



CESAR SCHOOL
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Teoria dos Grafos - 2024.2

Avaliação II
Grupo: ArestasGulosas

Alunos: Ana Beatriz Ximenes Alves e Caio Barreto de Albuquerque

RECIFE
2024

SUMÁRIO

INTRODUÇÃO	3
A) ATIVIDADES E ALOCAÇÃO DE HORAS	4
B) MANUAL DO USUÁRIO COM FUNCIONALIDADES	7
1. Criação de Grafos	7
2. Adicionar Vértices e Arestas	8
3. Importação de Grafos	10
4. Análise de Propriedades do Grafo	12
5. Visualização de Imagens	16
C) DESCRIÇÃO CÓDIGO FONTE	17
<i>a) Criação e Ativação da Virtual Environment (venv)</i>	<i>17</i>
<i>b) Instalar Dependências do Projeto</i>	<i>18</i>
<i>c) Configurar os Terminais</i>	<i>18</i>
<i>d) Acessar a Aplicação</i>	<i>18</i>
1. Backend	18
<i>A) /create_graph</i>	<i>19</i>
<i>B) /add_vertex</i>	<i>19</i>
<i>C) /add_edge</i>	<i>20</i>
<i>D) /get_graph_image</i>	<i>21</i>
<i>E) /load_graph_from_file</i>	<i>22</i>
<i>F) /load_graph_from_string</i>	<i>23</i>
<i>G) /graph_order_size</i>	<i>23</i>
<i>H) /adjacent_vertices</i>	<i>24</i>
<i>I) /vertex_degree</i>	<i>24</i>
<i>J) /are_adjacent</i>	<i>25</i>
<i>K) /shortest_path</i>	<i>25</i>
<i>L) /is_eulerian</i>	<i>26</i>
2. Comunicação via API	26
<i>A) createGraph(direcionado, valorado)</i>	<i>26</i>
<i>B) addVertex(vertice)</i>	<i>27</i>
<i>C) addEdge(origem, destino, peso)</i>	<i>27</i>
<i>D) getGraphImage()</i>	<i>27</i>
<i>E) loadGraphFromFile(file)</i>	<i>28</i>
<i>F) loadGraphFromString(data)</i>	<i>28</i>
<i>G) getGraphOrderSize()</i>	<i>28</i>
<i>H) getAdjacentVertices(vertice)</i>	<i>28</i>
<i>I) getVertexDegree(vertice)</i>	<i>28</i>
<i>J) areVerticesAdjacent(vertice1, vertice2)</i>	<i>29</i>

<i>K) getShortestPath(origem, destino)</i>	29
<i>L) getIsEurelian()</i>	29
3. Frontend	29
<i>A) handleCreateGraph</i>	30
<i>B) handleAddVertex</i>	30
<i>C) handleAddEdge</i>	30
<i>D) fetchGraphImage</i>	31
<i>E) fetchOrderSize</i>	31
<i>F) fetchIsEurelian</i>	31
<i>G) fetchAdjacentVertices</i>	31
<i>H) fetchVertexDegree</i>	32
<i>I) checkAdjacency</i>	32
<i>J) findShortestPath</i>	32
<i>K) handleLoadGraphFromFile</i>	32
<i>L) handleLoadGraphFromString</i>	33
<i>M) Estados e Campos Relacionados</i>	33

INTRODUÇÃO

Este relatório apresenta o desenvolvimento de um sistema destinado à criação, manipulação e análise de grafos, com foco na aplicação de conceitos teóricos e na resolução de desafios práticos, consistindo no desenvolvimento de uma aplicação para manipulação, visualização e análise de grafos, utilizando um backend em Flask e um frontend desenvolvido em React.

O projeto teve como objetivo principal criar uma ferramenta funcional e eficiente, capaz de atender às demandas de usuários com diferentes níveis de conhecimento técnico. O sistema permite a criação de grafos, além de diversas operações como adição de vértices e arestas, verificação de propriedades e carregamento de grafos via arquivo CSV ou string JSON. Ao longo do documento, detalhamos as etapas do projeto, desde o planejamento inicial, passando pela implementação e refinamento do código, até a elaboração de uma documentação abrangente.

Dessa forma, o objetivo principal do projeto é desenvolver um sistema que permita criar e manipular grafos de diferentes tipos, sejam direcionados ou não direcionados, valorados ou não valorados. Além disso, busca-se possibilitar a visualização clara e informativa da estrutura dos grafos, realizar análises diversas, como o cálculo de grau dos vértices, verificação de adjacência e busca de caminhos mínimos, bem como oferecer interfaces intuitivas que facilitem a interação tanto no backend quanto no frontend.

Por fim, o presente relatório inclui explicações detalhadas sobre as funcionalidades desenvolvidas, as escolhas técnicas realizadas e os desafios enfrentados, proporcionando uma visão clara do processo de desenvolvimento. Este material também serve como um guia prático e técnico, permitindo que usuários e desenvolvedores compreendam e utilizem plenamente as capacidades do sistema, além de oferecer uma base sólida para futuras expansões.

A) ATIVIDADES E ALOCAÇÃO DE HORAS

Para o desenvolvimento de um sistema interativo de manipulação de grafos, as atividades realizadas pela equipe buscaram atender aos requisitos propostos, incluindo a criação e manipulação de grafos, visualização gráfica ou textual, e execução de operações específicas como cálculo de menor caminho, verificação de adjacência e análise de propriedades estruturais como grafos eulerianos. Abaixo, descrevemos as etapas do projeto, justificando cada uma delas e indicando as horas alocadas pelos integrantes do grupo, conforme detalhado na tabela.

	Integrantes do Grupo		TOTAL
Atividade	Ana Beatriz Ximenes Alves	Caio Barreto de Albuquerque	
Compreendendo os Requisitos e Pesquisa de Ferramentas adequadas	2	3	5
Codificação e Desenvolvimento	12	18	30
Ajuste e Refatoração	4	6	10
Documentação	4	2	6
TOTAL	22	29	51

1. Compreendendo os Requisitos e Pesquisa de Ferramentas adequadas

Total de horas: 5

Nesta etapa inicial, analisamos os requisitos do sistema para compreender quais seriam as necessidades do projeto, destacando as funcionalidades obrigatórias como a inserção de vértices e arestas (individualmente e em lote), criação de grafos direcionados e não-direcionados, e operações como cálculo do menor caminho e

análise de propriedades como adjacência e grau dos vértices. Além disso, realizamos pesquisas para identificar ferramentas e bibliotecas que facilitam o desenvolvimento do sistema e o NetworkX foi selecionado tendo em vista a facilidade proporcionada na manipulação de grafos.

