****

**Trabajo Final para la Cátedra Estructuras de Datos en Python**

***Tema Principal:***

**Desarrollo de un Juego de Dragon Ball**

***Profesor:***

Angel Leonardo Bianco

***Alumnos:***

Gastón Daniel Dicundo

Alexander Molina

Cecilia Ayelen Senilliani

En el diseño del juego de Dragon Ball intentaremos crear un entorno dinámico y competitivo donde los personajes evolucionen y participen en torneos. A continuación, se detallan las decisiones de diseño, el análisis de complejidad y las justificaciones de las estructuras de datos elegidas.

Teoría y Realización Práctica

1. *Clases e Interfaces:*

Personaje

La clase Personaje contiene atributos claves como:

* nombre : Nombre del personaje.
* nivel\_poder : Fuerza del personaje.
* raza : Especie del personaje, esta puede influir en las habilidades y transformaciones.
* habilidades : Lista o conjunto de habilidades que posee el personaje.

Métodos

* Métodos de acción: Incluye métodos como subir\_nivel, incrementa el nivel de poder y adquirir\_habilidad para obtener nuevas habilidades. Estos serán fundamental para el progreso de los personajes.
* Gestión de combates: Se utilizará el patrón de estrategias para los diferentes tipos de combates.
* Método de información: Un método que presente los atributos y habilidades de los personajes.

*Definir un Personaje como una clase encierra los datos y funciones de cada uno en un solo lugar, mejorando la organización y reutilización del código.*

1. *Estructuras Recursivas:*

Funciones recursivas para Evolución de Poder

La evolución de poder mediante una función recursiva que incremente el nivel\_poder de acuerdo con las victorias en combate y la transformación del personaje. Este multiplicador de poder se basa en el número de combates y el tipo de enemigo derrotado. Validando que el incremento sea mayor que 0. Si no lo es, el nivel no se modifica y se muestra un mensaje claro.

*El enfoque recursivo permite la evolución gradual del personaje, calculando el poder de cada transformación y combate. Creando un sistema que se adapta al juego sin requerir muchas variables adicionales.*

1. *Árboles Binarios:*

Árbol Binario para clasificación de Personajes

Para organizar a los personajes según su nivel\_poder , se implementa un árbol de búsqueda donde cada uno contiene un personaje. Los nodos están de modo para la búsqueda rápida de los más fuertes o débiles.

*El BST es ideal para manejar búsquedas rápidas y eficaces de personajes por poder, ya sea, para encontrar más fuertes o más débiles en los torneos o en el desarrollo del juego. Permitiendo un mantenimiento ordenado de la lista de personajes en función de su evolución.*

1. *Árboles Generales:*

Árboles Generales para Habilidades

El árbol de habilidades se organiza en uno general, cada habilidad es un nodo y las técnicas son sus subnodos. Esta estructura permite que las habilidades tengan una jerarquía.

*El árbol general facilita la organización de habilidades, permitiendo que los personajes desbloqueen otras según su progreso. También admite una fácil expansión y flexibilidad para las necesidades de los jugadores en el juego.*

1. *Cola de Prioridades y Heap Binaria:*

Cola de Prioridades para Torneos

Se utilizará una cola de prioridades en un heap binario, donde los personajes con mayor nivel\_poder tienen prioridad. Asegurando que los que son más poderosos se enfrenten primeros en los torneos, lo que mantendrá la emoción y la competitividad en el juego.

*La cola de prioridad permite que los jugadores más poderosos sean seleccionados rápidamente para los enfrentamientos, optimizando la organización de torneos y permitiendo que los combates se realicen de manera justa y estratégica.*

1. *Análisis de Algoritmos:*

Métodos principales:

Clases Personaje

1. Método agregar\_habilidad:

* Tiempo: Si usamos una lista para almacenar habilidades, agregar una nueva habilidad toma *O*(1). Si necesitamos evitar duplicados, recorrer la lista toma *O*(*n*), donde *n* es el numero de habilidades.
* Espacio: Según cuantas habilidades tenga el personaje, ocupando *O*(*n*).

