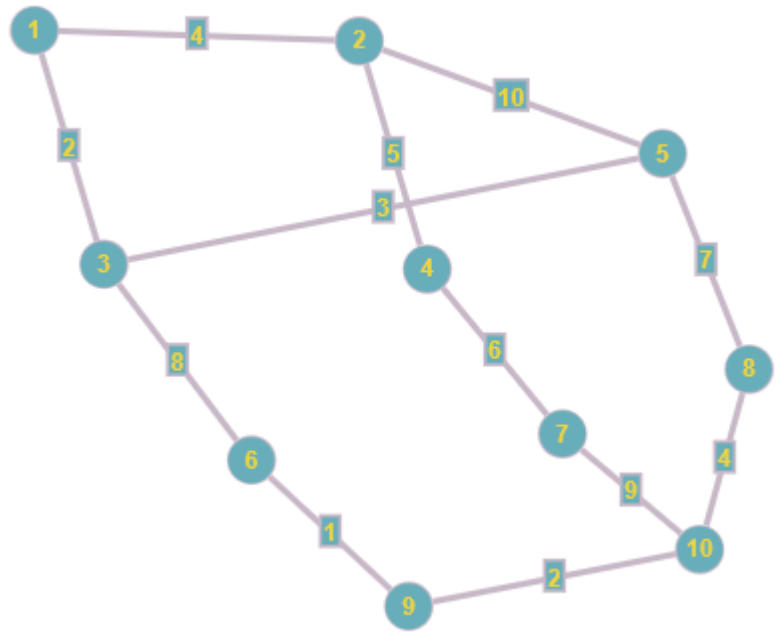
Bellman-Ford

```python
#Bellman-Ford
# se for representar a lista de arestas como uma aresta (v1,v2,peso) representando tanto a ida quanto
# a volta, lembre-se de adicionar mais um if, checando a volta.
# Se tiver duas arestas para representar uma aresta não direcional(ida e volta) mantenha assim
def bf(self,vertice_inicial):  1 usage
    #coloco todas distancias do vertice para todos outros como infinitas
    distancias = {vertice: float('inf') for vertice in self.lista_de_adjacencia}
    distancias[vertice_inicial] = 0
    # para armazenar os caminhos minimos
    caminhos = {vertice: [] for vertice in self.lista_de_adjacencia}
    caminhos[vertice_inicial] = [vertice_inicial]
    #antecessor de cada vértice no caminho minimo
    pai = {vertice: None for vertice in self.lista_de_adjacencia}

    # relaxando arestas
    for _ in range(self.n()-1):
        for u,v,peso in self.lista_de_arestas:
            # distancia de origem até u + peso de(u,v) tem que ser menor que distancia da origem a v
            if distancias[u] + peso < distancias[v]:
                distancias[v] = distancias[u] + peso
                caminhos[v] = caminhos[u] + [v]
                pai[v] = u


    # verificando ciclos de valores negativos: se isso é encontrado sempre será possivel fazer relaxamento
    # pois sempre dará para encurtar a distancia entre um vértice e outro
    for u,v,peso in self.lista_de_arestas:
        if distancias[v] > distancias[u] + peso:
            return False # encontrou um ciclo negativo pois foi possivel fazer mais relaxamentos
        # não encontrou ciclo negativo

    return distancias, caminhos, pai
```

O Bellman-Ford inicia com todos as distâncias como infinitas, distância do vértice inicial é inicializada para iniciar a partir dele
Tomei a liberdade de fazer um dicionário para armazenar os caminhos da origem até cada vértice, além do dicionário pai, que mostra o vértice antecessor à outro vértice naquele caminho
Depois relaxo as arestas fazendo as checagens necessárias, bem como adicionando no caminho e como pai de outro vértice.
Depois faço a verificação de caminhos negativos, o que seria um problema e por isso retornamos False, mas seria melhor fazer uma exceção aqui, pois pode dar um erro não personalizado, por estarmos atribuindo um valor booleano de uma forma que a linguagem não aceita(inseguranças do python)

OBS: essa função é para a forma que a representação das arestas está(duas arestas para representar uma aresta não direcional), caso seja a outra representação(uma aresta apenas para representar tanto ida quanto volta) precisaríamos acrescentar checagens(IF's) no primeiro FOR e mais uma condição no segundo FOR.