2. Codificação e Desenvolvimento

Total de horas: 30

Essa etapa constituiu o núcleo do projeto, concentrando-se no desenvolvimento efetivo do sistema e englobando diversas atividades principais. Primeiramente, foi implementada a funcionalidade de criação de grafos, permitindo a adição de vértices e arestas tanto individualmente quanto em lote, com validações rigorosas para evitar entradas inválidas. Além disso, o sistema ganhou flexibilidade ao possibilitar a criação de diferentes tipos de grafos, como direcionados e não-direcionados, bem como valorados e não-valorados.

No que diz respeito à visualização, utilizamos bibliotecas como Matplotlib para criar representações gráficas do grafo, complementadas por opções textuais, incluindo listas e matrizes de adjacência. Também desenvolvemos operações básicas, como cálculo da ordem e do tamanho do grafo, determinação do grau dos vértices e geração de listas de adjacência, contemplando listas de entrada e saída no caso de grafos direcionados. Atendendo a um requisito surpresa, foi implementada a funcionalidade de verificação de eulerianidade, baseada nos critérios de conectividade e grau dos vértices, permitindo identificar se o grafo é euleriano. Essa fase do projeto exigiu intensa colaboração da equipe para resolver problemas técnicos e garantir o atendimento aos requisitos, resultando no maior acúmulo de horas de trabalho entre todas as etapas.

3. Ajuste e Refatoração

Total de horas: 10

Após a implementação inicial, dedicamos um período significativo à refatoração do código, com o objetivo de aprimorar sua organização, legibilidade e desempenho geral. Durante esse processo, realizamos a otimização de algoritmos essenciais, como os responsáveis pelo cálculo do menor caminho e pela análise de adjacência, garantindo que o sistema mantivesse eficiência mesmo ao lidar com grafos de maior complexidade.

Também foi nessa etapa que corrigimos diversos bugs, incluindo inconsistências observadas no tratamento de grafos direcionados e problemas na geração de visualizações gráficas. Além disso, promovemos a padronização do código, reestruturando trechos para alinhar-se a boas práticas de programação, o que não apenas tornou a leitura e o entendimento do código mais fáceis, mas também facilitou sua manutenção futura.

4. Documentação

Total de horas: 6

Concluimos o projeto com a elaboração de uma documentação completa, abrangendo diferentes públicos e objetivos. Para os usuários finais, criamos um manual detalhado com orientações claras sobre como interagir com o sistema, incluindo instruções para criar grafos, realizar operações e interpretar os resultados apresentados. Paralelamente, desenvolvemos uma documentação técnica voltada a desenvolvedores, que inclui o registro da estrutura de dados empregada, descrições detalhadas dos algoritmos implementados e as justificativas para as principais decisões técnicas tomadas ao longo do desenvolvimento.

Além disso, adicionamos exemplos práticos que ilustram o uso das principais funcionalidades do sistema, como o cálculo do menor caminho e a análise de grafos eulerianos, permitindo que os leitores compreendam de forma aplicada o funcionamento do sistema. Essa documentação foi cuidadosamente projetada para atender tanto às necessidades de usuários quanto às de desenvolvedores interessados em ampliar ou adaptar a solução.

B) MANUAL DO USUÁRIO COM FUNCIONALIDADES

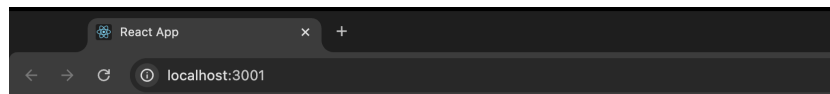
Como dito, a aplicação oferece uma série de funcionalidades robustas e versáteis, projetadas para atender a diferentes necessidades relacionadas à criação, manipulação e análise de grafos. A seguir, detalhamos cada funcionalidade de forma abrangente, com explicações claras sobre como utilizá-las e as possibilidades oferecidas.

1. Criação de Grafos

O ponto de partida para qualquer trabalho na aplicação é a criação de um grafo. Aqui, você pode definir se o grafo será direcionado ou não e se as arestas terão valores associados.

Passo a passo detalhado:

A) No topo da interface, você deverá escolher entre as opções disponíveis, referentes ao Tipo e Valoração do grafo que deseja construir:



Construtor de Grafos Interativo

Configurações do Grafo

☐ Direcionado ☐ Valorados

a) Referente ao Tipo de Grafo, ele poderá ser:

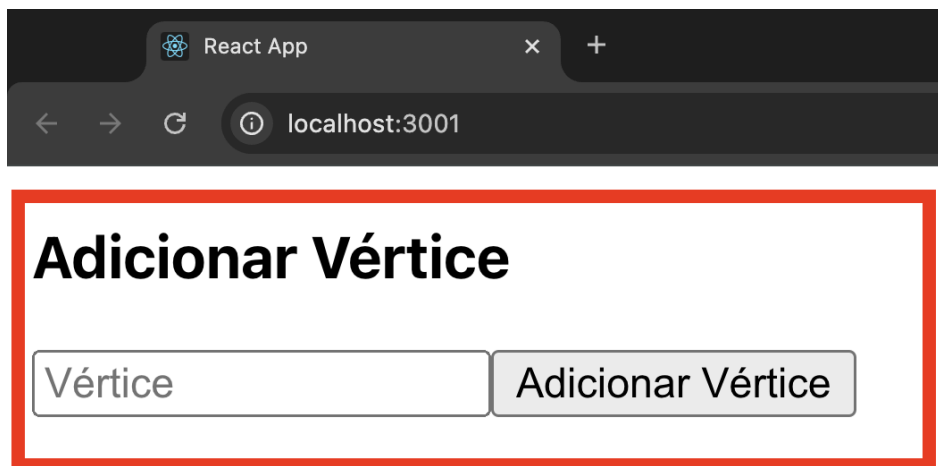
- i) Não direcionado: as conexões entre os vértices não possuem uma direção específica. Por exemplo, uma aresta entre A e B permite ir de A para B e vice-versa.
- ii) Direcionado: as arestas têm direção definida, ou seja, uma ligação de A para B não permite automaticamente a conexão de B para A.

- b) Já na Valoração, pode ser:
- i) Valorado: as arestas possuem pesos ou valores numéricos que representam características como distância, custo ou tempo.
 - ii) Não valorado: as arestas são simples conexões sem qualquer valor atribuído.
- B) Após definir essas características, clique em “Criar Grafo” e o grafo será inicializado com as configurações selecionadas.
- C) A partir desse momento, e somente após realizar esse passo, você terá uma área de trabalho pronta para receber vértices e arestas.

