5. Práctica grafos

TUIA - Programación 2

2025 - 2C

Grafos

Práctica 5

Representación de Grafos

Una forma posible de implementar el TAD Grafo (no es la única) es guardar una lista de los vértices y un diccionario que especifique, para vértice, una lista de sus vecinos. A continuación mostramos una implementación incompleta y un ejemplo de uso:

```python
from typing import Any

class Grafo:
  def __init__(self) -> None:
    self.vertices = []
    self.vecinos = {}

  def add_node(self, vertice: Any) -> None:
    # O(1)
    self.vertices.append(vertice)
    self.vecinos[vertice] = []

  def add_edge(self, vertice1: Any, vertice2: Any) -> None:
    # O(1)
    self.vecinos[vertice1].append(vertice2)
    self.vecinos[vertice2].append(vertice1)

  def get_adjacent(self, vertice: Any) -> list[Any]:
    # O(1)
    return self.vecinos[vertice]

  def get_nodes(self) -> list[Any]:
    # O(1)
    return self.vertices


# Ejemplo de uso
grafo = Grafo()
```

```python
grafo.add_node("A")
grafo.add_node("B")
grafo.add_node("C")

grafo.add_edge("A", "B")
grafo.add_edge("B", "C")
grafo.add_edge("C", "A")

print("Vértices:", grafo.get_nodes())
print("Vecinos de A:", grafo.get_adjacent("A"))
print("Vecinos de B:", grafo.get_adjacent("B"))
print("Vecinos de C:", grafo.get_adjacent("C"))
```

Ejercicio 1

Completar la implementación agregando los siguientes métodos:

- `remove_node(x)`: Remueve el nodo `x` (si existe) y todas sus aristas adyacentes.
- `remove_edge(x, y)`: Remueve la arista entre el nodo `x` y el nodo `y` (si existe).
- `are_adjacent(x, y)`: Devuelve `True` si `x` e `y` son adyacentes, `False` en caso contrario.
- `is_node(x)`: Devuelve `True` si `x` es un nodo del grafo, `False` en caso contrario.

Estime la complejidad temporal de cada una de las operaciones en función de la cantidad de vértices del grafo.

Ejercicio 2

Escriba una función `get_edges(G)` que reciba un grafo y devuelva una lista de las aristas del grafo. Tenga cuidado de no repetir aristas.

Ejercicio 3

Escriba una función `is_subgraph(G, H)` que decida si `H` es subgrafo de `G`.

Ejercicio 4

Escriba una función `induce(G, U)` que recibe un grafo `G` y una lista de vértices `U` y devuelva el grafo inducido en `G` por el conjunto `U`.

Ejercicio 5

Implemente el método `__eq__` para grafos para permitir comparar por igualdad. Dos grafos son iguales si tienen el mismo conjunto de vértices y la misma colección de aristas.

Ejercicio 6

Escriba una función `is_induced_subgraph(G, H)` que decida si `H` es subgrafo inducido de `G`, para algún conjunto de vértices.

Ejercicio 7

Escriba una función `is_complete(G)` que decida si `G` es el grafo completo.

Ejercicio 8

Dado un grafo `G=(V,E)`, un *clique* es un subconjunto de vértices $C \subseteq E$ tal que todos los vértices de `C` son adyacentes entre sí. En otras palabras, un *clique* es un subgrafo en el que cada vértice está conectado a todos los demás vértices del subgrafo. Esto equivale a decir que el subgrafo de `G` inducido por `C` es un grafo completo.

El tamaño de un *clique* es el número de vértices que contiene.

Dar una función `has_clique(G, k)` que decida si un grafo `G` tiene un clique de al menos `k` elementos.

Ayuda: Defina primero una función `subsets_of_size_k` que, dada una lista y un entero positivo `k`, devuelva una lista con todas las posibles listas de tamaño `k`.

Ejercicio 9

El complemento de un grafo `G=(V,E)` es un grafo `G'=(V,E')` que contiene exactamente los mismos vértices y los vértices `v` y `w` estan conectados si y solo si no lo están en `V`.

Defina una función `complement(G)` que dado un grafo `G`, devuelva el grafo complementario a `G`. La función debe ser pura, es decir, no debe modificar el grafo original.