Teste complementar em um grafo simples:



txt do grafo(entrada):

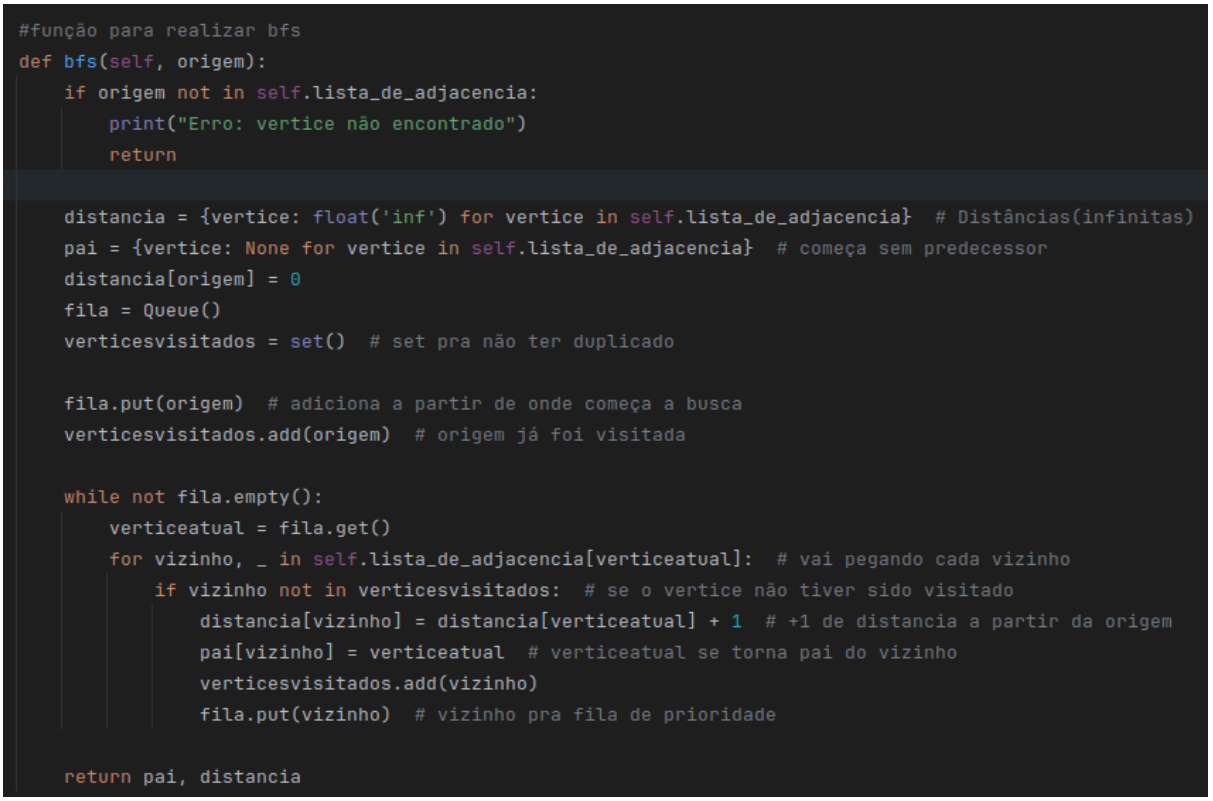
```
a 1 2 4
a 2 1 4
a 1 3 2
a 3 1 2
a 2 4 5
a 4 2 5
a 2 5 10
a 5 2 10
a 3 5 3
a 5 3 3
a 3 6 8
a 6 3 8
a 4 7 6
a 7 4 6
a 5 8 7
a 8 5 7
a 6 9 1
a 9 6 1
a 7 10 9
a 10 7 9
a 8 10 4
a 10 8 4
a 9 10 2
a 10 9 2
```

Saída para **origem: vértice 8:**
utilizando a biblioteca tabulate para formatar o print e visualizar melhor

