Trabalho Prático Teoria de Grafos

Arthur Freitas Jardim[1], **Wilken Henrique Moreira**[2]

[1 1]Pontifícia Universidade Católica de Minas Gerais (PUC Minas)
R. Dom José Gaspar, 500 - Coração Eucarístico, Belo Horizonte - MG, 30535-901

[22]Instituto de Ciências Exatas e Informática (ICEI)
Pontifícia Universidade Católica de Minas Gerais (PUC Minas) – Belo Horizonte, MG – Brazil

wilkenhenriquemoreira@gmail.com, arthurfreitasjardim@gmail.com

Abstract. *This report presents the practical work for the Graph Theory course, part of the Software Engineering program at PUC Minas. The project aims to develop a library for graph manipulation, implemented in Python. The library will include functionalities for graph representation and manipulation, such as creating graphs, adding and removing edges, and checking graph properties.*

Resumo. *Este relatório apresenta o trabalho prático da disciplina Teoria dos Grafos, do curso de Engenharia de Software da PUC Minas. O objetivo do projeto é desenvolver uma biblioteca para manipulação de grafos, implementada em Python. A biblioteca incluirá funcionalidades para representação e manipulação de grafos, como criação de grafos, adição e remoção de arestas, e verificação de propriedades dos grafos.*

1. Introdução

Este trabalho prático tem como objetivo aplicar os conhecimentos adquiridos na disciplina de Teoria dos Grafos, através do desenvolvimento de uma biblioteca para manipulação de grafos, implementada em Python. A escolha do Python se deve à sua simplicidade e expressividade, o que facilita o foco nos conceitos de grafos em vez de detalhes de implementaço.

A biblioteca desenvolvida abrange diversas funcionalidades essenciais para a manipulação de grafos, incluindo:

- Representação de grafos utilizando diferentes estruturas de dados, como matriz de adjacência, matriz de incidência e lista de adjacência.
- Funções básicas para manipulação de grafos, incluindo:
  - Criação de um grafo com X vértices, onde o número de vértices deve ser inserido pelo usuário.
  - Adição de arestas, com controle sobre o direcionamento.
  - Remoção de arestas.
  - Ponderação de vértices e arestas.
  - Rotulação de vértices e arestas.
  - Checagem de adjacência entre vértices e entre arestas.
  - Checagem da existência de arestas.
  - Checagem da quantidade de vértices e arestas.
  - Verificação de grafo vazio e grafo completo.

  - Verificação da conectividade do grafo, incluindo:
    - Grafos simplesmente conexos.
    - Grafos semi-fortemente conexos.
    - Grafos fortemente conexos.
  - Checagem da quantidade de componentes fortemente conexos utilizando o algoritmo de Kosaraju.
  - Checagem de pontes e articulações.
- Verificação de propriedades dos grafos, como adjacência entre vértices, existência de arestas, conectividade, e detecção de componentes fortemente conexos.

Além disso, o projeto é dividido em três partes:

1. **Desenvolvimento da biblioteca para manipulação de grafos**: Incluindo todas as funcionalidades básicas mencionadas.
2. **Implementação de algoritmos para detecção de pontes**: Utilizando tanto um método ingênuo quanto um método baseado no algoritmo de Tarjan, seguido pela aplicação do Algoritmo de Fleury para encontrar caminhos eulerianos.
3. **Exportação e visualização de grafos**: Implementação de funcionalidades para leitura e escrita de grafos em formatos compatíveis com o software Gephi, e geração de visualizações utilizando essa ferramenta.

Este relatório documenta o processo de desenvolvimento da biblioteca, os algoritmos implementados, os resultados obtidos, e as conclusões derivadas da análise dos grafos manipulados.

2. Atribuição de Responsabilidades

Parte	Responsabilidade	Responsável
Parte 1	Desenvolver biblioteca para manipulação de grafos	Wilken Henrique Moreira
	Representação de grafos (Matriz de Adjacência, Matriz de Incidência, Lista de Adjacência)	Wilken Henrique Moreira
	Funções básicas de manipulação de grafos (Criação, remoção de arestas, checagens diversas)	Wilken Henrique Moreira
	Ponderação e rotulação de vértices e arestas	Wilken Henrique Moreira
	Checagem de conectividade (simplesmente conexo, semi-fortemente conexo, fortemente conexo)	Arthur Freitas Jardim
	Checagem de componentes fortemente conexos com o algoritmo de Kosaraju	Arthur Freitas Jardim
Parte 2	Identificação de pontes (Método naive e método baseado em Tarjan)	Arthur Freitas Jardim
	Implementação de detecção de pontes com o Algoritmo de Fleury	Arthur Freitas Jardim
	Análise de tempos computacionais para diferentes tamanhos de grafo	Arthur Freitas Jardim
Parte 3	Implementação de leitura e salvamento de grafos em arquivos compatíveis com Gephi	Wilken Henrique Moreira
	Suporte aos formatos GEXF, GDF, GML, GraphML, Pajek NET, GraphViz DOT, CSV, UCINET DL, Tulip TPL, Netdraw VNA, Spreadsheet	Wilken Henrique Moreira
	Documentação dos formatos de arquivo	Wilken Henrique Moreira

Figure 1. Diagrama de Classe da Biblioteca de Manipulação de Grafos

3. Diagramas

Nesta seção, são apresentados os diagramas utilizados no desenvolvimento da biblioteca de manipulação de grafos. Os diagramas a seguir ilustram a estrutura do sistema e os principais casos de uso do software.

3.1. Diagrama de Classe

A escolha de dividir a implementação da biblioteca em três classes principais — *Aresta*, *Vertice* e *Grafo* — segue o paradigma de orientação a objetos (OO) e foi feita para garantir modularidade, flexibilidade e clareza no design. A separação dessas responsabilidades facilita a manutenção e evolução do código, permitindo que cada classe se concentre em uma parte específica da representação do grafo, sem criar dependências excessivas entre os componentes.

A classe *Vertice* encapsula as propriedades e comportamentos associados aos vértices, como o nome, atributos e métodos para manipulação desses dados. A classe *Aresta*, por sua vez, é responsável por representar a conexão entre dois vértices, contendo propriedades como peso, rótulo e direção, que são essenciais para operações como busca e manipulação de arestas. Por fim, a classe *Grafo* serve como a estrutura de gerenciamento principal, armazenando os vértices e arestas, além de fornecer os métodos para adicionar, remover e interagir com esses elementos.

