Proyecto Grafo - Implementación con Listas de Adyacencias

Julio Pérez

Abril - Julio 2024

1 Introducción

Sea $G = (V, E)$ un grafo dirigido, una posible forma de representarlo es usando listas de adyacencias. En esta implementación se tiene una lista de diccionarios donde las claves son los vértices y los valores son listas de los vértices adyacentes. La mayor ventaja de esta implementación es su flexibilidad, ya que permite agregar vértices y arcos de forma sencilla.

Para este proyecto, se desea que los estudiantes hagan una implementación en Python de la estructura de datos Grafo Dirigido usando listas de adyacencias como una clase. A continuación se detallan los métodos públicos que la clase.

2 Métodos Públicos

- `def add_vertex(self, vertex):`
  Agrega un vértice al grafo. Retorna `True` si el vértice es agregado con éxito. Retorna `False` si el vértice ya existe en el grafo.

- `def add_edge(self, from_vertex, to_vertex):`
  Conecta dos vértices con un arco dirigido. Retorna `True` si el arco es agregado con éxito. Retorna `False` si uno o ambos vértices no existen o el arco ya existe.

- `def remove_edge(self, from_vertex, to_vertex):`
  Desconecta dos vértices removiendo el arco dirigido. Retorna `True` si el arco es eliminado con éxito. Retorna `False` si uno o ambos vértices no existen o el arco no existe.

- `def contains_vertex(self, vertex):`
  Verifica si un vértice está presente en el grafo.

- `def get_inward_edges(self, vertex):`
  Retorna la lista de predecesores de un vértice. Es decir, retorna la lista de todos los vértices u tales que $(u, \texttt{vertex}) \in E$. Si el vértice no existe en el grafo, retorna `None`.

- `def get_outward_edges(self, vertex):`
  Retorna la lista de sucesores de un vértice. Es decir, retorna la lista de todos los vértices u tales que $(\texttt{vertex}, u) \in E$. Si el vértice no existe en el grafo, retorna `None`.

- `def get_vertices_connected_to(self, vertex):`
  Retorna la lista de vértices adyacentes a un vértice. Es decir, retorna la lista de todos los vértices u tales que $(\texttt{vertex}, u) \in E$ o $(u, \texttt{vertex}) \in E$. Si el vértice no existe en el grafo, retorna `None`.

- `def get_all_vertices(self):`
  Retorna la lista de todos los vértices del grafo

- `def remove_vertex(self, vertex):`
  Elimina un vértice del grafo. Retorna `True` si el vértice es eliminado con éxito. Retorna `False` si el vértice no existe en el grafo.

- `def size(self):`
  Retorna la cantidad de vértices que contiene el grafo.

- `def subgraph(self, vertices):`
  Retorna otra instancia de grafo donde el conjunto de vértices contiene solo aquellos vértices(V') presentes en la colección dada (`vertices`) y solo aquellos arcos asociados a esos vértices. Es decir, retorna $G' = (V', E')$ donde $E' = \{(u, v) \in E \mid u \in V' \land v \in V'\}$.

- `def complement(self):`
  Retorna el grafo complemento. Es decir, un grafo con los mismos vértices pero con todos los arcos que no existen en el grafo original. $E_{\text{comp}} = \{(u, v) \mid u, v \in V \land u \neq v : (u, v) \notin E\}$.

- `def copy(self):`
  Retorna una copia del grafo (`deepcopy(self)`)

Nota: Debe funcionar con cualquier objeto como vértice. Pueden asumir que __eq__ y __hash__ están correctamente implementado para cualquier objeto usado.

3 Entrega

Este proyecto se realizará en equipos de 2 ó 3 integrantes y debe ser subido a GitHub. Su repositorio debe contener la clase `AdjacencyListGraph.py` con la implementación, y un archivo `README.md` identificado con el nombre y número de carnet de los integrantes y una breve explicación de su implementación junto con una estimación de la complejidad computacional de cada método (Big O notation).

La fecha límite de entrega es el miércoles 11 de junio de 2024 a las 11:59 pm.