FUNDAMENTOS DE PROGRAMACIÓN

Proyecto Entregable: Entrega 03

La evaluación continua requiere de la realización de un proyecto a lo largo de todo el cuatrimestre que cuenta con un total de 3 entregas, cada una de ellas con una puntuación. El objetivo de la tercera entrega es diseñar e implementar una red social utilizando **grafos**, una estructura de datos fundamental para modelar relaciones entre entidades. En este proyecto, los **nodos** del grafo representarán los usuarios de la red social, mientras que las **aristas** modelarán las relaciones de amistad o conexión entre ellos.

Un **grafo** es una estructura matemática que se utiliza para modelar relaciones entre objetos. Se compone de:

- **Vértices** o **nodos**, que representan los objetos.
- **Aristas**, que representan las conexiones o relaciones entre los nodos.

Formalmente, un grafo $\boldsymbol{G}$ se define como un par $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$, donde:

- V es el conjunto de vértices.
- E es el conjunto de aristas, donde cada arista conecta dos vértices.

Los grafos pueden clasificarse en:

1. **Dirigidos**: Las aristas tienen un sentido (un nodo "origen" y un nodo "destino"). Cuando los grafos son dirigidos hay que definir también el tipo de recorrido posible:
    1. **FORWARD:** Recorrido hacia adelante, es decir, se sigue el orden normal de los sucesores de un nodo. Cuando se realiza el recorrido, se exploran los nodos vecinos o sucesores en el grafo en la dirección en la que se encuentran.
    2. **BACK:** Representa un recorrido hacia atrás, es decir, se exploran los predecesores de un nodo en lugar de sus sucesores. Este tipo de recorrido es útil en grafos dirigidos, donde se desean explorar los nodos hacia los cuales apuntan las aristas, en lugar de los nodos a los que apuntan las aristas.
2. **No dirigidos**: Las aristas no tienen dirección, es decir, la relación entre los nodos es bidireccional.
3. **Ponderados**: Las aristas tienen un peso o valor asociado (útil para representar costos, distancias, etc.).
4. **No ponderados**: Todas las aristas tienen el mismo valor o peso implícito.

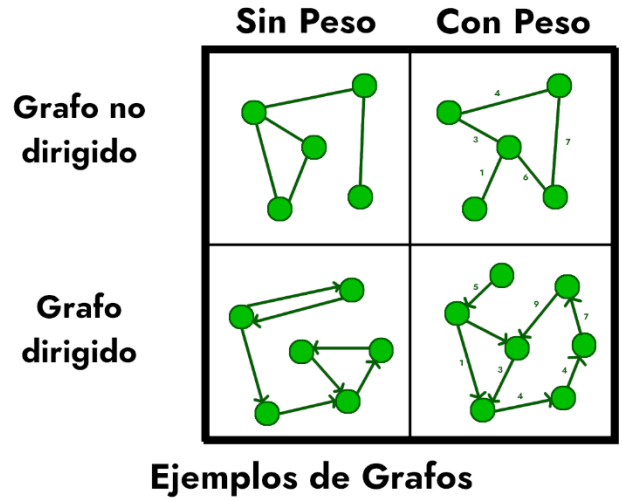

Ilustración 1: Tipos de Grafos. Fuente: https://uchile.progcomp.cl

Los **recorridos sobre grafos** son procesos que permiten visitar los nodos y aristas de un grafo siguiendo un patrón definido. Son fundamentales para analizar y procesar información en un grafo. Existen dos métodos principales de recorrido:

1. **Búsqueda en profundidad (DFS, *Depth-First Search*)**:
    - Se explora un camino completo desde un nodo inicial hasta el final antes de retroceder y explorar otros caminos.
    - Se utiliza una **pila** (puede ser la pila de llamadas del sistema en una implementación recursiva).
2. **Búsqueda en anchura (BFS, *Breadth-First Search*)**:
    - Se exploran todos los nodos a una misma distancia desde el nodo inicial antes de avanzar a la siguiente "capa".
    - Se utiliza una **cola** para gestionar los nodos a explorar.

