Projeto de Teoria dos Grafos: Uma Implementação de Representação de Grafos

Adrielson F. Justino[1]

[1]Programa de Pós-Graduação em Engenharia de Computação
e Sistemas (PECS) – Universidade Estadual do Maranhão (UEMA)
São Luís, MA – Brazil

adrielferreira28@gmail.com

1. Introdução

Este relatório descreve o projeto desenvolvido e executado no âmbito da disciplina de Projeto e Análise de Algoritmos, oferecida pela Universidade Estadual do Maranhão, como parte integrante do curso de Engenharia da Computação. O foco central deste trabalho reside na criação de uma representação de grafos utilizando a estrutura de dados da Matriz de Adjacência.

Os grafos, enquanto estruturas matemáticas, possuem a capacidade de modelar uma vasta gama de situações do mundo real, como redes de transporte, interações sociais em plataformas digitais, fluxo de informação em sistemas computacionais, dentre outros [Adali and Ortega 2018]. Sua aplicação é vasta e abrange diversos campos do conhecimento, incluindo ciência da computação, engenharia, matemática aplicada e física [Egilmez et al. 2016].

Uma das maneiras de representar esses grafos é através da estrutura de dados conhecida como Matriz de Adjacências, que consiste em uma estrutura de dados que representa grafos por meio de uma matriz *booleana* [Al-Sayegh and Farsangi 2020]. Nessa representação, os vértices do grafo são dispostos em linhas e colunas da matriz, e os elementos indicam a presença ou ausência de uma aresta entre dois vértices [Paramadevan and Sotheeswaran 2021]. Essa abordagem possibilita a análise eficiente de propriedades dos grafos, tais como a presença de ciclos, conectividade entre vértices, entre outras características importantes[Sahoo 2019]. Além disso, também pode ser usada para problemas de identificação de componentes conexos [Prajwala 2019].

O problema de identificação de componentes (ou conectividade) em grafos refere-se à tarefa de determinar se um grafo é conectado, ou seja, se há um caminho entre cada par de vértices no grafo[Prajwala 2019, Glück 2017]. Um componente conexo em um grafo é um subconjunto de vértices onde cada par de vértices está conectado por um caminho [Dörfler et al. 2018]. Se um grafo possui apenas um componente conexo, ele é considerado conectado; caso contrário, é desconectado.

A identificação de componentes conexos é uma questão fundamental em teoria dos grafos e tem diversas aplicações práticas [Prajwala 2019, Dörfler et al. 2018]. Por exemplo, em redes de comunicação, é essencial garantir que todos os dispositivos estejam interconectados para garantir a transmissão eficiente de dados [Sahoo 2019]. Em logística, a conectividade de redes de transporte é crucial para planejar rotas eficientes. Além disso, em sistemas de redes sociais, a identificação de componentes conexos pode

revelar grupos de usuários que estão interconectados, permitindo análises mais detalhadas sobre padrões de interação.

Esta implementação foi direcionada para resolver problemas de identificação de Componentes Conexos, com especial atenção ao uso do algoritmo de busca em profundidade (Depth-First Search - DFS). Além disso, foram demonstradas diversas operações sobre grafos, tais como determinar se dois vértices são adjacentes, o cálculo do grau de um vértice, a busca dos vizinhos de um vértice específico e a visita de todas as arestas do grafo.

Ao longo deste documento, é fornecida uma visão detalhada do processo de implementação dos algoritmos referidos, juntamente com uma análise dos resultados obtidos. Espera-se que este projeto não apenas ofereça uma compreensão clara do trabalho realizado, mas também contribua para o avanço do conhecimento nesta área específica.

2. Descrição do Experimento

A condução do projeto seguiu uma abordagem sistemática, dividida em duas etapas principais. A primeira etapa de Implementação de Representação de Grafos envolveu a leitura e transformação dos dados de entrada em uma estrutura de dados adequada; operações fundamentais com grafos e implementação do algoritmo de busca para identificação de componentes conexos. Na segunda etapa, foi desenvolvida a Visualização de Grafos com base na matriz gerada.

2.1. Representação de Grafos

Esta etapa é iniciada com a leitura e processamento da entrada, que consiste em um arquivo de texto (TXT) que possui descrição do grafo. A partir do arquivo de entrada, um algoritmo converte esses dados em uma estrutura de matriz de adjacência, garantindo uma representação eficiente do grafo.

Com a estrutura de matriz de adjacência estabelecida, foram determinadas as seguintes tarefas:

- Determinar Adjacência entre Dois Vértices (vX e vY);
- Cálculo do Grau de um Vértice Qualquer;
- Buscar Todos os Vizinhos de um Vértice Qualquer;
- Visitar Todas as Arestas do Grafo

Por fim, foi abordado especificamente o problema de conexidade em grafos dirigidos e não dirigidos com a aplicação do algoritmo DFS. A DFS é uma técnica recursiva que nos permite explorar os vértices do grafo em profundidade, encontrando subgrafos onde cada par de vértices está conectado por um caminho [Riansanti et al. 2018].

2.2. Visualização de Grafos

Realizar a apresentação gráfica do Grafo lido de um novo arquivo de texto gerado a partir do grafo armazenado no formato de que uma API ou biblioteca vai receber.

Na fase de Visualização de Grafos, o foco principal é transformar a estrutura de dados do grafo em uma representação visual compreensível e informativa. Uma abordagem comum é utilizar bibliotecas ou APIs especializadas em visualização de grafos,

como a Graphviz, D3.js, NetworkX (para Python), ou outras ferramentas similares. Essas bibliotecas oferecem uma variedade de funcionalidades para representar grafos de maneira estética e interativa.

Neste projeto foram delimitadas as seguintes tarefas:

- Criar arquivo de texto a ser importado pela biblioteca de visualização;
- Leitura do novo arquivo gerado a partir do grafo armazenado;
- Visualizar grafo.

2.3. Tecnologias Utilizadas

Para a implementação deste projeto, utilizamos a linguagem de programação *Python* (3.11.5), dada sua facilidade de uso e vasta gama de bibliotecas disponíveis. Em particular, empregamos as bibliotecas *NetworkX* (3.1) e *Matplotlib* (3.7.2). A *NetworkX*[1] é uma biblioteca especializada em estruturas de grafos e algoritmos relacionados, oferecendo uma variedade de ferramentas para manipulação e análise de grafos. Já o *Matplotlib*[2] é uma biblioteca amplamente utilizada para visualização de dados em *Python*, proporcionando recursos para criar gráficos e representações visuais do grafo.

3. Implementação Representação de Grafos

Nesta seção, é descrita a implementação da representação de grafos, perpassando desde o processo de entrada de arquivo; armazenamento na estrutura de dados; as funcionalidades desenvolvidas para manipulação e análise do grafo; e criação de arquivo de texto para visualização.

