

Biblioteca de Grafos - (Python)

Requisitos

- Para o funcionamento dessa biblioteca, basta ter o Python - a partir da versão **3.0** - instalada em sua máquina; além do módulo venv (v. ≥ 3.0). Responsável pelo provisionamento de um ambiente isolado de execução - de maneira a conter somente o que for necessário às funcionalidades da biblioteca - esse pacote pode ser instalado com auxílio da documentação linkada em sua versão.

Instalação

- Utilizaremos o gerenciador de pacotes padrão do Python, chamado de **pip**, para realizar o download das dependências do projeto. Tais acervos são facilitadores para a criação de estruturas primitivas, controle de tamanho de dados em bits e optimalidade em termos de memória e performance. As dependências merecedoras de destaque são:
 - numpy (v. ≥ 1.23.3)
 - HeapDict (v. ≥ 1.0.1)
- Em um terminal de comando (PowerShell em sistemas Windows, idealmente), digite:

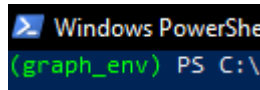
```
git clone https://github.com/RuasGAR/graph_library.git
cd C:/Users/${pasta de destino}

# Para ativar seu ambiente de execução:
# Linux e Mac
./graph_env/bin/activate
# Windows
.\graph_env\bin\Activate.ps1

# Baixando as dependências

pip install graph_env/requirements.txt
```

- Veja que agora seu terminal dispõe de uma marcação ao lado do caminho de diretório.

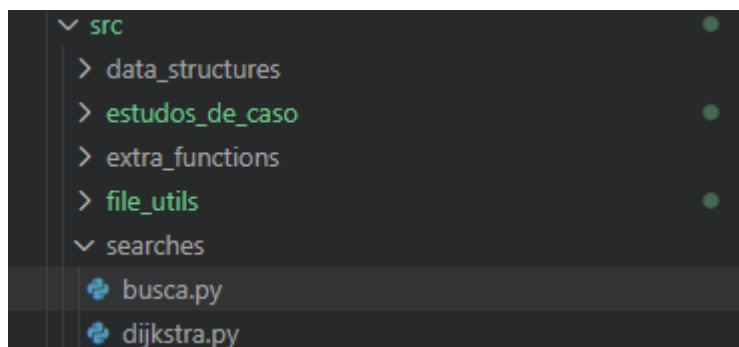


- Pronto! Todos os passos já foram dados e agora é só utilizar como quiser! Para uma melhor orientação de como rodar alguns testes, veja a seção “*Caso de Estudos*”.

Ideação e Estrutura

Linguagem e Paradigma

- Python é uma linguagem bastante flexível em termos de paradigma, sendo a **orientação a objeto** um de seus destaques mais recentes, em especial no desenvolvimento de APIs simples e leves. Na ampla maioria do código, é essa a abordagem utilizada para o desenvolvimento da biblioteca: os códigos são segmentados como classes e separadas em módulos.



Estrutura hierárquica do projeto: menores componentes de modularização são as classes

- Certas escolhas de concepção - cuja totalidade é comentada com mais detalhes na seção *Funcionalidades* - reúnem algoritmos com características similares para evitar sintetizar e reduzir instâncias necessárias. Um exemplo dessa natureza pode ser visto no arquivo “busca.py”, o qual inclui uma classe de Busca. Com somente uma instância dessa classe, podemos escolher o melhor algoritmo de busca para grafos sem peso - Breadth First Search ou Depth First Search.
- Por último - e certamente importante - faz-se importante a exposição do componente básico de praticamente todos os métodos de busca disponíveis na biblioteca. Estamos falando da classe Vertice, disponível no caminho `graph_env/src/data_structures/search_vertex.py`.

```

class Vertice:
    def __init__(
        self,
        valor: int,
        pai: int,
        peso: float = 0.0,
    ) -> None:
        self.valor: int = valor
        self.marcaador: bool = False
        self.pai: Union[int, None] = pai if pai else None
        self.nivel: int = 0
        self.peso = peso

```

Representação de Grafos

- Nossa biblioteca é capaz de representar grafos não direcionados, com pesos positivos ou com nenhum. Grafos com pesos só podem ser alocados em **Vetores de Adjacência**. Aqueles que não possuem intensidades associadas às arestas também podem ser representados por **Matrizes de Adjacência**, embora sempre tenhamos como recomendação a primeira estrutura de dados para menor consumo de memória e de processamento.

