1 INTRODUÇÃO

No trabalho a seguir, focaremos nos conceitos de circuitos Eulerianos e Hamiltonianos. O trabalho será dividido em duas partes principais. Na primeira, implementamos um algoritmo para verificar e encontrar circuitos Eulerianos, utilizando três estruturas de dados: matriz de incidência, matriz de adjacências e lista de adjacências. Na segunda parte, abordaremos uma condição necessária para grafos Hamiltonianos. Em ambas as partes analisaremos a complexidade do algoritmo em função da entrada e da estrutura utilizada e plotaremos um gráfico de complexidade para cada caso.

Para implementar os algoritmos, utilizaremos uma estrutura abstrata de grafos chamada $GrafoBase$, que servirá de interface para cada um dos tipos de estrutura de dados mostrados na Figura 1.

```python
class GrafoBase:
    def __init__(self, num_v=0):...

    def add_vertice(self):
        # Adiciona um vértice ao grafo
        pass

    2 usages
    def add_aresta(self, u, v):
        # Adiciona uma aresta entre os vértices u e v
        pass

    4 usages (1 dynamic)
    def arestas(self) -> list[tuple]:
        # Retorna uma lista de tuplas (u, v) que representam as arestas do grafo
        pass

    2 usages
    def vertices(self) -> list:
        pass

    1 usage
    def remove_aresta(self, u, v):
        # Remove a aresta entre os vértices u e v
        pass

    1 usage
    def remove_vertice(self, u):
        # Remove o vértice u do grafo
        pass

    3 usages
    def adjs(self, u) -> list[int]:
        # Retorna uma lista com os vértices adjacentes a u
        pass

    def ha_aresta(self, u, v):
        # O vértice u tem uma aresta com o vértice v?
        pass

    1 usage
    def grau(self, u) -> int:
        # O grau de um vértice é o número de arestas que incidem nele
        pass

    1 usage
    def tem_alguma_aresta(self, u):
        # O vértice tem alguma aresta?
        pass
```

Fig. 1. Interface $GrafoBase$

Além disso, serão implementadas duas estruturas abstratas mixin: uma para representar graficamente o grafo ($GrafoDisplayMixin$) e outra para ler o grafo a partir de um arquivo ($GrafoReadFileMixin$). O arquivo responsável pela leitura do grafo deve estar formatado de modo que cada linha representa um vértice e os vértices adjacentes estejam listados na mesma linha. Isso é mostrado na Figura 2.

```python
class GrafoDisplayMixin(GrafoBase):

    def __str__(self):...

    2 usages
    def display(self, filename='graph'):...

1 usage
class GrafoReadFileMixin(GrafoBase):
    1 usage
    @classmethod
    def read_graph(cls, file_name):
        with open(file_name, 'r') as f:
            # get the len of lines in file
            lines = len(f.readlines())
            graph_instance = cls(num_v=lines)
            visited = [False] * lines
            f.seek(0)
            for i in range(lines):...
        return graph_instance
```

Fig. 2. Interface $GrafoMixins$

Também teremos uma estrutura que estende as estruturas discutidas anteriormente e implementa alguns métodos com base nos métodos abstratos da primeira estrutura ($GrafoProprio$), conforme mostrado na Figura 3. Esses métodos serão usados nas estruturas de caminho euleriano ($GrafoEulerianoPathMixin$) e grafo hamiltoniano ($GrafoHamiltonianoMixin$), estes serão mostrados em mais detalhes nas próximas seções.

```python
class GrafoProrio(GrafoDisplayMixin, GrafoReadFileMixin):
    def __init__(self, num_v=0):...

    1 usage
    def vertices(self):...

    3 usages
    def bp(self, v, visited):...

    1 usage
    def eh_conectado(self):...

    @classmethod
    def __copy__(cls, self):...


1 usage
class GrafoEulerianoPathMixin(GrafoProrio):...


1 usage
class GrafoHamiltonianoMixin(GrafoProrio):...


8 usages
class Grafo(GrafoEulerianoPathMixin, GrafoHamiltonianoMixin):
    pass
```

Fig. 3. Interface Grafo

Vale constar que o método $eh\_conectado$, foi retirado da referência Eulerian path. Além disso, alguns dos códigos apresentados acima foram omitidos por questões de brevidade, mas podem ser encontrados no repositório https://github.com/VitorHugoOli/grafos_eda.git.