3.1. Leitura - Entrada Arquivo

Para realizar a leitura do arquivo de texto contendo a descrição do grafo, foi desenvolvida uma função em *Python* chamada `ler_grafo_txt`, conforme segue:

```
def ler_grafo_txt(nome_arquivo):
    grafo = {"vertices": set(), "arestas": []}

    with open(nome_arquivo, 'r') as arquivo:
        tipo_grafo = arquivo.readline().strip()

        for linha in arquivo:
            u, v = linha.strip().split(',')
            grafo["vertices"].add(u.strip())
            grafo["vertices"].add(v.strip())
            grafo["arestas"].append((u.strip(), v.strip()))

    grafo["vertices"] = sorted(list(grafo["vertices"]))

    print("Grafo:", grafo)
    print("Tipo de Grafo:", tipo_grafo)
```

[1] *NetworkX*: `https://NetworkX.org/documentation/stable/index.html`
[2] *Matplotlib*: `https://Matplotlib.org/`

```
    return grafo, tipo_grafo

nome_arquivo = "grafoND.txt"
grafo, tipo_grafo = ler_grafo_txt(nome_arquivo)
```

Essa função recebe como entrada o nome do arquivo a ser lido e retorna um dicionário representando o grafo e uma *string* indicando o tipo do grafo, Dirigido (D) ou Não Dirigido (ND). A função `ler_grafo_txt` inicia criando um dicionário vazio chamado `grafo`. Neste dicionário, a chave "vertices" é associada a um conjunto vazio e a chave "arestas" é associada a uma lista vazia. Essas estruturas servirão para armazenar os vértices e arestas do grafo, respectivamente.

Em seguida, é aberto o arquivo de texto especificado pelo parâmetro `nome_arquivo` utilizando a declaração `with open(nome_arquivo, 'r') as arquivo:`. O modo de leitura `'r'` é utilizado, garantindo que o arquivo seja aberto apenas para leitura. Isso garante uma manipulação segura do arquivo, com o *Python* se encarregando de fechá-lo automaticamente após o uso.

Dentro do bloco `with`, a função começa lendo a primeira linha do arquivo, que contém o tipo do grafo (se é D ou ND). O método `strip()` é utilizado para remover quaisquer espaços em branco extras no início e no final da linha. O tipo de grafo é então armazenado na variável `tipo_grafo`.

Após isso, a função itera sobre as linhas restantes do arquivo utilizando um *loop* `for linha in arquivo:`. Para cada linha, a função separa os vértices da aresta utilizando o método `split(',')`, que divide a linha em uma lista de *strings* utilizando a vírgula como separador. Os vértices são então adicionados ao conjunto de vértices do grafo, garantindo que não haja repetições, e a aresta é adicionada à lista de arestas do grafo. Na sequência, os vértices do grafo são ordenados em ordem alfabética utilizando a função `sorted()` e convertidos de volta para uma lista. Isso garante que os vértices sejam armazenados em uma ordem consistente na estrutura de dados.

Por fim, a função imprime o grafo e o tipo de grafo lido do arquivo para fins de verificação durante a execução do código, e retorna o dicionário `grafo` representando o grafo e a *string* `tipo_grafo` representando o tipo do grafo D ou ND. Essa função oferece uma maneira eficiente de ler e armazenar a descrição de um grafo a partir de um arquivo de texto, preparando os dados para serem utilizados em análises posteriores.

3.2. Armazenamento na Estrutura de Dados

Para representação do grafo em uma matriz de adjacência foi defina a função `matriz_adjacencia`, conforme demonstrado a seguir:

```
def matriz_adjacencia(grafo, tipo_grafo):
    num_vertices = len(grafo["vertices"])
    matriz = [[0] * num_vertices for _ in range(num_vertices)]

    for aresta in grafo["arestas"]:
        u_idx = grafo["vertices"].index(aresta[0])
```

```
            v_idx = grafo["vertices"].index(aresta[1])

            if tipo_grafo == "D":
                matriz[u_idx][v_idx] = 1
            elif tipo_grafo == "ND":
                matriz[u_idx][v_idx] = 1
                matriz[v_idx][u_idx] = 1

    return matriz
```

A função `matriz_adjacencia`, recebe como entrada um dicionário representando o grafo e uma *string* indicando o tipo do grafo D ou ND. Ela retorna uma matriz de adjacência representando o grafo.

Inicialmente `num_vertices = len(grafo["vertices"])` calcula o número de vértices no grafo, obtendo o comprimento da lista de vértices do dicionário do grafo. A matriz de adjacência inicializada com zeros `matriz = [[0] * num_vertices for _ in range(num_vertices)]` em que o tamanho da matriz é determinado pelo número de vértices no grafo. Cada entrada na matriz representa uma aresta entre dois vértices.

Na sequência, um *loop* `for` itera sobre todas as arestas no grafo. Neste laço são encontrados os índices do primeiro `u_idx` e segundo `v_idx` vértices das arestas na lista de vértices do grafo.

Então é realizado um bloco condicional que verifica se o grafo é D. Caso positivo, `matriz[u_idx][v_idx] = 1` define a entrada correspondente na matriz de adjacência como 1 para indicar a existência de uma aresta do vértice `u_idx` para o vértice `v_idx`. No entanto, caso seja ND, define a entrada correspondente na matriz de adjacência como 1 para indicar a existência de uma aresta do vértice `u_idx` para o vértice `v_idx`. Além disso, para grafos ND, define também a entrada oposta na matriz de adjacência como 1, para garantir que a matriz seja simétrica. Por fim, retorna a matriz de adjacência resultante.

3.3. Verificação de Adjacentes

A função `verifica_adjacentes` foi definida para verificar se dois vértices vX e vY qualquer são adjacentes. A função recebe como entrada a matriz de adjacência gerada e uma *string* indicando o tipo do grafo (D ou ND).

Inicialmente, a função solicita ao usuário para digitar os dois vértices que deseja verificar se são adjacentes. Para isso uma função adicional foi definida para os inputs dos vértices, conforme demonstrado a seguir:

```
def input_vertice(mensagem):
    vertice = input(mensagem)
    return vertice
```

A função `input_vertice` é chamada dentro da função `verifica_adjacentes` da seguinte forma:

```
def verifica_adjacentes(matriz_adj, tipo_grafo):
```

```
men_u = 'Digite o primeiro vértice: '
men_v = 'Digite o segundo vértice: '
```

Em seguida, a função `verifica_adjacentes` verifica se os vértices fornecidos pelo usuário estão presentes no conjunto de vértices do grafo. Se não estiverem, o usuário é solicitado a fornecer vértices válidos.

```
vertices = grafo["vertices"]

while True:
    u = input_vertice(men_u)
    if u in vertices:
        break
    else:
        print("Vértice inválido. \nPor favor, digite
        um dos seguites vértices: ", vertices)

while True:
    v = input_vertice(men_v)
    if v in vertices:
        break
    else:
        print("Vértice inválido. \nPor favor, digite
        um dos seguites vértices: ", vertices)
```

Para implementação do código anterior contou com auxílio do *ChatGPT 3.5*[3]. Após obter os vértices válidos, a função encontra os índices correspondentes desses vértices na lista de vértices do grafo `grafo["vertices"].index(u)` e `grafo["vertices"].index(v)`.

