

**FATEC SHUNJI NISHIMURA**  
**BIG DATA NO AGRONEGÓCIO**

Andreas Trisoglio, Daniel Chiaveri, Eduardo Alves, Sabrina Madeira, Victor Hugo

**CAMINHOS EULERIANOS, HAMILTONIANOS E COMPONENTES CONEXOS**

Pompeia/SP

2025

## **RESUMO**

Os grafos constituem uma das estruturas matemáticas mais versáteis e aplicadas na ciência moderna, oferecendo suporte para problemas de conectividade, otimização e modelagem de sistemas complexos. Entre os conceitos fundamentais da teoria dos grafos estão os caminhos e ciclos eulerianos, os caminhos e ciclos hamiltonianos e os componentes conexos, cada um responsável por descrever diferentes propriedades estruturais e comportamentais de um grafo. Os conceitos eulerianos, originados com o famoso Problema das Sete Pontes de Königsberg, tratam da possibilidade de percorrer todas as arestas sem repetição. Já os conceitos hamiltonianos, inspirados no jogo do Icosaedro de Hamilton, concentram-se na visita a todos os vértices sem repeti-los. Por fim, os componentes conexos permitem identificar partes independentes de um grafo, revelando subestruturas desconectadas. Este trabalho apresenta a fundamentação teórica, os algoritmos associados e exemplos de aplicação prática dessas três áreas essenciais no estudo dos grafos.

## 1 - INTRODUÇÃO

A teoria dos grafos consolidou-se como uma das áreas mais importantes da matemática discreta, fornecendo bases teóricas e ferramentas práticas para a resolução de problemas presentes em computação, engenharia, logística e diversas outras áreas. Entre os tópicos mais relevantes desse campo destacam-se os caminhos e ciclos eulerianos, os caminhos e ciclos hamiltonianos e os componentes conexos, que surgiram inicialmente a partir de curiosidades matemáticas ou problemas reais, mas que hoje desempenham papel central em modelagens computacionais. Os estudos sobre caminhos e ciclos eulerianos tiveram início com Leonhard Euler ao resolver o Problema das Sete Pontes de Königsberg, enquanto os conceitos hamiltonianos foram introduzidos por William Rowan Hamilton por meio de seu famoso jogo do Icosaedro. Complementarmente, o entendimento da conectividade entre vértices por meio de componentes conexos possibilita identificar estruturas internas de um grafo, sendo fundamental na análise de redes reais. Neste trabalho, serão explorados os fundamentos teóricos desses conceitos, seus principais algoritmos e algumas aplicações práticas que evidenciam sua importância.

### 1.1 - HISTÓRIA

O conceito dos Caminhos Eulerianos surgiu de um dos problemas mais famosos da história da matemática.

Na época, a cidade prussiana de Königsberg — hoje Kaliningrado, na Rússia — era cortada por um rio com duas ilhas conectadas por 7 pontes. Nisso, os moradores queriam saber: é possível passar por todos os locais da cidade atravessando todas as pontes somente uma vez?

Então, Leonhard Euler, grande nome da matemática, foi desafiado a resolver o problema.

Como conclusão, Euler transformou a cidade em um grafo, onde os locais da cidade seriam os vértices, e as pontes, arestas. Com isso, surgiu uma das primeiras regras para a criação de um grafo: se um vértice tem número ímpar de arestas, você entra e sai um número ímpar de vezes.

Mais de um século depois, surgiu o conceito dos Caminhos Hamiltonianos, este baseado em um jogo, o jogo do Icosaedro de William Rowan Hamilton.

Esse jogo tinha como princípio a figura de um icosaedro — poliedro de 20 faces — , em que cada vértice representasse uma cidade. O objetivo desse jogo era visitar todas as cidades exatamente uma única vez e retornar ao ponto de partida.

Resolvendo o próprio jogo, Hamilton não conseguiu chegar a uma conclusão específica, pois o problema era, em si, muito complexo.

Assim, a realidade do problema de Hamilton só foi entendida décadas depois, estudada por Dirac e diversos outros matemáticos posteriormente, enfim chegando à teoria do grafo.

## 2 - FUNDAMENTAÇÃO TEÓRICA

A saber, existem diversos conceitos relacionados à teoria dos grafos, vitais para o entendimento da pesquisa.