Otimizações e Boas Práticas

- Por padrão, variáveis e valores inteiros ou de ponto flutuante serão criados com 64 bits pelo Python. Entretanto, em nosso uso corriqueiro, notamos que esse número de alocação de armazenamento é um pouco exagerado: 32 bits já são capazes de representar valores inteiros na casa dos 5 bilhões, enquanto o uso máximo de nossos problemas tinha como limite superior a região dos 10 milhões de vértices. Dessa maneira, fizemos um intensivo uso da biblioteca **numpy** com o intuito de reduzir esse espaço de ocupação e, por consequência, aumentando a eficiência do código para a grande maioria dos casos.
- Para além disso, outra preocupação levantada foi a legibilidade do programa, em especial no que diz respeito ao entendimento de entrada e saída dos métodos e funções. Python não é uma linguagem fortemente tipada, mas não impede que utilizemos módulos como **typing** para tornar mais transparente as expectativas de um código. Essa estratégia, apesar de verbosística, facilita o debugging, os mecanismos de *hint* das IDEs mais populares e viabiliza de forma mais clara a colaboração da comunidade na elaboração do projeto.

Formato(s) de Entrada

- Até o momento (v_tal), a biblioteca permite o processamento de grafos não direcionados com pesos e sem pesos. O formato de entrada aceito pela biblioteca é bem similar nos dois casos, diferindo somente pela presença de um valor a mais quando as arestas possuem pesos. A escolha foi orientada pelo padrão acadêmico mais comum aos pesquisadores da área.

```
968
436 263
63 676
293 728
488 336
314 793
```

Sem peso

```
1000
180 251 7.39
243 294 6.33
22 416 4.67
364 502 6.36
200 997 7.09
```

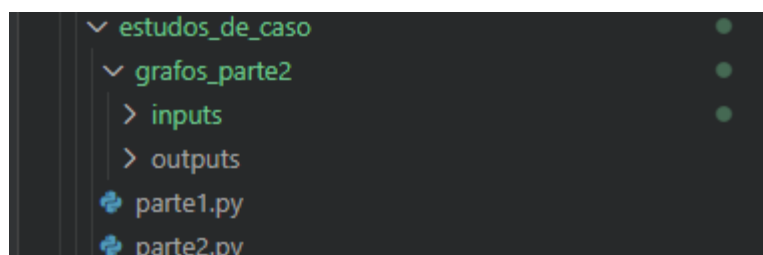
Com peso

Um arquivo .txt deve ser fornecido nos seguintes formatos:

- *Primeira linha:* nº de vértices
- *Da segunda linha em diante:* arestas codificadas em seus vértices incidentes, separados por um espaço.
- *Caso o grafo tenha pesos,* um terceiro valor (de ponto flutuante ou inteiro) pode ser incluído em cada aresta, a ser preenchido com um espaço de distância em relação aos vértices.

Como usar (foco nos estudos de caso)

- Em cada parcela do desenvolvimento da lib foram rodados estudos para averiguar a complexidade, viabilidade, qualidade e assertividade dos métodos implementados. Por uma questão de organização, desenvolvemos scripts relativos a cada uma das partes para enxugar a main.



- Dessa maneira, se quisermos verificar os resultados da “Questão 3” da parte 2, basta fazer o import do script relacionado. A parte 1 tem maior incidência de prints no terminal de execução como saída e monitoramento do programa, enquanto a parte 2 é um pouco mais inteligente, originando arquivos com as respostas correspondentes para cada grafo. Tais saídas podem ser encontradas no caminho `graph_env/src/estudos_de_caso/grafos_parte2/outputs`. E não se preocupe caso essa pasta não apareça de início: se esse for o caso, qualquer execução da parte 2 já criará o diretório automaticamente.
 - *Observação: todas as questões da parte 2 foram implementadas dessa maneira, tendo somente o caminho para armazenamento dos outputs como parâmetro. As da primeira exigem um pouco mais de desenvoltura com o estado do código.*

```
from estudos_de_caso.parte2 import questao1, questao2, questao3, questao4
from pathlib import Path
from os import getcwd

def __main__():

    caminho_grafos = Path(
        getcwd(), "graph_env", "src", "estudos_de_caso", "grafos_parte2", "inputs"
    )

    questao1(caminho_grafos)

    # questao2(caminho_grafos)

    # questao3(caminho_grafos)
```

Funcionalidades

Parte 1

Focada especialmente em grafos sem pesos, essa parcela da biblioteca fornece os algoritmos de base no estudo de Grafos. Além disso, também é possível investigar características relevantes do grafo inserido, como o seu grau mínimo.