1. Método subir\_nivel

* Tiempo: Validar el incremento y actualizar el nivel de poder es una operación constante, *O*(1).
* Espacio: No requiere almacenamiento adicional, por lo que es *O*(1).

1. Método mostrar\_info

* Tiempo: Mostrar la información de los atributos del personaje toma *O*(1).
* Espacio: También es *O*(1) ya que no se almacenan datos adicionales.

Función combate

* Tiempo: Comprar niveles de poder y actualizar el poder del ganador *O*(1).
* Espacio: No usamos estructuras adicionales, por lo que es *O*(1).

Función transformar

* Tiempo: Depende del numero de transformaciones, tomando *O*(*t*), donde *t* es el nivel de transformación.
* Espacio: La recursión utiliza memoria en la pila proporcional al número de transformaciones, *O*(*t*).

Árbol de Búsqueda Binaria (BST)

1. Inserción:

* Tiempo: Depende de la altura del árbol (*h*). Si el árbol esta balanceado, es *O*(log *n*). En el peor caso (si está desbalanceado), es *O*(*n*).
* Espacio: *O*(*h*), ya que la recursión para insertar utiliza memoria proporcional a la altura del árbol.

1. Búsqueda del más fuerte o más débil:

* Tiempo: Igual que la inserción, *O*(log *n*) en promedio y *O*(*n*) en el peor caso.
* Espacio: *O*(1) ya que no necesitamos memoria adicional para almacenar nodos.

1. Recorrido en orden:

* Tiempo: Visitar todos los nodos toma *O*(*n*).
* Espacio: La recursión utiliza *O*(*h*), donde *h* es la altura del árbol.

Cola de Prioridad con Heap Binario

1. Agregar personaje:

* Tiempo: Insertar en un heap toma *O*(log *n*), donde *n* es el número de personajes en la cola.
* Espacio: *O*(*n*), ya que el heap almacena todos los personajes.

1. Siguiente combate:

* Tiempo: Extraer el personaje con mayor prioridad toma *O*(log *n*).
* Espacio: *O*(1), ya que solo se devuelve el siguiente personaje.

1. Mostrar cola:

* Tiempo: Mostrar la cola completa requiere ordenar los elementos, *O*(*n* log *n*)
* Espacio: O(*n*) para almacenar los personajes en la cola.

*Para las batallas y la evolución de poder, nuestras operaciones son eficientes, con una complejidad mayormente constante (O(1)).*

*La organización de personajes, ya sea con un árbol binario o una cola de prioridad, depende de la estructura elegida:*

*El* ***BST*** *es más eficiente para búsquedas y recorridos ordenados, con O(log n) en promedio.*

*La* ***cola de prioridad*** *es ideal para priorizar combates, con O(log n) para operaciones clave como insertar y extraer.*

*Los árboles generales de habilidades son adecuados para gestionar jerarquías, con una complejidad lineal (O(n)) para recorrer todos los nodos.*

Codificación y algoritmos

1. Grafos: Vamos a crear una estructura que represente planetas como nodos y rutas espaciales como aristas. También incluiremos funcionalidades para agregar planetas y rutas, buscar rutas entre planetas, y permitir a los personajes viajar.

Características:

1. Nodos (planetas): Representan ubicaciones en el universo de Dragon Ball (Tierra, Namek, Vegeta, etc.).
2. Aristas (rutas espaciales): Representan conexiones entre planetas con una distancia asociada.
3. Funcionalidades:

* Agregar planetas: Permite expandir el universo.
* Agregar rutas: Define caminos entre planetas.
* Mostrar rutas: Visualiza las conexiones existentes.
* Búsqueda de ruta más corta: Implementa el algoritmo de Dijkstra para calcular la distancia mínima entre dos planetas.

1. Recorrido DFS y BFS:

DFS (Depth-First Search):

* Explora lo más profundo posible antes de retroceder.
* Útil para buscar caminos específicos o exhaustivos.
* Puede encontrar un camino más rápido en grafos con muchas ramas profundas.