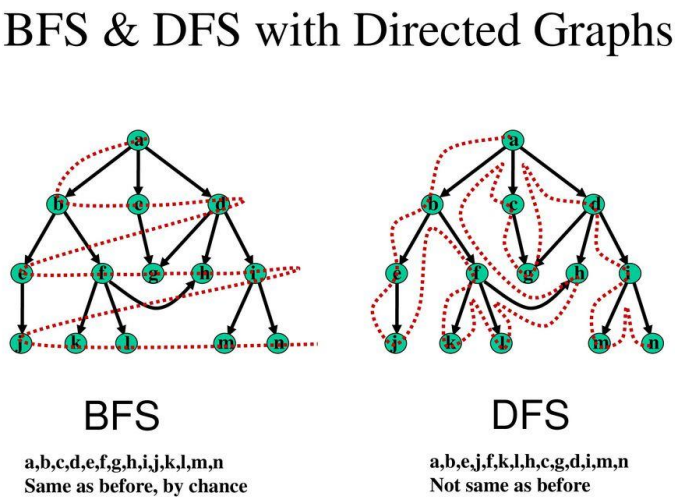

Ilustración 2: Búsqueda en grafos. Fuente :https://uchile.progcomp.cl

IMPORTANTE: En este entregable, se proporcionará un proyecto con parte del código ya implementado. Antes de comenzar, es fundamental que analice y comprenda los tipos y métodos ya disponibles, ya que serán la base para completar su entrega. A continuación, se especifican los tipos y métodos adicionales que deberá implementar, integrándolos correctamente con las estructuras y funcionalidades existentes.

TODO: Link al proyecto Github:

Resumen de los tipos presentes en este proyecto:

- **Grafo:** Representa la estructura abstracta de un grafo, donde se definen las relaciones entre nodos a través de métodos para obtener los sucesores de un nodo, el peso de una arista, y las aristas entre dos nodos específicos.
- **E_grafo:** Implementa una estructura de grafo concreta que puede ser dirigida o no dirigida, permitiendo operaciones como agregar vértices y aristas, calcular pesos de aristas, generar subgrafos, invertir un grafo, y visualizarlo gráficamente.
- **Recorrido:** Representa una estrategia general para recorrer un grafo, proporcionando métodos para obtener el camino, el árbol de recorridos, y el peso de los caminos desde un origen, así como para

determinar si existe un camino entre dos nodos. Los métodos de recorrido se implementan en clases derivadas, permitiendo algoritmos como búsqueda en profundidad o amplitud.

- **Recorrido en profundidad:** Implementa el algoritmo de búsqueda en profundidad (DFS) para recorrer un grafo, utilizando una pila para explorar los nodos lo más profundo posible antes de retroceder. A medida que recorre el grafo, construye un árbol de recorrido y calcula las distancias desde el nodo origen a los demás nodos.
- **Recorrido en anchura:** Implementa el algoritmo de búsqueda en amplitud (BFS) para recorrer un grafo, utilizando una cola para explorar los nodos de nivel en nivel. A medida que recorre el grafo, construye un árbol de recorrido y calcula las distancias (en términos de cantidad de aristas) desde el nodo origen a los demás nodos.
- **Usuario:** Representa a un usuario en el sistema, con atributos como DNI, nombre, apellidos y fecha de nacimiento.
- **Relación:** Representa una conexión entre usuarios en la red social.
- **Red_social:** Representa la red social como un grafo, donde los nodos son usuarios y las aristas son relaciones entre ellos.

A. E_grafo<Grafo[V,E]>

Métodos a implementar

- **__add_neighbors:** Añade un vértice alcnjunto de vecinos de otro vértice en el grafo.
- **__add_predecessors**: Añade un vértice al conjunto de predecesores de otro vértice en un grafo dirigido.
- **add_edge**: añade una arista entre un vértice origen y un vértice destino. El método debe asegurar que:
    - Los vértices de origen y destino existan en el grafo.
    - Los vértices no sean iguales (no se permitan bucles).
    - No se agregue una arista duplicada entre los mismos vértices.

  Preste especial atención al implementar este método, ya que deberá actualizar varias propiedades clave del grafo. Asegúrese de comprender cómo se gestionan las aristas, los vecinos, los predecesores y los órigenes/destinos para garantizar que todas las relaciones se mantengan correctamente. Además, deberá tener en cuenta si el grafo es dirigido o no dirigido.