2 Estruturas de dados

A seguir, será discutido cada uma das estruturas de dados implementadas para representar um grafo: lista de adjacência, matriz de adjacência e matriz de incidência. Para cada uma delas, será demonstrada a complexidade das estruturas de dados escolhidas na construção da representação do grafo, bem como os métodos que a própria estrutura possui. Com base nessas complexidades, serão realizadas análises posteriores dos algoritmos de circuito euleriano e requisitos necessários para o grafo hamiltoniano.

Para as três representações do grafo, escolheu-se utilizar a estrutura de listas contíguas[1]. Na Tabela 1, é mostrada a complexidade das principais operações que serão utilizadas nessas representações: *append* (inserção no final), *pop* (remoção de um item) e *access* (acesso a um item). Essas operações já foram amplamente discutidas em sala de aula, logo não se dará tanto enfoque na explicação da ordem de complexidade. Além disso, essas ordens de complexidades foram validadas na documentação da linguagem (tempo de complexidade).

TABLE 1
Complexidade das funções da lista contígua

Operação	C_{best}	C_{worst}
Append	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Remove/Pop	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Access	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Em cada uma das representações, será realizada a análise dos seguintes métodos abstratos: *remove_vertice* (remover um vértice), *remove_aresta* (remover uma aresta), *adjs* (retornar lista de vértices adjacentes), *ha_aresta* (verificar se há uma aresta entre dois vértices), *grau* (calcular o grau de um vértice) e *tem_alguma_aresta* (verificar se há alguma aresta conectada no vértice). Além destes, também será feita a análise dos métodos próprios: *vertices* (listar todos os vértices), *bp* (busca em profundidade), *eh_conectado* (verificar se o grafo é conectado) e *__copy__* (criar uma cópia do grafo).

Para as análises que se seguem, o tamanho da entrada será representado pelo número de vértices V e o número de arestas A, uma vez que todos os métodos e algoritmos discutidos estarão realizando operações sobre estas entradas.

2.1 Lista de adjacências

Como já discutido previamente, para a representação da lista de adjacências será utilizada a estrutura de dados lista contígua da própria linguagem de programação. Nesta estrutura, cada posição conterá uma lista com os vértices adjacentes e o índice dessa posição será o vértice em questão. Isso permite uma rápida consulta aos vértices adjacentes a um determinado vértice, além de facilitar a inserção e remoção de arestas.

Na figura 4, temos a implementação para essa representação.

```python
class GrafoLstAdj(Grafo):
    def __init__(self, num_v=0):
        super().__init__(num_v=num_v)
        self.grafo: list[list[int]] = [[] for _ in range(num_v)]
        pass

    1 usage (1 dynamic)
    def add_vertice(self):...

    1 usage (1 dynamic)
    def add_aresta(self, u, v):...

    def arestas(self):...

    1 usage (1 dynamic)
    def remove_vertice(self, u):
        if len(self.grafo[u]) == 0:
            return

        # Remove o vertice u das listas de adjacencia dos outros vertices
        for v in range(self.num_v):
            for i, w in enumerate(self.grafo[v]):
                analysis.COUNTER += 1
                if w == u:
                    self.grafo[v].pop(i)
                if i >= len(self.grafo[v]):
                    break
                w = self.grafo[v][i]
                if w > u:
                    self.grafo[v][i] -= 1

        self.grafo.pop(u)
        self.num_v -= 1

    def remove_aresta(self, u, v):
        analysis.COUNTER += self.grau(u) + self.grau(v)
        self.grafo[u].remove(v)
        self.grafo[v].remove(u)

    def adjs(self, u):
        analysis.COUNTER += 1
        return self.grafo[u]

    def ha_aresta(self, u, v):
        return v in self.grafo[u]

    2 usages
    def grau(self, u):
        analysis.COUNTER += 1
        return len(self.grafo[u])

    def tem_alguma_aresta(self, u):
        analysis.COUNTER += 1
        return len(self.grafo[u]) > 0
```