Por fim, a função verifica se os vértices são adjacentes, dependendo do tipo do grafo. Para grafos D `if tipo_grafo == "D"`, verifica-se se há uma entrada 1 na matriz de adjacência na posição correspondente aos vértices fornecidos por meio da expressão: `if (matriz_adj[u_idx][v_idx] == 1)`. Para grafos ND, verifica-se se há uma entrada 1 na matriz de adjacência nas posições correspondentes a ambos os vértices ou vice-versa utilizando a expressão: `if (matriz_adj[u_idx][v_idx] == 1 or matriz_adj[v_idx][u_idx] == 1)`. No final, imprime uma mensagem indicando se os vértices são adjacentes (expressão verdadeira) ou não são adjacentes (expressão falsa).

3.4. Calcular o grau de um vértice qualquer

A função `calcular_grau` calcula o grau de um vértice específico no grafo. Assim como a função anterior, ela recebe como entrada uma matriz de adjacência gerada e a *string* que indica o tipo do grafo. A função também solicita ao usuário para digitar um vértice do grafo por meio da função `input_vertice` e verifica se o vértice fornecido pelo usuário está presente no conjunto de vértices do grafo.

[3]*ChatGPT*: `https://chat.openai.com/`

Após obter um vértice válido, a função encontra o índice correspondente desse vértice na lista de vértices do grafo `u_idx = grafo["vertices"].index(u)`. Em seguida, a função `sum()` do *Python* calcula o grau do vértice somando todos os elementos na linha da matriz de adjacência correspondente ao vértice `grau = sum(matriz_adj[u_idx])`. Se o tipo do grafo for não dirigido, a função adiciona o valor da diagonal principal da matriz de adjacência correspondente ao vértice ao grau calculado, considerando o *loop* do vértice `grau += matriz_adj[u_idx][u_idx]`, em que `matriz_adj[u_idx][u_idx]` acessa o elemento na diagonal principal da matriz de adjacência, que representa as arestas que conectam o vértice a si mesmo. Por fim, `return grau` retorna o grau do vértice calculado.

3.5. Buscar todos os vizinhos de um vértice qualquer

Para verificar todos os vizinhos de um vértice qualquer foi definida a função `verifica_vizinhos`. A função também solicita a entrada de vértice e após obter um vértice válido fornecido pelo usuário encontra o índice correspondente desse vértice na lista de vértices do grafo e inicializa uma lista vazia para armazenar os vizinhos `vizinhos = []`:

Em seguida, a função percorre a linha correspondente ao vértice na matriz de adjacência por meio do laço `for i in range(len(matriz_adj[u_idx]))`. Para cada elemento da linha igual a 1 `(matriz_adj[u_idx][i] == 1)`, adiciona o vértice correspondente à lista de vizinhos utilizando a função `append` do *Python*, se ainda não estiver presente `(grafo["vertices"][i] not in vizinhos)`.

Nos grafos não direcionados, também verifica a coluna correspondente ao vértice na matriz de adjacência. Para cada elemento da coluna igual a 1, adiciona o vértice correspondente à lista de vizinhos, se ainda não estiver presente. No final, `return vizinhos` retorna a lista de vizinhos encontrados.

3.6. Visitar todas as arestas do grafo

Para percorrer todas as arestas do grafo foi definida a função `visitar_arestas`. Esta função recebe como entrada um dicionário representando o grafo e uma *string* indicando o tipo do grafo (D ou ND).

A função inicializa uma lista vazia `(arestas_visitadas = [])` para armazenar as arestas visitadas durante o percurso e um conjunto `(visitadas = set())` para rastrear as arestas que já foram visitadas. Além disso, ela obtém o número de vértices no grafo `(num_vertices = len(grafo["vertices"]))`.

```
    for i in range(num_vertices):
        for j in range(num_vertices):
            if (i, j) not in visitadas and matriz[i][j] == 1:
                if tipo_grafo == "D" or (tipo_grafo == "ND"
                and j >= i):
                    arestas_visitadas.append((grafo["vertices"]
                    [i], grafo["vertices"][j]))
                    visitadas.add((i, j))
                    if tipo_grafo == "ND":
                        visitadas.add((j, i))
```

Na sequência, foram definidos dois *loops* em que o primeiro *loop* `for i in range(num_vertices)` percorre todos os vértices do grafo, onde `i` representa o índice do vértice atual. Dentro do primeiro *loop*, o segundo *loop* `for j in range(num_vertices)` percorre novamente todos os vértices do grafo, onde `j` representa o índice do vértice com o qual `i` será comparado para formar uma aresta.

No segundo *loop* é realizada uma verificação `(if (i, j) not in visitadas and matriz[i][j] == 1)` se a aresta entre os vértices `i` e `j` não foi visitada `((i, j) not in visitadas)` e se há uma conexão entre esses vértices na matriz de adjacência `(matriz[i][j] == 1)`. Também é verificado se o grafo é direcionado `(tipo_grafo == "D")` ou se é não direcionado `(tipo_grafo == "ND")` e se o vértice `j` é maior ou igual ao vértice `i`, garantindo que as arestas não sejam contadas duas vezes em grafos não direcionados, caso essa condição seja verdadeira significa que foi encontrado uma aresta válida e ela é adicionada à lista `arestas_visitadas` com os vértices correspondentes `(arestas_visitadas.append((grafo["vertices"][i], grafo["vertices"][j])))`. Após visitar a aresta, ela é adicionada ao conjunto `visitadas` para não ser visitada novamente `(visitadas.add((i, j)))`.

Em grafos não direcionados `(if tipo_grafo == "ND")`, é adicionada a aresta reversa ao conjunto `visitadas` para garantir que ambas as direções da aresta sejam consideradas como visitadas `(visitadas.add((j, i)))`.

Por fim, `return arestas_visitadas` retorna a lista de arestas visitadas e, em seguida, essa lista pode ser impressa para mostrar quais arestas foram visitadas durante o percurso.