Vértice	Distância Mínima	Caminho	Pai
1	12	8 -> 5 -> 3 -> 1	3
2	16	8 -> 5 -> 3 -> 1 -> 2	1
3	10	8 -> 5 -> 3	5
4	19	8 -> 10 -> 7 -> 4	7
5	7	8 -> 5	8
6	7	8 -> 10 -> 9 -> 6	9
7	13	8 -> 10 -> 7	10
8	0	8	
9	6	8 -> 10 -> 9	10
10	4	8 -> 10	8

Os resultados foram validados a partir de um site(https://graphonline.top/en/) que você desenha grafos e consegue usar alguns algoritmos sobre eles, como encontrar caminhos mínimos a partir de um vértice

BFS:



```python
#função para realizar bfs
def bfs(self, origem):
    if origem not in self.lista_de_adjacencia:
        print("Erro: vertice não encontrado")
        return

    distancia = {vertice: float('inf') for vertice in self.lista_de_adjacencia}  # Distâncias(infinitas)
    pai = {vertice: None for vertice in self.lista_de_adjacencia}  # começa sem predecessor
    distancia[origem] = 0
    fila = Queue()
    verticesvisitados = set()  # set pra não ter duplicado

    fila.put(origem)  # adiciona a partir de onde começa a busca
    verticesvisitados.add(origem)  # origem já foi visitada

    while not fila.empty():
        verticeatual = fila.get()
        for vizinho, _ in self.lista_de_adjacencia[verticeatual]:  # vai pegando cada vizinho
            if vizinho not in verticesvisitados:  # se o vertice não tiver sido visitado
                distancia[vizinho] = distancia[verticeatual] + 1  # +1 de distancia a partir da origem
                pai[vizinho] = verticeatual  # verticeatual se torna pai do vizinho
                verticesvisitados.add(vizinho)
                fila.put(vizinho)  # vizinho pra fila de prioridade

    return pai, distancia
```

O algoritmo de bfs tem uma pequena verificação para saber se o vértice está no grafo, então inicializa as distâncias como infinitas e os pais como None, a distância da origem é 0 e é definido um conjunto de vértices que já foram visitados e uma fila para adicionar os adjacentes que ainda vão ser visitados, enquanto a fila não tiver vazia vai pegando o vértice e tirando da fila e olhando os vizinhos deles, se o vizinhos não tiverem sido visitados, incrementa a distância baseado no vértice atual, o pai do vizinho se torna o vértice atual e vizinho entra no conjunto de vértices visitados e é adicionado à fila pra ser visitado na próxima iteração.

Djikistra:

```python
#função para realizar djikstra
def djikstra(self, origem):
    if origem not in self.lista_de_adjacencia:  # verifica se o vertice esta na lista
        print("Erro: vertice não encontrado")
        return
    distancias = {vertice: float('inf') for vertice in self.lista_de_adjacencia} # esse distancias representa a distancia minima
    pai = {vertice: None for vertice in self.lista_de_adjacencia}

    distancias[origem] = 0

    filadeprioridade = [(0.0, origem)]

    while filadeprioridade:  # vai percorrendo enquanto tiver valor na fila de prioridade
        distatual, verticeatual = heapq.heappop(filadeprioridade)  # remove o menor elemento da fila

        if distatual > distancias[verticeatual]:
            continue

        for vizinho,peso in self.lista_de_adjacencia[verticeatual]:
            if distancias[vizinho] > distancias[verticeatual] + peso:  # verifica se é a menor distancia total
                distancias[vizinho] = distancias[verticeatual] + peso  # atualiza a distancia
                pai[vizinho] = verticeatual  # atualiza o pai do vértice
                heapq.heappush(filadeprioridade, item: (distancias[vizinho], vizinho))  # adiciona à fila

    return pai, distancias
```

Da mesma forma no algoritmo djikstra, existe aquela pequena verificação e a declaração inicial das distâncias infinitas e dos pais, a diferença é que essas distâncias vão ser as distâncias mínimas da origem para qualquer vértice, logo após é declarado uma fila de prioridade, pois a ideia é ir colocando valores na fila de prioridade e pegando o menor caminho e tirando sempre o menor, enquanto a fila de prioridade existir, vai verificando se a distância atual é a menor e depois vai olhando os vizinhos, verificando se a distância do vértice atual e o peso é maior que distância atual do vizinho(no começo por exemplo, essa distância seria infinita), se for, o vértice atual se torna pai do vizinho e a distância mínima é atualizada, dessa forma esse vizinho vai para a fila de prioridade para a partir dele encontrarmos o caminho mínimo dos outros vizinhos dele.

