Universidad Americana

Algoritmos y Estructuras de Datos

Documentación de ejercicios sobre Grafos

Integrantes:

Joshua Ordoñez

Michael Casco

Diego Rodriguez

Daniel Gutierrez

Docente:

Silvia Gigdalia Ticay López

Fecha:

13 de junio del 2025

Clase Grafo

Esta clase representa un grafo usando una lista de adyacencia. Puede ser dirigido o no dirigido, dependiendo del parámetro con el que se construya.

__init__(self, es_dirigido=False)

```
def __init__(self, es_dirigido= False):
    self.es_dirigido = es_dirigido
    self.adyacencia = {}
```

Constructor del grafo.

- Parámetro:
  es_dirigido (bool): Si el grafo es dirigido (True) o no dirigido (False). Por defecto es no dirigido.

- Atributos:

  - self.es_dirigido: Guarda si el grafo es dirigido.

  - self.adyacencia: Diccionario que representa la lista de adyacencia del grafo.

agregar_vertice(self, vértice)

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

Agrega un vértice al grafo.

- Parámetro:
  vértice: El nodo a agregar.

- Funcionamiento:
  Si el vértice no existe en el diccionario de adyacencia, se crea con una lista vacía.

agregar_arista(self, u, v, peso=1)

```
def agregar_arista(self, u, v, peso=1):
    self.agregar_vertice(u)
    self.agregar_vertice(v)
    self.adyacencia[u].append(v)
    if not self.es_dirigido:
        self.adyacencia[v].append(u)
```

Agrega una arista entre dos vértices.

- Parámetros:
  u: Vértice de origen.
  v: Vértice de destino.
  Peso: Peso de la arista.

- Funcionamiento:
  Inserta ambos vértices llamando a agregar_vertice y luego:
  - Agrega v a la lista de vecinos de u.
  - Si el grafo no es dirigido, también agrega u como vecino de v.

obtener_vecinos(self, vertice)

```
def obtener_vecinos(self, vertice):
    if vertice in self.adyacencia:
        return self.adyacencia[vertice]
    else:
        return []
```

Obtiene la lista de vecinos de un vértice.

- Parámetro:
  vértice: Vértice del cual se quieren los vecinos.

- Retorno:
  Lista de vecinos si existe el vértice, si no, una lista vacía.

- Funcionamiento:
  Si el vértice existe dentro del diccionario “adyacencia” retorna todos los valores ligados a la clave la cual es vértice.

Existe_arista(self, u, v)

```python
def existe_arista(self, u, v):
    if u in self.adyacencia:
        return v in self.adyacencia[u]
    return False
```

Verifica si existe una arista de u a v.

- Parámetros:
  u, v: Vértices a verificar.

- Retorno:
  True si v está en la lista de adyacencia de u, False en caso contrario.

bfs(self, inicio)

```python
def bfs(self, inicio):
    if inicio not in self.adyacencia:
        return []

    visitados = []
    cola = [inicio]
    orden_visita = []

    while cola:

        actual = cola.pop(0)
        if actual not in visitados:
            visitados.append(actual)
            orden_visita.append(actual)
            for vecino in self.obtener_vecinos(actual):
                if vecino not in visitados and vecino not in cola:
                    cola.append(vecino)

    return orden_visita
```

Recorrido en anchura (BFS).

- Parámetro:
  inicio: Vértice desde donde inicia la búsqueda.

- Retorno:
  Lista del orden en que se visitan los nodos.

- Funcionamiento:
  Usa lista de visitados para guardar los vértices ya visitados así como una cola la cual se inicializa con el vértice donde se desea iniciar. Luego se extrae el primer

elemento de la cola, si este se no se encuentra en visitados, es decir si aun no lo hemos recorrido añadirá el vértice en la lista de visitados, así como en la lista de “orden_visita”, esta última nos servirá para saber el orden en el que hemos recorrido el Grafo. Luego entramos a un bucle for, en el cual, se recorre los vecinos del vértice actual y los guardamos en la variable vecino. Si el vecino no se encuentra en la lista de vistos ni en la lista cola lo añadiremos en la cola.

dfs(self, inicio)

```python
def dfs(self, inicio):
    visitados = []
    orden_visita = []

    def _dfs(vertice):
        if vertice not in visitados:
            visitados.append(vertice)
            orden_visita.append(vertice)
            for vecino in self.obtener_vecinos(vertice):
                _dfs(vecino)

    if inicio in self.adyacencia:
        _dfs(inicio)

    return orden_visita
```

Recorrido en profundidad (DFS).

- Parámetro:
  inicio: Vértice desde donde inicia la búsqueda.

- Retorno:
  Lista del orden de visita en DFS.

- Funcionamiento:
  Usa una lista llamada visitados para ir guardando los vértices que ya han sido recorridos. La función comienza desde un vértice inicial y utiliza una función auxiliar interna llamada _dfs que se encarga de hacer el recorrido de forma recursiva. Esta función primero verifica si el vértice actual ya ha sido visitado; si no lo ha sido, lo añade a la lista visitados y también a la lista orden_visita, que se usa para registrar el orden en que se visitan los vértices. Luego entra en un bucle for donde recorre todos los vecinos del vértice actual. Por cada vecino, si este no ha sido visitado aún, se llama recursivamente a la función _dfs para continuar el recorrido desde ese vecino. De esta manera, el algoritmo profundiza lo más posible antes de retroceder, siguiendo el método de búsqueda en profundidad.

es_conexo(self)

```python
def es_conexo(self):
    if not self.adyacencia:
        return True

    incio = next(iter(self.adyacencia))
    visitados = self.bfs(incio)
    return len(visitados) == len(self.adyacencia)
```

Verifica si el grafo es conexo.

