Trabalho prático de Grafos

Pedro H. Rodrigues[1]

[1]Engenharia de Software – Pontificia Universidade Católica de Minas Gerais (Puc Minas)

Abstract. *This paper presents the implementation and documentation of a Python-based graph manipulation library developed as part of the coursework for the "Graph Theory and Computability" class. The library offers functionalities for creating, labeling, weighting, and exporting graphs, with support for multiple internal representations. The Gephi tool is used to visually validate the results.*

Resumo. *Este artigo apresenta a implementação e documentação de uma biblioteca em Python para manipulação de grafos, desenvolvida como parte do trabalho prático da disciplina de "Teoria dos Grafos e Computabilidade". A biblioteca oferece funcionalidades para criação, rotulagem, atribuição de pesos e exportação de grafos, com suporte a múltiplas representações internas. A ferramenta Gephi é utilizada para validação visual dos resultados.*

1. Informações gerais

Este documento serve como uma documentação para a biblioteca de manipulação de grafos desenvolvida no trabalho prático da disciplina de "Teoria dos Grafos e Computabilidade". O texto passará por três seções além desta:

- **Funcionalidades**: Descrição das principais funcionalidades da biblioteca.
- **Estrutura**: Detalhamento da arquitetura e organização do código.
- **Como Utilizar**: Instruções para uso da biblioteca.

2. Funcionalidades

A biblioteca desenvolvida tem como objetivo prover uma interface simples que possibilite a criação e manipulação de grafos. Abaixo, todas as funcionalidades desenvolvidas até momento são descritas e devidamente ilustradas com ajuda da ferramenta "Gephi".

2.1. Criar um grafo

Como a primeira e indispensável funcionalidade, temos a criação de um grafo. Para usufruir desta mecânica é necessário que o número de vértices seja previamente informado, assim, ao criar o grafo ele se iniciará com todos os vértices necessários, porém sem nenhuma aresta.

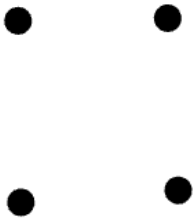

Figure 1. Criação de um grafo

2.2. Adicionar rótulos em vértices

Após a criação do Grafo, é possível adicionar rótulos customizados para cada vértice.

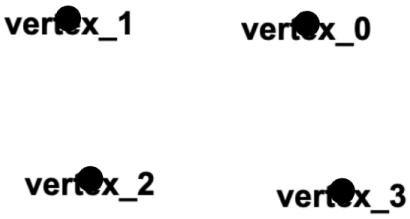

Figure 2. Vértices com rótulos

2.3. Adicionar peso em vértices

Além de rotular vértices, é possível atribuir um peso para eles.

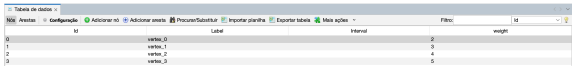

Figure 3. Vertices com pesos

2.4. Adicionar arestas

Dado dois vértices dentro do Grafo é possível adicionar uma aresta entre eles.

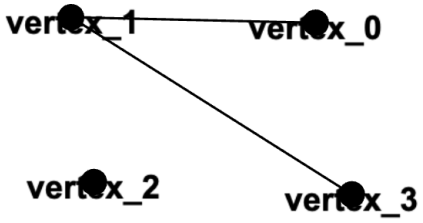

Figure 4. Vertices com arestas

2.5. Rotular arestas

Dado uma aresta existente, é possível adicionar um rótulo nela.

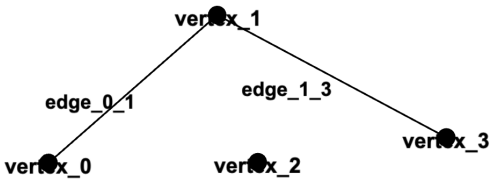

Figure 5. Arestas rotuladas

2.6. Adicionar peso as arestas

Além de adicionar um rótulo a uma aresta é também possível adicionar um peso. É importante frisar que ao adicionar o peso, a ferramenta de visualização Gephi pode distorcer a aresta, sendo preciso mexer em configurações para obter uma melhor visualização.

Figure 6. Arestas rotuladas

2.7. Exportar Grafo

Todo grafo criado pode ser exportado para ".gexf" e importado em qualquer ferramenta que consiga interpretar o arquivo gerado. A título de curiosidade, todas as imagens de grafos geradas neste documento foram feitas utilizando a funcionalidade de exportação da biblioteca.