Vértice mais distante:

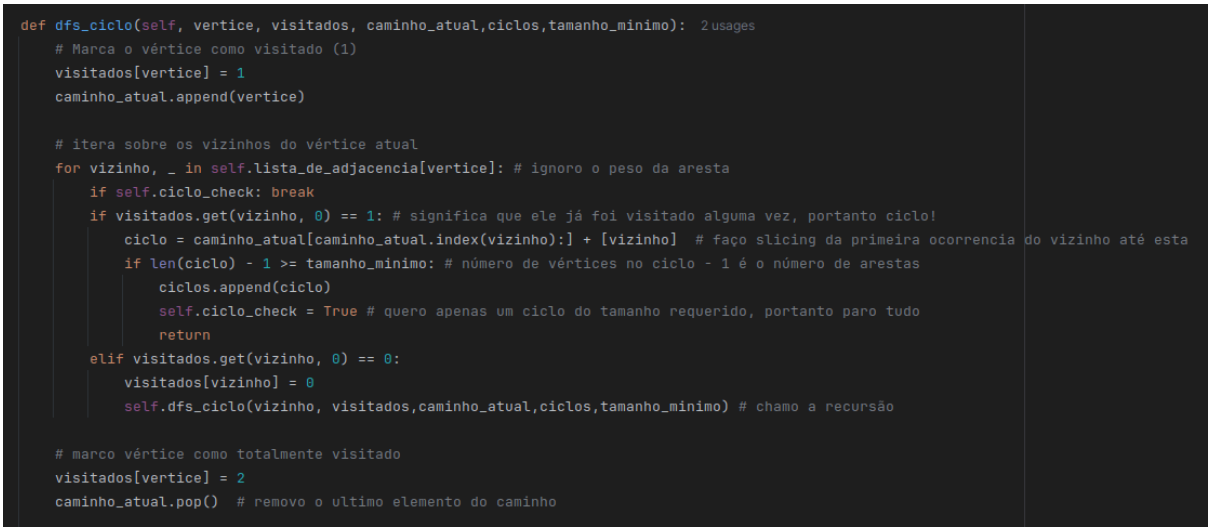
```python
def vertice_mais_distante(self,vertice):
    dist,_,_ = self.bf(vertice) # lista das distancias a partir de um vértice, usando bellman ford
    maior_distancia = -float('inf')
    vertice_maior = None
    for vertice in dist:
        if dist[vertice] > maior_distancia and dist[vertice] != float('inf'): # comparo a distancia dos vértices
            maior_distancia = dist[vertice]
            vertice_maior = vertice

    return maior_distancia,vertice_maior # retorno maior distancia e o vértice mais distante da minha origem
```

Caminho arestas maior que 10:

```python
def busca_caminho(self,origem,tamanho_minimo):  1 usage
    # irei fazer uma função interna pois a chamada da função recursiva é simples demais pra separar
    def dfs_caminho(vertice,caminho_atual, arestas_atual):
        # checa para cada chamada se o resultado atual corresponde ao criterio de arestas minimas
        # se não atender a isso, não vai retornar o caminho de arestas_atual e portanto continuará para uma nova origem
        # se retornar cai na checagem if resultado(não for nulo) retorna o resultado e pronto
        if len(arestas_atual) >= tamanho_minimo:
            return arestas_atual
        for vizinho, peso in self.lista_de_adjacencia[vertice]:
            if vizinho not in caminho_atual: # não posso ter ciclos
                nova_aresta = (vertice,vizinho,peso) #formato a aresta para retornar bonitinho no print
                resultado = dfs_caminho(vizinho,caminho_atual + [vizinho], arestas_atual + [nova_aresta])
                if resultado: # se o resultado(caminho não tiver vazio) retorna o caminho
                    return resultado
        return None
    return dfs_caminho(origem,  caminho_atual: [origem],  arestas_atual: []) # faz a chamada para a recursão
```