Essa abordagem modular, que segmenta claramente as responsabilidades de cada classe, facilita a implementação de novos algoritmos ou funcionalidades, já que cada parte do sistema pode ser modificada ou extendida independentemente. Além disso, ao seguir os princípios de OO, garantimos que o código seja mais reutilizável e fácil de testar, ao mesmo tempo que promove a coesão e o baixo acoplamento entre os componentes do sistema. O diagrama de classe ilustrado na Figura 2 apresenta a estrutura detalhada das relações entre as classes.

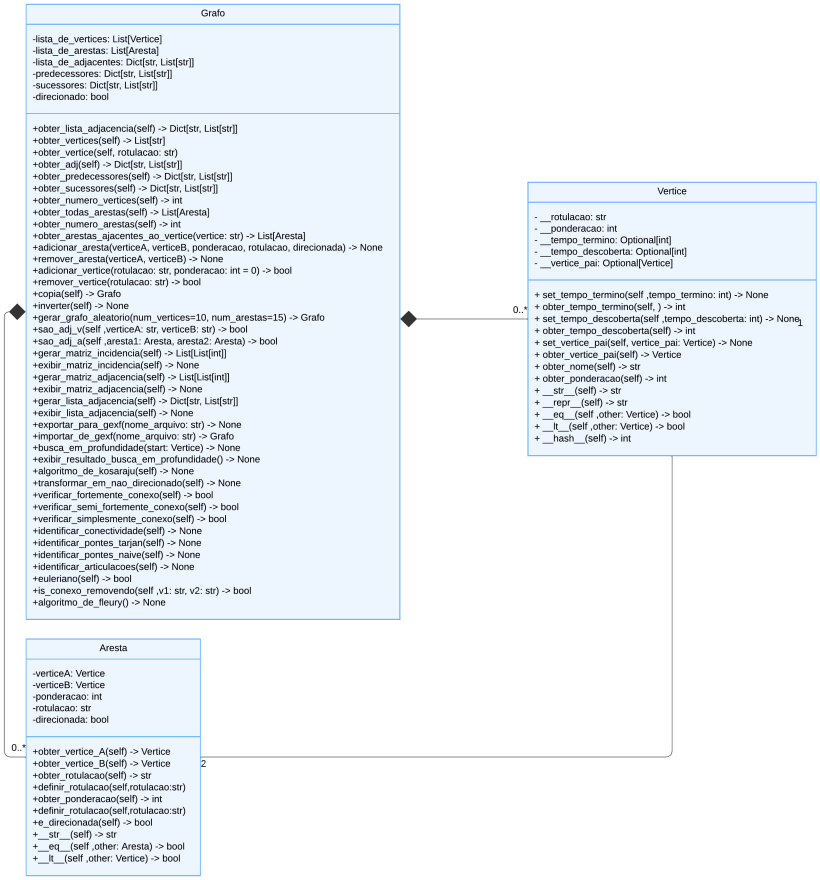


Figure 2. Diagrama de Classe da Biblioteca de Manipulação de Grafos

3.2. Diagrama de Caso de Uso

O diagrama de caso de uso ilustra as principais interações entre os usuários e o sistema, destacando os casos de uso essenciais para o funcionamento da biblioteca de grafos. Abaixo está o diagrama de caso de uso (ver Figura 3):

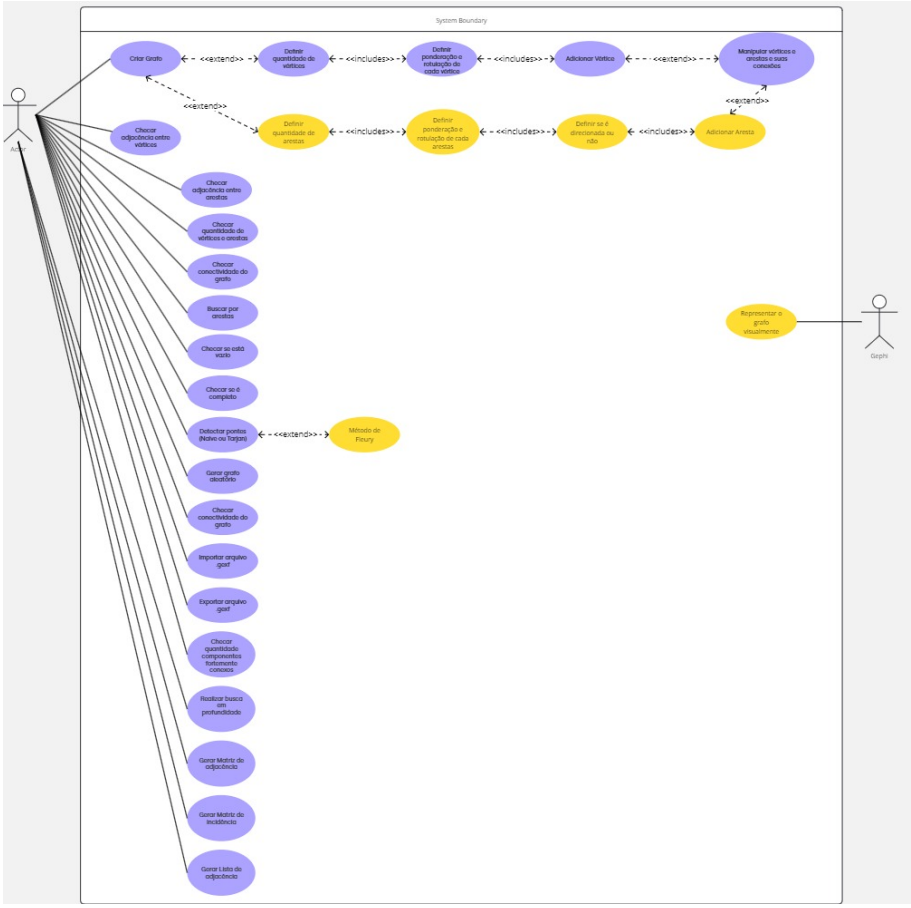


Figure 3. Diagrama de Caso de Uso da Biblioteca de Manipulação de Grafos

4. Algoritmos

Nesta seção, apresentamos as implementações de diversos algoritmos utilizados para operações sobre grafos. Esses algoritmos são essenciais para a análise e manipulação de estruturas de grafos em diversos contextos.