Segundo Samuel Jurkiewicz (2009), grafos são uma forma de representar um relacionamento entre entidades, podendo ser exibido na forma de um gráfico contendo vértices (as entidades) e arestas conectando esses vértices (o relacionamento entre as entidades); ou exibido de forma matemática utilizando listas, de maneira que ' $V = \{ v_1, v_2, \dots \}$ ' representa a lista dos vértices. ' $E = \{(v_1, v_2), \dots\}$ ' representa a lista de arestas; e ' $G = \{V, E\}$ ' representa o gráfico.

Para compreender alguns dos teoremas citados nesta pesquisa, é importante explicar o conceito de 'grau de um vértice', que caracterizado por Samuel Jurkiewicz (2009), significa a quantidade de arestas incidentes a esse mesmo vértice.

Outro conceito fundamental é a concepção de trilha, que, de acordo com Samuel Jurkiewicz (2009), representa uma sequência de arestas distintas.

Além disso, é importante citar que, explicado por Samuel Jurkiewicz (2009), a trilha fechada representa uma trilha em que o vértice de origem e a de destino são os mesmos.

Ademais, Samuel Jurkiewicz (2009) também informa que caminhos representam uma trilha em que todos os vértices são distintos.

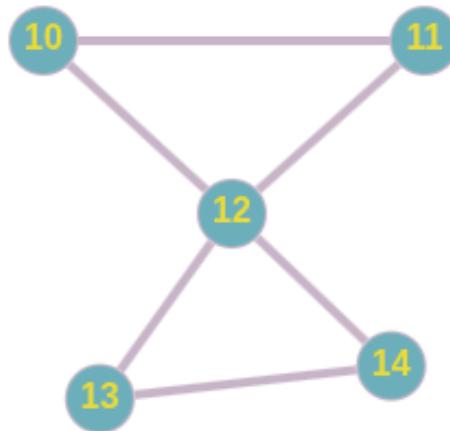
Na mesma linha, Samuel Jurkiewicz (2009) apresenta o conceito de ciclo como um caminho em que o vértice de origem e a de destino são os mesmos.

Como caracteriza Cícero Quarto (2021), ciclos eulerianos podem ser definidos a partir de uma série de características, sendo elas: Pertencer a um grafo conexo; ser uma trilha fechada conectando todas as arestas e voltando ao ponto de início, sem repetir arestas;

Um teorema que define se um grafo pode ser categorizado como ciclo euleriano é: “Um multi grafo conexo com, pelo menos, dois vértices tem um ciclo euleriano somente se cada um de seus vértices tiver grau par.” (QUARTO, 2021).

Exemplo de ciclo euleriano:

**Ilustração 1 - Grafo contendo ciclo euleriano**



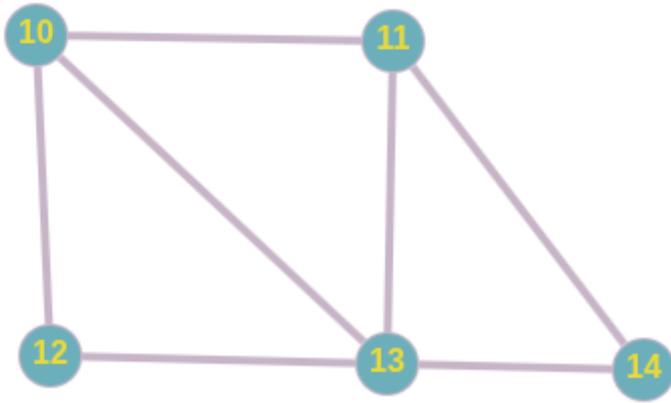
Fonte: Elaboração própria

Diferentemente dos ciclos eulerianos, os caminhos eulerianos seguem as mesmas premissas, com as exceções de não haver o retorno ao vértice de origem, e possuir exatamente dois vértices de grau ímpar.

Um teorema que define se um grafo pode ser categorizado como caminho euleriano é: “Um multi grafo conexo tem um caminho euleriano, mas não um ciclo euleriano se e somente se tiver exatamente dois vértices de grau ímpar.” (QUARTO, 2021).

Exemplo de caminho euleriano:

**Ilustração 2 - Grafo contendo caminho euleriano**



Fonte: Elaboração própria

Exemplos de utilização prática dos caminhos e ciclos eulerianos são vários, podendo ser citado desafios que envolvem percorrer uma única vez: “[...] cada estrada em uma rede de transporte; cada conexão em uma rede de serviços públicos; cada link em uma rede de comunicações; [...]” (QUARTO, 2021).

Entre os principais algoritmos, é possível citar o Algoritmo de Hierholzer, proposto em 1873, foi um dos primeiros a tratar ciclos eulerianos. “A idéia é, a partir de um vértice qualquer, percorrer arestas até retornar ao vértice inicial. Porém, desta forma, pode ser obtido um ciclo que não inclua todas as arestas do grafo”. (CARVALHO, 2019).