3.7. Aplicação em Busca

A DFS é um algoritmo utilizado para percorrer ou buscar em uma estrutura de dados, geralmente um grafo ou uma árvore, de forma que se explore o máximo possível em uma ramificação antes de retroceder. Esse algoritmo é amplamente utilizado em diversas aplicações, incluindo busca de caminhos, ordenação topológica, identificação de componentes conexas e detecção de ciclos em grafos.

O funcionamento básico da DFS é seguir um caminho o mais longe possível antes de retroceder e explorar outros caminhos. Ela utiliza uma pilha (ou recursão) para manter o controle dos vértices que ainda precisam ser explorados. Esse algoritmo é conhecido por ser bastante eficiente e fácil de implementar.

Um pseudocódigo genérico para a DFS pode ser definido da seguinte forma:

```
DFS(Grafo G, Vértice v):
    Marque o vértice v como visitado
    Para cada vértice vizinho u de v:
        Se u não foi visitado:
            DFS(G, u)
```

Este pseudocódigo mostra uma implementação recursiva da DFS, onde cada vértice é visitado e, em seguida, recursivamente visitamos todos os seus vizinhos não visitados. Neste estudo a implementação de DFS será aplicada no contexto de identificação de componentes conexas em grafos D ou ND, conforme a seção 3.7.1.

3.7.1. Implementação da Busca em Profundidade (DFS)

Para este projeto a função `dfs` foi definida para implementar a Busca em Profundidade em um grafo representado por uma matriz de adjacência. Na qual recebe como entrada a matriz de adjacência que representa o grafo, uma lista de vértices visitados, o vértice atual, uma lista para armazenar o componente conexo encontrado, o tipo do grafo (D ou ND) e a lista de vértices originais do grafo. A lista `vertices` é fundamental para as saídas representarem os vértices originais fornecidos no arquivo TXT, independente se são numéricos, caracteres ou strings. Conforme segue:

```
def dfs(matriz_adj, visitados, vertice, componente,
tipo_grafo, vertices):
    visitados[vertices.index(vertice)] = True

    for vizinho in range(len(matriz_adj)):
        if matriz_adj[vertices.index(vertice)][vizinho] == 1
        and not visitados[vizinho]:
            dfs(matriz_adj, visitados, vertices[vizinho],
            componente, tipo_grafo, vertices)
    componente.append(vertice)
```

Em consonância ao pseudocódigo para DFS inicialmente o vértice atual é marcado como visitado `(True)`, usando o índice do vértice na lista de vértices como uma referência `(visitados[vertices.index(vertice)])`. Na sequência deve-se percorrer sobre todos os vizinhos do vértice atual, isso é feito com o `loop for vizinho in range(len(matriz_adj))`: que percorre todas as colunas da matriz de adjacência. Então é verificado se existe uma aresta entre o vértice atual e o vizinho, em que a condicional `if matriz_adj[vertices.index(vertice)][vizinho] == 1 and not visitados[vizinho]` utiliza os índices na lista de vértices para acessar a matriz de adjacência, e se o vizinho não foi visitado anteriormente. Caso positivo, Chama recursivamente a função `dfs` para visitar o vizinho não visitado, atualizando o vértice atual para o vizinho.

Por fim, `componente.append(vertice)` adiciona o vértice atual ao componente conexo após a exploração de todos os seus vizinhos. A partir da função `dfs` é possível implementar as demais funções para identificar componentes conexas nos grafos D ou ND.

3.7.2. Encontrar Componentes Conexos em Grafos Não Direcionados

Em grafos não dirigidos, a conectividade entre os vértices é bidirecional, em que não há distinção entre vértices de origem e vértices de destino nas arestas. Assim, um componente conexo em um grafo não dirigido é um subconjunto de vértices onde existe um caminho entre cada par de vértices dentro desse subconjunto. Isso significa que todos os vértices de um componente conexo estão de alguma forma interligados, formando uma única unidade coesa.

Para encontrar os componentes conexos em um grafo não dirigido, pode-se utilizar

uma variedade de algoritmos, como a busca em profundidade (DFS) ou a busca em largura (BFS). Neste projeto foi implementado a DFS.

A função `encontrar_componentes_conexos_nao_direcionados` foi projetada para encontrar todos os componentes conexos em um grafo não direcionado representado por uma matriz de adjacência. Ela recebe como entrada a matriz de adjacência e a lista de vértices do grafo. Como demonstrado a seguir:

```
def encontrar_componentes_conexos_nao_direcionados(matriz_adj,
vertices):
    visitados = [False] * len(matriz_adj)
    componentes_conexos = []

    for vertice in vertices:
        if not visitados[vertices.index(vertice)]:
            componente = []
            dfs(matriz_adj, visitados, vertice, componente,
            tipo_grafo, vertices)
            componentes_conexos.append(componente)

    return componentes_conexos
```

A função Inicializa uma lista de booleanos `visitados` onde cada elemento corresponde a um vértice do grafo e é inicializado com o valor `False` indicando que nenhum vértice foi visitado até então. Também é definida uma lista vazia `componentes_conexos` que irá armazenar todos os componentes conexos encontrados no grafo.

Um *loop* é utilizado para iterar sobre todos os vértices do grafo, em que a estrutura de condição `if not visitados[vertices.index(vertice)]` é utilizada para verificar se o vértice atual não foi visitado ainda. Caso não tenha sido visitado é inicializado uma lista vazia `componente` para armazenar os vértices do componente conexo atual; a função de busca em profundidade (`dfs`) é chamada para explorar o grafo a partir do vértice atual, adicionando todos os vértices alcançáveis ao componente atual; e o componente conexo encontrado à lista de componentes conexos por meio da função `append`. Por fim, `return componentes_conexos` retorna a lista de todos os componentes conexos encontrados no grafo.

3.7.3. Encontrar Componentes Conexos em Grafos Dirigidos

Os grafos dirigidos são estruturas em que as arestas possuem uma direção específica, desta forma há uma distinção clara entre o vértice de origem e o vértice de destino de cada aresta. Nestes grafos, a conectividade entre os vértices pode ser analisada não apenas em termos de existência de caminhos, mas também em termos de componentes fortemente conexos.

Um componente fortemente conexo em um grafo dirigido é um subconjunto de vértices tal que, para cada par de vértices u e v pertencentes a esse componente, existe um caminho direcionado de u para v e outro caminho direcionado de v para u.

Para encontrar os componentes fortemente conexos em um grafo dirigido, foi utilizado o algoritmo de *Kosaraju*, que depende da identificação dos componentes conexos em um grafo transposto. O grafo transposto é obtido invertendo a direção de todas as arestas do grafo original.

