



23 de Noviembre de 2023

Actividad

Actividad 6

Estructuras Nodales

Entrega

- **Lugar:** En Canvas - **Actividad 6**
- **Fecha máxima de entrega:** 27 de Noviembre a las 20:00

Introducción

A medida que crece la popularidad del deporte *DCCaptura la bandera*, la necesidad de tener mejores jugadores y mejores equipos también lo hace.

Para apoyar la creación de su equipo, **Sir Hernán444** estudió las capacidades e interacciones entre sus jugadores y te pide a ti que la ayudes a representar esas interacciones para formar los equipos de la mejor manera y así poder competir con **Lily416** y **Joaking**.

Flujo del programa

El programa consiste en un grafo dirigido que contiene la información individual de cada uno de los jugadores, junto con las interacciones entre ellos y permite hacerle consultas sobre los datos que contiene. Decimos que un jugador es amigo de otro, si tiene una arista hacia él, decimos que dos jugadores son conocidos si es que existe un camino entre ellos y, finalmente, decimos que dos jugadores son compañeros si pertenecen al mismo equipo (sin importar si están conectados o no).

Dentro del grafo, cada jugador corresponde a un nodo, que guarda su nombre y velocidad, y existirá una arista entre dos jugadores si es que el jugador A ve al jugador B como un amigo. Cómo es un grafo dirigido, que A vea a B como amigo, no implica que B también vea a A como amigo... la amistad no es recíproca.

Archivos

Archivos de código

En el directorio de la actividad encontrarás los siguientes archivos con código:

- **Modificar** `equipo.py`: Aquí encontrarás la definición básica de las clases que debes implementar y completar, junto con un pequeño *main* que prueba las consultas.
- **No modificar** `test.py`: Este archivo ejecuta los *tests* asociados a las distintas consultas.

Entidades

No modificar `class Jugador:`

Clase que representa un nodo del grafo. Cada jugador guarda su información personal y tiene los siguientes métodos:

- **No modificar** `def __init__(self, nombre: str, velocidad: int) -> None:`
Inicializador de la clase. Asigna los siguientes atributos:

<code>self.nombre</code>	Un <code>str</code> que representa el nombre del jugador.
<code>self.velocidad</code>	Un <code>int</code> que representa la velocidad del jugador.

- **No modificar** `def __repr__(self) -> None:`
Método encargado del `repr` de la clase.

Modificar `class Equipo:`

Clase que representa al grafo con los datos de los jugadores. Posee los siguientes métodos:

- **No modificar** `def __init__(self) -> None:`
Inicializador de la clase. Define los siguientes atributos:

<code>self.jugadores</code>	Un <code>dict</code> que asocia el <code>id_jugador</code> de los jugadores con la instancia <code>Jugador</code> correspondiente.
<code>self.dict_adyacencia</code>	Un <code>defaultdict</code> que asocia cada <code>id_jugador</code> con una lista formada por los <code>id_jugador</code> de sus vecinos. En este caso, los vecinos de un jugador A serían todos los jugadores dentro del equipo que A considera amigos.

- **Modificar** `def agregar_jugador(self, id_jugador: int, jugador: Jugador) -> bool:`
Método encargado de agregar un jugador al grafo. Recibe el `id_jugador` y la instancia del jugador e intenta agregarlo al grafo.
 - Si no existe un jugador con ese `id_jugador`, lo añade al diccionario de jugadores y retorna `False`. En este método no debes modificar la variable `dict_adyacencia`.
 - Si ya existe un jugador con ese `id_jugador` en el equipo, no lo agrega y retorna `True`.
- **Modificar** `def agregar_vecinos(self, id_jugador: int, vecinos: list[int]) -> int:`
Método encargado de agregar los vecinos de un jugador al grafo. Recibe el `id_jugador` y una lista con los `id_jugador` de sus vecinos e intenta agregar todos los vecinos de la lista al diccionario de adyacencia.
 - En caso que el jugador no se encuentre en el **diccionario de jugadores**, este debe agregarse al **diccionario de adyacencia** como una entrada nueva, pero **no deben agregarse sus vecinos**. Luego de lo anterior, el método retorna `-1`.

- Si el jugador se encuentra en el diccionario de jugadores, se deben agregar los vecinos y retornar la cantidad de nuevos vecinos agregados. En este ultimo caso, si algún `id_jugador` de la lista `vecinos` ya existe previamente entre los vecinos del jugador, este `id_jugador` **no debe ser añadido nuevamente**, solamente se añaden los nuevos vecinos que no estaban previamente.
- **Modificar** `def peor_amigo(self, id_jugador: int) -> Jugador:`
Un jugador se dice “peor amigo” de otro si: es su amigo (vecino) y además la diferencia entre sus velocidades es la mayor entre todos los vecinos. El método recibe el `id_jugador` de un jugador, busca entre sus vecinos a su peor amigo y lo retorna.

Nota: Un jugador no puede ser su propio peor amigo. Puedes asumir que el `id_jugador` está en el equipo y que nunca existirá empate en la diferencia de velocidades. Si el jugador no tiene vecinos, el método debe retornar `None`.
- **Modificar** `def mejor_compañero(self, id_jugador: int) -> Jugador:`
Un jugador será el “mejor compañero” de otro si es que la diferencia entre sus velocidad es la menor en **todo el equipo**. Este método recibe el `id_jugador` de un jugador y retorna a su mejor compañero en el equipo.

Nota: Un jugador no puede ser su propio mejor compañero. Puedes asumir que el `id_jugador` está en el equipo y que nunca existirá empate en la diferencia de velocidades. Si el jugador es el único miembro del equipo, el método debe retornar `None`.
- **Modificar** `def mejor_conocido(self, id_jugador: int) -> Jugador:`
Decimos que dos jugadores están conectados si existe un camino entre ellos en el grafo. Un jugador será el “mejor conocido” de otro si es que la diferencia entre sus velocidad es la menor entre todos los jugadores conectados a este. Este método recibe el `id_jugador` de un jugador y retorna a su mejor conocido en el equipo.

Nota: Un jugador no puede ser su propio mejor conocido. Puedes asumir que el `id_jugador` está en el equipo y que nunca existirá empate en la diferencia de velocidades. Si el jugador no tiene vecinos, el método debe retornar `None`.
- **Modificar** `def distancia(self, id_jugador_1: int, id_jugador_2: int) -> int:`
Definimos la distancia entre dos jugadores como el tamaño del camino más corto entre ellos y decimos que la distancia entre un nodo y sí mismo es 0. Este método recibe los *ids* de dos jugadores y calcula la distancia entre estos. Si no existe camino entre los dos nodos, o si alguno de los dos `id_jugador` no está en el equipo, el método debe retornar `-1`.

Notas

- No puedes hacer *import* de otras librerías que no sean las ya entregadas en el archivo a completar.
- Recuerda que la ubicación de tu entrega es en **Canvas**. Puedes utilizar *git* para tener un respaldo de tu trabajo, pero finalmente se revisará lo subido al buzón de Canvas.
- Recuerda que esta evaluación presenta corrección **automatizada**. Si entregas un código que se cae al momento de correr los *tests*, será evaluado con 0 puntos.
- Se recomienda completar la actividad en el orden del enunciado.
- Si aparece un error inesperado, ¡léelo! Intenta interpretarlo y/o buscarlo en *Google*.