4.1. Busca em Profundidade (DFS)

A Busca em Profundidade (DFS) é um algoritmo que explora um grafo visitando um vértice e, em seguida, recursivamente explorando seus vértices adjacentes até que todos os vértices sejam visitados. Sua complexidade temporal é $O(V + E)$, onde V é o número de vértices e E é o número de arestas, já que cada vértice e aresta é percorrido uma única vez. Caso o vértice inicial não seja fornecido, o algoritmo começa pelo primeiro vértice da lista de vértices.

```
def algoritmo_de_kosaraju(self):
        if self.esta_vazio():
            print("O grafo está vazio. Adicione vértices e arestas antes de executar o algoritmo.")
            return []

        self.busca_em_profundidade()

        self.__lista_de_vertices.sort(
            key=lambda v: v.obter_tempo_termino() if v.obter_tempo_termino() is not None else float('-inf'),
            reverse=True
        )

        grafo_reverso = self.inverter()

        grafo_reverso.busca_em_profundidade()

        visitados = grafo_reverso.busca_em_profundidade()

        count = 0
        for vertice in visitados:
            pai = vertice.obter_vertice_pai()
            if pai is None:
                count+=1

        return count
```

Figure 4. Diagrama do algoritmo de Busca em Profundidade (DFS).

4.2. Algoritmo de Kosaraju

O algoritmo de Kosaraju é eficiente para encontrar os componentes fortemente conexos em um grafo direcionado. Sua complexidade é $O(V + E)$, onde V é o número de vértices e E o número de arestas. Isso ocorre porque o algoritmo realiza duas buscas em profundidade: a primeira para ordenar os vértices conforme o tempo de término e a segunda para identificar os componentes no grafo invertido. Além disso, a inversão do grafo também tem complexidade $O(V + E)$, pois cada vértice e aresta precisam ser processados. Portanto, o algoritmo é eficiente, com complexidade linear em relação ao número total de vértices e arestas.

A detecção de componentes fortemente conexos no grafo é realizada com o algoritmo de Kosaraju, que identifica subgrafos onde cada vértice é acessível de qualquer outro vértice dentro da componente. O processo é composto por duas etapas principais de busca em profundidade (DFS): uma no grafo original e outra no grafo invertido [1].

- **Verificação do Grafo Vazio:** O algoritmo começa verificando se o grafo está vazio. Caso o grafo não contenha vértices ou arestas, o método retorna uma mensagem informando a necessidade de adicionar elementos ao grafo.

- **Busca em Profundidade (DFS) no Grafo Original:** A primeira busca em profundidade é realizada no grafo original para determinar a ordem de término dos vértices.
- **Ordenação dos Vértices:** Após a DFS, os vértices são ordenados com base no tempo de término da busca, o que define a ordem em que serão visitados no grafo invertido.
- **Grafo Reverso:** O grafo original é invertido, ou seja, todas as arestas são direcionadas na direção oposta, formando o grafo reverso.
- **Segunda Busca em Profundidade no Grafo Reverso:** A busca em profundidade é executada novamente, mas agora no grafo invertido, seguindo a ordem dos vértices obtida na primeira busca.
- **Contagem das Componentes Fortemente Conexas:** O número de componentes fortemente conexas é determinado ao contar quantos vértices possuem um vértice pai nulo, indicando que são raízes de componentes fortemente conexas.

O método retorna o número de componentes fortemente conexas no grafo.

```
def algoritmo_de_kosaraju(self):
        if self.esta_vazio():
            print("O grafo está vazio. Adicione vértices e arestas antes de executar o algoritmo.")
            return []

        self.busca_em_profundidade()

        self.__lista_de_vertices.sort(
            key=lambda v: v.obter_tempo_termino() if v.obter_tempo_termino() is not None else float('-inf'),
            reverse=True
        )

        grafo_reverso = self.inverter()

        grafo_reverso.busca_em_profundidade()

        visitados = grafo_reverso.busca_em_profundidade()

        count = 0
        for vertice in visitados:
            pai = vertice.obter_vertice_pai()
            if pai is None:
                count+=1

        return count
```

Figure 5. Algoritimo de Kosaraju

4.3. Algoritmo de Tarjan

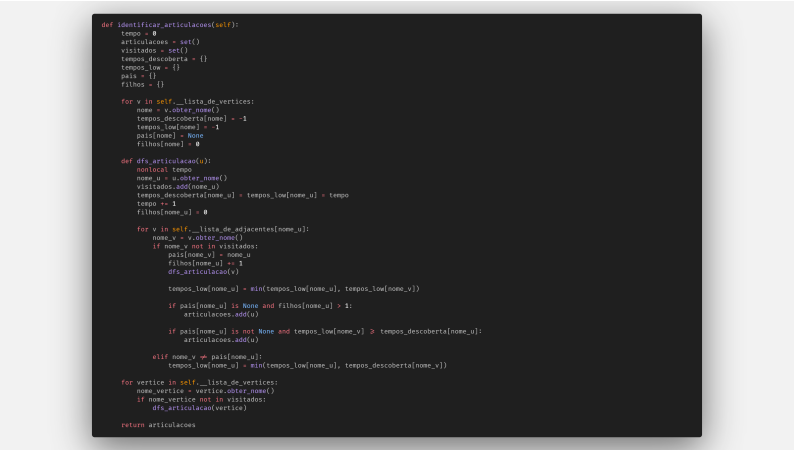
O algoritmo de Tarjan é utilizado para identificar componentes fortemente conexos em grafos direcionados, além de ser uma ferramenta eficaz para detectar pontes e vértices de articulação. Na implementação desta biblioteca, o algoritmo foi adaptado para identificar principalmente pontes e articulações, por meio dos métodos `identificar_pontes_tarjan` e `identificar_articulacoes`.

