Universidad Nacional de San Martín

Licenciatura en Ciencia de Datos

Algoritmos II

Formas de representación

El diseño de nuevos tipos abstractos de datos (TAD) basados en estructuras de grafos son generalmente **definidos para cada problema puntual**. **No** es frecuente encontrar un **TAD Grafo(a) paramétrico** que sirva de base para instanciar con otro TAD.

Cada una de ellas tendrá sus ventajas y

desventajas y la selección de cuál utilizar dependerá de:

- el problema a resolver
- los recursos físicos disponibles
- qué operaciones predominarán para consumir la estructura en el programa.

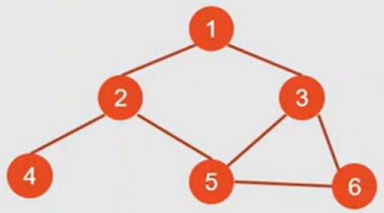
Formas de representación

- Conjuntos de nodos y aristas

V = {1, 2, 3, 4, 5, 6}

E = { (1,2), (1,3), (2,4), (2,5), (3,5), (3,6), (5,6) }   (enlace simple)

E = { (1,2), (2,1), (1,3), (3,1), (2,4), (4,2), ... }   (enlace doble)



Formas de representación

- Lista de adyacencia

$V = \{1, 2, 3, 4, 5, 6\}$

Nodos		Adyacentes
1	→	[2,3]
2	→	[1,4,5]
3	→	[1,5,6]
4	→	[2]
5	→	[2,3,6]
6	→	[3,5]

Formas de representación

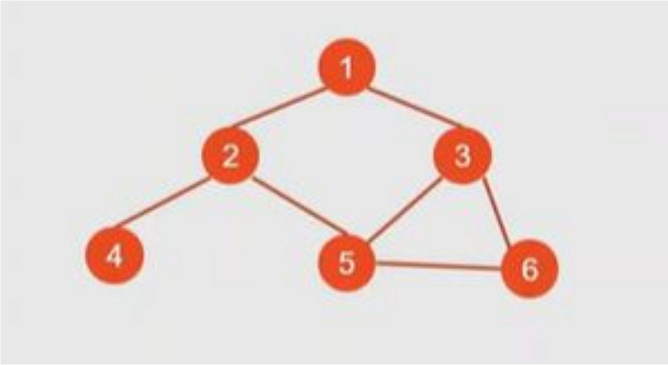
- Matriz de adyacencia

$$A: VxV = (a_{ij}) \quad a_{ij} = \begin{cases} 1 & si\ (i,j) \in \vec{E} \\ 0 & si\ (i,j) \notin \vec{E} \end{cases}$$

Nodos

Nodos	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	0	1	1	0
3	1	0	0	0	1	1
4	0	1	0	0	0	0
5	0	1	1	0	0	1
6	0	0	1	0	1	0

Formas de representación



1) **Conjunto de nodos y aristas**

V= {1,2,3,4,5,6}

E ={(1,2),(1,3),(2,4),(2,5),(3,5),(3,6),(5,6)}.

El conjunto de aristas es un **par ordenado que parte de un nodo origen y destino.**

En el ejemplo (1,2): nodo 1 se conecta con 2 y 2 con 1. Esto vale sólo para **GRAFOS NO DIRIGIDOS** (si fueran dirigidos sólo 1 se conecta con 2) → **<u>ENLACE SIMPLE</u> EN FORMA DE IMPLEMENTACIÓN**

En una búsqueda por ejemplo hay que buscar (1,2) o (2,1) para ver si existe la relación.

<u>ENLACE DOBLE</u>: mismo conjunto de **aristas** pero con los **dos pares ordenados (conexión inversa)**

E={(1,2),(2,1),(2,4),(4,2),(3,1),(1,3),(3,5),(5,3),, …}.

Tedioso de **mantener**, mayor **espacio**, pero útil para **BÚSQUEDAS** (podemos tener el conjunto de forma **ORDENADA**). Ej: si quiero ver si 3 se conecta con 5, **busco hasta donde empieza el 3 como origen**

Implementación

Implementar la variante del TAD Grafo (grafo simple) representado con conjunto de **nodos y aristas** con sus constructores y proyectores básicos. Los nodos del grafo tienen una **etiqueta única** para identificarlos y pueden asociarse entre ellos mediante aristas o arcos. Se puede crear un **grafo vacío**.