FS (Breadth-First Search):

* Explora todos los nodos al mismo nivel antes de ir más profundo.
* Garantiza encontrar el camino más corto en términos de número de pasos si todas las aristas tienen el mismo peso.

*Estos algoritmos complementan las funcionalidades del grafo, permitiendo búsquedas estratégicas y exploración del universo de Dragon Ball.*

1. Ordenamiento topológico:

* Cada nodo del grafo representa una habilidad.
* Una arista dirigida A→B significa que A debe ser dominada antes de desbloquear B.

Explicación del Algoritmo

* Se calcula el grado de entrada de cada nodo (cantidad de prerrequisitos).
* Se procesan primero los nodos con grado de entrada cero (sin prerrequisitos).
* Cada vez que un nodo es procesado, se reduce el grado de entrada de sus vecinos.
* Si todos los nodos son procesados, el grafo es un DAG válido, y se obtiene el ordenamiento.
* Si no, existe un ciclo en los prerrequisitos.

*Este enfoque es útil para planificar las etapas de entrenamiento en cualquier sistema de habilidades progresivas, asegurando que se respeten las jerarquías y dependencias.*

1. Problemas NP y Camino Mínimo:

Podemos aplicar el algoritmo para modelar el mapa del universo de Dragon Ball. Aquí cada planeta es un nodo, las conexiones entre ellos son aristas ponderadas (distancias), y buscamos la ruta más corta para recolectar todas las Esferas del Dragón distribuidas en distintos planetas.

Diseño del Problema

1. Nodos: Cada planeta del universo.
2. Aristas: Distancias entre planetas conectados.
3. Objetivo: Encontrar la ruta más corta que pase por los planetas donde están las Esferas del Dragón.

De esta manera:

* Dijkstra calcula la ruta más corta desde el planeta actual al siguiente objetivo.
* La función recolectar\_esferas repite este proceso hasta visitar todos los planetas que contienen las Esferas del Dragón.
* La ruta y la distancia total se construyen progresivamente.

*Esto asegura la recolección de las Esferas del Dragón de forma óptima en términos de distancia recorrida.*

import heapq

#Para la clase personaje

#Defi clase Personaje con los atributos básicos: nombre, nivel\_poder, habilidades, y raza. También le agregaremos métodos para subir de nivel y añadir habilidades.

class Personaje:

    def \_\_init\_\_(self, nombre, nivel\_poder, raza):

        self.nombre = nombre

        self.nivel\_poder = nivel\_poder

        self.raza = raza

        self.habilidades = []

    def agregar\_habilidad(self, habilidad):

        """Agrega una habilidad al personaje."""

        self.habilidades.append(habilidad)

    def subir\_nivel(self, incremento\_poder):

        """Aumenta el nivel de poder del personaje en función del incremento dado validando que el incremento sea un número positivo"""

        if incremento\_poder <= 0:

           print(f"El incremento debe ser positivo. {self.nombre} no ha cambiado su nivel de poder.")

        if not isinstance(incremento\_poder, (int, float)):

            print(f"El incremento debe ser un número.{self.nombre} no ha cambiado su nivel de poder.")

            return

        self.nivel\_poder += incremento\_poder

        print(f"{self.nombre} ha subido su nivel de poder a {self.nivel\_poder:.2f}")

    def mostrar\_info(self):

        """Muestra la información del personaje."""

        info = f"Nombre: {self.nombre}, Raza: {self.raza}, Nivel de Poder: {self.nivel\_poder}, Habilidades: {self.habilidades}"

        print(info)

#Gestion de combates

#Para los combates, podemos implementar una función de enfrentamiento entre dos personajes que compare el nivel de poder.

def combate(personaje1, personaje2):

    """Realiza un combate entre dos personajes y determina el ganador."""

    if personaje1.nivel\_poder > personaje2.nivel\_poder:

        print(f"{personaje1.nombre} ha ganado el combate contra {personaje2.nombre}")

        personaje1.subir\_nivel(personaje2.nivel\_poder \* 0.1)  # Ejemplo: el ganador aumenta un 10% del poder del oponente

    elif personaje2.nivel\_poder > personaje1.nivel\_poder:

        print(f"{personaje2.nombre} ha ganado el combate contra {personaje1.nombre}")

        personaje2.subir\_nivel(personaje1.nivel\_poder \* 0.1)

    else:

        print("El combate ha terminado en empate.")