As funcionalidades implementadas, **para grafos não direcionados e sem pesos**, são:

- Obtenção de grau mínimo, máximo, médio e da mediana do grau de um grafo sem pesos não direcionado.
- Fornecimento de número de arestas e de vértices de um grafo sem pesos não direcionado.
- BFS e DFS → completas, essas formas de busca têm como principal diferença a garantia de optimalidade em cenários finitos: enquanto a primeira garante o encontro do melhor caminho, a segunda é sensível a estruturas com loops e, portanto, pode não encontrar a melhor trajetória de um vértice ao outro. A BFS utiliza filas em sua implementação, enquanto a DFS faz uso de pilhas.
 - O time de desenvolvimento da linguagem Python oferece uma biblioteca de estruturas primitivas chamada **collections**, a qual contém uma estrutura chamada de **deque**. Apelido de *double ended queue*, esse construto pode ser utilizado tanto como filas (FIFO) como stacks (FILO), já que podemos escolher adicionar ou retirar itens no início ou no fim da estrutura.
- Identificação de componentes conexas, bem como a quantificação dos vértices em cada uma delas. Utiliza BFS como base.

Parte 2

A etapa 2 consiste na extensão dos principais objetivos de análise em um grafo para instâncias que tenham pesos positivos. Introduzindo técnicas de paradigmas gulosos, podemos agora calcular o custo de percorrer determinado caminho, bem como determinar conjuntos de vértices cuja que sejam alcançáveis entre si.

Mais especificamente, você poderá utilizar os seguintes algoritmos:

- Dijkstra, implementado com vetor de distâncias → com complexidade $O(n^2)$, a versão tradicional desse famoso algoritmo é capaz de identificar o custo de todos os outros nós para um ponto de partida escolhido pelo usuário. No entanto, deve-se ter cautela com seu uso: para grafos muito grandes, sua complexidade pode ser impeditiva, seja em termos de memória, seja de processamento (principalmente).

```
def dijkstra_com_vetor(grafo: VetorAdj, vertice_s: int) -> List[Dict]:

    # Criação de vértices de todos os vértices, da fila de descobertos
    vertices: List[Dict] = [
        Vertice(x, np.int32(-1), np.inf) for x in range(1, grafo.num_vertices + 1)
    ]

    # vai funcionar como se fosse uma fila, constantemente reordenada

    descobertos: List[Dict] = []

    # Setando distância do inicial e colocando-o entre os descobertos
    vertices[vertice_s - 1].peso = 0
    descobertos.append(vertices[vertice_s - 1])
```

Estruturas auxiliares para o algoritmo de Dijkstra com vetor

- Dijkstra, implementado com fila de prioridade → instanciadas como heap, as filas de prioridade nos possibilita o acesso e extração de mínimos em tempo $O(1)$ e $O(\log n)$, respectivamente. Ambas são bem menores do que a complexidade de acesso do vetor, que no pior caso exige uma passagem por todos os n vértices disponíveis. Dessa forma, a complexidade total de Dijkstra torna-se $O((n+m) \log n)$, o que é menor do que n^2 em casos de grafos de baixa densidade.
 - Por se tratarem de estruturas básicas na computação, a heap possui inúmeras implementações disponíveis em forma de biblioteca para uma grande gama de linguagens. O algoritmo de Dijkstra requer que tenhamos o poder de alterar o custo de um determinado nó, o que significa alterar a estrutura global como um todo. A biblioteca encontrada para tal finalidade foi a HeapDict, que mistura dicionários com as operações de **decrease-key** e **extract min** das heaps.

```
teste = heapdict.heapdict
for i in range(10):
    teste[i] = np.inf
teste[8] = 0.0
```

Exemplo de utilização da HeapDict.

Aqui são criados 10 itens na heap, cujas chaves são os identificadores de vértices e o valor, seu custo associado.

- Árvore geradora mínima, utilizando o algoritmo de Prim: muito similar a Dijkstra, o algoritmo de Prim é capaz de obter os subgrafos nos quais os nós estão todos conectados, assegurando uma das configurações possíveis de se obter essa propriedade com o menor custo possível. Foi implementada usando a estratégia de heaps.

```
for vizinho in grafo.percorrer_vizinhos(u):  
    valor_vizinho = vizinho[0]  
    peso_u_vizinho = vizinho[1]  
  
    try:  
        if (  
            distancias[valor_vizinho] > peso_u_vizinho  
        ): # comparamos somente com as arestas locais  
            distancias[valor_vizinho] = np.float32(peso_u_vizinho)  
            pais[valor_vizinho - 1] = u  
    except KeyError:  
        continue
```

Em destaque, podemos ver o fator de disparidade com Dijkstra: na exploração de vértices, não nos preocupamos com o menor custo acumulado, mas sim com aquele que garante o menor próximo passo. No exemplo, se a distância conhecida para um vértice for maior do que a distância de u a esse mesmo nó, então devemos substituir o custo associado, e atualizar seu parentesco.