Fig. 4. Implementação lista de adjacência

- **remove_vertice**: A complexidade deste método é $C(V) = V * V - 1 * V - 1 = O(V^3)$, no qual precorrerá todos os vértices e para cada um destes percorrera todos os suas arestas($V - 1$, grafo completamente conectado), e na aresta que conter o vértice a ser deletado, será executado a função pop que tem pior caso como $V - 1$(exclui o primeiro item do vetor). O melhor caso seria quando o vértice não possui-se aresta, $O(1)$.
- **remove_aresta**: A complexidade deste método é $C(V) = V - 1 = O(V)$ no pior caso e $O(1)$ no melhor caso. O pior e melhor caso é definido pela complexidade do método remoção da lista.
- **adjs**: A complexidade deste método é $O(1)$, pois apenas retorna a lista de adjacências do vértice u.
- **ha_aresta**: A complexidade deste método é $O(V)$, onde V é o número de vértices. No pior caso, a complexidade será dominada pela busca de v na lista de adjacências de u.
- **grau**: A complexidade deste método é $O(1)$, já que apenas retorna o tamanho da lista de adjacências do vértice u.
- **tem_alguma_aresta**: A complexidade deste método é $O(1)$, pois apenas verifica se a lista de adjacências

1. Em Python, as listas são contíguas e alocadas dinamicamente, logo não é necessário declarar o tamanho inicial das mesmas

do vértice u possui algum elemento.
- **vertices**: A complexidade deste método é $O(V)$, pois cria uma nova lista contendo todos os vértices do grafo.
- **bp**: A complexidade deste método é $O(V)$, pois realiza uma busca em profundidade a partir do vértice v, percorrendo todos os vértices, como o método $adjs$ é $O(1)$, somente recurção sobre os vértices tera peso na análise. Para o mlehor caso caso o vértice em questão não tenha vizinhos, será $O(1)$.
- **eh_conectado**: A complexidade deste método é $O(V)$, já que encontra um vértice com grau maior que 0 é $O(V)$, realiza uma busca em profundidade (bp) em $O(V)$ e verifica se todos os vértices foram visitados em $O(V)$.
- **__copy__**: A complexidade deste método é $O(V+A)$, pois realiza uma cópia profunda do grafo, copiando todos os vértices e suas listas de adjacências.

2.2 Matriz De Adjacência

Na representação de matriz de adjacência do grafo, será criada uma matriz de listas contíguas, na qual cada posição representa um vértice. Dentro destas posições, há uma lista com todos os outros vértices do grafo. Caso haja uma aresta entre dois vértices, o valor 1 será adicionado na posição correspondente na matriz. Essa representação permite consultas rápidas sobre a existência de arestas entre vértices, mas pode ser ineficiente em termos de espaço para grafos esparsos.

Na figura 5, temos a implementação para essa representação.

```python
class GrafoMtrAdj(Grafo):
    def __init__(self, num_v=0):
        super().__init__(num_v=num_v)
        self.grafo = [[0 for _ in range(num_v)] for _ in range(num_v)]

    1 usage (1 dynamic)
    def add_vertice(self):...

    1 usage (1 dynamic)
    def add_aresta(self, u, v):...

    def arestas(self):...

    1 usage (1 dynamic)
    def remove_vertice(self, u):
        self.grafo.pop(u)
        for v in self.grafo:
            v.pop(u)
        self.num_v -= 1

    def remove_aresta(self, u, v):
        self.grafo[u][v] -= self.grafo[u][v]
        self.grafo[v][u] -= self.grafo[v][u]

    1 usage
    def adjs(self, u):  # O(V)
        return [v for v in range(self.num_v) if self.grafo[u][v] > 0]

    def ha_aresta(self, u, v):
        return self.grafo[u][v] > 0

    def grau(self, u):
        return sum(self.grafo[u])

    def tem_alguma_aresta(self, u):  # O(V)
        for i in self.grafo[u]:
            if i > 0:
                return True
```