- **edge_weight:** Este método debe devolver el peso de la arista que conecta los vértices sourceVertex y targetVertex. El peso se obtiene a través de la función de peso (_weight) definida en el grafo.
- **add_vertex**: Este método debe agregar un nuevo vértice al grafo. Si el vértice no existe en el grafo, se debe añadir y devolver True. Si el vértice ya está presente, no se debe agregar y se debe devolver False.
- **edge_source:** Este método debe devolver el vértice de origen de una arista e.
- **edge_target:** Este método debe devolver el vértice de destino de una arista e.
- **vertex_set:** Devuelve el conjunto de vértices del grafo. Es decir, devuelve el conjunto de todos los nodos almacenados en el grafo.
- **contains_edge**: Este método debe verificar si existe una arista entre los vértices origen y destino. Devuelve True si la arista existe, y False en caso contrario.

- **predecessors**: permite obtener los predecesores de un vértice en el grafo. En un grafo dirigido, los predecesores de un vértice son aquellos vértices que apuntan hacia el vértice dado. En un grafo no dirigido, no existen predecesores en el sentido tradicional, por lo que el método devolverá los vecinos del vértice.
- **sucessors**: permite obtener los sucesores de un vértice en un grafo. Los sucesores son los vértices que están conectados directamente al vértice dado, es decir, aquellos que están en el conjunto de vecinos de ese vértice. Sin embargo, dependiendo del tipo de recorrido (FORWARD o BACK), el comportamiento del método cambia. Si el tipo de recorrido es FORWARD se deben devolver los vecinos del vértice, si el tipo de recorrido es BACK se deben devolver los predecesores del vértice.
- **inverse_graph**: este método genera el grafo inverso de un grafo dirigido. Si el grafo es no dirigido, simplemente debe devolver el grafo original, ya que invertir las aristas en grafos no dirigidos no tiene sentido. Invertir un grafo significa invertir sus aristas, es decir, si en el grafo original existe una arista que va de source a target, en el grafo inverso esta arista irá de target a source.

B. Recorrido <ABC, Generic[V,E]>

Atributos

- **tree,** de tipo dict[V, tuple[Optional[V], float]]**,** protegida. Es un diccionario que almacena la información de los nodos visitados durante el recorrido. Las claves de dicho diccionario son los vértices y los valores una tupla tuple[Optional[V], float] que contiene dos elementos: el predecesor del vértice y el costo o distancia acumulada para llegar a ese vértice desde el nodo de inicio.
- **path,** de tipo list[V], protegida. Es una lista que guarda el camino o secuencia de vértices visitaddos durante el recorrido.
- **grafo,** de tipo Grafo[V, E], protegida. Grafo sobre el que se realiza el recorrido.

Otros métodos

- **path_to_origin*:*** permite construir el camino hacia el origen a partir de un vértice dado. Para ello, el método sigue los predecesores de cada vértice hasta llegar al nodo raíz. El método devuelve una lista de vértices que forman este camino, comenzando desde el vértice source hasta el nodo raíz.
- **origin:** permite determinar el origen de un recorrido a partir de un vértice dado. El recorrido sigue los predecesores registrados en el árbol de recorrido, hasta llegar al nodo raíz.
- **groups:** organiza los vértices del grafo en un diccionario, donde las claves son los vértices de origen y el valor es el conjunto de vértices que pertenecen a ese grupo. Los vértices que tienen el mismo origen se agrupan juntos.

C. Recorrido_en_profundidad <Recorrido[V, E]>

Métodos de factoría

- **of:** Método de factoría que se utiliza para crear una nueva instancia de la clase a partir de un grafo.

Otros métodos

- **traverse:** Realiza un recorrido en profundidad (DFS) del grafo comenzando desde el vértice source que se le pasa como parámetro de entrada. Utilizando una pila, el algoritmo visita los vértices coonectados de manera recursiva, marcando los vértices visitados y registrando el recorrido en el árbol del recorrido.

D. Usuario

Este tipo representa un usuario en el sistema, con atributos como DNI, nombre, apellidos y fecha de nacimiento.

Propiedades

- **dni**, de tipo str: El DNI del usuario, que debe seguir el formato de 8 dígitos seguidos de una letra.
- **nombre**, de tipo str: El nombre del usuario.
- **apellidos**, de tipo str: Los apellidos del usuario.
- **fecha_nacimiento**, de tipo date: La fecha de nacimiento del usuario, que debe ser anterior a la fecha actual.

Métodos de factoría

- **of:** Método de factoría que crea e inicializa una nueva instancia de la clase.
- **parse:** Método de factoría que recibe una cadena de texto con la información de un usuari9o yla convierte en una instancia del tipo. La cadena de texto tiene el siguiente formato: *"45718832U,Carlos,Lopez,1984-01-14"*

Representación como cadena:
dni - nombre

E. Relacion

Este tipo representa una relación entre dos usuarios, con atributos como ID, número de interacciones y días activos. La clase permite crear instancias de Relacion con un identificador único y proporciona una representación detallada de la relación.