Implementar los **métodos** del grafo (no necesariamente con recursividad), con el siguiente__init__:

```
    # Constructor de la clase, inicializa el grafo con conjuntos vacíos de vértices y aristas
 def __init__(self, vertices: set[T] = set(), aristas: set[tuple[T, T]] = set()) -> None:
     self.vertices = vertices  # Conjunto de vértices o nodos del grafo
     self.aristas = aristas    # Conjunto de aristas, que son pares de vértices
```

```
def agregar_nodo(self, nodo: T)->None:
def agregar_arista(self, origen: T, destino: T)->None:
def eliminar_arista(self, origen: T, destino: T) -> None:
def eliminar_nodo(self, nodo: T)->None:
def es_vecino_de(self, nodo: T, otro_nodo: T) -> bool:
def vecinos_de(self, nodo: T) -> set[T]:
```

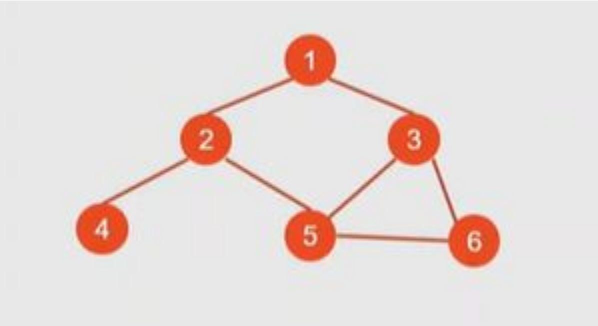
Formas de implementación

2) **Lista de adyacencia**

Parte también de un conjunto de vértices:

V= {1,2,3,4,5,6} y se complementa con una “lista de adyacencia” donde cada nodo apunta a una lista de otros nodos vecinos.

Es **más performante en almacenamiento de espacio** respecto de la próxima que vamos a ver y es muy práctica y útil para el **acceso a los vecinos**.



Nodos		Adyacentes
1	→	[2,3]
2	→	[1,4,5]
3	→	[1,5,6]
4	→	[2]
5	→	[2,3,6]
6	→	[3,5]

Implementación

Implementar la variante del TAD Grafo (grafo simple) representado con **lista de adyacencia** con sus constructores y proyectores básicos. Los nodos del grafo simplemente **tienen una etiqueta única** para identificarlos y pueden asociarse entre ellos mediante aristas o arcos. **Se puede crear un grafo vacío.**

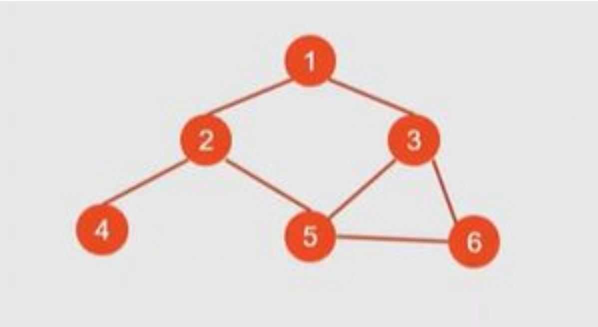
Implementar las funciones (no necesariamente con recursividad):

El __init__ es:

```
def __init__(self):
        # Inicializa el grafo como un diccionario donde las claves son nodos y los valores son conjuntos de nodos vecinos
        self.lista_adyacencia: dict[T, set[T]] = {}
```

```
def agregar_nodo(self, nodo: T):
def agregar_arista(self, origen: T, destino: T):
def eliminar_arista(self, origen: T, destino: T):
def eliminar_nodo(self, nodo: T):
def es_vecino_de(self, nodo: T, otro_nodo: T) -> bool:
def vecinos_de(self, nodo: T) -> list[T]:
```

Formas de representación



3) Matriz de adyacencia

Se establece una **matriz cuadrada de VxV** donde cada **índice** de fila (i) o **columna** (j) representa el **identificador del nodo.**

Grafos no dirigidos: es **simétrica**, en **dirigidos no**.

Problema: grafos con muchos nodos, el **espacio** requerido en **memoria** será así con costo cuadrático. En grafos **ponderados**, se puede asignar el **peso** de la arista en lugar de 1. Se debe tener cuidado de cómo modelar la no conexión ya que el **0 puede representar una conexión de costo 0.** Posible solución: matriz podría no ser de Entero sino un TAD nuevo que determine el peso.

	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	0	1	1	0
3	1	0	0	0	1	1
4	0	1	0	0	0	0
5	0	1	1	0	0	1
6	0	0	1	0	1	0

$$A = (a_{ij}) \quad a_{ij} = \begin{cases} 1 \; si \; (i,j) \in \vec{E} \\ 0 \; si \; (i,j) \notin \vec{E} \end{cases}$$

Implementación

Implementar la variante del TAD Grafo (grafo simple) representado con **matriz de adyacencia** con sus constructores y proyectores básicos. Los nodos del grafo simplemente **tienen una etiqueta única** para identificarlos y pueden asociarse entre ellos mediante aristas o arcos. Se puede crear un **grafo vacío**.

Implementar las funciones (no necesariamente con recursividad):

Tener en cuenta que la función __init__ es la siguiente:

```
    def __init__(self):
        self.matriz_adyacencia: list[list[int]] = []
        self.nodos: list[T] = []
```

```
def agregar_nodo(self, valor_nodo: T):
def agregar_arista(self, nodo1: T, nodo2: T):
def eliminar_nodo(self, nodo: T):
def eliminar_arista(self, nodo1: T, nodo2: T):
def es_vecino_de(self, nodo1: T, nodo2: T) -> bool:
def vecinos_de(self, nodo: T)->list[T]:
```