A função `encontrar_componentes_fortemente_conexos` foi projetada para encontrar todos os componentes fortemente conexos em um grafo representado por uma matriz de adjacência. Entretanto, foi inicialmente definida uma função auxiliar intitulada de `encontrar_ordem_vertices` para encontrar a ordem correta dos vértices a serem visitados, conforme explicado a seguir:

A função `encontrar_ordem_vertices` visa encontrar a ordem dos vértices em um grafo direcionado representado por uma matriz de adjacência. Ela também depende da função `dfs` para explorar o grafo e determinar a ordem dos vértices.

```
def encontrar_ordem_vertices(matriz_adj, vertices):
    visitados = [False] * len(matriz_adj)
    ordem = []

    for vertice in vertices:
        if not visitados[vertices.index(vertice)]:
            componente = []
            dfs(matriz_adj, visitados, vertice, componente,
            tipo_grafo, vertices)
            ordem.extend(componente)
    return ordem
```

A parte principal da função reside na inicialização de uma lista vazia `ordem` que irá armazenar a ordem dos vértices e no *loop* que percorre todos os vértices do grafo `(for vertice in vertices)`. Se o vértice atual não foi visitado ainda, é inicializada a lista vazia `componente` que guarda os vértices do componente conexo atual, a função (`dfs`) é chamada novamente a partir do vértice atual, e todos os vértices alcançáveis ao componente atual são adicionados. Após isso, `ordem.extend(componente)` estende a lista `ordem` com os vértices do componente conexo encontrado, preservando a ordem de visitação. E no fim, `return ordem` retorna a ordem dos vértices encontrados no grafo.

Após a definição da função `encontrar_ordem_vertices` prosseguiu-se para implementação da função `encontrar_componentes_fortemente_conexos`, conforme segue:

```
def encontrar_componentes_fortemente_conexos(matriz_adj,
vertices):
    visitados = [False] * len(matriz_adj)
    ordem = encontrar_ordem_vertices(matriz_adj, vertices)
    grafo_transposto = [[matriz_adj[j][i] for j in
    range(len(matriz_adj))] for i in range(len(matriz_adj))]
    componentes_conexos = []

    for vertice in reversed(ordem):
```

```
        if not visitados[vertices.index(vertice)]:
            componente = []
            dfs(grafo_transposto, visitados, vertice,
            componente, tipo_grafo, vertices)
            componentes_conexos.append(componente)

    return componentes_conexos
```

A função também inicializa a lista `visitados`. Chama a função `ordem = encontrar_ordem_vertices()` que será explicada em seguida e calcula o grafo transposto, trocando as linhas pelas colunas da matriz de adjacência (`[[matriz_adj[j][i] for j in range(len(matriz_adj))] for i in range(len(matriz_adj))]`), esse passo é fundamental para a correta identificação dos componentes fortemente conexos. Também inicializa a lista vazia `componentes_conexos`.

Na sequência, é utilizado o *loop* `for vertice in reversed(ordem)` para percorrer todos os vértices do grafo, na ordem inversa obtida pela função `encontrar_ordem_vertices`. Se o vértice atual não foi visitado ainda, é inicializada a lista vazia `componente`. No entanto, agora a função (`dfs`) é chamada para explorar o grafo transposto e adiciona todos os vértices alcançáveis ao componente atual. Com isso (`componentes_conexos.append(componente)`) adiciona o componente fortemente conexo encontrado à lista de componentes conexos. E no final, `return componentes_conexos` retorna a lista de todos os componentes fortemente conexos encontrados no grafo.

3.7.4. Imprimir Componentes Conexos

A função `encontrar_e_imprimir_componentes` é responsável por imprimir os componentes conexos em um grafo, seja ele direcionado ou não direcionado. Conforme demonstrado a seguir:

```
def encontrar_e_imprimir_componentes(matriz_adj, tipo_grafo,
vertices):
    if tipo_grafo == "ND":
        componentes_nao_direcionados = encontrar_componentes_
        conexos_nao_direcionados(matriz_adj, vertices)
        print("Componentes Conexos em Grafo Não Direcionado:",
        componentes_nao_direcionados)
    elif tipo_grafo == "D":
        componentes_fortemente_conexos = encontrar_
        componentes_fortemente_conexos(matriz_adj, vertices)
        print("Componentes Fortemente Conexos em Grafo
        Direcionado:", componentes_fortemente_conexos)
```

Basicamente a função verifica se o grafo é não direcionado e imprime os componentes conexos encontrados no grafo não direcionado. Se não for, chama a

função encontrar_componentes_fortemente_conexos para encontrar os componentes fortemente conexos em um grafo direcionado e imprime os componentes fortemente conexos encontrados no grafo direcionado.

3.8. Criação do arquivo de texto para visualização do grafo

A última função definida na parte de Representação de Grafos foi criar_arquivo_grafo responsável por criar um arquivo de texto contendo informações sobre o grafo, como o tipo do grafo (D ou ND) e suas arestas.

```
def criar_arquivo_grafo(matriz_adj, tipo_grafo):
    matriz = matriz_adj
    arestas = visitar_arestas(matriz_adj, tipo_grafo)

    with open("grafo_info.txt", "w") as f:
        if tipo_grafo == 'ND':
            f.write(f"ND\n")
        else:
            f.write(f"D\n")
        for aresta in arestas:
            f.write(f"{aresta[0]} {aresta[1]}\n")
```

Inicialmente a variável matriz recebe a matriz de adjacência (matriz_adj). Enquanto arestas armazena resultantes da utilização da função visitar_arestas descrita na seção 3.6. Em seguida with open("grafo_info.txt", "w") as f abre um novo arquivo grafo_info.txt em modo de escrita.

A partir disso, é realizado a verificação do tipo de grafo, em que se o grafo for ND no arquivo será escrito no arquivo o tipo do grafo (ND para não direcionado) seguido de uma quebra de linha. Caso o grafo seja direcionado, escreve no arquivo o tipo do grafo (D para direcionado) seguido de uma quebra de linha. Por fim, itera sobre todas as arestas do grafo (for aresta in arestas) e escreve no arquivo as arestas do grafo, onde cada linha contém os vértices inicial e final separados por um espaço.

4. Implementação da Visualização de Grafos

Nesta Seção, é descrita a implementação da Visualização de Grafos, perpassando pelo processo de Leitura do Grafo do novo Arquivo de Texto, até Visualização do Grafo.

4.1. Ler Grafo de um Arquivo de Texto

Primeiramente foi definida a função ler_grafo_txt responsável por ler as informações do tipo de grafo de um arquivo de texto gerado. Onde (with open(nome_arquivo, 'r') as arquivo) abre o arquivo especificado pelo nome (nome_arquivo) em modo de leitura ('r'). Em seguida é feita a leitura a primeira linha do arquivo, que contém o tipo de grafo (tipo_grafo = arquivo.readline().strip()) e remove espaços em branco extras utilizando o método strip(). Por fim, return tipo_grafo retorna o tipo de grafo lido do arquivo.