- Retorno:
  True si todos los vértices son alcanzables desde un nodo cualquiera, False si no.

- Funcionamiento:
  Usa bfs para recorrer desde un nodo arbitrario. Si el número de nodos visitados es igual al total de vértices, es conexo.

encontrar_camino(self, inicio, fin)

```python
def encontrar_camino(self, inicio, fin):
    if inicio not in self.adyacencia or fin not in self.adyacencia:
        return []


    padres = {inicio: None}
    cola =  [inicio]

    while cola:
        actual = cola.pop(0)
        if actual == fin:
            break
        for vecino in self.obtener_vecinos(actual):
            if vecino not in padres:
                padres[vecino] = actual
                cola.append(vecino)

    if fin not in padres:
        return []

    camino = []
    actual = fin
    while actual is not None:
        camino.insert(0, actual)
        actual = padres[actual]

    return camino
```

Encuentra un camino entre dos nodos usando BFS.

- Parámetros:
  inicio: Nodo inicial.
  fin: Nodo final.

- Retorno:
  Lista con el camino desde inicio hasta fin, o lista vacía si no hay camino.

- Funcionamiento:
  Esta función busca encontrar un camino entre dos vértices dados, inicio y fin, utilizando un recorrido en anchura (BFS). Primero verifica si ambos vértices existen en el grafo; si no es así, retorna una lista vacía. Luego se inicializa una cola con el nodo de inicio y un diccionario llamado padres, que guarda el nodo del cual proviene cada vértice. A continuación, se realiza el recorrido: se extrae un nodo de la cola y, si es el nodo destino (fin), se rompe el bucle porque ya se encontró el camino. Si no, se recorren sus vecinos; si un vecino no está en padres, se guarda como hijo del nodo actual y se agrega a la cola. Una vez terminado el recorrido, se verifica si el nodo fin fue alcanzado. Si no está en padres, se retorna una lista vacía indicando que no hay camino. Si fue alcanzado, se reconstruye el camino desde fin hacía inicio usando el diccionario de padres, insertando cada vértice al inicio de la lista camino, hasta volver al origen. Finalmente, se retorna la lista completa con el camino encontrado.

Implementación

El siguiente código realiza la creación y manipulación de un grafo no dirigido, utilizando una clase llamada Grafo que es importada desde un módulo externo llamado módulos. A través del código se demuestra cómo agregar vértices, establecer aristas entre ellos, incluir un vértice desconectado y realizar pruebas para verificar la conectividad del grafo y la existencia de caminos entre vértices.

1. Importación del módulo

```
from modulos import Grafo
```

Se importa la clase Grafo desde un archivo o módulo llamado módulos. Se asume que esta clase contiene los métodos necesarios para crear y operar sobre grafos, tales como agregar vértices, agregar aristas, comprobar la conectividad, y buscar caminos.

2. Creación del grafo

Se instala un grafo no dirigido, es decir, las conexiones (aristas) entre los vértices no tienen dirección: si hay una arista entre A y B, también se considera que existe entre B y A.

3. Agregado de vértices

```
for v in ['A', 'B', 'C', 'D', 'E']:
    grafo_nd.agregar_vertice(v)
```

Se agregan al grafo cinco vértices identificados por letras: 'A', 'B', 'C', 'D' y 'E'.

---

4. Agregado de aristas

```
aristas = [('A', 'B'), ('A', 'C'), ('B', 'D'), ('C', 'D'), ('D', 'E')]
for u, v in aristas:
    grafo_nd.agregar_arista(u, v)
```

Se agregan conexiones entre los vértices, lo que define la estructura del grafo. Por ejemplo, 'A' está conectado con 'B' y 'C', 'D' está conectado con 'B', 'C' y 'E'.

---

5. Agregado de vértice desconectado

```
grafo_nd.agregar_vertice('F')
```

Se añade un vértice 'F' que no tiene ninguna arista, es decir, no está conectado con ningún otro nodo. Esto sirve para probar la conectividad general del grafo.

---

6. Pruebas de funcionalidad

```
print("¿El grafo es conexo?:", grafo_nd.es_conexo())
```

Este método comprueba si el grafo es conexo, es decir, si todos los vértices están conectados directa o indirectamente entre sí. Dado que 'F' está aislado, el grafo no será conexo.

```
print("Camino de 'A' a 'E':", grafo_nd.encontrar_camino('A', 'E'))
```

Busca un camino entre los vértices 'A' y 'E'. En este caso, sí existe un camino (por ejemplo, A → C → D → E).

```
print("Camino de 'A' a 'Z':", grafo_nd.encontrar_camino('A', 'Z'))
```

Busca un camino entre 'A' y 'Z'. Como 'Z' no ha sido agregado al grafo, no se encontrará ningún camino.