2.8. Operações gerais

Além das funcionalidades já citadas, a biblioteca também contém funcionalidades gerais, como:

- Deletar aresta
- Checar se vértices são adjacentes
- Checar se arestas são adjacentes
- Checar se uma aresta é incidente a um vértice
- Verificar se o grafo é completo
- Verificar se o grafo está vazio
- Verificar se uma aresta existe
- Obter quantidade de arestas e vértices
- Obter areas

3. Estrutura

A biblioteca contém a seguinte estrutura dentro do diretório src

- **graph.py**: Classe principal da biblioteca que oferece uma interface unificada para operações como criação de arestas, verificação de adjacências, e exportação de grafos.
- **models/graph_components/vertex.py**: Define a estrutura dos vértices do grafo. Inclui suporte a atributos como `WEIGHT` e `LABEL`, permitindo a adição de informações personalizadas aos vértices.
- **models/graph_components/edge.py**: Define a estrutura das arestas do grafo. Suporta atributos como `WEIGHT` e `LABEL`, que podem ser usados para ponderação e identificação de arestas.
- **models/graph_representations/graph_representation.py**: Classe abstrata que define a interface comum para todas as representações de grafos. Inclui métodos abstratos para operações como criação de arestas, verificação de adjacências, e exportação.
- **models/graph_representations/graph_representations_types.py**: Define os tipos de representações de grafos disponíveis (`ADJACENCY_LIST`, `ADJACENCY_MATRIX`, `INCIDENCE`).
- **models/graph_representations/adjacency/adjacency_matrix_representation.py**: Implementa a representação de grafos usando matriz de adjacência.
- **models/graph_representations/adjacency/adjacenty_list_representation.py**: Implementa a representação de grafos usando listas de adjacência.
- **models/graph_representations/incidence/incidence_representation.py**: Implementa a representação de grafos por incidência.

De uma maneira um pouco mais visual abaixo temos a representação da estrutura dos arquivos

```
|-- src/
    |-- graph.py
    |-- models/
        |-- graph_components/
        |   |-- edge.py
        |   |-- vertex.py
        |-- graph_representations/
            |-- graph_representation.py
            |-- graph_representations_types.py
```

```
|-- adjacency/
|   |-- adjacency_matrix_representation.py
|   |-- adjacenty_list_representation.py
|-- incidence/
    |-- incidence_representation.py
```

4. Como utilizar

O primeiro passo é instalar a dependência `networkx` que permite a geração do arquivo ".gexf", para fazer isso basta executar o comando `pip install -r requirements.txt`. Logo após, através da classe **Graph** localizada no arquivo "graph.py" você já pode começar a manipular

4.1. Inicialização

A classe **Graph** aceita dois parâmetros durante a inicialização:

- `quantity_of_vertices`: Número inteiro que define a quantidade de vértices do grafo.
- `representations` (opcional): Lista contendo os tipos de representação interna que o grafo utilizará.

Por padrão, a representação baseada em incidência é sempre incluída, independentemente das escolhas do usuário. Caso o parâmetro `representations` não seja fornecido, todas as representações disponíveis (matriz de adjacência, lista de adjacência e matriz de incidência) serão utilizadas simultaneamente.

4.2. Exemplo de uso

```
from src.models.graph import Graph
from src.models.graph_representations.
   graph_representations_types import GraphRepresentationType
# Grafo apenas com represetation de lista de adjacencia
grafo = Graph(
   quantity_of_vertices=5,
   representations=[GraphRepresentationType.ADJACENCY_LIST]
)

# Grafo com todas as representations possiveis
grafo_completo = Graph(quantity_of_vertices=10)

# Adicionando arestas
grafo.create_edge(0, 1) # Cria uma aresta entre os vertices 0 e
   1
grafo.create_edge(1, 2) # Cria uma aresta entre os vertices 1 e
   2

#Verificando adjacencias
if grafo.is_vertexes_adjacent(0, 1):
   print("Vertices 0 e 1 sao adjacentes")
```

```
# Exportando o grafo para visualizacao no Gephi
arquivo_exportado = grafo.export_graph()
print(f"Grafo exportado para: {arquivo_exportado}")
```

As operações são executadas em todas as representações internas automaticamente, garantindo consistência dos dados independentemente da representação escolhida para consulta.

5. Análise do Repositório core do Vuejs

Para efeitos de analisar o comportamento da biblioteca e fazer um trabalho utilizando alguns conceitos aprendidos durante a matéria, foi coletado todas as issues e pull requests do repositório core do vuejs.

5.1. Coleta de dados

Conforme mencionado anteriormente, foram coletadas todas as issues e pull requests do repositório. Em seguida, identificaram-se os autores desses objetos, bem como os autores dos comentários e reações associados. A partir desses dados, foi construído um grafo não direcionado, em que cada vértice representa um usuário e cada aresta corresponde a uma interação entre usuários. O peso das arestas foi definido com base na soma das interações realizadas, conforme descrito a seguir:

- Usuário A abre PR e usuário B faz merge: Peso 3
- Usuário B aprova/solicita mudanças no PR de A: Peso 2
- Usuário B comenta na Issue/PR de A: Peso 2
- Usuário A mencioinou B em algum comentário: Peso 1
- Usuário A reagiu a comentário de B: Peso 1

Abaixo uma visualização completa do grafo:

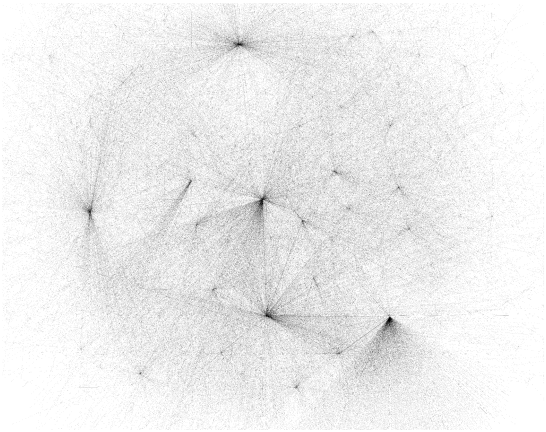

Figure 7. Grafo final

5.2. Análise

Com o grafo já montado seis perguntas foram feitas, nas próximas seções será apresentado a resposta para cada uma das perguntas e como a análise foi feita.