2. Adicionar Vértices e Arestas

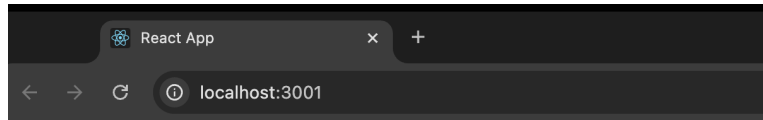
Com o grafo criado, o próximo passo é definir os componentes que o formam, sendo eles vértices e arestas. Os vértices representam os pontos ou entidades do grafo, enquanto as arestas representam as conexões entre eles.

Adicionando Vértices:



The screenshot shows a web browser window with the title 'React App' and the address bar displaying 'localhost:3001'. The main content area features a large heading 'Adicionar Vértice' in bold black text. Below the heading is a form with a text input field containing the placeholder text 'Vértice' and a button labeled 'Adicionar Vértice'. The entire form is enclosed in a red rectangular border.

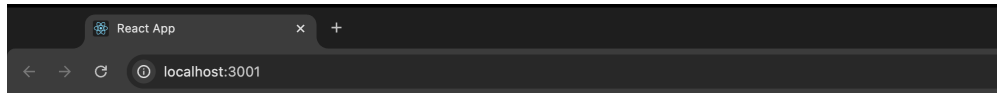
- A) Navegue até a seção “Adicionar Vértices”.
- B) Insira o nome ou identificador único do vértice (ex.: “A”, “B”, “1” ou qualquer rótulo desejado).
- C) Clique no botão “Adicionar Vértice”.
- D) O vértice será exibido na interface gráfica, representado como um nó circular.



Adicionando Arestas:

A screenshot of a web browser window showing a form titled 'Adicionar Aresta'. The form has two input fields labeled 'Origem' and 'Destino', and a button labeled 'Adicionar Aresta'. The form is highlighted with a red border.

- A) Vá até a seção “Adicionar Arestas”.
- B) Escolha os vértices de origem e destino entre os já adicionados ao grafo. Ressalta-se que só é possível criar arestas entre vértices que já foram criados anteriormente.
- C) Caso o grafo seja valorado, insira o peso da aresta. Por exemplo, em um grafo que representa rotas, o peso pode ser a distância entre os pontos.



Construtor de Grafos Interativo

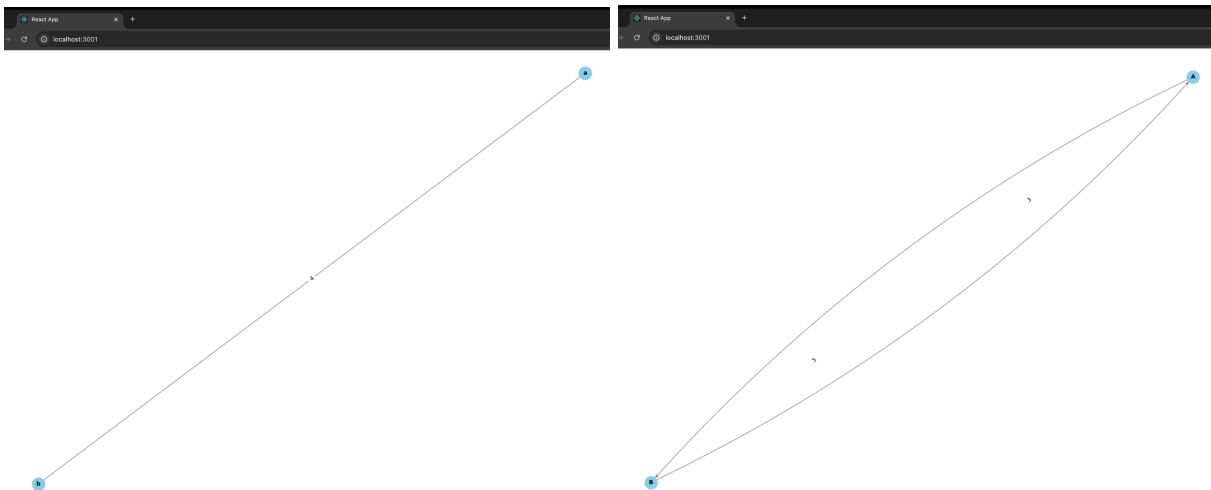
Configurações do Grafo

☐ Direcionado ☒ Valorados

Adicionar Vértice

Adicionar Aresta

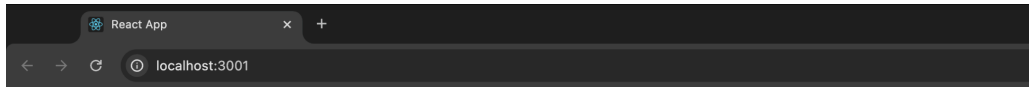
D) Clique em “Adicionar Aresta”. A conexão será representada como uma linha entre os vértices selecionados, com o eventual peso disponível acima.



Vejamos que esse processo é repetitivo e flexível, permitindo a construção de grafos pequenos e simples ou complexos e detalhados.

3. Importação de Grafos

Caso você já tenha dados estruturados, a aplicação permite carregar grafos completos a partir de arquivos ou strings. Atualmente, os formatos suportados são CSV (arquivo) e JSON (string).



Carregar Grafo

Choose File

No file chosen

Carregar Grafo do Arquivo

Insira o JSON para o grafo

Carregar Grafo de String

Importando um grafo de arquivo CSV:

```
GrafoEmLote.csv X
backend > GrafoEmLote.csv > data
1 Source,Target,Weight
2 0,1183,340
3 0,4407,183
4 1,2177,357
5 1,4230,480
6 2,425,224
7 2,4489,94
```

- A) Certifique-se de que o arquivo esteja no formato correto, onde as colunas estão nomeadas, na ordem exata, em colunas “Source”, “Target” e “Weight”, referentes às definições dos vértices de Origem e Destino, e, opcionalmente, o Peso das arestas.
- B) Acesse a opção “Carregar Grafo do Arquivo” e carregue o arquivo diretamente do seu dispositivo.
- C) O grafo será gerado automaticamente, exibindo todos os vértices e arestas com base nas informações fornecidas.

