Grafos parte 1 - Complexidade de Algoritmos

1 - Implementação Implementar uma classe representando um grafo e suas operações.

implement-grafo.py > ...

```python
class Graph:
    def __init__(self):
        self.graph = {}

    def add_vertex(self, vertex):
        self.graph[vertex] = []

    def add_edge(self, vertex1, vertex2):
        self.graph[vertex1].append(vertex2)
        self.graph[vertex2].append(vertex1)

    def get_neighbors(self, vertex):
        return self.graph.get(vertex, [])

    def __str__(self):
        return str(self.graph)

# Exemplo de uso
graph = Graph()
graph.add_vertex('A')
graph.add_vertex('B')
graph.add_vertex('C')
graph.add_edge('A', 'B')
graph.add_edge('B', 'C')

print(graph.get_neighbors('A'))  # Saída: ['B']
print(graph.get_neighbors('B'))  # Saída: ['A', 'C']
print(graph)  # Saída: {'A': ['B'], 'B': ['A', 'C'], 'C': ['B']}


# esta é uma implementação básica de um grafo não direcionado
# onde cada vértice é representado por um nó e as arestas são
# representadas pelas conexões entre nós.

# A função 'add_vertex' adiciona um vértice ap grafo
# 'add_edge adiciona uma aresta entre dois vértices
# 'get_neighbors' retorna os vizinhos de uma vértice
# A função '__str__' é usada para imprimir o grafo
```

2- Um grafo conectado é um grafo onde todos os vértices estão conectados. Dado um grafo G, como é possível verificar se um grafo é conectado ou não?

Para verificar se um grafo é conectado ou não, você pode usar uma busca em profundidade (DFS) ou uma busca em largura (BFS) a partir de um vértice arbitrário, e então verificar se todos os vértices foram visitados durante a travessia.

Aqui está um exemplo de como você poderia implementar essa verificação usando busca em profundidade (DFS) em python:

implent-dfs.py > ...

```python
class Graph:
    def __init__(self):
        self.graph = {}

    def add_vertex(self, vertex):
        self.graph[vertex] = []

    def add_edge(self, vertex1, vertex2):
        self.graph[vertex1].append(vertex2)
        self.graph[vertex2].append(vertex1)

    def dfs(self, start_vertex, visited):
        visited.add(start_vertex)
        for neighbor in self.graph[start_vertex]:
            if neighbor not in visited:
                self.dfs(neighbor, visited)

    def is_connected(self):
        start_vertex = list(self.graph.keys())[0]  # Escolhe um vértice arbitrário como ponto de partida
        visited = set()
        self.dfs(start_vertex, visited)

        # Verifica se todos os vértices foram visitados
        return len(visited) == len(self.graph)

# Exemplo de uso
graph = Graph()
graph.add_vertex('A')
graph.add_vertex('B')
graph.add_vertex('C')
graph.add_edge('A', 'B')
graph.add_edge('B', 'C')

print(graph.is_connected())  # Saída: True

# Neste exemplo, a função 'is_connect' usa a busca de profundidade (DFS) para visitar todos os vértices do grafo a partir de um vértice inicial, e então verifica se todos ps vértices foram visitados. Se todos os vértices foram visitados, o grafo é considerado conectado
```

3. Um grafo acíclico é um grafo onde para qualquer caminho $\{v_1, v_2, \ldots, v_k\}$, $v_i \neq v_j$ onde $i \neq j$ e $i \in \{1, \ldots, k\}, j \in \{1, \ldots, k\}$. Como é possível encontrar um ciclo em um grafo?

Para encontrar um ciclo de um grafo, você pode usar a busca em profundidade (DFS) e acompanhar se você retornar a um vértice já visitado durante a travessia. Se você retornar a um vértice visitado (diferente do vértice anterior), então há um ciclo no grafo.

Aqui está um exemplo de como você pode modificar a classe ‘Graph’ para verificar a presença de um ciclo usando a busca em profundidade:

```python
class Graph:
    def __init__(self):
        self.graph = {}

    def add_vertex(self, vertex):
        self.graph[vertex] = []

    def add_edge(self, vertex1, vertex2):
        self.graph[vertex1].append(vertex2)

    def has_cycle(self):
        visited = set()
        stack = set()

        def dfs(node):
            visited.add(node)
            stack.add(node)

            for neighbor in self.graph[node]:
                if neighbor not in visited:
                    if dfs(neighbor):
                        return True
                elif neighbor in stack:
                    return True

            stack.remove(node)
            return False

        for vertex in self.graph:
            if vertex not in visited:
                if dfs(vertex):
                    return True

        return False

# Exemplo de uso
graph = Graph()
graph.add_vertex('A')
graph.add_vertex('B')
graph.add_vertex('C')
graph.add_edge('A', 'B')
graph.add_edge('B', 'C')
graph.add_edge('C', 'A')

print(graph.has_cycle())  # Saída: True

# Neste exemplo, a função 'has_cycle'
# utiliza a busca em profundidade modificada para rastrear se algum vértice
# é visitado mais de uma vez durante a travessia, indicando a presença de um cliclo
```

Aluna: Luana Roza de Oliveira