UNIVERSIDAD AMERICANA

Algoritmos y Estructuras de Datos

Documentación

Estudiantes:

- Diego Armando Urbina Aviles
- Julio Cesar Delgadillo Pineda
- Emmanuel Leonardo Aguilar Novoa

Docente:

Silvia Gigdalia Ticay Lopez

```python
class Grafo:
    def __init__(self, es_dirigido=False):
        self.grafo = {}
        self.es_dirigido = es_dirigido

    def agregar_vertice(self, vertice):
        if vertice not in self.grafo:
            self.grafo[vertice] = []

    def agregar_arista(self, u, v, peso=1):
        if u not in self.grafo:
            self.agregar_vertice(u)
        if v not in self.grafo:
            self.agregar_vertice(v)

        if not self.es_dirigido:
            self.grafo[u].append((v, peso))
            self.grafo[v].append((u, peso))
        else:
            self.grafo[u].append((v, peso))

    def obtener_vecinos(self, vertice):
        if vertice not in self.grafo:
            raise ValueError(f"El vértice {vertice} no existe.")
        return [v[0] for v in self.grafo[vertice]]

    def existe_arista(self, u, v):
        if u not in self.grafo or v not in self.grafo:
            return False
        for vecino, _ in self.grafo[u]:
            if vecino == v:
                return True
        return False
```

La clase Grafo se utiliza para representar un grafo. El constructor __init__ inicializa un diccionario vacío llamado grafo, donde las claves son los vértices y los valores son listas de sus vértices vecinos. El parámetro es_dirigido especifica si el grafo es dirigido o no. Si es dirigido, las aristas solo se agregarán en una dirección, mientras que en un grafo no dirigido se agregarán en ambas direcciones.

Método **agregar_vertice(self, vértice)**: Este método agrega un vértice al grafo si no existe previamente.

Método **agregar_arista(self, u, v, peso=1)**: Agrega una arista entre los vértices u y v. Si el grafo es no dirigido, se agrega la arista en ambas direcciones; si es dirigido, solo se agrega de u a v. El peso de la arista es opcional, con un valor predeterminado de 1.

Método obtener_vecinos(self, vertice): Devuelve los vértices adyacentes a un vértice dado. Si el vértice no existe, se lanza un error.

Método existe_arista(self, u, v): Verifica si existe una arista entre los vértices u y v. Retorna True si existe, y False en caso contrario.

```python
from collections import deque
from grafo import Grafo

def bfs(grafo, inicio):
    visitados = set()
    cola = deque([inicio])
    recorrido = []

    while cola:
        vertice = cola.popleft()
        if vertice not in visitados:
            visitados.add(vertice)
            recorrido.append(vertice)
            for vecino, _ in grafo.grafo[vertice]:
                if vecino not in visitados:
                    cola.append(vecino)
    return recorrido
```

El método bfs implementa el algoritmo de búsqueda en amplitud (BFS). Comienza en el vértice inicio, marca los vértices como visitados a medida que se procesan, y los agrega a una cola para ser explorados. Los vértices adyacentes a un vértice procesado se agregan a la cola si aún no han sido visitados. Este algoritmo recorre el grafo en "niveles", procesando todos los vértices a una distancia determinada antes de pasar a los vértices a mayor distancia.

visitados: Un conjunto que mantiene un registro de los vértices visitados.

cola: Una cola utilizada para almacenar los vértices a procesar, implementada con deque para eficiencia en las operaciones de inserción y eliminación.

recorrido: Una lista que almacena los vértices visitados en el orden en que fueron procesados.

```
def dfs(grafo, inicio):
    visitados = set()
    recorrido = []
    _dfs_recursivo(grafo, inicio, visitados, recorrido)
    return recorrido

def _dfs_recursivo(grafo, vertice, visitados, recorrido):
    visitados.add(vertice)
    recorrido.append(vertice)
    for vecino, _ in grafo.grafo[vertice]:
        if vecino not in visitados:
            _dfs_recursivo(grafo, vecino, visitados, recorrido)
```

El método dfs realiza una búsqueda en profundidad (DFS). Comienza desde el vértice inicio y explora completamente cada vértice antes de retroceder. Utiliza recursión para visitar los vértices adyacentes. Los vértices visitados se agregan a un conjunto visitados para evitar procesar un vértice más de una vez. A medida que se visitan los vértices, se agregan a la lista recorrido.

_dfs_recursivo: Es el método recursivo que realiza la búsqueda en profundidad. Para cada vértice, se exploran todos sus vecinos no visitados.

visitados: Un conjunto que rastrea los vértices que ya han sido procesados.

recorrido: Una lista que guarda el orden de los vértices visitados.

```python
def es_conexo(grafo):
    """Verifica si el grafo no dirigido es conexo"""
    if not grafo.grafo:
        return True  # Un grafo vacío es considerado conexo
    visitados = set()
    dfs_recursivo(grafo, next(iter(grafo.grafo)), visitados, [])
    return len(visitados) == len(grafo.grafo)

def encontrar_camino(grafo, inicio, fin):
    """Encuentra un camino entre inicio y fin"""
    visitados = set()
    padres = {inicio: None}
    cola = deque([inicio])

    while cola:
        vertice = cola.popleft()
        if vertice == fin:
            camino = []
            while vertice is not None:
                camino.insert(0, vertice)
                vertice = padres[vertice]
            return camino
        for vecino, _ in grafo.grafo[vertice]:
            if vecino not in visitados:
                visitados.add(vecino)
                padres[vecino] = vertice
                cola.append(vecino)
    return []  # Si no existe camino
```

es_conexo: Este método verifica si el grafo es conexo. Para ello, realiza una búsqueda en profundidad (DFS) desde un vértice arbitrario y verifica si todos los vértices fueron visitados. Si el número de vértices visitados es igual al número total de vértices en el grafo, el grafo es conexo.

encontrar_camino: Este método utiliza BFS para encontrar un camino entre los vértices inicio y fin. Si el camino existe, reconstruye y devuelve el camino a partir de los "padres" de cada vértice, que se almacenan mientras se recorre el grafo. Si no se encuentra un camino, devuelve una lista vacía.