Universidad Nacional Guillermo Brown
Cátedra: Estructuras de Datos en Python
II Cuatrimestre 2024

Trabajo Final Estructura de Datos

Integrantes del Grupo

1. Florencia Molina - dni: 37.834.497
2. Junior Flores - dni: 95.122.461
3. Matias Gonzalez - dni: 37.477.163

Enlace al repositorio Repositorio.

Tabla de Contenidos

Introducción

Descripción

Este proyecto tiene como objetivo, desarrollar en grupo un juego de Dragon Ball. El propósito de este proyecto es realizar un juego para la cátedra de estructura de datos , y poder implementar todos los temas que vimos y luego realizar una defensa individual de dicho proyecto. Este trabajo tiene como carácter evaluativo con nota que sera sumada al primer parcial.
El proyecto se puede por grupos de 2 o 3 personas, utilizando el lenguaje python y librerías que sean necesarias.
Fecha de entrega 28/11 18hs.

Objetivos

- Creación de un un juego por turnos de DBZ.
- Creación y gestión de personajes, batallas y evolución de poderes.
- Poder gestionar una forma de trabajo en grupo para la realización del proyecto.
- Entregar el trabajo en tiempo y forma.

Planificación

Análisis del mercado

La relevancia de realizar este proyecto de un juego de DBZ es la de implementar todo lo aprendido en la cátedra involucrando compañeros y la búsqueda de una forma de trabajo en común para el realización del proyecto.
Poder jugar diferentes pensamientos y avanzar en grupo, divertirnos en la creación del mismo.

Publico Objetivo:

- Docente de la cátedra.
- Compañeros futuros colegas que quieran implementar mejoras.
- Publico general de video juegos.

Alcance del proyecto

Funciones principales

- Elegir un personaje.
- El personaje tiene que subir de niveles.
- El personaje tiene que adquirir nuevas habilidades.
- Realizar entrenamientos.
- Jugar un torneo.
- Jugar una batalla aleatoria.
- Los personajes pueden recorrer planetas del universo de dragon ball z.
- Los personajes pueden buscar las esferas del dragon.

Límites del proyecto

- Crear un juego que sea por turnos.
- Crear una interfaz de usuario que sea por terminal.

Metodología

Metodología y proceso de trabajo optamos en trabajar con reuniones semanales a disponibilidad de cada uno.

Tecnologías

- Visual studio code
- Python
- Meet
- Drive
- Discord
- Repositorio de Github
- Grupo de WhatsApp

AGENDA DE REUNIONES
Con el fin de llevar encuentros semanales para conversar acerca del trabajo final, se crearon distintas vías de comunicación entre las cuales se nombran las siguientes:

- Chat grupal en Whatsapp para facilitar la comunicación entre los integrantes.
- Drive para compartir y editar documentos.
- Canal en Discord para compartir material y llevar a cabo reuniones.
- Repositorio en GitHub para subir actividades del proyecto.
- Reuniones a través de Meet para mantener conversaciones.
  A continuación se deja detallado el cronograma de encuentros a lo largo del trabajo:
  Semana 1
  Viernes 01/11/24: Se llevó a cabo el primer encuentro en Meet donde cada integrante se presentó y se delinearon las primeras ideas acerca del trabajo final. Fue en esta instancia donde se acordó crear un canal en Discord y un repositorio en GitHub para compartir material y actividades realizadas por cada integrante.
  Semana 2
  Viernes 08/11/24: Mediante una reunión de Meet se logró avanzar en las primeras unidades del trabajo. En una puesta en común, se dialogó y se brindaron opiniones sobre lo trabajado hasta el momento.
  Semana 3
  Viernes 15/11/24: Se utilizó por primera vez el canal de voz de Discord en el cual se fueron compartiendo propuestas y dificultades y, al mismo tiempo, se buscaron soluciones en conjunto para abordar dichas problemáticas. Además, se acordó seguir avanzando en las unidades siguientes.
  Semana 4
  Viernes 21/11/24: Se realizo una reunion donde cada uno mostro en lo que estuvo trabajando y mostrar los cambios que cada uno estuvo haciendo para luego juntarlos en el repositorio.
  Semana 5

Viernes 29/11/24: En esta fecha tenemos planeado juntarnos para poder charlar sobre el proyecto en general

Análisis

Especificación de requerimientos

Requerimientos funcionales

1. Sistema de combate:

- Implementar un sistema de combate estratégico que permita la interacción del jugador con diferentes habilidades y movimientos desde la terminal.
- Desarrollar habilidades únicas para cada clase de personaje.