Pontes são arestas cuja remoção desconecta o grafo, e articulações são vértices cuja remoção aumenta o número de componentes conexos. Pontes são identificadas quando o valor `low-link` de um vértice adjacente é maior que o tempo de descoberta do vértice atual. Já os vértices de articulação são detectados quando um vértice possui mais de um filho ou quando o `low-link` de um vértice adjacente é maior ou igual ao tempo de descoberta do vértice. [3]

A detecção de pontes e articulações é fundamental para entender a estrutura de conectividade do grafo e sua vulnerabilidade a falhas. A complexidade do algoritmo é $O(V + E)$, onde V representa o número de vértices e E o número de arestas, devido à execução de uma única busca em profundidade sobre o grafo.

```
def identificar_pontes_tarjan(self):
        visitados = set()
        low = {}
        descobertas = {}
        pontes = []
        t = 0

        for v in self.__lista_de_vertices:
            low[v.obter_nome()] = float('inf')
            descobertas[v.obter_nome()] = -1

        def dfs_pontes(v):
            nonlocal t
            visitados.add(v.obter_nome())
            descobertas[v.obter_nome()] = low[v.obter_nome()] = t
            t += 1
            for adj in self.__lista_de_adjacentes[v.obter_nome()]:

                if adj.obter_nome() not in visitados:
                    adj.set_vertice_pai(v)
                    dfs_pontes(adj)

                    low[v.obter_nome()] = min(low[v.obter_nome()], low[adj.obter_nome()])

                    if low[adj.obter_nome()] > descobertas[v.obter_nome()]:
                        pontes.append((v.obter_nome(), adj.obter_nome()))

                elif adj.obter_nome() != (v.obter_vertice_pai().obter_nome() if v.obter_vertice_pai() else None):
                    low[v.obter_nome()] = min(low[v.obter_nome()], descobertas[adj.obter_nome()])

        for v in self.__lista_de_vertices:
            if v.obter_nome() not in visitados:
                dfs_pontes(v)

        pontes.reverse()
        return pontes
```

Figure 6. Diagrama do algoritmo de Tarjan para detecção de pontes.

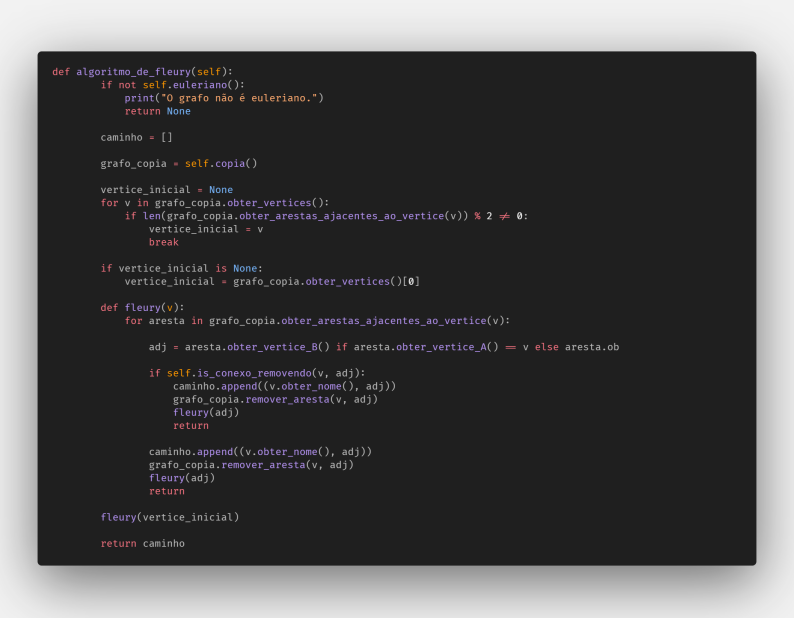


```
def identificar_articulacoes(self):
        tempo = 0
        articulacoes = set()
        visitados = set()
        tempos_descoberta = {}
        tempos_low = {}
        pais = {}
        filhos = {}

        for v in self.__lista_de_vertices:
            nome = v.obter_nome()
            tempos_descoberta[nome] = -1
            tempos_low[nome] = -1
            pais[nome] = None
            filhos[nome] = 0

        def dfs_articulacao(u):
            nonlocal tempo
            nome_u = u.obter_nome()
            visitados.add(nome_u)
            tempos_descoberta[nome_u] = tempos_low[nome_u] = tempo
            tempo += 1
            filhos[nome_u] = 0

            for v in self.__lista_de_adjacentes[nome_u]:
                nome_v = v.obter_nome()
                if nome_v not in visitados:
                    pais[nome_v] = nome_u
                    filhos[nome_u] += 1
                    dfs_articulacao(v)

                    tempos_low[nome_u] = min(tempos_low[nome_u], tempos_low[nome_v])

                    if pais[nome_u] is None and filhos[nome_u] > 1:
                        articulacoes.add(u)

                    if pais[nome_u] is not None and tempos_low[nome_v] >= tempos_descoberta[nome_u]:
                        articulacoes.add(u)

                elif nome_v != pais[nome_u]:
                    tempos_low[nome_u] = min(tempos_low[nome_u], tempos_descoberta[nome_v])

        for vertice in self.__lista_de_vertices:
            nome_vertice = vertice.obter_nome()
            if nome_vertice not in visitados:
                dfs_articulacao(vertice)

        return articulacoes
```

Figure 7. Diagrama do algoritmo de Tarjan para detecção de pontes.

4.4. Algoritmo de Fleury

O algoritmo de Fleury é utilizado para encontrar um caminho ou ciclo Euleriano em um grafo. Este algoritmo percorre o grafo removendo arestas de forma a garantir que um ciclo Euleriano seja formado, respeitando as condições de Euler. [2]



```
def algoritmo_de_fleury(self):
        if not self.euleriano():
            print("O grafo não é euleriano.")
            return None

        caminho = []

        grafo_copia = self.copia()

        vertice_inicial = None
        for v in grafo_copia.obter_vertices():
            if len(grafo_copia.obter_arestas_ajacentes_ao_vertice(v)) % 2 != 0:
                vertice_inicial = v
                break

        if vertice_inicial is None:
            vertice_inicial = grafo_copia.obter_vertices()[0]

        def fleury(v):
            for aresta in grafo_copia.obter_arestas_ajacentes_ao_vertice(v):

                adj = aresta.obter_vertice_B() if aresta.obter_vertice_A() == v else aresta.ob

                if self.is_conexo_removendo(v, adj):
                    caminho.append((v.obter_nome(), adj))
                    grafo_copia.remover_aresta(v, adj)
                    fleury(adj)
                    return

                caminho.append((v.obter_nome(), adj))
                grafo_copia.remover_aresta(v, adj)
                fleury(adj)
                return

        fleury(vertice_inicial)

        return caminho
```

Figure 8. Diagrama do algoritmo de Fleury para encontrar um caminho ou ciclo Euleriano.