#Transformaciones

#Podemos implementar una función recursiva que aplique multiplicadores a nivel\_poder.

def transformar(personaje, multiplicador, nivel\_transformacion):

    """Aplica una transformación recursiva al personaje según su nivel de transformación."""

    if nivel\_transformacion == 0:

        return personaje.nivel\_poder

    else:

        personaje.nivel\_poder \*= multiplicador

        return transformar(personaje, multiplicador, nivel\_transformacion - 1)

#Implementacion de Arbol de Busqueda

#Clasifica a los personajes de acuerdo a su nivel de poder. El personaje con el mayor nivel de poder estará en el nodo más a la derecha del árbol.  El personaje con el menor nivel de poder estará en el nodo más a la izquierda del árbol.

class NodoBST:

    def \_\_init\_\_(self, personaje):

        self.personaje = personaje  # El personaje en este nodo

        self.izquierdo = None  # Hijo izquierdo (personajes con menor poder)

        self.derecho = None    # Hijo derecho (personajes con mayor poder)

class ArbolPersonajes:

    def \_\_init\_\_(self):

        self.raiz = None  # El árbol comienza vacío

    def insertar(self, personaje):

        """Inserta un personaje en el árbol de acuerdo a su nivel de poder."""

        if self.raiz is None:

            self.raiz = NodoBST(personaje)

        else:

            self.\_insertar\_nodo(self.raiz, personaje)

    def \_insertar\_nodo(self, nodo, personaje):

        """Método recursivo para insertar un nodo en el árbol."""

        if personaje.nivel\_poder < nodo.personaje.nivel\_poder:

            if nodo.izquierdo is None:

                nodo.izquierdo = NodoBST(personaje)

            else:

                self.\_insertar\_nodo(nodo.izquierdo, personaje)

        else:

            if nodo.derecho is None:

                nodo.derecho = NodoBST(personaje)

            else:

                self.\_insertar\_nodo(nodo.derecho, personaje)

    def buscar\_personaje\_mas\_fuerte(self):

        """Devuelve el personaje con el mayor nivel de poder (más a la derecha)."""

        if self.raiz is None:

            return None

        return self.\_buscar\_mas\_fuerte(self.raiz)

    def \_buscar\_mas\_fuerte(self, nodo):

        """Recorre el árbol hasta encontrar el personaje más fuerte."""

        while nodo.derecho:  # El personaje más fuerte está más a la derecha

            nodo = nodo.derecho

        return nodo.personaje

    def buscar\_personaje\_mas\_debil(self):

        """Devuelve el personaje con el menor nivel de poder (más a la izquierda)."""

        if self.raiz is None:

            return None

        return self.\_buscar\_mas\_debil(self.raiz)

    def \_buscar\_mas\_debil(self, nodo):

        """Recorre el árbol hasta encontrar el personaje más débil."""

        while nodo.izquierdo:  # El personaje más débil está más a la izquierda

            nodo = nodo.izquierdo

        return nodo.personaje

    def en\_orden(self):

        """Muestra los personajes en orden (de menor a mayor nivel de poder)."""

        self.\_en\_orden\_recursivo(self.raiz)

    def \_en\_orden\_recursivo(self, nodo):

        """Recorrido en orden para imprimir los personajes."""

        if nodo:

            self.\_en\_orden\_recursivo(nodo.izquierdo)

            print(f"{nodo.personaje.nombre}: {nodo.personaje.nivel\_poder}")

            self.\_en\_orden\_recursivo(nodo.derecho)

#Arboles generales

#Este modelo organiza habilidades de manera eficiente, permite jerarquías claras y soporta la expansión futura, ideal para un juego donde las habilidades progresan y evolucionan.