2. Progresión del Personaje:

- Crear un sistema de niveles y experiencia que permita al jugador avanzar y mejorar a medida que avanza en el juego.
- Desarrollar un sistema de entrenamiento que permita mejorar las habilidades del personaje.
- Desarrollar un sistema que el personaje pueda viajar por diferentes planetas y poder entrenar en ellos.

3. Interacción del Jugador:

- Diseñar una interfaz de usuario desde la terminal que permite al jugador explorar el juego y pueda participar en múltiples desafíos.

Requerimientos no funcionales

1. Rendimiento y Optimización

- Garantizar un rendimiento fluido y estable del juego, evitando errores de ejecución.
- Optimizar el uso de recursos para garantizar que el juego funcione de manera eficiente en dispositivos de PC.

2. Estilo Visual y Narrativa:

- Desarrollar un menu intuitivo para el usuario.
- Crear una narrativa atractiva que sumerge al jugador en el mundo del juego.

3. Estabilidad:

- Realizar pruebas para garantizar la estabilidad del juego, minimizando errores, cierres inesperados y fallos técnicos.

Historia de usuarios y criterios de aceptación

En esta sección se estudia a detalle las distintas audiencias a las que va dirigido el juego

- Docente de la cátedra : Quien nos dará una devolución critica y con nota del trabajo realizado.
- Compañeros de la clase, futuros colegas que nos darán su opinion o podrán mejorar el proyecto aportando sus propias ideas.
- Jugadores RPG tradicionales: Personas que disfrutan de la profundidad narrativa y el progreso de los personajes, aspectos típicamente asociados con los RPG clásicos.
- Público general de videojuegos: Este segmento, aunque amplio, puede ser atraído por una narrativa atractiva, mecánicas de juego interesantes y un estilo visual único.

Diseño

- **Herramientas y entorno de desarrollo**
    - Visual studio code
    - Git
    - Github
- **Mecánica y dinámica del juego**
    - Sistema de habilidades, los personajes utilizan habilidades exclusivos de las diferentes clases de personajes. Estas clases están representadas en una lista de habilidades.
    - Sistema de combate, los jugadores pueden utilizar ataques normales, o ataques especiales (habilidades), o cargar energía (ki) para la utilización de estas habilidades.
      El sistema esta diseñado para que sea un combate por turnos hasta que algún jugador se quede sin vida. Por cada turno se actualiza la vida, energía(ki), experiencia, nivel de los personajes.
    - Progresión del personaje, el protagonista avanza de nivel por medio de la experiencia(combates, entrenamientos), y va adquiriendo nuevas habilidades.
    - Una variedad de personajes están disponibles en el juego para elección del usuario. Estas clases ya están predefinidas en la interfaz permitiendo al usuario empezar el juego rápidamente.
    - Exploración y misiones el personaje puede explorar distintos planetas y realizar entrenamientos o misiones.
- **Arquitectura general de la aplicación**
- La arquitectura de la aplicacion , la decidimos dividir en 2 partes, una carpet llamada clase, donde estan todas las clases del juego
- luego una carpeta llamada personajes donde en esta carpeta hay 2 archivos, donde se intancian personajes para la utilizacion del juego.
- luego para la realizacion de pruebas del juego se creo un archivo por fuera de las carpetas donde llama a todas las clases y personajes necesario para realizar pruebas aisladas

- Se creo una carpeta llamada trabajo practico donde estan todos los recursos que utilizamos para la creacion del trabajo practico , como imagenes, videos diagramas

Desarrollo

Clases e Interfaces

Clase Personaje

Introducción

La clase Personaje modela las características y comportamientos de un personaje en el juego basado en Dragon Ball. Incluye atributos esenciales para el combate, transformación, progresión de nivel, y exploración de planetas, junto con métodos que definen su funcionalidad

Atributos del Personaje
Cada atributo representa una característica específica del personaje.

```
- nombre: Nombre del personaje (tipo str).
- vida: Puntos de vida del personaje (tipo int).
- raza: Raza a la que pertenece (tipo str).
- estado: Estado actual del personaje ("Normal", "defensivo", etc.).
- ki: Nivel de Ki actual del personaje (tipo int).
- max_ki: Máximo nivel de Ki permitido en la transformación actual.
- transformaciones: Árbol de transformaciones del personaje.
- transformacion_actual: Transformación actual activa.
- habilidades: Árbol de habilidades disponibles.
- exp: Experiencia acumulada del personaje.
- max_exp: Experiencia necesaria para alcanzar el próximo nivel.
- nivel_de_poder: Nivel de poder actual del personaje.
- nivel: Nivel del personaje (tipo int, por defecto 1).
- max_ki_base: Máximo nivel de Ki base, sin transformaciones (tipo int, por
defecto 10,000).
- combates_ganados: Número de combates ganados (tipo int, por defecto 1).
- planeta_actual: Planeta donde se encuentra el personaje actualmente (tipo
str, por defecto "Tierra").
```