Fig. 5. Interface $GrafoBase$

- **remove_vertice**: A complexidade deste método é sempre $O(V^2)$, pois remove a linha correspondente ao vértice u na matriz de adjacência em O(V) e remove a coluna correspondente ao vértice u em todas as linhas restantes em $O(V^2)$.
- **remove_aresta**: A complexidade deste método é sempre $O(1)$, pois apenas decrementa os valores correspondentes na matriz de adjacência.
- **adjs**: A complexidade deste método é sempre $O(V)$, pois percorre toda a linha correspondente ao vértice u na matriz de adjacência.
- **ha_aresta**: A complexidade deste método é sempre $O(1)$, pois apenas verifica o valor correspondente na matriz de adjacência.
- **grau**: A complexidade deste método é sempre $O(V)$, pois soma todos os elementos da linha correspondente ao vértice u na matriz de adjacência.
- **tem_alguma_aresta**: A complexidade deste método no pior caso é $O(V)$, pois verifica todos os elementos da linha correspondente ao vértice u na matriz de adjacência. No melhor caso é $O(1)$, caso o vértice inicial tenha aresta com este.
- **vertices**: A complexidade deste método é sempre $O(V)$, pois percorre todos os vértices do grafo e cria uma lista com eles.
- **bp**: A complexidade deste método é $O(V^2)$ no pior caso, pois a função de busca em profundidade pode visitar todos os vértices e, a cada visita, chama o método adjs que tem complexidade $O(V)$. No melhor caso, a complexidade é $O(V)$ quando o grafo é desconexo e o vértice inicial não possui vizinhos, realizando apenas a chamada ao método adjs para o vértice inicial.
- **eh_conectado**: A complexidade deste método é $O(V^2)$ no pior caso, pois verifica a existência de arestas para cada vértice em $O(V^2)$ (devido ao método tem_alguma_aresta), realiza uma busca em profundidade em $O(V^2)$ (levando em conta a complexidade do método adjs) e, em seguida, verifica se todos os vértices foram visitados em $O(V^2)$ (devido ao método adjs). No melhor caso, a complexidade é $O(V^2)$ quando o grafo é desconexo e a verificação inicial encontra um vértice sem arestas, realizando apenas a chamada ao método tem_alguma_aresta para cada vértice.
- **__copy__**: A complexidade deste método é $O(V^2)$, pois realiza uma cópia profunda da matriz de adjacência VxV.

2.3 Matriz de incidência

Na matriz de incidência, a representação do grafo será por meio de uma matriz de listas contíguas, em que cada posição representa um vértice e cada lista dentro do vértice representa uma aresta do grafo. Neste caso, cada coluna da matriz corresponde a uma aresta, e cada linha corresponde a um vértice. Um valor 1 na posição (i, j) indica que o vértice i é incidente à aresta j. Essa representação é útil quando se deseja analisar as arestas do grafo (e.x: handshake), mas também pode ser ineficiente em termos de espaço para grafos com muitas arestas.