Também podemos usar de exemplo o algoritmo de Fleury, como explica Carvalho (2019), esse algoritmo foi proposto em 1883 e também usa grafo reduzido induzido pelas arestas ainda não marcadas pelo algoritmo. De início, nenhuma das arestas está marcada, e, a partir de um vértice aleatório, uma aresta que obedeça a regra da ponte é escolhida para ser percorrida e inserida no ciclo.

A regra da ponte, é tratada de maneira a qual: “Se uma aresta  $\{v, w\}$  é uma ponte no grafo reduzido, então  $\{v, w\}$  só deve ser escolhida pelo algoritmo de Fleury caso não haja outra opção.” (CARVALHO, 2019).

Como caracteriza Cícero Quarto (2021), ciclos hamiltonianos podem ser definidos a partir de uma série de características, sendo elas: Pertencer a um grafo conexo; ser uma trilha fechada conectando todos os vértices e voltando ao ponto de início, sem repetir vértices.

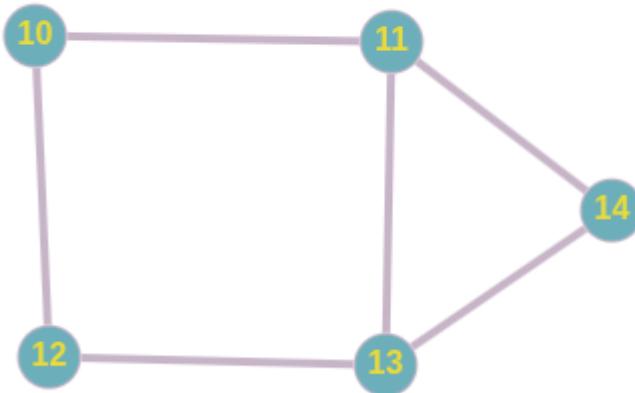
Segundo Cícero Quarto (2021), diferentemente dos ciclos eulerianos, a utilização de teoremas para definir se um grafo possuí um ciclo hamiltoniano é

menos conclusiva, pois, apesar da condição ser atingida significar que o grafo pode ser classificado como ciclo hamiltoniano, o não atendimento da condição não é suficiente para negar a possibilidade dele ser um ciclo hamiltoniano. Os dois principais teoremas são:

- Teorema de Dirac: seja  $G$  um grafo simples com 3 ou mais vértices. Se o grau de cada vértice for maior ou igual a metade do número de vértices, então  $G$  possui um ciclo hamiltoniano;
- Teorema de Ore: seja  $G$  um grafo de  $n$  vértices. Se a soma dos graus de cada par de vértice não-adjacente for maior ou igual a  $n$ , então  $G$  possui um ciclo hamiltoniano.

Exemplo de ciclo hamiltoniano:

**Ilustração 3 - Grafo contendo ciclo hamiltoniano**

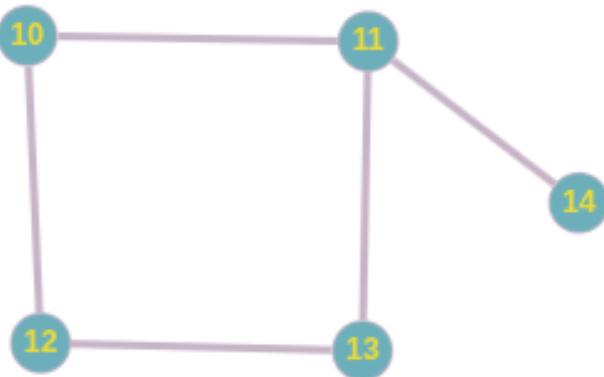


Fonte: Elaboração própria

Diferentemente dos ciclos hamiltonianos, os caminhos hamiltonianos seguem as mesmas premissas, com a exceção de não haver o retorno ao vértice de origem.

Exemplo de caminho hamiltoniano:

**Ilustração 4 - Grafo contendo caminho hamiltoniano**



Fonte: Elaboração própria

Segundo Picolo e Silva (2021), exemplos de utilização prática dos caminhos e ciclos hamiltonianos envolvem casos de otimização, como usar o problema do caixeiro viajante, em que é necessário passar por um conjunto de cidades, sem repeti-las, pelo menor trajeto possível, para decidir rotas de pontos turísticos.