Métodos

Métodos básicos de la clase

init

```
-  Constructor de la clase. Inicializa todos los atributos y realiza una
evolución inicial de poder basada en los combates ganados.
```

Métodos de combate

ganar_combate(self)

```
- Registra la victoria de un combate e incrementa los combates ganados. Además, evoluciona el poder del personaje.
```

ataque_basico(self,enemigo)

```
- Realiza un ataque básico sobre el enemigo, infligiendo daño igual al nivel de poder del personaje. Requiere al menos 1,000 de Ki.
```

recibir_daño(self,daño_recibido, personaje)

```
- Reduce la vida del personaje dependiendo del daño recibido. Si el personaje está en estado defensivo, el daño se reduce a la mitad.
```

defender(self)

```
- Cambia el estado del personaje a defensivo, reduciendo a la mitad el daño recibido en el próximo ataque.
```

usar_habilidad(self,habilidad_nombre, enemigo)

```
- Utiliza una habilidad específica del árbol de habilidades, siempre que el personaje tenga suficiente Ki y cumpla con los requisitos de transformación.
```

Métodos de transformación

transformarse(self,nombre_transformacion)

```
- Realiza la transformación del personaje si cumple con los requisitos de Ki y transformación previa. Multiplica el nivel de poder y recalcula el Ki máximo.
```

Métodos de progresión

evolucionar_poder(self,combates_ganados=None, multiplicador=2)

```
- Método recursivo que incrementa el nivel de poder basado en los combates ganados o un multiplicador especificado.
```

subir_nivel(self)

```
- Incrementa el nivel del personaje si la experiencia actual supera el máximo necesario. Actualiza atributos como vida, Ki máximo, y experiencia máxima.
```

calcular_max_ki(self, multiplicador=None)

```
- Calcula el máximo de Ki basado en el nivel del personaje y un multiplicador opcional.
```

actualizar_max_exp(self)

```
- Recalcula la experiencia máxima necesaria para subir al siguiente nivel.
```

Métodos de exploración

viajar_a(self,grafo, planeta_destino, metodo_busqueda='bfs')

```
- Permite al personaje viajar a un planeta conectado al actual, utilizando un método de búsqueda (BFS o DFS) para encontrar rutas.
```

escapar_a(self,grafo, planeta_destino)

```
- Cambia directamente el planeta actual del personaje, siempre que el planeta destino exista en el grafo.
```

Métodos de utilidad

mostrar_stats(self)

```
- Muestra las estadísticas actuales del personaje, incluyendo vida, Ki, nivel, transformaciones, habilidades, y experiencia.
```

cargar_ki(self,incremento)

```
- Incrementa gradualmente el nivel de Ki hasta alcanzar el máximo permitido,
mostrando el progreso en tiempo real.
```

incrementar_atributos(self)

```
- Aumenta los atributos de vida y nivel de poder proporcionalmente al nivel del
personaje.
```

Funcionalidades clave

- Transformaciones y habilidades: El personaje puede evolucionar y utilizar habilidades dependiendo de su Ki y transformación actual.
- Progresión de nivel: Gana experiencia al participar en combates, lo que permite mejorar sus atributos y desbloquear nuevas habilidades.
- Exploración: Viaja entre planetas conectados en el grafo del juego, enriqueciendo la experiencia de exploración.
- Combate dinámico: Incluye estados defensivos, ataques básicos, y el uso de habilidades únicas.

Analisis Algoritmico

Cargar Ki
El método cargar_ki incrementa el nivel de ki del personaje en intervalos constantes hasta alcanzar su capacidad máxima (max_ki).

Complejidad Temporal

- Mejor caso: Si ki ya está al máximo, la complejidad es O(1) porque no entra en el bucle.
- Peor caso: Si ki inicia desde 0 y se carga en incrementos constantes, la complejidad es O(n), donde n = max_ki / incremento.
- Ejemplo: Para max_ki = 1000 y incremento = 100, habrá 10 iteraciones.
  La pausa de time.sleep no afecta la complejidad algorítmica, aunque ralentiza el tiempo de ejecución.

Complejidad Espacial

- O(1): Solo se utilizan variables locales como incremento y el atributo ki.