4.5. Algoritmo de Naive

O algoritmo Naive é utilizado para identificar pontes em um grafo. Uma ponte é uma aresta cuja remoção aumenta o número de componentes conectados do grafo, ou seja, desconecta uma parte do grafo. Este algoritmo verifica cada aresta individualmente, removendo-a e verificando se o grafo continua conectado. Se a remoção da aresta desconectar o grafo, essa aresta é considerada uma ponte.

```
def identificar_pontes_naive(self):
        pontes = []
        for aresta in self.__lista_de_arestas:
            verticeA = aresta.obter_vertice_A().obter_nome()
            verticeB = aresta.obter_vertice_B().obter_nome()
            ponderacao = aresta.obter_ponderacao()
            rotulacao = aresta.obter_rotulacao()
            direcionada = aresta.e_direcionada()

            self.remover_aresta(verticeA, verticeB)

            if self.identificar_conectividade() == "não conexo":
                pontes.append((verticeA, verticeB))

            self.adicionar_aresta(verticeA, verticeB, ponderacao, rotulacao, direcionada)

        print("Naive", pontes)
        return pontes
```

Figure 9. Diagrama do algoritmo de Naive para encontrar pontes.

5. Manipulação do Grafo

Nesta seção, são descritas as funções básicas implementadas para a manipulação do grafo. A seguir, apresentamos as manipulações realizadas em nossa biblioteca de grafos, incluindo a criação, modificação e verificação de propriedades dos grafos.

5.1. Criação de um Grafo

A criação de um grafo é realizada apenas instanciando a classe `Grafo(isDirecionado=True)`. O usuário pode especificar se o grafo será direcionado ou não-direcionado. Por padrão, caso não seja preenchido, o grafo é criado como `False`, e após a inserção de uma aresta direcionada, ele é automaticamente definido como `True`.

5.1.1. Criação de um Grafo Aleatório

O método `gerar_grafo_aleatorio(num_vertices=10, num_arestas=15, max_arestas_por_vertice=None, direcionadas=None)` é utilizado para criar um grafo aleatório com um número definido de vértices e arestas. O grafo pode ser direcionado ou não-direcionado, e as arestas podem ser ponderadas e rotuladas de maneira aleatória. O número de vértices (`num_vertices`) e de arestas (`num_arestas`) são especificados pelo usuário, e o método inclui verificações para garantir que o número de arestas não exceda o máximo permitido, conforme a configuração do grafo. Este método é útil para testes, permitindo a geração de grafos com diferentes características.

5.2. Adição e Remoção de Vértices

A adição e remoção de vértices são operações fundamentais para a manipulação de grafos. Estas funções permitem ao usuário inserir ou excluir vértices no grafo, alterando sua estrutura conforme necessário.

O método `adicionar_vertice(rotulacao: str, ponderacao: int = 0)` permite ao usuário adicionar um novo vértice ao grafo. O parâmetro `rotulacao` é utilizado para identificar o vértice, enquanto `ponderacao` permite atribuir um valor de peso ao vértice, sendo este um parâmetro opcional que por padrão é definido como 0.

O método `remover_vertice(rotulacao: str) -> bool` permite ao usuário remover um vértice específico do grafo. O parâmetro `rotulacao` é utilizado para identificar o vértice a ser removido. O método retorna `True` se o vértice for removido com sucesso e `False` caso o vértice não exista no grafo.

Ao remover um vértice, todas as arestas associadas a ele também são removidas, o que altera a estrutura do grafo conforme a representação escolhida (matriz de adjacência, matriz de incidência ou lista de adjacência). Além disso, as listas de predecessores e sucessores também são atualizadas para refletir a remoção do vértice de acordo com o tipo do grafo.

5.3. Adição e Remoção de Arestas

A função de adição de arestas permite ao usuário inserir novas arestas entre vértices do grafo. A adição pode ser direcionada ou não, dependendo das configurações fornecidas pelo usuário. A remoção de arestas é realizada ao remover as conexões entre dois vértices especificados.

O método `adicionar_aresta(verticeA, verticeB, ponderacao, rotulacao, direcionada)` permite adicionar uma aresta

entre dois vértices, com a possibilidade de definir se a aresta é direcionada ou não. Caso a aresta seja direcionada, ela será inserida no grafo apenas se não houver outra aresta com os mesmos vértices de origem e destino. Para arestas não direcionadas, a verificação leva em conta a relação entre os dois vértices de forma bidirecional. Se a aresta for adicionada com sucesso, o método atualizará as estruturas de adjacência, predecessores e sucessores do grafo.

O método `remover_aresta(verticeA, verticeB)` permite remover uma aresta entre dois vértices. A remoção envolve a exclusão da aresta na lista de arestas, além de atualizar as listas de adjacência, sucessores e predecessores dos vértices envolvidos. A remoção considera se o grafo é direcionado ou não, alterando a estrutura de acordo com o tipo de aresta presente no grafo. Ao remover a aresta, as conexões associadas a ela são igualmente removidas, o que altera a conectividade entre os vértices.

5.4. Ponderação e Rotulação de Vértices e Arestas

A ponderação e a rotulação de vértices e arestas permitem associar valores e identificadores a esses elementos do grafo. A ponderação envolve a atribuição de um valor numérico (peso) a vértices e arestas, o que pode ser útil para algoritmos como o de Dijkstra, que exigem a consideração de pesos. Já a rotulação consiste em atribuir um nome ou identificador único a cada vértice ou aresta, facilitando a manipulação e a identificação no grafo.

Esses valores são definidos no momento da criação do vértice ou aresta, por meio da inserção dos parâmetros no construtor das classes `Aresta` ou `Vertice`. Entretanto, tanto a ponderação quanto a rotulação podem ser alteradas a qualquer momento, oferecendo flexibilidade para modificar os dados durante a execução do programa.

A obtenção e definição de sua rotulação e ponderação podem ser feitas com os seguintes métodos:

- `obter_nome()` para acessar o nome (rotulação) do vértice.
- `definir_nome(rotulacao: str)` para definir ou alterar a rotulação do vértice.
- `obter_ponderacao()` para acessar o valor de ponderação do vértice.
- `definir_ponderacao(ponderacao: int)` para definir ou alterar a ponderação do vértice.

Esses métodos permitem ao usuário acessar e modificar as informações associadas a cada vértice ou aresta conforme necessário.