Propiedades/Atributos

- **id**, de tipo int: El ID único de la relación, generado automáticamente cada vez que se crea una nueva instancia de Relacion.
- **interacciones**, de tipo int: El número de interacciones que ha tenido la relación entre los usuarios.
- **dias_activa**, de tipo int: El número de días activos que ha tenido la relación.
- **xx_num**, de tipo int, privada: Un contador interno que incrementa cada vez que se crea una nueva instancia de Relacion para asegurar que el ID sea único.

Métodos de factoría

- **of:** Crea e inicializa una nueva instancia de la clase Relación, asignando un ID único y tomando como parámetros el número de interacciones y los días activos de la relación.

Representación como cadena:

({id} - días activa: {dias_activa} - num interacciones {interacciones})

F. Red_social(E_grafo[Usuario, Relacion])

Este tipo representa una red social modelada como un grafo donde los nodos son usuarios y las aristas son relaciones entre ellos. La clase Red_social extiende la clase E_grafo y proporciona métodos para crear e interactuar con redes sociales, gestionando usuarios y sus relaciones.

Atributos

- **usuarios_dni**, de tipo dict, privada: Un diccionario que almacena los usuarios, indexados por su DNI. Permite acceder rápidamente a los usuarios dentro de la red social por su identificador único.

Métodos de factoría

- **of:** Método de factoría que crea e inicializa una nueva instancia de Red_social a partir del tipo de grafo y del tipode recorrido. Por defecto el tipo del grafo será no dirigidoy el tipo de recorrido será inverso (BACK).
- **parse:** Método de factoría que lee dos archivos (uno con información de usuarios y otro con relaciones) y crea una instancia de **Red_social**. Los usuarios y las relaciones se leen desde los archivos proporcionados, se agregan al grafo y se establecen las conexiones correspondientes entre ellos.

Representación como cadena:

({id} - días activa: {dias_activa} - num interacciones {interacciones})

ANEXO

A continuación, se muestra a modo de guía una posible salida por consola de cada uno de los tests a realizar.

Test_grafo.py

```
********** Nº Predecesores de cada vértice
18909774Z - Maria -- 2
952871880 - Pedro -- 0
55039956S - David -- 3
60412985S - Maria -- 2
951577320 - Pedro -- 2
58127458W - Maria -- 1
56427434U - Elena -- 3
71894470A - Carlos -- 0
25143909I - Lucia -- 3
45718832U - Carlos -- 6
82007713N - Carlos -- 3
16274768S - Juan -- 3
76929765H - Juan -- 0
63506915L - Lucia -- 1
62258675I - Laura -- 2
92322186A - Pedro -- 3
85707754E - Jorge -- 1
61832964Y - Pedro -- 1
10115245D - Ana -- 3
87345530M - Ana -- 1

********** Nº Vecinos de cada vértice
18909774Z - Maria -- 2
952871880 - Pedro -- 0
55039956S - David -- 3
60412985S - Maria -- 2
951577320 - Pedro -- 2
58127458W - Maria -- 1
56427434U - Elena -- 3
```

71894470A - Carlos -- 0

25143909I - Lucia -- 3

45718832U - Carlos -- 6

82007713N - Carlos -- 3

16274768S - Juan -- 3

76929765H - Juan -- 0

63506915L - Lucia -- 1

62258675I - Laura -- 2

92322186A - Pedro -- 3

85707754E - Jorge -- 1

61832964Y - Pedro -- 1

10115245D - Ana -- 3

87345530M - Ana – 1

Test_recorrido_en_anchura.py

El camino más corto desde 25143909I hasta 87345530M es: [Usuario(dni='87345530M', nombre='Ana', apellidos='Rodriguez', fecha_nacimiento=datetime.date(1964, 3, 2)), Usuario(dni='18909774Z', nombre='Maria', apellidos='Diaz', fecha_nacimiento=datetime.date(1995, 1, 5)), Usuario(dni='25143909I', nombre='Lucia', apellidos='Lopez', fecha_nacimiento=datetime.date(1955, 6, 7))]

La distancia mínima es: 2 pasos.

Test_recorrido_en_profundidad.py

No hay conexión directa entre 25143909I y 76929765H.