Entre os principais algorítmos é possível citar o Algoritmo de Held-Karp, proposto em 1962 por Bellman e por Held e Karp para resolver o problema do caixeiro viajante (TSP). Reis (2025) caracteriza o Algoritmo de Held-Karp como uma solução de complexidade  $O((n^2) * (2^n))$ , que utiliza programação dinâmica para resolver o TSP ao minimizar o custo de um ciclo Hamiltoniano, visitando as cidades exatamente uma vez e retornando à origem.

Quanto à definição de grafos:

Um grafo  $G$  é chamado conexo se existe um caminho para cada par de vértices de  $G$ . Caso contrário, é chamado desconexo. Um grafo desconexo possui partes conexas que são chamadas de componentes. Um grafo conexo possui um único componente conexo enquanto um grafo desconexo possui vários componentes conexos. (Souza, 2013)

De acordo com Samuel Jurkiewicz (2009), componentes são uma parte do grafo em que os vértices, ou conjunto de vértices conectados entre si, não possuem arestas os conectando ao restante dos vértices do mesmo grafo. Complementando, um componente é um subgrafo de um grafo  $G$  desconexo, em que nenhum de seus vértices faz parte de outro subgrafo de  $G$ .

Ademais, como é caracterizado por Cardoso (2005), componentes conexos são subgrafos maximais do grafo  $G$ , ou seja, eles são subgrafos em que todos os pares vértices estão conectados e que possuem o máximo de vértices de  $G$  de

modo que se for adicionado mais um vértice de G a esse subgrupo, ele perderá a propriedade de conexo e se tornará um subgrafo desconexo.

Como exemplo de utilização prática de componentes conexos, Almeida (2020) apresenta a setorização de sistemas de abastecimento de água.

Ao buscar por componentes em grafos utiliza-se os algoritmos de BFS E DFS.

Segundo INTERVIEW CAKE, o BFS é um algoritmo que realiza a busca nível a nível em um grafo. Partindo de um ponto de partida, visita todos os seus vizinhos imediatos, depois os vizinhos destes e assim por diante, até percorrer por todo o grafo.

Já o algoritmo de DFS pode ser tido como:

"A busca em profundidade" (DFS, do inglês Depth-First Search ) é um método para explorar uma árvore ou grafo. Em uma DFS, você percorre um caminho o mais profundamente possível antes de voltar e tentar um caminho diferente.

A busca em profundidade é como caminhar por um labirinto de milho. Você explora um caminho, chega a um beco sem saída e volta para tentar um caminho diferente." (INTERVIEW CAKE,[s.d.]).

Esses dois algoritmos são amplamente usados devido a sua característica de de travessia, ou seja, exploram todos os vértices alcançáveis a partir de um ponto de partida.

### **3 - METODOLOGIA / ALGORÍTMOS**

#### **3.1 - ALGORITMO EULERIANO**

Como entrada, a função com o código recebe uma lista de adjacência estruturada em dicionário.

##### **Ilustração 5 - Escopo da Função.**

```
def grafo_euleriano(grafo: dict) -> None:
```

Fonte: Elaboração própria.

Para identificar grafos com caminhos e ciclos eulerianos, foi utilizado o algoritmo de Hierholzer, que utiliza a estrutura de *stacks* para a execução.

Primeiro, é verificada a existência de um caminho ou de um ciclo euleriano a partir dos graus dos vértices existentes. Se houver no grafo exatamente dois vértices com grau ímpar, o código trata a entrada como um caminho euleriano. Caso todos os graus de todos os vértices seja par, o código trata a entrada como ciclo euleriano. Por último, se nenhuma das opções anteriores forem satisfeitas, o código encerrará e dirá que não se trata de um grafo euleriano. Assim, o código pode definir o vértice pelo qual iniciará a busca, o *start point*.

**Ilustração 6 - Código que Identifica a Existência de um Caminho ou um Ciclo Euleriano**

```
# criando o ponto de partida do grafo
start_point = None

# descobrindo a existência de vértices com grau ímpar
grau_impar: list = [v for v in grafo if len(grafo[v]) % 2 == 1]

# definindo se é caminho, ciclo ou grafo não euleriano
if len(grau_impar) == 0:
    print("Ciclo Euleriano")
    start_point = next(iter(grafo))
elif len(grau_impar) == 2:
    print("Caminho Euleriano")
    start_point = grau_impar[0]
else:
    print("O Grafo não é Euleriano")
    return
```

Fonte: Elaboração própria.

Após a definição do tipo de grafo, são criadas as listas que armazenarão, respectivamente, os vértices visitados pelo algoritmo e os vértices colocados em *stack*.