#Los jugadores pueden desbloquear habilidades según cumplan requisitos previos.

#Es fácil añadir nuevas técnicas al árbol.

#Facilita la integración con un sistema de juego basado en niveles o puntos de habilidad.

class NodoHabilidad:

    def \_\_init\_\_(self, nombre):

        self.nombre = nombre

        self.subhabilidades = []

    def agregar\_subhabilidad(self, subhabilidad):

        self.subhabilidades.append(subhabilidad)

    def mostrar\_arbol\_de\_habilidades(self, nivel=0):

        print(" " \* nivel \* 2 + f"- {self.nombre}")

        for sub in self.subhabilidades:

            sub.mostrar\_arbol\_de\_habilidades(nivel + 1)

# Creación del árbol de habilidades

kamehameha = NodoHabilidad("Kamehameha")

potenciado = NodoHabilidad("Kamehameha potenciado")

potenciado.agregar\_subhabilidad(NodoHabilidad("Kamehameha x10"))

potenciado.agregar\_subhabilidad(NodoHabilidad("Kamehameha dual"))

instantaneo = NodoHabilidad("Kamehameha instantáneo")

instantaneo.agregar\_subhabilidad(NodoHabilidad("Kamehameha instantáneo x10"))

electrico = NodoHabilidad("Kamehameha eléctrico")

electrico.agregar\_subhabilidad(NodoHabilidad("Kamehameha eléctrico avanzado"))

# Construcción del árbol

kamehameha.agregar\_subhabilidad(potenciado)

kamehameha.agregar\_subhabilidad(instantaneo)

kamehameha.agregar\_subhabilidad(electrico)

# Visualización del árbol

kamehameha.mostrar\_arbol\_de\_habilidades()

#Cola de prioridad

#Heap Binaria= Utilizamos un heap binario maximo, el cual prioriza  a los personajes con niveles de poder más altos.

#Esta estructura garantiza que los personajes más poderosos enfrenten primero a otros combatientes, optimizando la organización del torneo y mejorando la experiencia del juego.

class TorneoColaDePrioridades:

    def \_\_init\_\_(self):

        # Usaremos una lista para almacenar el heap

        # En Python, heapq implementa un min-heap por defecto, así que usaremos valores negativos para simular un max-heap.

        self.cola = []

    def agregar\_personaje(self, nombre, nivel\_poder):

        # Insertamos el personaje con nivel de poder negativo para simular max-heap

        heapq.heappush(self.cola, (-nivel\_poder, nombre))

    def siguiente\_combate(self):

        # Extraemos el personaje con mayor nivel de poder

        if self.cola:

            nivel\_poder, nombre = heapq.heappop(self.cola)

            return nombre, -nivel\_poder

        else:

            return None, None

    def mostrar\_cola(self):

        # Mostrar la cola en orden de prioridad

        personajes = [(-nivel\_poder, nombre) for nivel\_poder, nombre in self.cola]

        return sorted(personajes, reverse=True)  # Ordenamos por prioridad (mayor nivel de poder primero)

-----

class NodoPlaneta:

    def \_\_init\_\_(self, nombre):

        self.nombre = nombre

        self.vecinos = {}  # Diccionario para almacenar planetas conectados y su distancia

    def agregar\_vecino(self, planeta, distancia):

        self.vecinos[planeta] = distancia  # Agregar conexión con otro planeta

class GrafoUniverso:

    def \_\_init\_\_(self):

        self.planetas = {}  # Diccionario para almacenar todos los nodos del grafo

    def agregar\_planeta(self, nombre):

        """Agrega un nuevo planeta al grafo."""

        if nombre not in self.planetas:

            self.planetas[nombre] = NodoPlaneta(nombre)

    def agregar\_ruta(self, origen, destino, distancia):