Na figura 6, temos a implementação para essa representação.

```python
class GrafoMtrInc(Grafo):

    def __init__(self, num_v=0):
        super().__init__(num_v=num_v)
        self.grafo: list[list[int]] = [[] for _ in range(num_v)]
        self.num_a = 0

    1 usage (1 dynamic)
    def add_vertice(self):...

    1 usage (1 dynamic)
    def add_aresta(self, u, v):...

    def arestas(self):...

    1 usage (1 dynamic)
    def remove_vertice(self, u):
        incident_edges = [i for i, value in enumerate(self.grafo[u]) if value > 0]
        for edge_index in reversed(incident_edges):
            for i in range(self.num_v):
                self.grafo[i].pop(edge_index)
            self.num_a -= 1
        self.grafo.pop(u)
        self.num_v -= 1

    def remove_aresta(self, u, v):
        edge_index = None
        for i in range(self.num_a):
            if self.grafo[u][i] > 0 and self.grafo[v][i] > 0:
                edge_index = i
                break
        if edge_index is not None:
            for i in range(self.num_v):
                self.grafo[i].pop(edge_index)
            self.num_a -= 1

    def adjs(self, u):
        adjacents = []
        for i, value in enumerate(self.grafo[u]):
            if value > 0:
                for j in range(self.num_v):
                    Analysis.COUNTER += 1
                    if j != u and self.grafo[j][i] > 0:
                        adjacents.append(j)
        return adjacents

    def ha_aresta(self, u, v):
        for i in range(self.num_a):
            if self.grafo[u][i] > 0 and self.grafo[v][i] > 0:
                return True
        return False

    def grau(self, u):
        return sum(self.grafo[u])

    def tem_alguma_aresta(self, u):
        for i in range(self.num_a):
            if self.grafo[u][i] > 0:
                return True
```

Fig. 6. Interface $GrafoBase$

- **remove_vertice**: A complexidade deste método é $O(V^2 \times A)$, para o pior caso, visto que, para cada aresta incidente ($V-1$, grafo completamente conectado), terá que percorrer todos os vértices (V), realizando a remoção da aresta incidente (A). Já no melhor caso, grafo sem arestas, não será necessário realizar nenhuma iteração, logo $O(1)$.
- **remove_aresta**: A complexidade deste método é $O(V \times A)$, já que é necessário percorrer todos os vértices e arestas do grafo para remover uma aresta. No melhor caso, $O(A)$, somente será necessário percorrer as arestas verificando se a aresta recebida existe.
- **adjs**: A complexidade deste método é $C(n) = (V-1) * V = O(V^2)$, pois, em um grafo completamente conectado, será necessário percorrer todas as arestas incidentes no vértice $(V-1)$ e, para cada uma destas, percorrer todos os vértices. O melhor caso é $O(1)$, no qual o vértice não incide em nenhum outro vértice.
- **ha_aresta**: A complexidade deste método é $O(A)$, no pior caso, pois é necessário percorrer todas as arestas do grafo para verificar se há uma aresta entre os vértices u e v. O melhor caso é quando a aresta procurada é a primeira do grafo, ou seja, quando o grafo é completo, assim a complexidade é $O(1)$.
- **grau**: A complexidade deste método é $O(A)$, já que é necessário percorrer todas as arestas do grafo para calcular o grau de um vértice.
- **tem_alguma_aresta**: A complexidade deste método é $O(A)$, no pior caso, já que é necessário percorrer todas as arestas do grafo para verificar se um vértice possui alguma aresta. No melhor caso, a complexidade é $O(1)$, caso a posição de uma aresta do vértice seja a primeira.
- **vertices**: A complexidade deste método é $O(V)$, já que é necessário percorrer todos os vértices do grafo.
- **bp**: A complexidade deste método no pior caso é $C(n) = (V-1) \times V \times (V/2) = O(V^3)$, uma vez que ele fará V chamadas recursivas para cada vértice e, em cada chamada, executará o método $adjs$ discutido previamente. O melhor caso é $O(1)$, já que ele não teria vértices adjacentes a serem verificados.
- **eh_conectado**: A complexidade deste método no pior caso é $O(V^3)$, tanto pela função bp, quanto pelo último loop que, a cada chamada, executa o método $adjs$ discutido previamente. O melhor caso é O(V), pois realiza ao menos o loop pelos vértices verificando se algum possui alguma aresta.
- **__copy__**: A complexidade deste método é $O(V \times A)$, já que é necessário copiar todos os elementos da matriz de incidência.