Conclusión
El método es lineal respecto al rango entre ki inicial y max_ki. Es eficiente para casos donde el rango de carga no es extremadamente grande.

Evolucionar poder

El método evolucionar_poder es una herramienta clave en el diseño del juego, ya que permite simular cómo crece la fuerza de un personaje tras ganar combates o experimentar transformaciones especiales. El análisis de su eficiencia nos ayuda a entender cómo maneja los recursos del sistema y qué tan bien está optimizado para estos cálculos.

- Cuando se pasa un número a combates_ganados, el método se ejecuta de forma recursiva. Por cada combate, el nivel de poder del personaje se multiplica por un factor llamado multiplicador, que determina cuánto crece su fuerza. Además, se suma experiencia (por ejemplo, 50 puntos) y se verifica si es necesario subir de nivel mediante el método subir_nivel. Después de eso, el método se llama a sí mismo, reduciendo en uno el número de combates restantes. Este proceso se repite hasta que no queden combates por simular.

- Este enfoque recursivo tiene un comportamiento predecible: el número de veces que se ejecuta depende directamente de la cantidad de combates ganados. Si hay 10 combates, el método hará 10 llamadas, más una final cuando los combates lleguen a 0. Por cada una de esas llamadas, las operaciones internas (como la multiplicación para calcular el poder o la suma de experiencia) son rápidas y siempre tardan lo mismo, por lo que podemos decir que el tiempo total que toma el método crece de manera proporcional al número de combates. En otras palabras, si duplicamos los combates, el tiempo que tomará también se duplicará.

- En cuanto al espacio de memoria, cada vez que se hace una llamada recursiva, se guarda información en la "pila" del programa, que es una parte de la memoria reservada para este tipo de operaciones. Si hay muchas llamadas consecutivas (por ejemplo, 100 combates), esta pila puede ocupar bastante espacio. Sin embargo, en un caso típico con unos pocos combates, esto no debería ser un problema significativo.

- Cuando combates_ganados es None, el método sigue un camino diferente. En lugar de usar recursión, aplica una transformación directa: el poder del personaje se multiplica una sola vez, y otros atributos como el máximo de ki (max_ki) y la vida también se ajustan en proporción. Este tipo de cálculo es mucho más rápido, ya que no necesita repetirse varias veces. Por eso, en este caso, tanto el tiempo que tarda como el espacio que ocupa en memoria son constantes, sin importar cuán grande sea el multiplicador.

- Lo interesante de este método es que combina dos tipos de crecimiento: uno gradual y otro instantáneo, lo que lo hace muy útil para un juego donde el progreso del personaje debe sentirse natural pero también emocionante en momentos clave. Sin embargo, usar recursión para manejar los combates puede ser menos eficiente si se planean muchos enfrentamientos seguidos, porque podría consumir más memoria de la necesaria. Esto se puede solucionar limitando el número de combates por llamada o reemplazando la recursión por un bucle, aunque esto último haría que el código sea un poco menos elegante.

Conclusión

El método evolucionar_poder está diseñado de manera eficiente para manejar la evolución del personaje en diferentes escenarios, equilibrando crecimiento gradual y transformaciones puntuales. Su uso de recursión permite un flujo lógico y natural para el manejo de múltiples combates consecutivos, pero a costa de utilizar más espacio en la pila de llamadas, lo que puede ser un riesgo si se manejan grandes

cantidades de combates en una sola ejecución. Por otro lado, su enfoque para las transformaciones directas es rápido y eficiente, ya que no depende de recursión ni de procesos iterativos.

Complejidad

En términos de complejidad, el método es lineal (O(n)) para escenarios con recursión, y constante (0(1)) para transformaciones, lo que lo hace razonablemente óptimo para un videojuego. Sin embargo, si se espera que el número de combates sea muy alto, una alternativa como un bucle podría ser más segura y eficiente.

En general, este método es una solución práctica y funcional para un juego donde el progreso del personaje debe ser dinámico y emocionante. Mantiene un equilibrio entre un diseño limpio y una implementación eficiente, lo que lo hace ideal para simular el crecimiento y las mecánicas de poder en un entorno interactivo.