Importando um grafo em formato de string (JSON):

- A) Prepare uma string JSON no formato que tenha os índices “vertices” e “arestas”, cujo primeiro é uma lista de todos os nós que existirão no grafo e o segundo uma lista de listas com as posições “origem”, “destino” e “peso”, nessa ordem.



```
{  
  "vertices": ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"],  
  "arestas": [  
    ["A", "B", 3],  
    ["A", "C", 1],  
    ["B", "D", 4],  
    ["B", "E", 2],  
    ["C", "F", 5],  
    ["D", "G", 6],  
    ["E", "H", 3],  
    ["F", "I", 7],  
    ["G", "J", 4],  
    ["H", "A", 2],  
    ["I", "B", 3],  
    ["J", "C", 5],  
    ["A", "G", 8],  
    ["C", "H", 1],  
    ["E", "F", 2],  
    ["I", "J", 6]  
  ]  
}
```

- B) Cole a string no campo de importação.
C) Clique em “Carregar Grafo de String” e o grafo será exibido.

Por fim, verifica-se que a funcionalidade de importação é ideal para trabalhar com dados grandes ou pré-processados, pois elimina o trabalho manual de definição manual do grafo.

4. Análise de Propriedades do Grafo

Uma das características mais poderosas da aplicação é sua capacidade de realizar análises sobre o grafo, que ajudam a explorar a estrutura e as propriedades de forma eficiente.

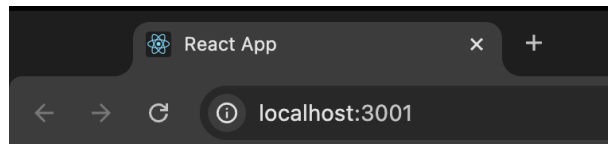


The screenshot shows a web browser window with the title 'React App' and the address bar displaying 'localhost:3001'. The page content includes several sections with buttons:

- Operações no Grafo**: A button labeled 'Exibir Ordem e Tamanho'.
- Verificar Eureliano**: A button labeled 'Verificar Eureliano'.
- Exibir Vértices Adjacentes**: A text input field labeled 'Vértice para adjacentes' followed by a button labeled 'Exibir Vértices Adjacentes'.
- Exibir Grau do Vértice**: A text input field labeled 'Vértice para grau' followed by a button labeled 'Exibir Grau do Vértice'.
- Verificar Adjacência**: Two text input fields labeled 'Origem' and 'Destino', followed by a button labeled 'Verificar Adjacência'.
- Calcular Caminho Mais Curto**: Two text input fields labeled 'Origem' and 'Destino', followed by a button labeled 'Calcular Caminho Mais Curto'.

Propriedades disponíveis:

- Ordem e Tamanho: Retorna o número total de vértices (ordem) e arestas (tamanho) no grafo.



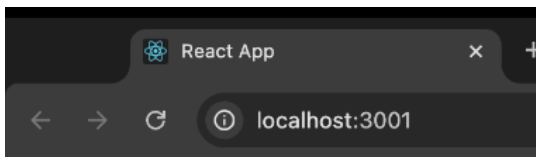
Operações no Grafo

Exibir Ordem e Tamanho

Ordem: 2, Tamanho: 2

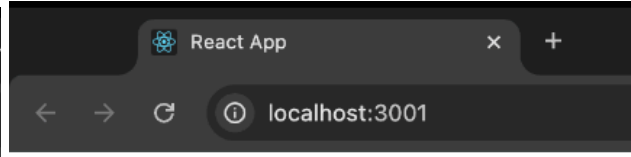
- Circuito Euleriano: Verifica se o grafo contém um caminho que percorre todas as arestas exatamente uma vez. Ressalta-se que, para que um grafo tenha um circuito euleriano, ele precisa ser conexo (em grafos não-direcionados) e todos

os seus vértices devem ter grau par. Em grafos direcionados, é necessário que cada vértice tenha o mesmo grau de entrada e saída.



Verificar Eureliano

Verificar Eureliano



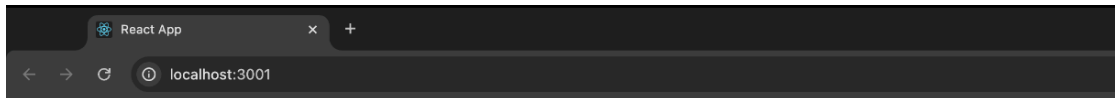
Verificar Eureliano

Verificar Eureliano

O grafo é Euleriano!

O grafo não é Euleriano.

- Lista de Adjacências: Apresenta, para cada vértice do grafo, os outros vértices com os quais ele está conectado diretamente por uma aresta.

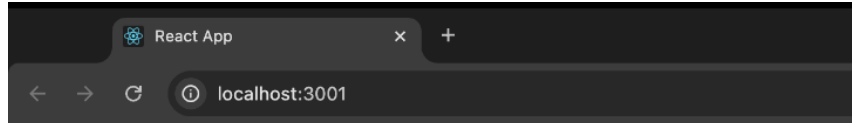


Exibir Vértices Adjacentes

A Exibir Vértices Adjacentes

Adjacentes: {"adjacentes_entrada":["B"],"adjacentes_saida":["B","C"]}

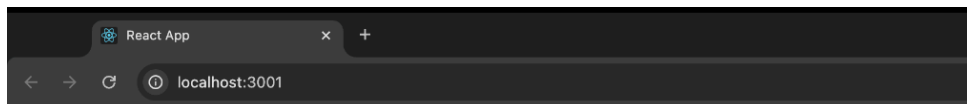
- Grau dos Vértices: Exibe o número de conexões (arestas) de cada vértice. Em grafos direcionados, distingue entre grau de entrada e grau de saída, de forma que o grau de Entrada é o número de arestas que chegam ao vértice e o grau de Saída o que saem do vértice.



Exibir Grau do Vértice

Grau: {"grau_entrada":1,"grau_saida":2}

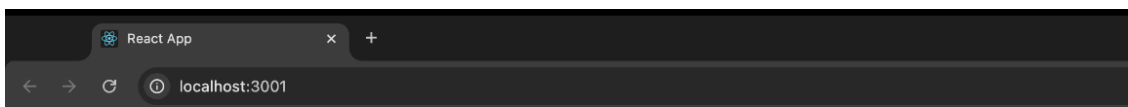
- Verificação de Adjacência: Determina se dois vértices específicos estão diretamente conectados por uma aresta.



Verificar Adjacência

Os vértices são adjacentes? Sim

- Caminhos Mínimos: Calcula a menor distância ou custo entre dois vértices.



Calcular Caminho Mais Curto

Custo do Caminho Mais Curto: 1

Caminho: A -> C

Dessa forma, para realizar uma análise:

- A) Acesse a propriedade desejada, disponibilizada na interface da aplicação.

- B) Forneça os parâmetros necessários (por exemplo, selecione dois vértices para calcular um caminho mínimo).
- C) Clique no botão da ação correspondente e os resultados serão apresentados na tela, com informações detalhadas.