3 Análise da Complexidade do Algoritmo de Circuito Euleriano

O algoritmo de circuito euleriano é composto por duas funções principais: A função `eh_euleriano` e a `circuito_euleriano`. O algoritmo implementado é baseado no algoritmo de Hierholzer's Hierholzer's Algorithm, que é utilizado para encontrar um circuito euleriano. Na imagem 7, conseguimos ver a implementação destas funções.

O método `eh_euleriano` é responsável por verificar se o grafo é euleriano. Para isso, primeiramente verifica se o grafo é conectado utilizando o método `eh_conectado`. Caso o grafo não seja conectado, retorna `False`. Em seguida, o método conta o número de vértices com grau ímpar[2]. Se houver mais de dois vértices com grau ímpar, o método retorna `False`, indicando que o grafo não é euleriano. Caso contrário, retorna `True`, indicando que o grafo é euleriano.

A função `circuito_euleriano` inicia verificando se o grafo é euleriano utilizando o método `eh_euleriano`, em sequencia inicializa uma pilha vazia e insere o vértice inicial, que é o vértice 0 neste caso. Em seguida, entra em um loop enquanto a pilha não estiver vazia. Dentro do loop, o algoritmo verifica os vértices adjacentes ao vértice atual (o vértice no topo da pilha). Se houver algum vértice adjacente, o algoritmo seleciona um deles como vizinho, adiciona esse vizinho à pilha e remove a aresta entre o vértice atual e o

2. Se houver um vértice com grau ímpar ele é semi euleriano

```python
class GrafoEulerianoPathMixin(GrafoProrio):
    1 usage
    def eh_euleriano(self):
        if not self.eh_conectado():
            return False

        odd = 0

        for i in range(self.num_v):
            if self.grau(i) % 2 != 0:
                odd += 1

        if odd > 2:
            return False
        return True

    def circuito_euleriano(self):
        circuit = []

        if not self.eh_euleriano():
            return circuit

        stack: deque = deque()
        stack.append(0)
        count = 0

        while len(stack) > 0:
            count += 1

            current_vertex = stack[-1]

            adjs = self.adjs(current_vertex)

            if len(adjs) > 0:  # Grau do vértice é maior que 0
                neighbor = adjs[0]
                stack.append(neighbor)
                self.remove_aresta(current_vertex, neighbor)
            else:
                circuit.append(stack.pop())

        circuit.reverse()
        return circuit
```

Fig. 7. Algorítimo para achar o circuito euleriano

vizinho. Caso contrário, se não houver vértices adjacentes, o algoritmo adiciona o vértice atual ao circuito e o remove da pilha. Ao final do loop, o algoritmo inverte a ordem dos vértices no circuito e o retorna como resultado.

A seguir, analisaremos a complexidade do algoritmo em relação às três estruturas mencionadas anteriormente.

Matriz de Incidência: No método `eh_euleriano`, a verificação de conectividade pelo `eh_conectado` tem complexidade $\mathcal{O}(V^3)$ no pior caso e $\mathcal{O}(V)$ no melhor. A análise dos graus dos vértices tem complexidade $\mathcal{O}(V * A)$. Logo, a complexidade final é $\mathcal{O}(V^3)$ no pior caso e $\mathcal{O}(V * A)$ no melhor. No método `circuito_euleriano`, a complexidade no pior caso será $\mathcal{O}(A^2 * V^2)$ e no melhor $\mathcal{O}(A^2)$, considerando a execução dos métodos `adjs` e `remove_aresta`.