O *stack* terá como primeiro vértice aquele anteriormente definido como *start point*.

**Ilustração 7 - Listas que Armazenam os Vértices Visitados e o Stack no Código.**

```
# criando o algoritmo com base em stacks
visitados = []
stack = [start_point]
```

Fonte: Elaboração própria.

Enquanto houver vértices armazenados no stack, o algoritmo se repetirá em *loop*, sempre pegando o último vértice do *stack* para realizar a busca.

#### Ilustração 8 - Início do Loop e Último Vértice da Lista de Adjacência.

```
while stack:
    # pegando o último vértice do stack
    vertice = stack[-1]
```

Fonte: Elaboração própria.

O algoritmo, então, identifica os vértices adjacentes ao vértice que pegou do *stack*, removendo a aresta que os liga da lista de adjacência e colocando aquele vértice adjacente no fundo do *stack*.

Caso não haja mais vértices adjacentes possíveis, o vértice é colocado na lista de vértices visitados.

#### Ilustração 9 - Algoritmo da Busca.

```
# Se o vértice tiver qualquer outro vértice adjacente,
# remover a aresta que os liga e colocá-lo no stack
if grafo[vertice]:
    adjacente = grafo[vertice].pop()
    grafo[adjacente].remove(vertice)

    stack.append(adjacente)

# caso não, colocar o vértice na lista de visitados
else:
    visitados.append(stack.pop())
```

Fonte: Elaboração própria.

Por fim, após esvaziar o *stack*, é imprimida a lista de vértices visitados e finalizada a função.

**Ilustração 10 - Finalização do Código.**

```
# encerrando o algoritmo
print("Vértices visitados:", visitados[::-1])
return
```

Fonte: Elaboração própria.

### 3.2 - ALGORITMO HAMILTONIANO

Para a identificação de um Ciclo Hamiltoniano em um grafo foi utilizado uma busca exaustiva com Backtracking, mesmo sendo a maneira mais lenta, ainda assim se demonstra ser a mais usada e efetiva para esse tipo de identificação sem contar os teoremas.

A função hamiltoniano recebe como entrada o Grafo (Matriz de Adjacência), o vértice atual e o caminho, o mesmo sendo recebido como None para inicialização da verificação.

**Ilustração 11 - Ínicio de Função**

```
def hamiltoniano(grafo:dict, vertice, caminho=None):
```

Fonte: Elaboração própria.

O Código faz a verificação através de uma busca exaustiva com backtracking. Iterando sobre todos os vértices e verificando todos os vizinhos para identificar o caminho, verifica se a quantidade de vértices é igual ao caminho percorrido, se essa condição for verdadeira, verifica se o primeiro item “caminho[0]” é igual ao vértice atual, se for encerra um ciclo, caso não seja, o Grafo não é hamiltoniano.

### Ilustração 12 - Função para identificar se é hamiltoniano

```
# Para quando o caminho terminar todos os vertices
if len(caminho) == n_vertices:
    # Verifica se conectou para fechar um ciclo
    if caminho[0] in grafo.get(vertice, []):
        return caminho
    else:
        return None

# Itera nos vizinhos
for vizinho in grafo.get(vertice, []):
    # Verifica se o vizinho já foi visitado
    if vizinho not in caminho:
        # Adiciona Vertice ao caminho
        caminho.append(vizinho)

        # Chamada recursiva
        resultado = hamiltoniano(grafo, vizinho, caminho)
        if resultado is not None:
            return resultado
```

Fonte: Elaboração própria.

Durante a etapa inicial é verificado se o caminho é vazio para que seja possível começar os testes, e iniciar a contagem de vértices para identificação do grafo.

### Ilustração 13 - Variáveis de inicialização

```
# Para recursividade não falhar
if caminho is None:
    caminho = [vertice]

# Quantidade de Vertices
n_vertices = len(grafo)
```

Fonte: Elaboração própria.

Nesta, o algoritmo faz a verificação do ciclo, em que, se o primeiro item do caminho for igual ao vértice atual, é encerrado o ciclo.

### Ilustração 14 - Verifica o Ciclo

```

# Para quando o caminho terminar todos os vertices
if len(caminho) == n_vertices:
    # Verifica se conectou para fechar um ciclo
    if caminho[0] in grafo.get(vertice, []):
        return caminho
    else:
        return None

```

Fonte: Elaboração própria.

Pelo fato do algoritmo utilizar backtracking e pilhas, foi usada a estratégia de recursividade para a verificação do resultado. Desta maneira, ao verificar que um vizinho ainda não foi visitado, ele é adicionado ao caminho e é testado o resultado com o vizinho, tornando-o o vértice atual.