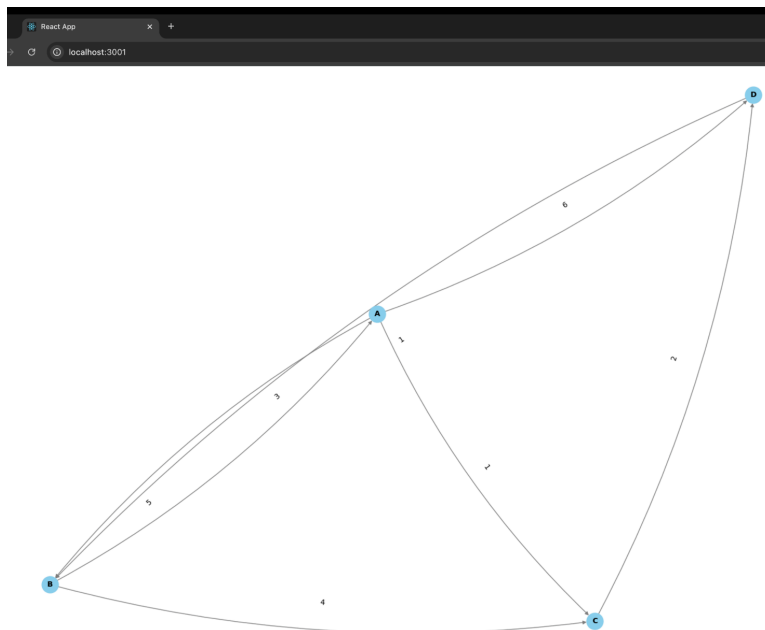
5. Visualização de Imagens

Sendo a visualização gráfica essencial para compreender a estrutura do grafo, a aplicação oferece uma interface intuitiva que exibe os vértices como nós e as arestas como linhas conectando-os.

Como funciona:

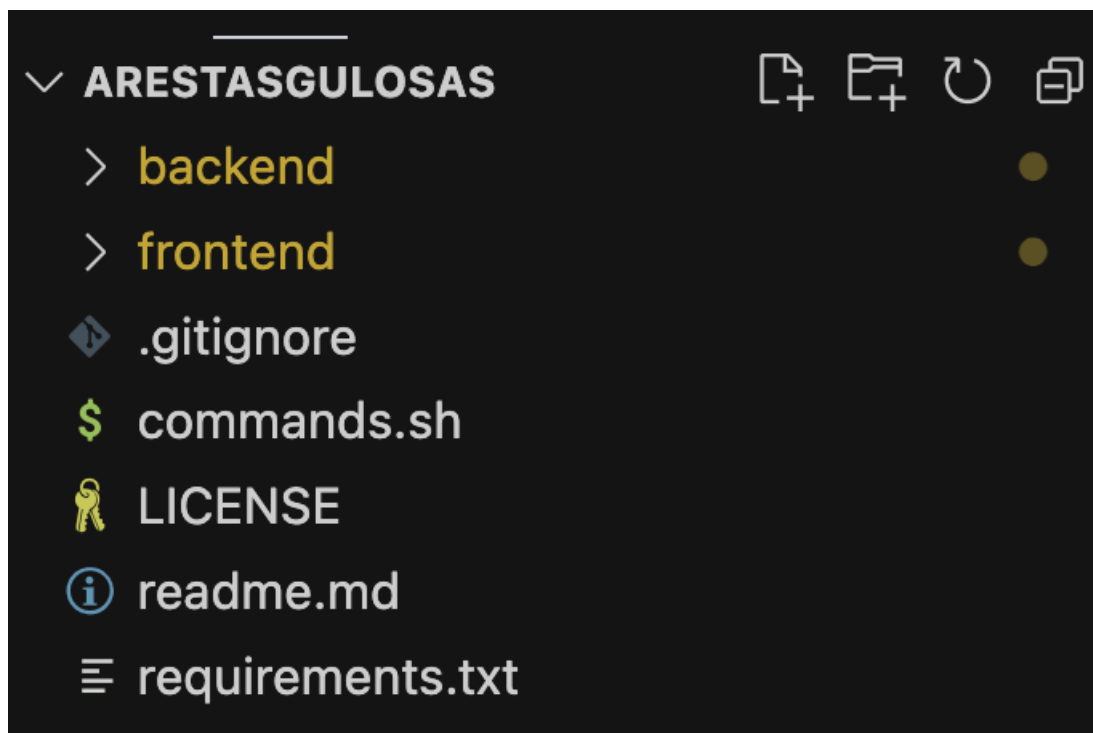
- A) Clique na opção “Visualizar Grafo”.

O grafo será renderizado automaticamente, com vértices e arestas ajustados em uma disposição que facilita a visualização. Caso o grafo seja valorado, os pesos das arestas são exibidos junto às conexões. O arquivo será gerado no formato PNG, pronto para ser usado em relatórios ou apresentações.



C) DESCRIÇÃO CÓDIGO FONTE

O projeto foi dividido em três principais componentes: backend, frontend e comunicação via API. Tivemos o código do Backend implementado em Python utilizando Flask, o Frontend implementado em React e a comunicação entre ambos realizada por meio de endpoints REST.



Para rodar a aplicação corretamente, recomenda-se seguir os passos abaixo utilizando um ambiente virtual Python (venv) para organizar as dependências. Em casos de dúvida, verificar o arquivo `commands.sh`, que contém os comandos necessários para rodar a aplicação.

a) Criação e Ativação da Virtual Environment (venv)

- Crie um ambiente virtual na pasta principal do projeto;
 - `python -m venv venv`
- Ative o ambiente virtual:

- No Windows: `venv\Scripts\activate`
- No Linux/MacOS: `source venv/bin/activate`

b) Instalar Dependências do Projeto

- Instale as dependências listadas no arquivo requirements.txt utilizando o pip;
 - `pip install -r requirements.txt`

c) Configurar os Terminais

- Terminal 1 (Backend):
 - Navegue até a pasta backend: `cd backend`
 - Execute o Flask: `flask run`

O servidor backend estará rodando localmente, normalmente no endereço `http://127.0.0.1:5000`.

- Terminal 2 (Frontend):
 - Abra outro terminal e navegue para a pasta frontend: `cd frontend`
 - Instale as dependências do frontend: `npm install`
 - Construa o projeto: `npm run build`
 - Inicie o servidor do frontend: `npm start`

O servidor frontend estará rodando localmente, normalmente no endereço `http://localhost:3000`.

d) Acessar a Aplicação

Com o backend e o frontend rodando, você poderá acessar a aplicação no navegador, no endereço local fornecido pelo frontend (geralmente `http://localhost:3000`).