        """Crea una conexión entre dos planetas con una distancia específica."""

        if origen in self.planetas and destino in self.planetas:

            self.planetas[origen].agregar\_vecino(self.planetas[destino], distancia)

            self.planetas[destino].agregar\_vecino(self.planetas[origen], distancia)  # Grafo no dirigido

    def mostrar\_rutas(self):

        """Muestra las conexiones y distancias entre planetas."""

        for planeta in self.planetas.values():

            print(f"Planeta {planeta.nombre} está conectado con:")

            for vecino, distancia in planeta.vecinos.items():

                print(f"  - {vecino.nombre} a una distancia de {distancia} unidades.")

    def buscar\_ruta\_mas\_corta(self, origen, destino):

        """Implementa el algoritmo de Dijkstra para encontrar la ruta más corta entre dos planetas."""

        import heapq

        if origen not in self.planetas or destino not in self.planetas:

            print("Uno o ambos planetas no existen.")

            return None

        # Inicialización de distancias y cola de prioridad

        distancias = {planeta: float('inf') for planeta in self.planetas}

        distancias[origen] = 0

        cola = [(0, origen)]  # (distancia acumulada, nodo actual)

        padres = {origen: None}

        while cola:

            distancia\_actual, planeta\_actual = heapq.heappop(cola)

            if planeta\_actual == destino:

                break

            for vecino, distancia in self.planetas[planeta\_actual].vecinos.items():

                distancia\_total = distancia\_actual + distancia

                if distancia\_total < distancias[vecino.nombre]:

                    distancias[vecino.nombre] = distancia\_total

                    padres[vecino.nombre] = planeta\_actual

                    heapq.heappush(cola, (distancia\_total, vecino.nombre))

        # Reconstrucción del camino más corto

        camino = []

        actual = destino

        while actual:

            camino.insert(0, actual)

            actual = padres.get(actual)

        print(f"La ruta más corta de {origen} a {destino} es:")

        print(" -> ".join(camino))

        print(f"Con una distancia total de {distancias[destino]} unidades.")

from collections import deque

class GrafoUniverso:

    def \_\_init\_\_(self):

        self.planetas = {}  # Diccionario para almacenar todos los nodos del grafo

    def agregar\_planeta(self, nombre):

        """Agrega un nuevo planeta al grafo."""

        if nombre not in self.planetas:

            self.planetas[nombre] = NodoPlaneta(nombre)

    def agregar\_ruta(self, origen, destino, distancia):

        """Crea una conexión entre dos planetas con una distancia específica."""

        if origen in self.planetas and destino in self.planetas:

            self.planetas[origen].agregar\_vecino(self.planetas[destino], distancia)

            self.planetas[destino].agregar\_vecino(self.planetas[origen], distancia)  # Grafo no dirigido

    def mostrar\_rutas(self):

        """Muestra las conexiones y distancias entre planetas."""

        for planeta in self.planetas.values():

            print(f"Planeta {planeta.nombre} está conectado con:")

            for vecino, distancia in planeta.vecinos.items():

                print(f"  - {vecino.nombre} a una distancia de {distancia} unidades.")

    def dfs(self, origen, destino):

        """Búsqueda en profundidad (DFS) para encontrar un camino entre dos planetas."""

        visitados = set()

        camino = []

        def \_dfs(actual):

            if actual == destino:

                return True

            visitados.add(actual)

            camino.append(actual)

            for vecino in self.planetas[actual].vecinos:

                if vecino.nombre not in visitados:

                    if \_dfs(vecino.nombre):

                        return True

            camino.pop()  # Retroceder si no se encuentra un camino

            return False

        if origen not in self.planetas or destino not in self.planetas:

            print("Uno o ambos planetas no existen.")

            return None

        if \_dfs(origen):

            print(f"Camino encontrado (DFS): {' -> '.join(camino + [destino])}")

        else:

            print("No se encontró un camino (DFS).")