Matriz de Adjacências: No método `eh_euleriano`, a verificação de conectividade tem complexidade $\mathcal{O}(V^2)$. A análise dos graus possui complexidade $\mathcal{O}(V)$. Assim, a complexidade final é $\mathcal{O}(V^2)$. No método `circuito_euleriano`, utiliza o método `eh_euleriano` e, em seguida, percorre todas as arestas do grafo. A complexidade desse loop será $C(V, A) = A * V * 1 = O(A * V)$, considerando que a cada iteração são executados os métodos `adjs` e `remove_aresta`. Assim a ordem de complexidade final do algorítimo é $\mathcal{O}(A * V)$.

Lista de Adjacências: No método `eh_euleriano`, a verificação de conectividade tem complexidade $\mathcal{O}(V)$ e a análise dos graus $\mathcal{O}(1)$. Logo, a complexidade final é $\mathcal{O}(V)$. No método `circuito_euleriano`, considerando a execução dos métodos `adjs` e `remove_aresta`, a complexidade final é $\mathcal{O}(V * A)$.

A lista de adjacências é a estrutura que apresenta a menor complexidade, seguida pela matriz de adjacências e, por último, a matriz de incidência. Nós gráficos 8, 9, 10, temos o plot da contagem da operação básica pelo tamanho da entrada, neles conseguimos notar essa diferença mais atenuada.