5.5. Checagem de Adjacência entre Vértices

A função `sao_adj_v` verifica se dois vértices estão conectados por uma aresta, ou seja, se são adjacentes. Esta operação é fundamental para diversos algoritmos de grafos, como aqueles baseados em busca em profundidade e busca em largura.

A função percorre a lista de arestas do grafo e verifica se existe uma aresta que conecta os dois vértices. Se a aresta for direcionada, a função checa se a direção e os vértices correspondem exatamente. Caso a aresta seja não-direcionada, a função verifica a existência da aresta em qualquer direção entre os dois vértices. Se a aresta que conecta os vértices for encontrada, a função retorna `True`, indicando que os vértices são adjacentes. Caso contrário, a função retorna `False`.

5.6. Checagem de Adjacência entre Arestas

Similar à checagem de adjacência entre vértices, a checagem de adjacência entre arestas determina se duas arestas compartilham um vértice comum. Isso é importante para entender como as arestas estão conectadas no grafo e para diversas operações e algoritmos que lidam com a estrutura das conexões entre as arestas.

A função `sao_adj_a` verifica se duas arestas são adjacentes, ou seja, se elas compartilham pelo menos um vértice. Para isso, ela obtém os vértices de cada aresta e compara se algum vértice de uma aresta é igual a algum vértice da outra. Se houver um vértice comum entre as duas arestas, a função retorna `True`, indicando que as arestas são adjacentes. Caso contrário, ela retorna `False`, indicando que as arestas não são adjacentes.

5.7. Checagem da Existência de Arestas

Essa função permite verificar se uma aresta específica existe no grafo. A verificação é realizada através da lista de arestas do grafo, que armazena as conexões entre os vértices. A função `buscar_aresta` recebe os nomes de dois vértices e percorre a lista de arestas para verificar se existe uma aresta entre esses vértices. Se a aresta for direcionada, ela verifica se o vértice A está conectado ao vértice B na direção correta. Para arestas não direcionadas, a função verifica se os vértices estão conectados em ambas as direções. Caso a aresta seja encontrada, a função retorna `True`, caso contrário, retorna `False`, indicando que a aresta não existe.

5.8. Checagem da Quantidade de Vértices e Arestas

A função responsável pela checagem da quantidade de vértices e arestas retorna o número total de vértices e arestas presentes no grafo. Essa informação é útil para análise de complexidade de algoritmos que operam sobre o grafo, ajudando na otimização e no entendimento do impacto das operações realizadas.

5.9. Verificação de Grafo Vazio e Completo

O grafo vazio é aquele que não possui arestas, enquanto o grafo completo é aquele em que cada par de vértices está conectado por uma aresta. Funções específicas foram implementadas para verificar essas condições no grafo.

O método `esta_vazio()` retorna `True` caso o grafo não contenha vértices, verificando se o número de vértices é igual a zero.

Já o método `e_completo()` verifica se o grafo é completo. Ele primeiro calcula o número máximo de arestas possíveis com base no número de vértices. Em grafos não direcionados, o número máximo de arestas é dado pela fórmula $\frac{V \times (V-1)}{2}$, onde V é o número de vértices. Para grafos direcionados, o número máximo de arestas é $V \times (V-1)$. Se o número de arestas não for igual ao número máximo esperado, o método retorna `False`. Em seguida, o método verifica se cada vértice está conectado a todos os seus adjacentes, garantindo que todas as arestas possíveis estão presentes no grafo.

5.10. Verificação de Conectividade

A conectividade do grafo é uma das propriedades fundamentais, que indica se todos os vértices estão conectados entre si. A verificação da conectividade é realizada em três diferentes níveis:

- **Grafo simplesmente conexo**: Todos os vértices estão conectados por algum caminho.
- **Grafo semi-fortemente conexo**: Existe um caminho entre os vértices, mas pode não ser possível fazer esse caminho em ambas as direções (direcionado).
- **Grafo fortemente conexo**: Existe um caminho entre todos os pares de vértices em ambas as direções (direcionado).

5.10.1. Verificação de Grafo Fortemente Conexo

A função `verificar_fortemente_conexo()` verifica se um grafo é fortemente conexo. Primeiramente, ela checa se o grafo está vazio utilizando `esta_vazio()`. Caso o grafo não esteja vazio, a função executa o algoritmo de Kosaraju por meio de `algoritmo_de_kosaraju()`, que retorna o número de componentes fortemente conexos. O grafo é considerado fortemente conexo se houver apenas um componente, retornando `True`; caso contrário, retorna `False`.

5.10.2. Verificação de Grafo Semi-Fortemente Conexo

A verificação de um grafo semi-fortemente conexo é realizada através de uma busca em profundidade no grafo direcionado. Se todos os vértices forem alcançáveis, mas em apenas uma direção, o grafo é considerado semi-fortemente conexo. A função `verificar_semi_fortemente_conexo()` realiza a busca em profundidade e verifica se o número de vértices visitados é igual ao número total de vértices no grafo. Se a condição for satisfeita, o grafo é semi-fortemente conexo, retornando `True`, caso contrário, retorna `False`.

5.10.3. Verificação de Grafo Simplesmente Conexo

A verificação de um grafo simplesmente conexo ocorre quando o grafo é transformado em um grafo não-direcionado. Se todos os vértices estiverem conectados, o grafo é considerado simplesmente conexo. A função `verificar_simplesmente_conexo()` transforma o grafo em não-direcionado e realiza uma busca em profundidade. Se o número de vértices visitados for igual ao número total de vértices, o grafo é considerado simplesmente conexo, retornando `True`, caso contrário, retorna `False`.

5.11. Checagem de Componentes Fortemente Conexos

A checagem de componentes fortemente conexos em um grafo direcionado é realizada utilizando o algoritmo de Kosaraju, conforme detalhado na subseção 4.2. O algoritmo identifica subgrafos em que cada vértice é acessível de qualquer outro vértice dentro da componente. A implementação desse algoritmo envolve duas buscas em profundidade, sendo uma no grafo original e outra no grafo invertido. O número de componentes fortemente conexos é determinado ao contar os vértices que possuem um vértice pai nulo na segunda busca em profundidade. A análise completa do algoritmo de Kosaraju pode ser consultada na subseção mencionada.

6. Representação de Grafos

Nesta seção, abordamos diferentes formas de representar grafos em memória. Cada representação tem suas vantagens e é escolhida dependendo das operações que se deseja otimizar, como buscas de arestas, vértices ou verificação de conexões. As representações mais comuns são a Matriz de Adjacência, a Matriz de Incidência e a Lista de Adjacência.