    def bfs(self, origen, destino):

        """Búsqueda en amplitud (BFS) para encontrar un camino entre dos planetas."""

        visitados = set()

        cola = deque([(origen, [origen])])  # (planeta actual, camino recorrido)

        while cola:

            actual, camino = cola.popleft()

            if actual == destino:

                print(f"Camino encontrado (BFS): {' -> '.join(camino)}")

                return

            visitados.add(actual)

            for vecino in self.planetas[actual].vecinos:

                if vecino.nombre not in visitados:

                    cola.append((vecino.nombre, camino + [vecino.nombre]))

        print("No se encontró un camino (BFS).")

# Creación del universo

universo = GrafoUniverso()

universo.agregar\_planeta("Tierra")

universo.agregar\_planeta("Namek")

universo.agregar\_planeta("Vegeta")

universo.agregar\_planeta("Kaiosama")

universo.agregar\_ruta("Tierra", "Namek", 100)

universo.agregar\_ruta("Namek", "Vegeta", 50)

universo.agregar\_ruta("Tierra", "Kaiosama", 200)

universo.agregar\_ruta("Kaiosama", "Vegeta", 150)

# Mostrar las rutas

universo.mostrar\_rutas()

# Búsqueda de un camino con DFS

universo.dfs("Tierra", "Vegeta")

# Búsqueda de un camino con BFS

universo.bfs("Tierra", "Vegeta")

from collections import defaultdict, deque

class GrafoHabilidades:

    def \_\_init\_\_(self):

        self.habilidades = defaultdict(list)  # Diccionario para almacenar las relaciones entre habilidades

    def agregar\_habilidad(self, habilidad, prerequisito=None):

        """

        Agrega una habilidad al grafo, con un prerequisito opcional.

        """

        if prerequisito:

            self.habilidades[prerequisito].append(habilidad)  # prerequisito -> habilidad

        if habilidad not in self.habilidades:

            self.habilidades[habilidad] = []  # Aseguramos que la habilidad esté en el grafo

    def orden\_topologico(self):

        """

        Realiza el ordenamiento topológico de las habilidades.

        Devuelve una lista con el orden de entrenamiento.

        """

        # Calcular los grados de entrada de cada nodo

        grado\_entrada = {habilidad: 0 for habilidad in self.habilidades}

        for prerequisitos in self.habilidades.values():

            for habilidad in prerequisitos:

                grado\_entrada[habilidad] += 1

        # Cola para los nodos con grado de entrada 0

        cola = deque([habilidad for habilidad, grado in grado\_entrada.items() if grado == 0])

        orden = []

        while cola:

            actual = cola.popleft()

            orden.append(actual)

            for vecino in self.habilidades[actual]:

                grado\_entrada[vecino] -= 1

                if grado\_entrada[vecino] == 0:

                    cola.append(vecino)

        # Verificar si el grafo tiene ciclos

        if len(orden) != len(self.habilidades):

            print("Error: Existe un ciclo en las habilidades, no es posible realizar el ordenamiento topológico.")

            return None

        return orden

# Ejemplo: Planificación de habilidades

grafo = GrafoHabilidades()

# Agregar habilidades y sus prerequisitos

grafo.agregar\_habilidad("Kamehameha avanzado", "Kamehameha")

grafo.agregar\_habilidad("Kamehameha dual", "Kamehameha avanzado")

grafo.agregar\_habilidad("Kamehameha instantáneo", "Kamehameha")

grafo.agregar\_habilidad("Kamehameha eléctrico", "Kamehameha avanzado")

grafo.agregar\_habilidad("Kamehameha eléctrico avanzado", "Kamehameha eléctrico")