Ciclos(arestas maior ou igual a 5):

```python
# função base para a chamada do dfs_ciclo, inicia a recursão
def busca_ciclo(self,tamanho_minimo):  1 usage
    # inicia todos os visitados com o status("cor") 0, nenhum foi visitado ou processado
    visitados = {vertice: 0 for vertice in self.lista_de_adjacencia}
    ciclos = []
    ciclos_arestas = []
    #caso seja um grafo conexo só usaremos o primeiro vértice
    for vertice in self.lista_de_adjacencia:
        if visitados[vertice] == 0:
            # vertice de inicio, lista de visitados e a "cor", lista para armazenar caminho, lista para armazenar
            # ciclo e o tamanho minimo do ciclo que queremos retornar
            self.dfs_ciclo(vertice,visitados, caminho_atual: [],ciclos,tamanho_minimo)
        if self.ciclo_check: break

    # função para formatar as arestas de maneira correta para retornar a aresta, e não o vértice
    for ciclo_vertices in ciclos:
        ciclo_arestas = []
        for i in range(len(ciclo_vertices) - 1):
            origem = ciclo_vertices[i]
            destino = ciclo_vertices[i + 1]
            # Procura o peso da aresta entre origem e destino
            for v, p in self.lista_de_adjacencia[origem]:
                if v == destino:
                    ciclo_arestas.append((origem, destino, p))
                    break
        ciclos_arestas.append(ciclo_arestas)
    return ciclos_arestas
```



```python
def dfs_ciclo(self, vertice, visitados, caminho_atual,ciclos,tamanho_minimo):  2 usages
    # Marca o vértice como visitado (1)
    visitados[vertice] = 1
    caminho_atual.append(vertice)

    # itera sobre os vizinhos do vértice atual
    for vizinho, _ in self.lista_de_adjacencia[vertice]: # ignoro o peso da aresta
        if self.ciclo_check: break
        if visitados.get(vizinho, 0) == 1: # significa que ele já foi visitado alguma vez, portanto ciclo!
            ciclo = caminho_atual[caminho_atual.index(vizinho):] + [vizinho]  # faço slicing da primeira ocorrencia do vizinho até esta
            if len(ciclo) - 1 >= tamanho_minimo: # número de vértices no ciclo - 1 é o número de arestas
                ciclos.append(ciclo)
                self.ciclo_check = True # quero apenas um ciclo do tamanho requerido, portanto paro tudo
                return
        elif visitados.get(vizinho, 0) == 0:
            visitados[vizinho] = 0
            self.dfs_ciclo(vizinho, visitados,caminho_atual,ciclos,tamanho_minimo) # chamo a recursão

    # marco vértice como totalmente visitado
    visitados[vertice] = 2
    caminho_atual.pop()  # removo o ultimo elemento do caminho
```

Está tudo bem explicado no código, mas basicamente tenho uma função que serve como base para iniciar o DFS recursivo que irá checar o status(“cor”) de cada vértice, e testar se está sendo formado um ciclo, isso é feito a perceber que um vértice que já foi visitado está sendo visitado novamente a partir de outro vértice.
O busca_ciclo também implementa um “formatador”(não exatamente necessário mas ta ai) que é o segundo ‘for’, para eu **transformar minha lista de vértices, em arestas**, buscando seus vizinhos e pesos do caminho que formou o ciclo, e assim formatando como (vértice, vizinho, peso)
Pra além disso também é necessário comentar que o número de arestas está sendo verificado a partir da condição de (if(len(ciclo) - 1 >= tamanho_minimo) ou seja, se tenho um caminho com 6 vértices, eu sei que tenho 5 arestas neste caminho, pois as arestas serão n-1 vértices

Dígrafo:

No dígrafo, as únicas mudança foram duas:

- A forma de colocar as arestas no grafo, baseado no grafo disponibilizado para testes, no qual apenas colocamos uma “orientação”, ou seja, consideramos apenas uma direção, de ida, e não a de volta(baseada na ordem que as arestas aparecem no documento), isso foi feito apenas fazendo um if(u<v)
- **Isso significa, de forma intuitiva, que estou considerando apenas as arestas na qual o vértice de “origem” dela, ou seja, o vértice na qual a aresta “sai”, é um número ímpar**
- Bem como também adicionamos uma lista de graus de entrada, para ir somando as arestas que entram no vértice, isso ajuda a otimizar a consulta do grau de entrada de um vértice, que se for feito iteração de toda a lista de adjacências demora MUITO

```python
def adicionarAresta(self, u, v, peso=1):
    # note que não preciso adicionar o v, já que pela forma que o documento está, ele passará por todos vertices
    # e portanto o u já bastará
    self.adicionarVertice(u)
    # uma das diferença do grafo pra digrafo, no caso para esse representação
    # coloco apenas uma vez a aresta (1,2,peso) não considerando (2,1,peso) para manter direcional
    if u<v:
        self.lista_de_adjacencia[u].append((v, peso)) # apenas adiciono uma direção
        self.lista_de_arestas.add((u,v,peso))
    #adiciono na lista de graus de entrada a aresta que está entrando no vértice de destino
    #isso ajuda otimizar o tempo pra consultar o grau do vértice de um dígrafo
    self.grau_entrada[v] = self.grau_entrada.get(v, 0) + 1
```

- A outra mudança foi na verificação do grau de um vértice, onde precisamos considerar tanto a entrada de arestas naquele vértice, quanto a saída.

```python
# retorna o grau do vertice
# uma das diferenças do grafo pra digrafo
def d(self, vertice):
    grau_entrada = len(self.lista_de_adjacencia[vertice])
    grau_saida = 0

    for vertice in self.lista_de_adjacencia:
        for tupla in self.lista_de_adjacencia[vertice]:
            if tupla[0] == vertice:
                grau_entrada +=1
    return grau_entrada + grau_saida
```

Dessa forma a consulta vai acontecer, mas é MUITO demorado, portanto com a ajuda da lista de graus, incrementada durante a adição das arestas temos:

```python
def d(self,vertice):
    grau_saida = len(self.lista_de_adjacencia[vertice])
    grau_entrada = self.grau_entrada[vertice]

    return grau_saida + grau_entrada
```

O motivo de não haver mais mudanças é porque as outras funções se baseiam na forma que o grafo está representado, sendo necessário mudar apenas a representação dele, ou seja, a forma que as arestas aparecem no documento. Seja dando entender que é um grafo ou que é um dígrafo.
Obviamente que caso eu quisesse tornar isso modular para TODOS os grafos, seria importante que todos os grafos seguisse um padrão de apenas ter uma direção em cada aresta, e deixando para o programador decidir se quer fazer com que aquele grafo seja representado como um não direcional(colocando a ida e a volta pra cada aresta, ou seja se aparece apenas (1,2), adicionaria a volta (2,1)) ou direcional(apenas adicionando as arestas como vem no documento).

Testes:

Função para ler o grafo:

```
def ler_grafo(path):  1 usage
    arestas = []
    arquivo = open(path, 'r')
    for linha in arquivo: #passa por todas linhas no arquivo
        if linha.startswith('a'): # por padrão, no documento, cada linha começa com 'a'
            _, v1, v2, peso = linha.split() # ignora o 'a' e atribui os valores a v1,v2 e peso
            arestas.append((int(v1), int(v2), float(peso)))
    return arestas
```

```
g = Grafo()
# g = Digrafo()
for aresta in arestas:
    g.adicionarAresta(aresta[0], aresta[1], aresta[2])
```

Grafo:

g.mind():

```
vertice_menor, d1 = g.mind()
print(vertice_menor,d1)
print(g.viz(6))
```

```
/usr/bin/python3.9 /home/puma633/MEGA downloads/BiblioGrafo/src/biblioteca/main.py
6 1
[(5, 774.0)]
1.4057033061981201

Process finished with exit code 0
```

Primeiro número é o vértice e o segundo é o grau dele
Retornei também o vizinho do vértice 6 após ver qual era o vértice de menor grau, só pra referência
este último número, representa uma função da biblioteca time, para ver o tempo de execução do código, desde a leitura do grafo até a execução da função especificada

g.maxd():

```python
vertice_maior, d2 = g.maxd()
print(vertice_maior,d2)
print(g.viz(140961))
```

```
/usr/bin/python3.9 /home/puma633/MEGA downloads/BiblioGrafo/src/biblioteca/main.py
140961 8
[(140960, 813.0), (140963, 671.0), (140980, 1519.0), (141020, 1929.0), (141026, 2481.0), (141105, 1439.0), (141110, 313.0), (140982, 456.0)]
1.396022081375122
```