6.1. Representação de Grafos

Os grafos podem ser representados de diversas formas, e a escolha da melhor forma depende do tipo de operação e da estrutura do grafo. As três representações mais utilizadas são:

6.1.1. Representação de Grafos Utilizando Matriz de Adjacência

A matriz de adjacência é uma matriz quadrada onde cada linha e cada coluna correspondem a um vértice do grafo. Se existe uma aresta entre dois vértices, o valor na célula correspondente da matriz é preenchido com um valor específico (geralmente 1 ou o peso da aresta). Se não houver aresta, o valor é 0 (ou infinito no caso de grafos ponderados). Esta representação é eficiente para grafos densos e permite verificar rapidamente a existência de uma aresta entre dois vértices, mas consome mais memória para grafos esparsos.

Matriz de Adjacência (linhas = vértices, colunas = vértices):

	1	2	3	4	5	6	7	8	9	10
1 :	0	0	0	0	0	0	0	0	0	0
2 :	0	0	0	0	1	0	0	0	0	0
3 :	0	0	0	0	0	0	0	0	1	0
4 :	0	0	0	0	0	0	0	0	0	0
5 :	0	1	1	1	0	0	1	0	0	0
6 :	0	1	0	0	0	0	0	0	0	0
7 :	0	0	1	0	1	0	0	0	0	0
8 :	0	1	0	0	1	0	0	0	1	0
9 :	0	1	1	0	0	0	0	0	0	0
10:	0	0	0	1	1	0	0	0	0	0

Figure 10. Exemplo de representação de matriz de adjacência no Gephi.

6.1.2. Representação de Grafos Utilizando Matriz de Incidência

A matriz de incidência é uma matriz onde as linhas representam os vértices e as colunas representam as arestas. Cada célula na matriz indica se um vértice está incidente a uma aresta. Para grafos direcionados, a célula recebe um valor positivo se o vértice for o ponto de origem da aresta e um valor negativo se for o ponto de destino. Para grafos não direcionados, as células indicam que o vértice está conectado à aresta.

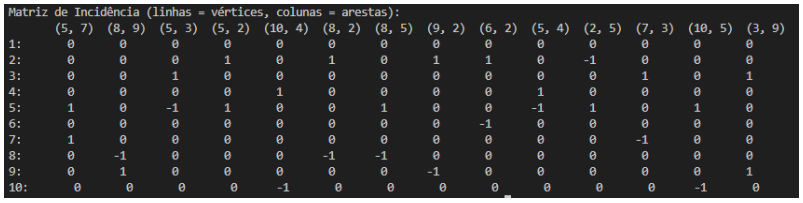


Figure 11. Exemplo de representação de matriz de incidência no Gephi.

6.1.3. Representação de Grafos Utilizando Lista de Adjacência

Na lista de adjacência, cada vértice é armazenado junto com uma lista de seus vértices adjacentes. Para grafos direcionados, se há uma aresta do vértice A para o vértice B, então o vértice B será adicionado à lista de adjacência de A. Para grafos não direcionados, ambos os vértices envolvidos na aresta aparecerão nas respectivas listas de adjacência. A lista de adjacência é uma das representações mais eficientes em termos de memória para grafos esparsos, já que armazena apenas as conexões existentes.

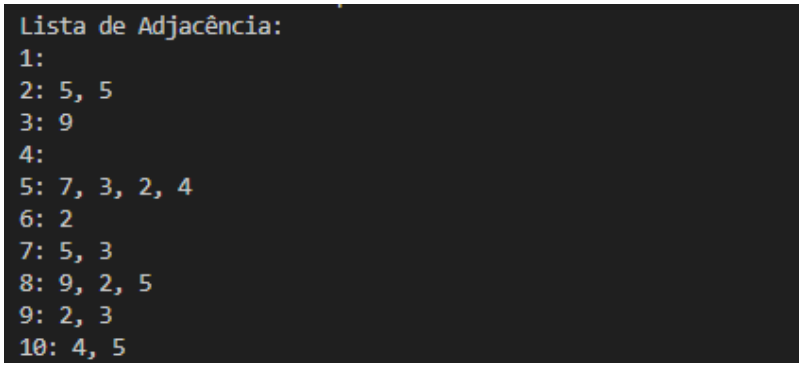


Figure 12. Exemplo de representação de lista de adjacência no Gephi.

7. Gephi e o Formato GEXF

O **Gephi** é um software popular para visualização e análise de grafos, amplamente utilizado em diversas áreas como ciência de redes e biologia. Ele suporta uma variedade de formatos de arquivo, incluindo GEXF, GDF, GML, GraphML, Pajek NET, GraphViz DOT, CSV, UCINET DL, Tulip TPL, entre outros. Esses formatos permitem a importação e exportação de grafos para diferentes ferramentas e plataformas. Contudo, o formato **GEXF (Graph Exchange XML Format)** se destaca por sua flexibilidade e escalabilidade, sendo ideal para representar grafos complexos com diversos atributos e conexões temporais. O GEXF é particularmente útil para trabalhar com grandes volumes de dados e grafos dinâmicos, tornando-se a escolha preferencial para análise de grafos de larga escala. [4]

Na biblioteca desenvolvida, é possível importar e exportar grafos em formato `.GEXF` através das funções `exportar_para_gexf()` e `importar_de_gexf(nome_arquivo: str)`. Essas funções permitem a conversão da estrutura interna do grafo para o formato XML e vice-versa, facilitando a integração com o Gephi. Uma vez exportado, o grafo pode ser renderizado e analisado

detalhadamente no Gephi, aproveitando suas capacidades avançadas de visualização e manipulação de grafos. O GEXF armazena informações detalhadas sobre vértices e arestas, incluindo atributos e propriedades temporais, proporcionando maior flexibilidade e eficiência na análise de redes complexas. Sua escalabilidade é essencial para sistemas que precisam processar grafos grandes e dinâmicos, tornando-o uma opção sólida para visualização e análise de grafos em projetos de grande porte.