1. Backend

No projeto, o backend é responsável pelo gerenciamento das operações com os grafos, como criação, adição de vértices e arestas, cálculo de propriedades (graus, adjacências, caminhos mínimos) e geração de imagens dos grafos. As principais bibliotecas utilizadas foram o NetworkX, para manipulação de grafos, e o Matplotlib, para geração de imagens dos grafos. Ainda, ressalta-se o uso do Pandas, para manipulação de arquivos CSV. Dessa forma, o backend contém os endpoints da aplicação, presentes no arquivo `app.py`, que fornecem funcionalidades relacionadas à manipulação de grafos. Abaixo está uma descrição detalhada de cada um:

A) `/create_graph`

```
@app.route("/create_graph", methods=["POST"])
def create_graph():
    global grafo, is_directed, is_weighted
    data = request.json
    is_directed = data.get("direcionado", False)
    is_weighted = data.get("valorado", False)
    grafo = nx.DiGraph() if is_directed else nx.Graph()
    return jsonify({"message": "Grafo criado com sucesso!"})
```

Cria um novo grafo vazio, permitindo configurar como direcionado ou não-direcionado e ponderado ou não-ponderado.

- Método: POST
- Entrada:
 - direcionado (booleano): Define se o grafo é direcionado.
 - valorado (booleano): Define se o grafo é ponderado.
- Saída:
 - Mensagem de confirmação.

B) `/add_vertex`

```
@app.route("/add_vertex", methods=["POST"])
def add_vertex():
    vertice = request.json.get("vertice")
    grafo.add_node(vertice)
    return jsonify({"message": f"Vértice {vertice} adicionado com sucesso!"})
```

Adiciona um novo vértice ao grafo.

- Método: POST
- Entrada:
 - vertice (string): Nome do vértice a ser adicionado.
- Saída:
 - Mensagem indicando que o vértice foi adicionado com sucesso.

C) `/add_edge`

```
@app.route("/add_edge", methods=["POST"])
def add_edge():
    origem = request.json.get("origem")
    destino = request.json.get("destino")
    peso = request.json.get("peso", 1)

    if is_weighted:
        grafo.add_edge(origem, destino, weight=peso)
    else:
        grafo.add_edge(origem, destino)

    return jsonify({"message": f"Aresta de {origem} para {destino} com peso {peso} adicionada com sucesso!"})
```

Adiciona uma aresta entre dois vértices, podendo incluir um peso, se o grafo for valorado.

- Método: POST
- Entrada:
 - origem (string): Vértice de origem.
 - destino (string): Vértice de destino.
 - peso (opcional, número): Peso da aresta.
- Saída:
 - Mensagem indicando que a aresta foi adicionada.

D) `/get_graph_image`

```
@app.route("/get_graph_image", methods=["GET"])
def get_graph_image():
    plt.figure(figsize=(16, 12))
    pos = nx.spring_layout(grafo)
    labels = nx.get_edge_attributes(grafo, 'weight') if is_weighted else None

    nx.draw(
        grafo, pos, with_labels=True, node_color='skyblue', node_size=500,
        font_size=10, font_weight='bold', edge_color='gray',
        connectionstyle="arc3,rad=0.1"
    )

    if labels:
        for (u, v), weight in labels.items():
            if grafo.has_edge(v, u) and (v, u) in labels:
                nx.draw_networkx_edge_labels(grafo, pos, edge_labels={(u, v): weight}, label_pos=0.3)
                nx.draw_networkx_edge_labels(grafo, pos, edge_labels={(v, u): labels[(v, u)]}, label_pos=0.7)
            else:
                nx.draw_networkx_edge_labels(grafo, pos, edge_labels={(u, v): weight})

    buf = BytesIO()
    plt.savefig(buf, format="png")
    plt.close()
    buf.seek(0)
    img_base64 = base64.b64encode(buf.read()).decode("utf-8")
    return jsonify({"image": img_base64})
```

Gera uma representação gráfica do grafo em formato de imagem e retorna a imagem codificada em base64.

- Método: GET
- Saída:
 - Imagem do grafo em base64.

E) `/load_graph_from_file`

```
@app.route("/load_graph_from_file", methods=["POST"])
def load_graph_from_file():
    if 'file' not in request.files:
        return jsonify({"error": "Nenhum arquivo enviado."}), 400

    file = request.files['file']
    try:
        data = pd.read_csv(file)
    except Exception as e:
        return jsonify({"error": f"Erro ao processar o arquivo CSV: {str(e)}"}), 400

    if not {'Source', 'Target', 'Weight'}.issubset(data.columns):
        return jsonify({"error": "Arquivo CSV deve conter as colunas: Source, Target, Weight"}), 400

    for _, row in data.iterrows():
        origem = str(row['Source'])
        destino = str(row['Target'])
        peso = int(row['Weight'])

        if origem not in grafo:
            grafo.add_node(origem)
        if destino not in grafo:
            grafo.add_node(destino)

        if is_weighted:
            grafo.add_edge(origem, destino, weight=peso)
            print(f"Aresta de {origem} para {destino} com peso {peso} adicionada com sucesso!")
        else:
            grafo.add_edge(origem, destino)
            print(f"Aresta de {origem} para {destino} adicionada com sucesso!")

    return jsonify({"message": "Grafo carregado do arquivo CSV com sucesso!"})
```

Carrega um grafo a partir de um arquivo CSV contendo informações de vértices e arestas.

- Método: POST
- Entrada:
 - Arquivo CSV com as colunas: Source, Target, Weight.
- Saída:
 - Mensagem indicando sucesso ou erro.

F) `/load_graph_from_string`

```
@app.route("/load_graph_from_string", methods=["POST"])
def load_graph_from_string():
    data = request.json
    vertices = data.get("vertices", [])
    arestas = data.get("arestas", [])
    for vertice in vertices:
        grafo.add_node(vertice)
    for aresta in arestas:
        origem, destino, peso = aresta
        if is_weighted:
            grafo.add_edge(origem, destino, weight=peso)
        else:
            grafo.add_edge(origem, destino)
    return jsonify({"message": "Grafo carregado da string com sucesso!"})
```

Carrega um grafo a partir de informações fornecidas em formato JSON.

- Método: POST
- Entrada:
 - vertices (lista): Lista de vértices.
 - arestas (lista de tuplas): Cada tupla contém origem, destino e peso.
- Saída:
 - Mensagem indicando sucesso.