#### Ilustração 15 - Looping e Recursividade

```

# Itera nos vizinhos
for vizinho in grafo.get(vertice, []):
    # Verifica se o vizinho já foi visitado
    if vizinho not in caminho:
        # Adiciona Vértice ao caminho
        caminho.append(vizinho)

        # Chamada recursiva
        resultado = hamiltoniano(grafo, vizinho, caminho)
        if resultado is not None:
            return resultado

        # Backtracking: remove apenas se a tentativa falhou
        caminho.pop()
return None

```

Fonte: Elaboração própria.

Por fim, se for encontrado um ciclo hamiltoniano no grafo, a sequência de vértices que representa o ciclo é retornada, se não, retorna nulo para uso posterior no programa.

### 3.3 - ALGORITMO DE COMPONENTES CONEXOS

Para a identificação e contagem dos componentes conexos em um grafo, utiliza-se a técnica de travessia, como a Busca em Largura (BFS) ou a Busca em Profundidade (DFS). O algoritmo a seguir utilizou a BFS para explorar todos os vértices alcançáveis a partir de um ponto inicial. Ao garantir que todos os vértices de um componente sejam visitados antes de passar para o próximo, é possível contar quantos subgrafos maximais conexos existem.

A função `achar_componentes(grafo)` recebe como entrada o grafo representado por uma matriz de adjacência, onde `grafo[i][j] = 1` indica uma aresta entre o vértice *i* e o vértice *j*.

Quanto à definição da função e inicialização, a primeira etapa define a função principal e inicializa as estruturas de dados essenciais para o rastreamento do estado da busca. A lista `visitados` é usada para armazenar os índices dos vértices que já foram explorados, evitando ciclos e recontagens. A variável `componentes_conexos` é o contador que será incrementado a cada novo componente encontrado.

#### Ilustração 16 - Menu e Inicialização de variáveis

```
14  def achar_componentes(grafo):
15      """
16          Função para encontrar o número de componentes conexos em um grafo representado por uma matriz de adjacência.
17          args:
18              grafo (list of list of int): Matriz de adjacência representando o grafo.
19              Exemplo:
20              grafo = [
21                  [0, 1, 0],
22                  [1, 0, 0],
23                  [0, 0, 0]
24              ]
25      """
26      # Inicializa uma lista para rastrear os vértices já visitados
27      visitados = []
28      # Inicializa o contador de componentes conexos
29      componentes_conexos = 0
30
```

Fonte: Elaboração própria.

Em seguida, sobre a iteração sobre os vértices e o início da busca, o algoritmo itera sobre todos os vértices do grafo. Se um vértice ainda não foi visitado, isso significa que ele pertence a um novo componente conexo. Neste ponto, o contador `componentes_conexos` é incrementado, e o vértice inicial é adicionado à lista `visitados` e à fila (ou stack, no caso de DFS) para iniciar a travessia.

#### Ilustração 17 - BFS

```

31     # Itera sobre cada vértice do grafo, usando enumerate para obter o índice (número do vértice)
32     for vertice_num, vertice in enumerate(grafo):
33
34         # Verifica se o vértice atual ainda não foi visitado
35         if vertice_num not in visitados:
36
37             visitados.append(vertice_num)
38             componentes_conexos += 1
39
40         # Inicializa uma fila para a busca em largura (BFS) a partir deste vértice
41         fila = [vertice_num]

```

Fonte: Elaboração própria.

Abordando a BFS, ela é aplicada para explorar todos os vértices do componente atual. O processo continua enquanto a fila não estiver vazia. O algoritmo retira o primeiro vértice da fila e verifica todos os seus vizinhos. Se um vizinho possui uma conexão (conexao == 1) e ainda não foi visitado, ele é marcado como visitado e adicionado ao final da fila para posterior exploração.

### Ilustração 18 - BFS continuação

```

42     # Enquanto houver vértices na fila
43     while fila:
44
45         # Algoritmo de busca por largura
46         # Itera sobre as conexões do vértice na frente da fila
47         for idx, conexao in enumerate(grafo[fila[0]]):
48             # Se há uma conexão (conexao == 1) e o vértice conectado não foi visitado
49             if conexao == 1 and idx not in visitados:
50                 # Marca o vértice conectado como visitado
51                 visitados.append(idx)
52                 # Adiciona o vértice conectado à fila para depois
53                 fila.append(idx)
54             # Remove o vértice da frente da fila após explorar suas conexões
55             fila.pop(0)

```

Fonte: Elaboração própria.