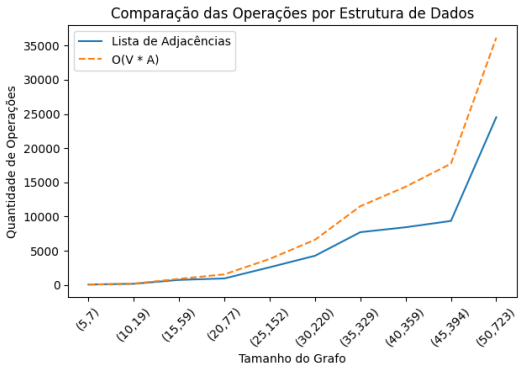

Fig. 8. Plot lista encadeada

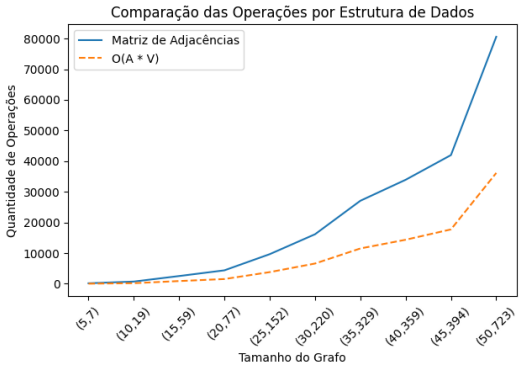

Fig. 9. Plot matriz de ajacencia

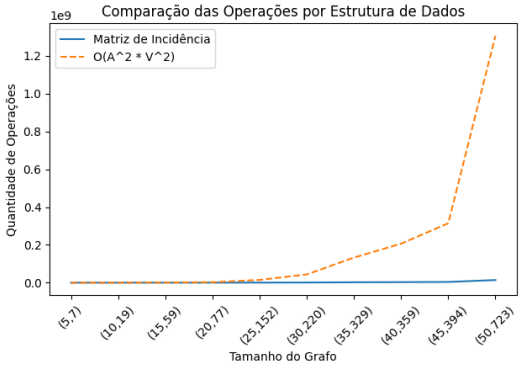

Fig. 10. Plot matriz de incidencia

4 Análise da Complexidade do Algoritmo de Grafo Hamiltoniano

O algoritmo de grafo hamiltoniano é composto principalmente pela função `eh_hamiltoniano`. Esta função verifica se a condição necessária para um grafo ser Hamiltoniano é satisfeita. A implementação do algoritmo é mostrada no código 12.

```python
class GrafoHamiltonianoMixin(GrafoProrio):
    2 usages
    @staticmethod
    def todos_subconjuntos(s):
        """Gera todos os subconjuntos próprios não vazios de S...."""
        return list(chain.from_iterable(combinations(s, r) for r in range(1, len(s))))

    1 usage
    def num_componentes_conexas_apos_remocao(self, vertices_to_remove):
        """Calcula o número de componentes conexas após a remoção de um conjunto de vértices...."""
        H: GrafoProrio = self.__copy__(self)

        # H = G - S
        count = 0
        for v in vertices_to_remove:
            i = v - count
            H.remove_vertice(i)
            count += 1

        visited = [False] * H.num_v
        num_components = 0

        # ω(H)
        for v in range(H.num_v):
            if not visited[v]:
                num_components += 1
                H.bp(v, visited)

        return num_components

    def eh_hamiltoniano(self):
        """Verifica se a condição necessária para um grafo ser Hamiltoniano é satisfeita...."""

        V = set(self.vertices())  # qualquer subconjunto próprio não vazio S ⊂ V

        for S in self.todos_subconjuntos(V):
            if self.num_componentes_conexas_apos_remocao(S) > len(S):  # vale a relação ω(G - S) ≤ |S|
                return False

        return True
```

Fig. 11. Algorítimo para achar o circuito euleriano

A seguir, analisaremos a complexidade do algoritmo e de seus métodos auxiliares em relação à lista de adjacências, dados que está e a que apresenta melhores ordens complexidade em geral.

Lista de Adjacências: A função `eh_hamiltoniano` é composta por três partes principais: geração de todos os subconjuntos próprios não vazios de vértices (S), cálculo do número de componentes conexas após a remoção de um conjunto de vértices ($\omega(G-S)$), e verificação se $\omega(G-S) \leq |S|$.

1. A geração de todos os subconjuntos próprios não vazios de vértices é realizada pelo método `todos_subconjuntos`. Este método tem complexidade $\mathcal{O}(2^V)$, onde V é o número de vértices. Isso ocorre porque há $2^V - 2$ subconjuntos próprios não vazios. A complexidade do método `todos_subconjuntos` é dada por:

$$\mathcal{O}(2^V) \tag{1}$$

2. O cálculo do número de componentes conexas após a remoção de um conjunto de vértices é realizado pelo método `num_componentes_conexas_apos_remocao`. Este método envolve a cópia do grafo e a remoção de vértices. A cópia do grafo tem complexidade $\mathcal{O}(V + A)$. A remoção de vértices tem complexidade $\mathcal{O}(V * S)$, onde S é o tamanho do conjunto de vértices a serem removidos. A busca em profundidade (bp) tem complexidade $\mathcal{O}(V)$. Portanto, a complexidade total deste método é:

$$\mathcal{O}((V + A) + V * S + V) = \mathcal{O}(V * S + 2V + A)) \tag{2}$$

3. A verificação se $\omega(G - S) \leq |S|$ é realizada em tempo constante, ou seja, $\mathcal{O}(1)$.

A complexidade final do algoritmo `eh_hamiltoniano` é dominada pela geração de todos os subconjuntos próprios não vazios de vértices e pelo cálculo do número de componentes conexas após a remoção de um conjunto de vértices. Portanto, a complexidade final é:

$$\mathcal{O}(2^V * (V * S + 2V + A)) = \mathcal{O}(2^V * (V * S + 2 * V + A)) \tag{3}$$

Abaixo segue um gráfico para demostração do desempenho do algorítimo, nele temos a contagem de operações básicas no eixo y e tamanho da entrada no eixo x. 12

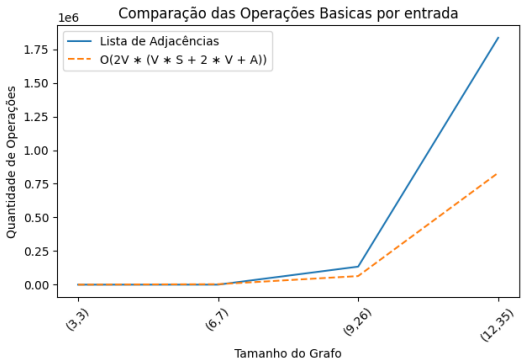

Fig. 12. Gráfico hamiltoniano