5.2.1. Quem são os 5 usuários mais influentes?

Para responder essa pergunta os 5 vértices com os maiores pesos foram encontrados

- yyx990803 com um peso total de 10320
- edison1105 com um peso total de 6389
- LinusBorg com um peso total de 4004
- github-actions com um peso total de 3646
- posva com um peso total de 3275

Uma coisa interessante é que se observar os pontos mais pretos do grafo perceberá que se tratam justamente destes usuários.

5.2.2. Remover quem gera maior fragmentação?

Para responder essa questão o seguinte foi feito

- Montar os componentes do grafo
- Para cada vértice do grafo fazer sua remoção
- Montar os novos componentes do grafo
- Fazer isso até encontrar o vértice que faz o maior número possível de componentes

O resultado final foi que se removermos o vértice do usuário pkg-pr-new teremos uma adição de 790 novos componentes, isso pode ser ilustado pela imagem abaixo observando o aumento de áreas mais cinzas, representando uma quantidade menor de conexões

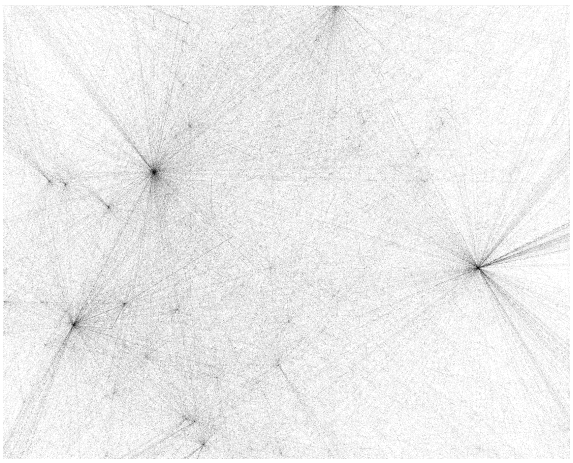

Figure 8. Grafo com maior fragmentação

5.2.3. Quais grupos naturais existem?

Para responder essa questão foram encontrados os componentes do grafo que possuem mais de 1 vértice. Usando está abordagem foram encontrados 23 grupos, sendo o maior deles composto por 10596 vértices

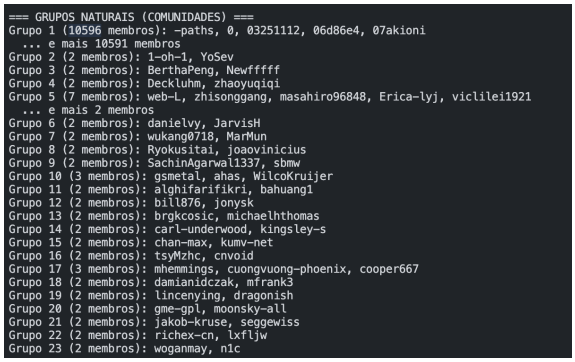

Figure 9. Grupos naturais

5.2.4. Qual o nível (percentual) de conexão da comunidade?

Para responder essa questão foi pensado no seguinte:

- Calcuar a quantidade máxima de combinações de pares no grafo ([n*(n-1)]/2)
- Para cada vértice do grafo faz uma busca em largura para descobrir a quais vértices ele se conecta diretamente e indiretamente
- Divide a quantidade de conexões encontradas pela quantidade total possível de conexões

Dito isso a resposta foi de 99.04 porcento de conexão

5.2.5. Dado um usuário, quem são os mais próximos?

Para realizar esse experimento, foi pego o usuário mais influente ”yyx990803” e realizado os seguintes passos

- Obter arestas do vértice
- Obter os maiores pesos

O resultado (limitado a 5) foi o seguinte:

- edison1105: peso total de 572
- renovate: peso total de 432
- sxzz: peso total de 273
- HcySunYang: peso total de 266
- pikax: peso total de 240

5.2.6. Dado um usuário, quem são os mais próximos que não interagem?

Para realizar esse experimento, foi pego o usuário mais influente ”yyx990803” e realizado os seguintes passos

- Obter a distância de cada vértice através de uma busca em largura (usuários diretos tem distância igual a 1 e indiretos igual a 2)
- Listar usuários com distância igual a 2 (nem tão próximo nem tão distante)

O resultado (limitado a 5) foi o seguinte:

- types
- xesxen
- WesCook
- shawfix
- ferreraa