Por fim, quanto a finalização e a saída, após a conclusão da iteração sobre todos os vértices e o esvaziamento das filas de busca, todos os componentes conexos foram identificados e contados. A função, então, exibe o resultado final.

## 4 - RESULTADOS (exemplos de execução)

Durante os testes, foi utilizado a lista de adjacências construída internamente ao programa como o grafo principal do programa, contendo 5 vértices nomeados de ‘A’ até ‘E’, com o intuito de aumentar a agilidade e reprodutividade dos casos de testes. A estrutura utilizada é exibida na ilustração 19.

### Ilustração 19 - Lista de adjacências

```
grafo = {
    'A': ['B', 'C', 'D'],
    'B': ['A', 'C', 'E'],
    'C': ['A', 'B', 'D', 'E'],
    'D': ['A', 'C', 'E'],
    'E': ['B', 'C', 'D']
}
```

Fonte: Elaboração própria.

A função `grafo_euleriano()` foi executada para identificar se o grafo satisfazia as condições necessárias para a existência de um ciclo, ou caminho Euleriano.

Conforme apresentado na Ilustração 20, a saída obtida foi:

#### Ilustração 20 - Verificação da classificação Euleriana

```
==== MENU DE ANÁLISE DE GRAFOS ====
1 - Verificar se é Euleriano
2 - Encontrar ciclo Hamiltoniano
3 - Contar componentes conexos
4 - Sair
-----
Escolha uma opção (1-4): 1

==== VERIFICAÇÃO EULERIANA ====
0 Grafo não é Euleriano
```

Fonte: Elaboração própria.

A função `hamiltoniano()` foi executada para identificar se um caminho ou ciclo Hamiltoniano, a partir de um vértice selecionado, e exibi-lo. Conforme apresentado na Ilustração 21, a saída obtida foi:

#### Ilustração 21 - Resultado da busca por ciclo Hamiltoniano

```
==== MENU DE ANÁLISE DE GRAFOS ====
1 - Verificar se é Euleriano
2 - Encontrar ciclo Hamiltoniano
3 - Contar componentes conexos
4 - Sair
-----
Escolha uma opção (1-4): 2

==== CICLO HAMILTONIANO ====
Digite o vértice inicial (A, B, C, D, E): c
Ciclo Hamiltoniano encontrado: C -> A -> B -> E -> D
```

Fonte: Elaboração própria.

A contagem de componentes conexos foi realizada por meio da conversão do dicionário que representa a lista de adjacências em uma matriz de adjacências, por meio da função dict\_para\_matriz(), seguida de um algoritmo de busca em largura. Conforme apresentado na Ilustração 22, a saída obtida foi:

#### Ilustração 22 - Resultado da contagem de componentes conexos

```
==== MENU DE ANÁLISE DE GRAFOS ====
1 - Verificar se é Euleriano
2 - Encontrar ciclo Hamiltoniano
3 - Contar componentes conexos
4 - Sair
-----
Escolha uma opção (1-4): 3

==== COMPONENTES CONEXOS ====
Vértices: ['A', 'B', 'C', 'D', 'E']
Número de componentes conexos: 1
```

Fonte: Elaboração própria.

A ilustração 23 exibe o encerramento do programa após finalizado os testes.

#### Ilustração 23 - Encerramento do programa

```
==== MENU DE ANÁLISE DE GRAFOS ====
1 - Verificar se é Euleriano
2 - Encontrar ciclo Hamiltoniano
3 - Contar componentes conexos
4 - Sair
-----
Escolha uma opção (1-4): 4
Saindo do programa...
```

Fonte: Elaboração própria.

## 5 - CONCLUSÃO

A análise dos caminhos e ciclos eulerianos, hamiltonianos e dos componentes conexos evidencia a profundidade e a aplicabilidade da teoria dos grafos na solução de problemas contemporâneos. Os conceitos eulerianos permitem compreender situações em que é necessário percorrer todas as arestas de uma rede sem repetição, enquanto os conceitos hamiltonianos lidam com a visita única a todos os vértices, sendo essenciais em problemas de otimização, como o caixeiro viajante. Já os componentes conexos fornecem uma visão estruturada da conectividade de um grafo, permitindo identificar subgrupos independentes, fator fundamental em áreas como redes sociais, sistemas de abastecimento, visão computacional e engenharia. Em conjunto, esses tópicos demonstram como problemas inicialmente teóricos se transformaram em ferramentas indispensáveis para a computação moderna. O estudo e implementação dos algoritmos apresentados reforçam a relevância desses conceitos, destacando seu papel na construção de soluções eficientes para desafios reais.