Novamente, printei o vizinho do vértice após verificar com a função qual era o maior grau, apenas para referência

Caminho com 10 arestas ou mais:

```python
caminho = g.busca_caminho( origem: 1, tamanho_minimo: 10)
for origem, destino, peso in caminho:
    print(origem, destino, peso)
```

Apenas especifico a origem(1 nesse caso), e o tamanho mínimo do meu caminho, ou seja o tanto de arestas que quero no caminho

```
/usr/bin/python3.9 /home/puma633/MEGA downloads/BiblioGrafo/src/biblioteca/main.py
1 1363 2428.0
1363 1358 1178.0
1358 1355 1827.0
1355 1274 1111.0
1274 1143 2155.0
1143 1144 1414.0
1144 1136 1254.0
1136 1131 2737.0
1131 1111 3997.0
1111 1099 1345.0
1.3666105270385742
```

Para 20 arestas agora:

```
/usr/bin/python3.9 /home/puma633/MEGA downloads/BiblioGrafo/src/biblioteca/main.py
1 1363 2428.0
1363 1358 1178.0
1358 1355 1827.0
1355 1274 1111.0
1274 1143 2155.0
1143 1144 1414.0
1144 1136 1254.0
1136 1131 2737.0
1131 1111 3997.0
1111 1099 1345.0
1099 1100 2578.0
1100 1101 1573.0
1101 1124 1343.0
1124 1122 1713.0
1122 1128 549.0
1128 1129 4088.0
1129 1174 1188.0
1174 1130 1539.0
1130 1127 1406.0
1127 1079 2304.0
1.4037978649139404
```

Ciclo de 5 ou mais arestas:

```python
ciclos = g.busca_ciclo(5)
for aresta in ciclos:
    for vertice, vizinho, peso in aresta:
        print(vertice,vizinho,peso)
```

Com pelo menos 5:

```
/usr/bin/python3.9 /home/puma633/MEGA downloads/BiblioGrafo/src/biblioteca/main.py
1130 1127 1406.0
1127 1079 2304.0
1079 1078 3593.0
1078 1150 1511.0
1150 1151 5673.0
1151 1130 974.0
1.3804748058319092
```

Com pelo menos 10:

```
/usr/bin/python3.9 /home/puma633/MEGA downloads/BiblioGrafo/src/biblioteca/main.py
258082 258083 3932.0
258083 258062 972.0
258062 258061 473.0
258061 258060 496.0
258060 258055 579.0
258055 258041 1549.0
258041 258040 259.0
258040 258039 347.0
258039 258038 1364.0
258038 258034 428.0
258034 258015 2232.0
258015 258011 628.0
258011 258010 799.0
258010 258009 361.0
258009 258012 639.0
258012 258014 1908.0
258014 258037 565.0
258037 258042 362.0
258042 258082 979.0
1.3556499481201172

Process finished with exit code 0
```

Vértice mais distante:

```
maior_distancia, vertice = g.vertice_mais_distante(129)
print(maior_distancia)
print(vertice)
```

```
220049 220052 720.0
220048 220050 531.0
220044 224782 463.0
iteracao 478 ***********
220044 220045 425.0
220052 220048 1009.0
220048 220050 531.0
iteracao 479 ***********
1437303.0
90644
384.7497103214264

Process finished with exit code 0
```

Maior distancia: 1437303

Vértice mais distante: 90644
tempo de processamento: 769 segundos = 12 minutos(meu computador i3 11 geração)
tempo de processamento: 384 segundos = 6 minutos(i7 13650HX)
óbvio que os dois usei apenas 1 núcleo e não fiz multithreading nem multiprocessing então enfim

Eu sei que esse é o resultado correto por comparações com resultados de colegas, mas o detalhe é:

- meu computador não foi capaz de processar isso em tempo hábil
- Deu 12 minutos pois adicionei uma flag, onde eu parava o algoritmo se não houvessem atualizações
- Nesse caso isso deu certo, pois o algoritmo convergiu relativamente “rápido”, porém o algoritmo, mesmo sem haver atualizações nas distâncias, continua rodando, pois faz v-1 iterações sobre todo o grafo, a flag otimiza isso, parando prematuramente.