4.2. Visualizar Grafo

A partir da leitura do arquivo foi definida a função `visualizar_grafo` responsável por visualizar o grafo representado pelo arquivo de texto, utilizando a biblioteca *NetworkX* para a criação e visualização do grafo. Conforme demonstrado a seguir:

```
def visualizar_grafo(tipo_grafo):
    if tipo_grafo == 'ND':
        G = nx.read_edgelist("grafo_info.txt",
        create_using=nx.Graph())
    else:
        G = nx.read_edgelist("grafo_info.txt",
        create_using=nx.DiGraph())
    pos = nx.spring_layout(G, k=0.5, iterations=30)
    nx.draw(G, with_labels=True, node_color="skyblue",
    node_size=1500, font_size=12, font_weight="bold",
    arrows=True)
    plt.show()
```

Inicialmente é verifica se o tipo de grafo é ND. Se o grafo for não direcionado, utiliza-se a função `nx.read_edgelist` para ler o arquivo de texto e criar um grafo não direcionado `(nx.Graph())`. Caso o grafo seja direcionado. é utilizado a função `nx.read_edgelist` para ler o arquivo de texto e criar um grafo direcionado `(nx.DiGraph())`.

Na sequência, são definidos os parâmetros necessários para utilizar a função de desenho da biblioteca *NetworkX*.

- `G`: O grafo a ser desenhado. - `with_labels=True`: Indica que os rótulos dos nós serão exibidos no desenho. - `node_color="skyblue"`: Define a cor dos nós do grafo. Neste caso, os nós serão coloridos com a tonalidade de azul celeste. - `node_size=1500`: Define o tamanho dos nós do grafo. Neste caso, os nós terão o tamanho de 1500 unidades. - `font_size=12`: Define o tamanho da fonte dos rótulos dos nós. - `font_weight="bold"`: Define o peso da fonte dos rótulos dos nós. Neste caso, os rótulos serão exibidos em negrito. - `arrows=True`: Indica que as arestas do grafo terão setas para representar a direção das conexões, caso o grafo seja direcionado.

- G: o grafo a ser desenhado.
- with_labels: em que `True` indica que os rótulos dos nós serão exibidos no desenho.
- node_color: define a cor dos nós do grafo. Neste caso, os nós serão coloridos com a tonalidade de azul-celeste definido por *"skyblue"*.
- node_size: define o tamanho dos nós do grafo. Neste caso, os nós terão o tamanho de 1500 unidades.
- font_size: define o tamanho da fonte dos rótulos dos nós.
- font_weight: define o peso da fonte dos rótulos dos nós. Neste caso, os rótulos serão exibidos em negrito definido `"bold"`.
- arrows: Em que `True` significa que as arestas do grafo terão setas para representar a direção das conexões, caso o grafo seja direcionado.

Por fim, `plt.show()` exibe o gráfico na tela.

5. Demonstração

Nesta seção, serão apresentados exemplos de saídas dos grafos, abrangendo desde a representação dos grafos em matriz de adjacência, passando pelas operações definidas na Seção 2.1, até a identificação de componentes conexas. Além disso, será fornecida a visualização gráfica de cada grafo utilizado nos exemplos.

5.1. Testes com grafo Dirigido

Por convenção, para todos os testes com grafo Dirigido foi utilizado um arquivo de texto com as seguintes configurações:

```
D
A,E
B,A
B,F
B,E
C,A
C,C
D,C
E,D
F,G
G,B
I,H
J,J
```

A Figura 1 apresenta o resultado da representação do grafo na Estrutura de Dados de Matriz de Adjacência. Essa representação foi obtida a partir da função `matriz_adjacencia()` definida na seção 3.2.

```
[[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
 [1, 0, 0, 0, 1, 1, 0, 0, 0, 0],
 [1, 0, 1, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
 [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]]
```

Figure 1. Grafo Dirigido representado em Matriz de Adjacência
Fonte: Elaborado pelo Autor.

A representação gráfica obtida pela função `visualizar_grafo` descrita na seção 2.2 pode ser observada na Figura 2.

A partir da representação do Grafo em Estrutura de Matriz de adjacência foram realizadas as seguintes operações em grafos com os respectivos resultados.

Com a estrutura de matriz de adjacência estabelecida, foram determinadas as seguintes tarefas:

Determinar Adjacência entre Dois Vértices (vX e vY

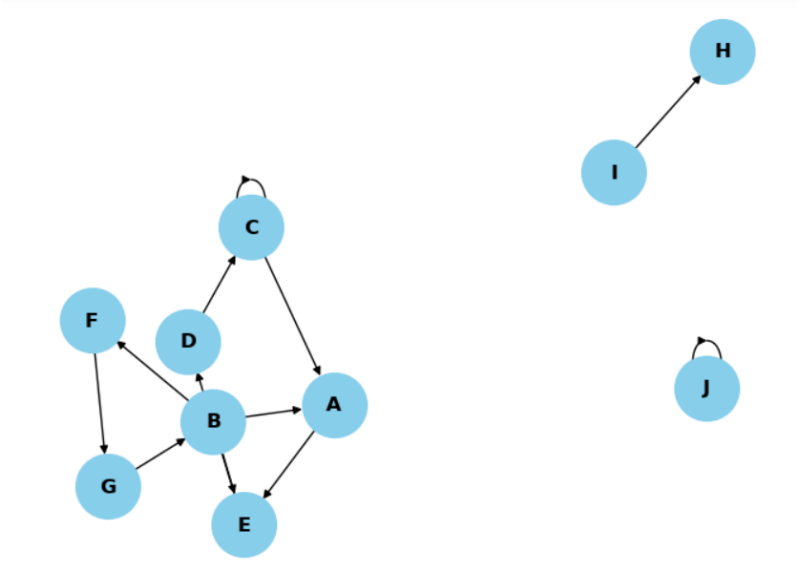

Figure 2. Representação visual do Grafo Dirigido
Fonte: Elaborado pelo Autor.

A Figura 3 mostra o resultado da utilização da função `verifica_adjacentes(matriz_adj, tipo_grafo)` descrita na seção 3.3. Em que se espera como resultado que os dois vértices inseridos (A e E) sejam adjacentes. Desta forma, observa-se que o resultado foi satisfatório.

```
verifica_adjacentes(matriz_adj, tipo_grafo)

Digite o primeiro vértice: A
Digite o segundo vértice: E
O vértice A é adjacente de E
```

Figure 3. Exemplo de vértices adjacentes em Grafo Dirigido
Fonte: Elaborado pelo Autor.

Enquanto a Figura 4 mostra o resultado da utilização da função `verifica_adjacentes()` em que se espera como resultado que os dois vértices inseridos (B e A) não sejam adjacentes. Desta forma, observa-se que o resultado foi satisfatório.