# Calcular el orden topológico

orden\_entrenamiento = grafo.orden\_topologico()

if orden\_entrenamiento:

    print("Orden de entrenamiento de habilidades:")

    print(" -> ".join(orden\_entrenamiento))

import heapq

class NodoPlaneta:

    def \_\_init\_\_(self, nombre):

        self.nombre = nombre

        self.vecinos = {}  # Diccionario para almacenar planetas conectados y su distancia

    def agregar\_vecino(self, planeta, distancia):

        self.vecinos[planeta] = distancia  # Agregar conexión con otro planeta

class GrafoUniverso:

    def \_\_init\_\_(self):

        self.planetas = {}  # Diccionario para almacenar todos los nodos del grafo

    def agregar\_planeta(self, nombre):

        """Agrega un nuevo planeta al grafo."""

        if nombre not in self.planetas:

            self.planetas[nombre] = NodoPlaneta(nombre)

    def agregar\_ruta(self, origen, destino, distancia):

        """Crea una conexión entre dos planetas con una distancia específica."""

        if origen in self.planetas and destino in self.planetas:

            self.planetas[origen].agregar\_vecino(self.planetas[destino], distancia)

            self.planetas[destino].agregar\_vecino(self.planetas[origen], distancia)  # Grafo no dirigido

    def buscar\_ruta\_mas\_corta(self, origen, destino):

        """Implementa el algoritmo de Dijkstra para encontrar la ruta más corta entre dos planetas."""

        if origen not in self.planetas or destino not in self.planetas:

            print("Uno o ambos planetas no existen.")

            return None

        # Inicialización de distancias y cola de prioridad

        distancias = {planeta: float('inf') for planeta in self.planetas}

        distancias[origen] = 0

        cola = [(0, origen)]  # (distancia acumulada, nodo actual)

        padres = {origen: None}

        while cola:

            distancia\_actual, planeta\_actual = heapq.heappop(cola)

            if planeta\_actual == destino:

                break

            for vecino, distancia in self.planetas[planeta\_actual].vecinos.items():

                distancia\_total = distancia\_actual + distancia

                if distancia\_total < distancias[vecino.nombre]:

                    distancias[vecino.nombre] = distancia\_total

                    padres[vecino.nombre] = planeta\_actual

                    heapq.heappush(cola, (distancia\_total, vecino.nombre))

        # Reconstrucción del camino más corto

        camino = []

        actual = destino

        while actual:

            camino.insert(0, actual)

            actual = padres.get(actual)

        return camino, distancias[destino]

    def recolectar\_esferas(self, origen, planetas\_esferas):

        """

        Encuentra la mejor ruta para recolectar todas las Esferas del Dragón.

        Utiliza Dijkstra para calcular las distancias entre el origen y cada planeta con esferas.

        """

        visitados = set()

        ruta\_completa = []

        distancia\_total = 0

        actual = origen

        while planetas\_esferas:

            distancias = []

            for planeta in planetas\_esferas:

                camino, distancia = self.buscar\_ruta\_mas\_corta(actual, planeta)

                distancias.append((distancia, camino))

            # Seleccionar la siguiente esfera más cercana

            distancias.sort(key=lambda x: x[0])  # Ordenar por distancia

            distancia\_minima, mejor\_camino = distancias[0]

            # Actualizar variables

            ruta\_completa.extend(mejor\_camino[:-1] if ruta\_completa else mejor\_camino)

            distancia\_total += distancia\_minima

            actual = mejor\_camino[-1]

            planetas\_esferas.remove(actual)

        print("Ruta para recolectar todas las Esferas del Dragón:")

        print(" -> ".join(ruta\_completa))

        print(f"Distancia total recorrida: {distancia\_total} unidades.")

*Conclusión*

El diseño del juego de Dragon Ball se centra en la eficiencia y en la capacidad de expansión. Cada estructura de datos fue seleccionada para cumplir con un propósito específico, optimizando tanto el rendimiento como la escalabilidad del juego.

La clase Personaje y los patrones de diseño aplicados permiten una implementación modular, mientras que las estructuras avanzadas, como árboles binarios y colas de prioridad, aseguran que el juego se mantenga rápido y emocionante.

Este enfoque garantiza un juego atractivo, donde la evolución y los torneos se manejan de forma eficaz y dinámica.