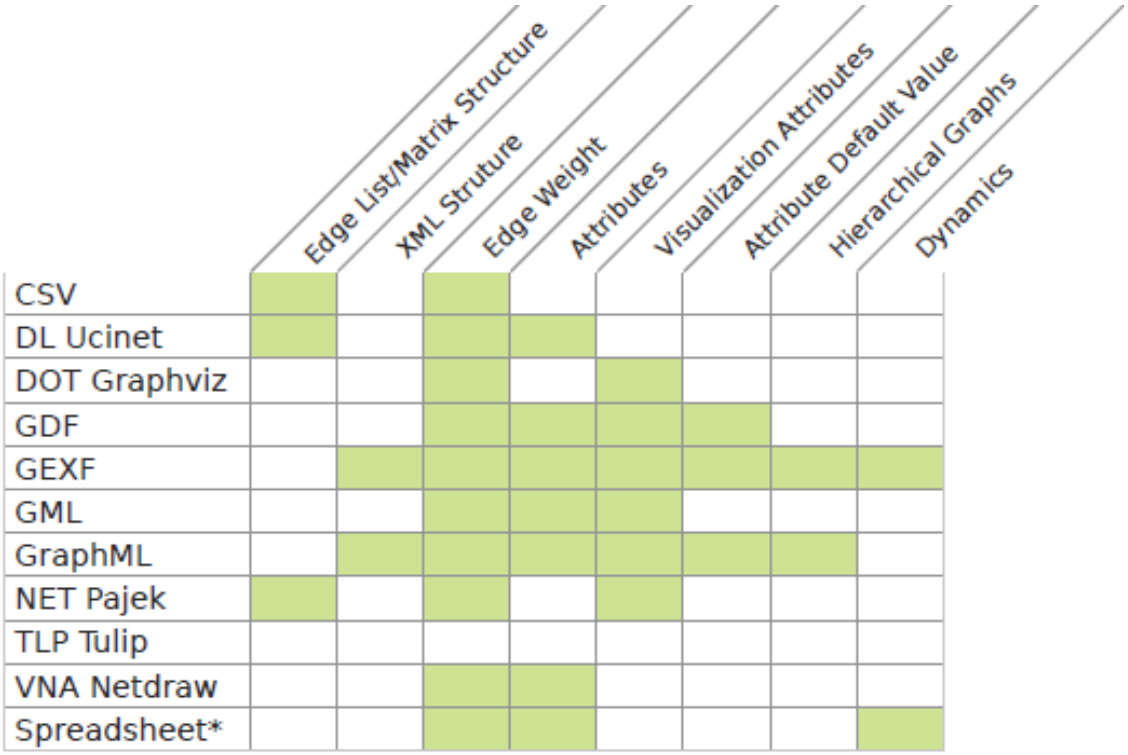


Figure 13. Formatos de Arquivo Suportados pelo Gephi

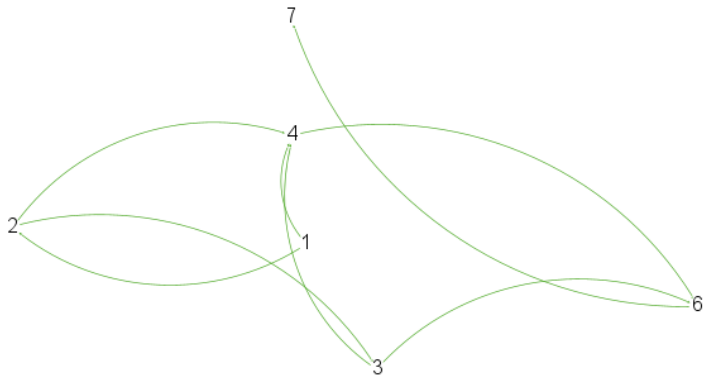


Figure 14. Exemplo de grafo gerado em .gexf no aplicativo Gephi.

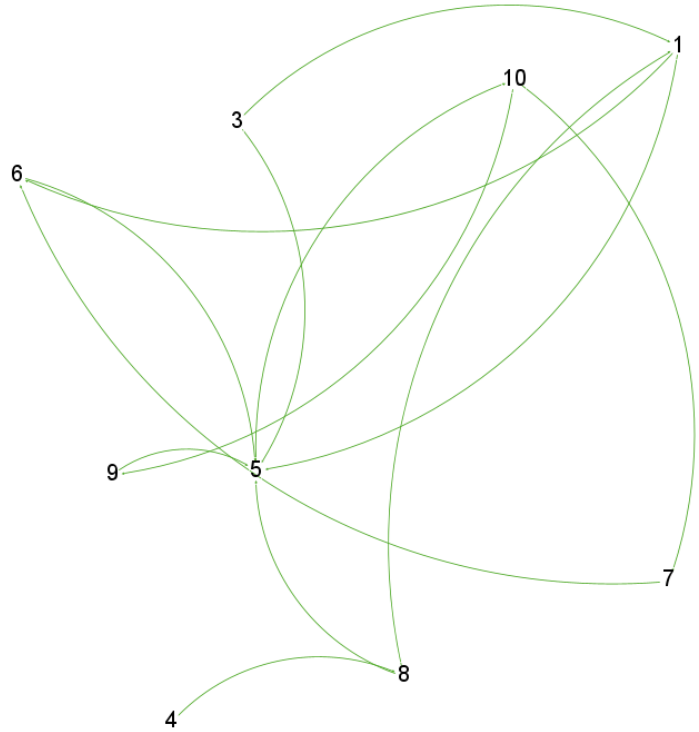


Figure 15. Exemplo de grafo gerado em .gexf no aplicativo Gephi.

References

[1] GeeksforGeeks. *Kosaraju's Algorithm for Strongly Connected Components*. Disponível em: https://www.youtube.com/watch?v=QlGuaHT1lzA&t=286s. Acesso em: 24 nov. 2024.

[2] GeeksforGeeks. *Fleury's Algorithm for Finding Eulerian Path*. Disponível em: https://www.youtube.com/watch?v=8MpoO2zA2l4. Acesso em: 24 nov. 2024.

[3] GeeksforGeeks. *Tarjan's Algorithm for Strongly Connected Components*. Disponível em: https://www.youtube.com/watch?v=qrAub5z8FeA. Acesso em: 24 nov. 2024.

[4] Dr. Martin Grandjean. *GEPHI - Introduction to Network Analysis and Visualization*. Disponível em: https://www.youtube.com/watch?v=371n3Ye9vVo&list=PLk_jmmkw5S2BqnYBqF2VNPcszY93-ze49. Acesso em: 24 nov. 2024.