Cálculo do Grau de um Vértice Qualquer

A Figura 5 mostra o resultado da utilização da função `calcular_grau()` descrita na seção 3.4. Em que se espera como resultado que o grau do vértice inserido (B) seja igual 3. Desta forma, observa-se que o resultado foi satisfatório.

Buscar Todos os Vizinhos de um Vértice Qualquer

A Figura 6 mostra o resultado da utilização da função `verifica_vizinhos()` descrita na seção 3.5. Em que se espera como resultado que os vizinhos do vértice inserido (B) sejam os vértices A, E e F. Desta forma, observa-se que o resultado foi satisfatório.

```
verifica_adjacentes(matriz_adj, tipo_grafo)
```

```
Digite o primeiro vértice: E
Digite o segundo vértice: A
O vértice E não é adjacente de  A
```

Figure 4. Exemplo de vértices não adjacentes em Grafo Dirigido
Fonte: Elaborado pelo Autor.

```
print("Grau de vértice:", calcular_grau(matriz_adj, tipo_grafo))
```

```
Digite um vértice do grafo: B
Grau de vértice: 3
```

Figure 5. Exemplo de grau de vértice em Grafo Dirigido
Fonte: Elaborado pelo Autor.

Visitar Todas as Arestas do Grafo Dirigido

A Figura 7 mostra o resultado da utilização da função `visitar_arestas()` descrita na seção 3.5. Em que se espera como resultado que as arestas visitadas sejam: `A-E, B-A, B-F, B-E, C-A, C-C, D-C, E-D, F-G, G-B, I-H, J-J`. Desta forma, observa-se que o resultado foi satisfatório.

Identificar componentes conexas em grafo dirigido

A Figura 8 mostra o resultado da identificação de componentes conexos em grafo dirigido. Este processo está descrito na seção 3.7. Para este exemplo se espera como resultado a identificação de 5 componentes: componente 1 `(H)`, componente 2 `(I)`, componente 3 `(J)`, componente 4 `(F, G, H)` e componente 5 `(E, D, C, A)`. Desta forma, observa-se que o resultado foi satisfatório.

5.2. Testes com grafo Não Dirigido

Por convenção, para todos os testes com grafo Não Dirigido foi utilizado um arquivo de texto com as seguintes configurações:

```
D
A,E
B,A
B,F
```

```
print("Vizinhos:", verifica_vizinhos(matriz_adj, tipo_grafo))
```

```
Digite um vértice do grafo: B
Vizinhos: ['A', 'E', 'F']
```

Figure 6. Exemplo de vizinhos de um vértice em Grafo Dirigido
Fonte: Elaborado pelo Autor.

```
arestas_visitadas = visitar_arestas(grafo, tipo_grafo)
print("Arestas do grafo visitadas: ", arestas_visitadas)
```

```
Arestas do grafo visitadas:  [('A', 'E'), ('B', 'A'), ('B', 'E'), ('B', 'F'), ('C', 'A'), ('C', 'C'), ('D', 'C'), ('E', 'D'),
('F', 'G'), ('G', 'B'), ('I', 'H'), ('J', 'J')]
```

Figure 7. Exemplo da consulta de todas as arestas em Grafo Dirigido
Fonte: Elaborado pelo Autor.

```
vertices = grafo["vertices"]
encontrar_e_imprimir_componentes(matriz_adj, tipo_grafo, vertices)
```

```
Componentes Fortemente Conexos em Grafo Direcionado: [['J'], ['I'], ['H'], ['F', 'G', 'B'], ['E', 'D', 'C', 'A']]
```

Figure 8. Exemplo da identificação de componentes conexas em Grafo Dirigido
Fonte: Elaborado pelo Autor.

```
B,E
C,A
C,C
D,C
E,D
F,G
G,B
I,H
J,J
```

A Figura 9 apresenta o resultado da representação do grafo não dirigido na Estrutura de Dados de Matriz de Adjacência. Essa representação foi obtida a partir da função `matriz_adjacencia()` definida na seção 3.2.

```
[[0, 1, 1, 1, 0, 0, 0, 0, 1],
 [1, 0, 1, 0, 0, 0, 0, 0, 0],
 [1, 1, 0, 0, 0, 0, 0, 0, 0],
 [1, 0, 0, 0, 0, 0, 0, 1, 0],
 [0, 0, 0, 0, 0, 1, 0, 0, 0],
 [0, 0, 0, 0, 1, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 1, 0, 0],
 [0, 0, 0, 1, 0, 0, 0, 1, 1],
 [1, 0, 0, 0, 0, 0, 0, 1, 0]]
```

Figure 9. Grafo Não Dirigido representado em Matriz de Adjacência
Fonte: Elaborado pelo Autor.

A representação gráfica obtida pela função `visualizar_grafo` descrita na seção 2.2 pode ser observada na Figura 10.

A partir da representação do Grafo não dirigido em Estrutura de Matriz de adjacência foram realizadas as seguintes operações em grafos com os respectivos resultados.

Com a estrutura de matriz de adjacência estabelecida, foram determinadas as seguintes tarefas:

Determinar Adjacência entre Dois Vértices (vX e vY

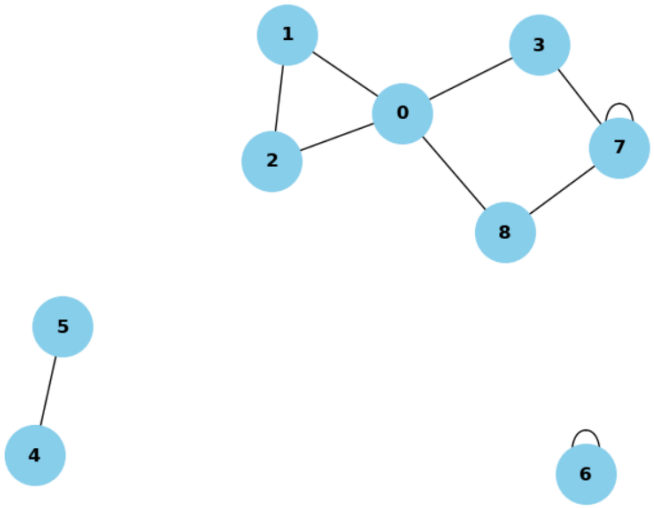

Figure 10. Representação visual do Grafo Não Dirigido
Fonte: Elaborado pelo Autor.

A Figura 11 mostra o resultado da utilização da função `verifica_adjacentes(matriz_adj, tipo_grafo)` descrita na seção 3.3. Em que se espera como resultado que os dois vértices inseridos (0 e 8) sejam adjacentes. Desta forma, observa-se que o resultado foi satisfatório.

```
verifica_adjacentes(matriz_adj, tipo_grafo)

Digite o primeiro vértice: 0
Digite o segundo vértice: 8
Os vértices 0 e 8 são adjacentes
```

Figure 11. Exemplo de vértices adjacentes em Grafo Não Dirigido
Fonte: Elaborado pelo Autor.