Flag:

```
# relaxando arestas
for _ in range(self.n()-1):
    atualizou = False
    for u,v,peso in self.lista_de_arestas:
        # distancia de origem até u + peso de(u,v) tem que ser menor que distancia da origem a v
        if distancias[u] + peso < distancias[v]:
            distancias[v] = distancias[u] + peso
            caminhos[v] = caminhos[u] + [v]
            pai[v] = u
            print(u,v,peso)
            atualizou = True
    if not atualizou: # nessa iteração não houve atualizações nas distancias
        break
```

Note a flag “atualizou” onde checo se naquela iteração houve alguma atualização na distância

Teste em um computador de um familiar com uma configuração muita boa e ainda sim, o máximo que consegui reduzir foi pela metade o tempo de convergencia, sendo tempo demais para fazer todas iterações. Mas o que posso acrescentar é em qual iteração os algoritmos convergiram:
Para o Grafo não direcionado convergiu na iteração 478

Dígrafo:

O dígrafo é representado apenas com arestas de “ida” oq quero dizer com isso é q vou acrescentando a lista de adjacências e listas de arestas apenas a primeira ocorrência de uma dada “aresta” no documento, ou seja se tenho (1,2,803) e logo na próxima linha (2,1,803), apenas considero a primeira(1,2,803)

Isso significa, de maneira intuitiva, que estou apenas considerando as arestas no qual o vértice na qual a aresta sai é um número impar
Não vou colocar a main aqui, pois nada muda do grafo pra cá, apenas que declaro g.digrafo() invés de g.grafo()

g.mind():

```
C:\Users\guilh\PycharmProjects\BiblioG
6 1
1.599731206893921
```

Veja que aqui deu o mesmo do grafo não direcionado, teoricamente aqui daria mais, porém como usei a representação considerando a apenas a "ida" como no documento aparece (5,6,774) e depois (6,5,774), então consideramos só o primeiro, portanto apenas a aresta de entrada

g.maxd():

```
C:\Users\guilh\PycharmProjects\BiblioGr
140961 15
1.4621176719665527
```

já aqui temos o mesmo vértice como resultado, porém de grau 8 foi para 15, pois ele tem todas aquelas arestas saindo dele e mais várias outras entrando nele

Ciclos:

Da forma que o grafo foi representado, como especifiquei, não foram achados ciclos

Caminho com 10 arestas ou mais:

Com 10 arestas ou mais:

```
C:\Users\guilh\PycharmProjects
1 1363 2428.0
1363 1364 1095.0
1364 1366 515.0
1366 1367 1611.0
1367 1368 2445.0
1368 1388 1454.0
1388 1391 452.0
1391 1393 893.0
1393 1395 445.0
1395 1398 1234.0
1.6479780673980713
```

Com 20 arestas ou mais:

```
C:\Users\guilh\PycharmProjects
1 1363 2428.0
1363 1364 1095.0
1364 1366 515.0
1366 1367 1611.0
1367 1368 2445.0
1368 1388 1454.0
1388 1391 452.0
1391 1393 893.0
1393 1395 445.0
1395 1398 1234.0
1398 1399 2023.0
1399 1430 796.0
1430 1479 888.0
1479 1480 1292.0
1480 1481 749.0
1481 1482 1425.0
1482 1483 817.0
1483 1484 503.0
1484 1492 1726.0
1492 1494 373.0
1.7085444927215576
```

Vértice mais distante:

```
iteracao 44 **************
3985 3989 667.0
4063 4064 503.0
iteracao 45 **************
152298.0
3989
13.673832654953003

Process finished with exit code 0
```

Também com a flag para interromper quando não houvesse mais atualizações
Dígrafo converge em 44 iterações
Fui até quase 10000 para testar.

Distância: 152298
Vértice mais distante: 3989

FIM!