G) `/graph_order_size`

```
@app.route("/graph_order_size", methods=["GET"])
def graph_order_size():
    ordem = grafo.number_of_nodes()
    tamanho = grafo.number_of_edges()
    return jsonify({"ordem": ordem, "tamanho": tamanho})
```

Retorna a ordem (número de vértices) e o tamanho (número de arestas) do grafo.

- Método: GET
- Saída:
 - ordem (inteiro): Número de vértices.
 - tamanho (inteiro): Número de arestas.

H) `/adjacent_vertices`

```
@app.route("/adjacent_vertices", methods=["POST"])
def adjacent_vertices():
    vertice = request.json.get("vertice")
    if grafo.is_directed():
        adjacentes_saida = list(grafo.successors(vertice))
        adjacentes_entrada = list(grafo.predecessors(vertice))
        return jsonify({"adjacentes_saida": adjacentes_saida, "adjacentes_entrada": adjacentes_entrada})
    else:
        adjacentes = list(grafo.neighbors(vertice))
        return jsonify({"adjacentes": adjacentes})
```

Retorna os vértices adjacentes a um vértice específico.

- Método: POST
- Entrada:
 - vertice (string): Nome do vértice.
- Saída:
 - Lista de vértices adjacentes.
 - Para grafos direcionados: listas de adjacências de entrada e saída.

I) `/vertex_degree`

```
@app.route("/vertex_degree", methods=["POST"])
def vertex_degree():
    vertice = request.json.get("vertice")
    if vertice not in grafo:
        return jsonify({"error": f"0 vértice '{vertice}' não existe no grafo."}), 400

    if grafo.is_directed():
        grau_saida = grafo.out_degree(vertice)
        grau_entrada = grafo.in_degree(vertice)
        return jsonify({"grau_saida": grau_saida, "grau_entrada": grau_entrada})
    else:
        grau = grafo.degree(vertice)
        return jsonify({"grau": grau})
```

Calcula o grau de um vértice. Para grafos direcionados, retorna o grau de entrada e o grau de saída.

- Método: POST
- Entrada:
 - vertice (string): Nome do vértice.

- Saída:
 - Grau do vértice.
 - Para grafos direcionados: grau_saida e grau_entrada.

J) `/are_adjacent`

```
@app.route("/are_adjacent", methods=["POST"])
def are_adjacent():
    vertice1 = request.json.get("vertice1")
    vertice2 = request.json.get("vertice2")
    adjacente = grafo.has_edge(vertice1, vertice2)
    return jsonify({"adjacente": adjacente})
```

Verifica se dois vértices estão conectados por uma aresta.

- Método: POST
- Entrada:
 - vertice1 (string): Primeiro vértice.
 - vertice2 (string): Segundo vértice.
- Saída:
 - adjacente (booleano): Indica se os vértices estão conectados.

K) `/shortest_path`

```
@app.route("/shortest_path", methods=["POST"])
def shortest_path():
    origem = request.json.get("origem")
    destino = request.json.get("destino")
    try:
        custo = nx.shortest_path_length(grafo, source=origem, target=destino, weight='weight')
        caminho = nx.shortest_path(grafo, source=origem, target=destino, weight='weight')
        return jsonify({"custo": custo, "caminho": caminho})
    except nx.NetworkXNoPath:
        return jsonify({"message": f"Não existe caminho entre {origem} e {destino}."}), 404
```

Calcula o caminho mais curto entre dois vértices e o custo associado.

- Método: POST
- Entrada:
 - origem (string): Vértice de origem.
 - destino (string): Vértice de destino.

- Saída:
 - custo (número): Custo total do caminho mais curto.
 - caminho (lista): Sequência de vértices no caminho mais curto.

L) `/is_eulerian`

```
@app.route("/is_eulerian", methods=["GET"])
def is_eulerian():
    if nx.is_eulerian(grafo):
        return jsonify({"euleriano": True, "message": "O grafo é Euleriano!"})
    else:
        return jsonify({"euleriano": False, "message": "O grafo não é Euleriano."})
```

Verifica se o grafo é euleriano.

- Método: GET
- Saída:
 - euleriano (booleano): Indica se o grafo é euleriano.
 - Mensagem explicativa.

2. Comunicação via API

Como dito, a comunicação entre o backend e o frontend é realizada por meio de endpoints REST, com as operações disponíveis incluindo criação de grafos, adição de vértices/arestas, carregamento de grafos via arquivo ou string JSON, e obtenção de propriedades do grafo, com os serviços disponibilizados por meio de chamadas axios no frontend, conectando-se ao backend Flask.

As funções da API utilizam o Axios para enviar requisições HTTP para um backend rodando no endereço `http://127.0.0.1:5000`. Cada função corresponde a uma funcionalidade do backend para gerenciar e operar sobre grafos e pode-se observar a atuação da API no arquivo `API.js`, presente no Frontend da aplicação.

A) `createGraph(direcionado, valorado)`

Cria um novo grafo no backend, com as propriedades especificadas (direcionado e/ou valorado).

- Método HTTP: POST
- Rota Backend consumida: `/create_graph`
- Parâmetros:
 - `direcionado`: booleano indicando se o grafo será direcionado.
 - `valorado`: booleano indicando se o grafo terá pesos nas arestas.

B) `addVertex(vertice)`

Adiciona um novo vértice ao grafo existente no backend.

- Método HTTP: POST
- Rota Backend consumida: `/add_vertex`
- Parâmetros:
 - `vertice`: nome do vértice a ser adicionado.

C) `addEdge(origem, destino, peso)`

Adiciona uma aresta entre dois vértices no grafo. O peso é incluído caso o grafo seja valorado.

- Método HTTP: POST
- Rota Backend consumida: `/add_edge`
- Parâmetros:
 - `origem`: vértice de origem da aresta.
 - `destino`: vértice de destino da aresta.
 - `peso` (opcional): peso da aresta, se o grafo for valorado.

D) `getGraphImage()`

Retorna a imagem ou representação visual do grafo gerado no backend, permitindo que o cliente visualize a estrutura do grafo.

- Método HTTP: GET
- Rota Backend consumida: `/get_graph_image`

E) `loadGraphFromFile(file)`

Envia um arquivo para o backend, que é utilizado para criar ou carregar um grafo com base nos dados contidos no arquivo.

- Método HTTP: POST
- Rota Backend consumida: `/load_graph_from_file`
- Parâmetros:
 - `file`: arquivo CSV contendo os dados do grafo.

F) `loadGraphFromString(data)`

Carrega um grafo no backend a partir de um JSON fornecido como string.

- Método HTTP: POST
- Rota Backend consumida: `/load_graph_from_string`
- Parâmetros:
 - `data`: JSON contendo a definição do grafo.