## REFERÊNCIAS

ALMEIDA, I. C. A. de. **APLICAÇÃO DE ALGORITMO DE TEORIA DOS GRAFOS PARA SETORIZAÇÃO DE SISTEMAS DE ABASTECIMENTO.** Fundação Universidade Federal de Mato Grosso do Sul Faculdade de Engenharias, Arquitetura e Urbanismo e Geografia. Disponível em: <https://posgraduacao.ufms.br/portal/trabalho-arquivos/download/7776>. Acesso em: 24 nov. 2025

CARDOSO, D. M. **Teoria dos Grafos e Aplicações.** Departamento de Matemática da Universidade de Aveiro Mestrado em Matemática - 2004/2005. Disponível em: [https://www.researchgate.net/profile/Domingos-Cardoso-2/publication/260300819\\_Teoria\\_dos\\_Grafos\\_e\\_Aplicacoes/links/004635399c4cbe9410000000/Teoria-dos-Grafos-e-Aplicacoes.pdf](https://www.researchgate.net/profile/Domingos-Cardoso-2/publication/260300819_Teoria_dos_Grafos_e_Aplicacoes/links/004635399c4cbe9410000000/Teoria-dos-Grafos-e-Aplicacoes.pdf). Acesso em: 24 nov. 2025

CARVALHO, Marco Antonio M. **BCC204 - Teoria dos Grafos**. 2019. Universidade Federal de Ouro Preto, Ouro Preto, 2019. Disponível em: [http://www.decom.ufop.br/marco/site\\_media/uploads/bcc204/19\\_aula\\_19.pdf](http://www.decom.ufop.br/marco/site_media/uploads/bcc204/19_aula_19.pdf). Acesso em: 24 nov. 2025

INTERVIEW CAKE. **Breadth-First Search (BFS)**: Concept | Java. [Online]. San Francisco: Interview Cake, [s.d.]. Disponível em: <<https://www.interviewcake.com/concept/java/bfs>>. Acesso em: 26 nov. 2025.

INTERVIEW CAKE. **Depth-First Search (DFS)**: Concept | Java. [Online]. San Francisco: Interview Cake, [s.d.]. Disponível em: <<https://www.interviewcake.com/concept/java/dfs>>. Acesso em: 26 nov. 2025.

JURKIEWICZ, S. **Grafos - Uma Introdução**. COPPE/UFRJ. Disponível em: <http://www.obmep.org.br/docs/apostila5.pdf>. Acesso em: 22 nov. 2025.

OLIVEIRA, Gabriel Fernandes de. **O Problema do Carteiro Chinês**. 2020. Trabalho de conclusão de curso Universidade de São Paulo, São Paulo, 2020. Disponível em: <https://www.linux.ime.usp.br/~gafeol/mac0499/chinese-postman/tex/main.pdf>. Acesso em: 24 nov. 2025

PICOLO, A. P.; SILVA, R. S. **Aplicação do Problema do Caixeiro Viajante para determinação de rotas turísticas**. Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul (IFRS), Campus Veranópolis – RS – Brasil. 16 nov. 2021. Disponível em: <https://ifrs.edu.br/veranopolis/wp-content/uploads/sites/10/2022/04/Artigo-TCC-Ana-Paula-Picolo-2021-Pub.pdf>. Acesso em: 23 nov. 2025.

QUARTO, C. **Caminhos e Ciclos Eulerianos e Hamiltonianos**. DECOMP/UEMA. 1 mar. de 2021. Disponível em: <https://cicerocq.wordpress.com/wp-content/uploads/2021/03/caminhos-e-ciclos-eulerianos-e-hamiltonianos-1.pdf>. Acesso em: 21 nov. 2025.

REIS, Renato Gomes dos. O problema do caixeiro viajante: uma abordagem quântica com codificação em múltiplas bases. 2025. Dissertação (Mestrado em Ciência da Computação) - Faculdade de Ciências, Universidade Estadual Paulista, Bauru, 2025. Disponível em: <https://repositorio.unesp.br/entities/publication/42308bf0-e456-43a3-93e5-47ffbd72f48c>. Acesso em: 25 nov. 2025.

SOUZA, Audemir Lima de. **Teoria dos grafos e aplicações**. 2013. 78 f. Dissertação (Mestrado em Matemática) - Universidade Federal do Amazonas, Manaus, 2013. Disponível em: <http://tede.ufam.edu.br/handle/tede/4788>. Acesso em: 24 nov. 2025