Enquanto a Figura 12 mostra o resultado da utilização da função `verifica_adjacentes()` em que se espera como resultado que os dois vértices inseridos (0 e 7) não sejam adjacentes. Desta forma, observa-se que o resultado foi satisfatório.

Cálculo do Grau de um Vértice Qualquer

A Figura 13 mostra o resultado da utilização da função `calcular_grau()` descrita na seção 3.4. Em que se espera como resultado que o grau do vértice inserido (7) seja igual 4. Desta forma, observa-se que o resultado foi satisfatório.

Buscar Todos os Vizinhos de um Vértice Qualquer

A Figura 14 mostra o resultado da utilização da função `verifica_vizinhos()` descrita na seção 3.5. Em que se espera como resultado que os vizinhos do vértice inserido (0) sejam os vértices 1, 2, 3 e 8. Desta forma, observa-se que o resultado foi satisfatório.

```
verifica_adjacentes(matriz_adj, tipo_grafo)
```

```
Digite o primeiro vértice: 0
Digite o segundo vértice: 7
O vértices 0 e 7 não são adjacentes
```

Figure 12. Exemplo de vértices não adjacentes em Grafo Não Dirigido
Fonte: Elaborado pelo Autor.

```
print("Grau de vértice:", calcular_grau(matriz_adj, tipo_grafo))
```

```
Digite um vértice do grafo: 7
Grau de vértice: 4
```

Figure 13. Exemplo de grau de vértice em Grafo Não Dirigido
Fonte: Elaborado pelo Autor.

Visitar Todas as Arestas do Grafo Não Dirigido

A Figura 15 mostra o resultado da utilização da função `visitar_arestas()` descrita na seção 3.5. Em que se espera como resultado que as arestas visitadas sejam: `0-1, 0-2, 0-3, 0-8, 1-2, 3-7, 4-5, 6-6, 7-7, 7-8`. Desta forma, observa-se que o resultado foi satisfatório.

Identificar componentes conexas em Grafo Não Dirigido

A Figura 16 mostra o resultado da identificação de componentes conexos em Grafo Não Dirigido. Este processo está descrito na seção 3.7. Para este exemplo se espera como resultado a identificação de 3 componentes: componente 1 `(2, 1, 8, 7, 3, 0)`, componente 2 `(5, 4)`, componente 3 `(6)`. Desta forma, observa-se que o resultado foi satisfatório.

6. Conclusão

O projeto desenvolvido no âmbito da disciplina de Projeto e Análise de Algoritmos proporcionou explorar e compreender a implementação de algoritmos e estruturas de dados fundamentais para a manipulação e análise de grafos. Ao longo do experimento, foi possível aplicar conceitos teóricos aprendidos em sala de aula em um contexto prático, enfrentando desafios reais de representação, análise e visualização de grafos.

Na primeira etapa do experimento, os esforços foram direcionados na

```
print("Vizinhos:", verifica_vizinhos(matriz_adj, tipo_grafo))
```

```
Digite um vértice do grafo: 0
Vizinhos: ['1', '2', '3', '8']
```

Figure 14. Exemplo de vizinhos de um vértice em Grafo Não Dirigido
Fonte: Elaborado pelo Autor.

```
arestas_visitadas = visitar_arestas(grafo, tipo_grafo)
print("Arestas do grafo visitadas: ", arestas_visitadas)
```

```
Arestas do grafo visitadas:  [('0', '1'), ('0', '2'), ('0', '3'), ('0', '8'), ('1', '2'), ('3', '7'), ('4', '5'), ('6', '6'),
('7', '7'), ('7', '8')]
```

Figure 15. Exemplo da consulta de todas as arestas em Grafo Não Dirigido
Fonte: Elaborado pelo Autor.

```
vertices = grafo["vertices"]
encontrar_e_imprimir_componentes(matriz_adj, tipo_grafo, vertices)
```

```
Componentes Conexos em Grafo Não Direcionado: [['2', '1', '8', '7', '3', '0'], ['5', '4'], ['6']]
```

Figure 16. Exemplo da identificação de componentes conexas em Grafo Não Dirigido
Fonte: Elaborado pelo Autor.

Implementação de Representação de Grafos. Em que foi abordado o problema de identificação de componentes conexos, implementando o algoritmo DFS para determinar a conectividade do grafo.

Na segunda etapa, foi realizada a Visualização de Grafos, onde se buscou transformar a estrutura de dados do grafo em uma representação visual compreensível e informativa. Foram utilizadas as bibliotecas *NetworkX e Matplotlib* em *Python* para criar visualizações gráficas dos grafos.

Os resultados obtidos durante os testes foram satisfatórios, demonstrando a eficácia das implementações desenvolvidas. Por meio dos testes, verificou-se que os algoritmos foram capazes de lidar com uma variedade de grafos, tanto pequenos quanto grandes, e produzir resultados consistentes.

Em suma, este projeto não apenas fortaleceu a compreensão dos conceitos fundamentais relacionados a grafos e algoritmos de busca, mas também nos proporcionou uma base para explorar aplicações mais avançadas no campo da Ciência da Computação e Engenharia.

7. References

References

Adali, T. and Ortega, A. (2018). Applications of graph theory. *Proc. IEEE*, 106:784–786.

Al-Sayegh, A. and Farsangi, E. N. (2020). Basys-mtb: An integrative structural simulation platform based on adjacency matrices. *Adv. Eng. Softw.*, 142:102772.

Dörfler, F., Simpson-Porco, J., and Bullo, F. (2018). Electrical networks and algebraic graph theory: Models, properties, and applications. *Proceedings of the IEEE*, 106:977–1005.

Egilmez, H. E., Pavez, E., and Ortega, A. (2016). Graph learning from data under laplacian and structural constraints. *IEEE Journal of Selected Topics in Signal Processing*, 11:825–841.

Glück, R. (2017). Algebraic investigation of connected components. pages 109–126.

Paramadevan, P. and Sotheeswaran, S. (2021). Properties of adjacency matrix of a graph and it's construction. *Journal of Science*.

Prajwala, N. B. (2019). A scientific research analysis to identify number of components in a graph. *International Journal of Recent Technology and Engineering*.

Riansanti, O., Ihsan, M., and Suhaimi, D. (2018). Connectivity algorithm with depth first search (dfs) on simple graphs. *Journal of Physics: Conference Series*, 948.

Sahoo, G. (2019). Complex adjacency spectra of digraphs. *Linear and Multilinear Algebra*, 69:193 – 207.