G) `getGraphOrderSize()`

Retorna a ordem (número de vértices) e o tamanho (número de arestas) do grafo atual.

- Método HTTP: GET
- Rota Backend consumida: `/graph_order_size`

H) `getAdjacentVertices(vertice)`

Retorna a lista de vértices adjacentes ao vértice informado.

- Método HTTP: POST
- Rota Backend consumida: `/adjacent_vertices`
- Parâmetros:
 - `vertice`: nome do vértice.

I) `getVertexDegree(vertice)`

Retorna o grau do vértice especificado (número de arestas conectadas a ele).

- Método HTTP: POST
- Rota Backend consumida: `/vertex_degree`
- Parâmetros:
 - `vertice`: nome do vértice.

J) `areVerticesAdjacent(vertex1, vertice2)`

Verifica se dois vértices são adjacentes (ou seja, se há uma aresta conectando-os).

- Método HTTP: POST
- Rota Backend consumida: `/are_adjacent`
- Parâmetros:
 - `vertex1`: vértice de origem.
 - `vertice2`: vértice de destino.

K) `getShortestPath(origem, destino)`

Calcula o menor caminho entre dois vértices e retorna o custo e a sequência de vértices que formam o caminho.

- Método HTTP: POST
- Rota Backend consumida: `/shortest_path`
- Parâmetros:
 - `origem`: vértice de origem do caminho.
 - `destino`: vértice de destino do caminho.

L) `getIsEurelian()`

Verifica se o grafo atual é Euleriano (possui um circuito ou caminho de Euler).

- Método HTTP: GET
- Rota Backend consumida: `/is_eulerian`

3. Frontend

Já o frontend fornece uma interface amigável para interação com o grafo, por onde o usuário pode criar grafos, adicionar vértices/arestas, carregar dados de arquivos e visualizar propriedades. Nele, pode-se determinar o controle de grafos direcionados e/ou valorados e realizar operações como verificar adjacência, calcular caminhos mínimos, e testar se o grafo é Euleriano. Dessa forma, o frontend é estruturado com componentes que permitem:

- Configuração inicial do grafo (direcionado/valorado).
- Adição de vértices e arestas com formulários dinâmicos.
- Exibição das propriedades do grafo, como ordem, tamanho, grau e adjacência.

Referente às funções e lógicas aplicadas, que estão disponíveis no arquivo `GraphControls.js`, temos:

A) `handleCreateGraph`

Cria um grafo vazio com as características definidas (direcionado e/ou valorado).

- Funcionamento:
 - Chama a função `createGraph` da API, passando as informações de direcionamento e valoração.
 - Atualiza a imagem do grafo chamando `fetchGraphImage`.

B) `handleAddVertex`

Adiciona um novo vértice ao grafo.

- Funcionamento:
 - Chama a função `addVertex` da API com o nome do vértice.
 - Limpa o campo de entrada do vértice (`setVertice("")`).
 - Atualiza a imagem do grafo com `fetchGraphImage`.

C) `handleAddEdge`

Adiciona uma aresta entre dois vértices no grafo.

- Funcionamento:
 - Verifica se o grafo é valorado para determinar se o peso será enviado.

- Chama a função `addEdge` da API com os vértices de origem e destino, além do peso (se houver).
- Limpa os campos de entrada (`setOrigem("")`, `setDestino("")`, `setPeso("")`).
- Atualiza a imagem do grafo com `fetchGraphImage`.

D) `fetchGraphImage`

Atualiza a visualização do grafo carregando a imagem mais recente.

- Funcionamento:
 - Chama a função `getGraphImage` da API, que retorna a imagem atual do grafo.
 - Atualiza o estado `setGraphImage` com a nova imagem.

E) `fetchOrderSize`

Recupera e exibe a ordem (número de vértices) e o tamanho (número de arestas) do grafo.

- Funcionamento:
 - Chama a função `getGraphOrderSize` da API.
 - Atualiza o estado `setOrderSize` com os valores recebidos.

F) `fetchIsEurelian`

Verifica se o grafo é euleriano.

- Funcionamento:
 - Chama a função `getIsEurelian` da API.
 - Atualiza o estado `setEurelian` com o resultado da verificação.

G) `fetchAdjacentVertices`

Retorna os vértices adjacentes a um vértice específico.

- Funcionamento:
 - Verifica se um vértice foi informado.

- Chama a função `getAdjacentVertices` da API com o vértice informado.
- Atualiza o estado `setAdjacentVertices` com os vértices adjacentes.

H) `fetchVertexDegree`

Calcula e exibe o grau de um vértice (número de conexões).

- Funcionamento:
 - Verifica se o vértice foi informado.
 - Chama a função `getVertexDegree` da API com o nome do vértice.
 - Atualiza o estado `setVertexDegree` com o grau retornado.

I) `checkAdjacency`

Verifica se dois vértices são adjacentes (conectados por uma aresta).

- Funcionamento:
 - Verifica se ambos os vértices de origem e destino foram informados.
 - Chama a função `areVerticesAdjacent` da API.
 - Atualiza o estado `setAdjacency` com true ou false.

J) `findShortestPath`

Calcula o caminho mais curto entre dois vértices.

- Funcionamento:
 - Verifica se os vértices de origem e destino foram informados.
 - Chama a função `getShortestPath` da API.
 - Atualiza o estado `setShortestPath` com o custo e a sequência de vértices do caminho mais curto.

K) `handleLoadGraphFromFile`

Carrega um grafo a partir de um arquivo CSV.

- Funcionamento:
 - Verifica se um arquivo foi selecionado.

- Chama a função `loadGraphFromFile` da API, passando o arquivo.
- Exibe mensagens de erro ou sucesso.
- Atualiza a imagem do grafo com `fetchGraphImage`.

L) `handleLoadGraphFromString`

Carrega um grafo a partir de uma string JSON.

- Funcionamento:
 - Verifica se o JSON inserido é válido.
 - Chama a função `loadGraphFromString` da API, passando os dados JSON.
 - Exibe mensagens de erro caso o JSON seja inválido.
 - Atualiza a imagem do grafo com `fetchGraphImage`.

M) *Estados e Campos Relacionados*

- `direcionado` e `valorado`: Controlam as opções de configuração do grafo (se é direcionado ou contém valores nas arestas).
- `vertice`, `origem`, `destino`, `peso`: Capturam as entradas do usuário para operações com vértices e arestas.
- `file` e `jsonString`: Armazenam o arquivo e a string JSON para carregamento do grafo.
- `orderSize`, `eurelian`, `adjacentVertices`, `vertexDegree`, `adjacency`, `shortestPath`: Armazenam os resultados das operações realizadas no grafo.