Clase Saiyajin

La clase Saiyajin hereda de la clase base Personaje, y representa una raza específica de personajes en el sistema del juego. Los Saiyajin tienen características únicas como transformaciones específicas y habilidades avanzadas.

class Saiyajin(Personaje):

```
- La clase extiende la funcionalidad de la clase Personaje, adaptándola para
incluir propiedades únicas de los Saiyajin.
```

Método Constructor

def init(self, nombre: str, arbol_habilidades_data, combates_ganados):

```
- Descripción:
    - Inicializa un objeto Saiyajin con los valores base típicos de esta raza,
como transformaciones, habilidades y estadísticas iniciales.
    - Utiliza árboles específicos para habilidades y transformaciones.

- Parámetros:

    - nombre (str): Nombre del personaje Saiyajin.
    - arbolhabilidadesdata (dict): Datos necesarios para construir el árbol de
habilidades.
    - combates_ganados (int): Cantidad de combates ganados por el Saiyajin.

- Implementación:

    - Llama al constructor de la clase base (super().__init) para configurar
los atributos heredados.
    - Inicializa un árbol de transformaciones exclusivo de los Saiyajin
```

```
mediante el método crear_arbol_transformaciones.
    - Configura las habilidades utilizando crear_arbol_habilidades.
    - Calcula el nivel de poder inicial basado en los combates ganados.
```

Métodos Públicos

str(self)

```
- Descripción:
    - Devuelve una representación en forma de cadena del Saiyajin, mostrando su
nombre y nivel de poder.

- Salida:

    - Una cadena como: "Goku, Nivel de poder: 9000".
```

ataque_especial(self,enemigo)

```
- Descripción:
    - Realiza un ataque especial propio de los Saiyajin. Este ataque inflige un
daño significativo al enemigo si el personaje tiene suficiente Ki.

- Parámetros:

    - enemigo (Personaje): Instancia del personaje enemigo que recibe el daño.
- Efectos:

    - Reduce el Ki del Saiyajin en 5000.
    - Inflige daño equivalente al doble de su nivel de poder.
- Restricciones:

    - Si el Ki del Saiyajin es menor a 5000, no puede ejecutar el ataque.
```

Métodos Auxiliares

crear_arbol_transformaciones(self)

```
- Descripción:
    - Genera y devuelve un árbol de transformaciones que representa la
jerarquía de formas del Saiyajin.

    - Transformaciones Incluidas:

        - Base: Sin requisitos, forma inicial.
        - Super Saiyajin (SSJ): Requiere 2000 de Ki y estar en la forma base.
```

```
        - Super Saiyajin 2 (SSJ2): Requiere 5500 de Ki y estar en SSJ.
        - Super Saiyajin 3 (SSJ3): Requiere 7000 de Ki y estar en SSJ2.
        - Super Saiyajin 4 (SSJ4): Requiere 8500 de Ki y estar en SSJ3.
    Estructura del Árbol:

        - Cada transformación es un nodo (NodoTransformacion) con enlaces
jerárquicos
```

crear_arbol_habilidades(self,arbol_habilidades_data)

```
- Descripción:
    - Construye el árbol de habilidades del Saiyajin utilizando datos
proporcionados.

- Parámetros:

    - arbol_habilidades_data (dict): Información estructurada para generar las
habilidades del personaje.
- Salida:

    - Un objeto ArbolHabilidades configurado según los datos recibidos.
```

Clase Androide

La clase Androide hereda de la clase base Personaje y representa una raza única con características distintivas. Los Androides destacan por su capacidad de transformarse en fases avanzadas y almacenar grandes cantidades de Ki.

class Androide(Personaje):

```
- Esta clase extiende la funcionalidad de Personaje para incluir
transformaciones y habilidades específicas de los Androides.
```

Método Constructor

def init(self, nombre: str, arbol_habilidades_data, combates_ganados=0):

```
- Descripción:
    - Inicializa un objeto Androide con atributos predeterminados que reflejan
las capacidades de los Androides, como un almacenamiento de Ki más alto y una
serie de transformaciones.

- Parámetros:
```

```
    - nombre (str): Nombre del personaje Androide.
    - arbol_habilidades_data (dict): Datos para construir el árbol de
habilidades del Androide.
    - combates_ganados (int, opcional): Cantidad de combates ganados. Valor
predeterminado: 0.
- Implementación:

    - Llama al constructor de la clase base (super().init) para configurar los
atributos heredados.
    - Asigna un máximo de Ki significativamente mayor al de otras razas.
    - Configura el árbol de transformaciones y habilidades del Androide.
    - Calcula el nivel de poder inicial según los combates ganados.
```

Métodos Públicos

str()

```
- Descripción:
    - Devuelve una representación en forma de cadena del Androide, mostrando su
nombre y nivel de poder.

- Salida:

    - Una cadena como: "Androide17 Nivel de poder: 9000".
```

ataque_especial(self,enemigo)

```
- Descripción:
    - Ejecuta un ataque especial característico de los Androides. El ataque
inflige un daño significativo al enemigo si el Androide tiene suficiente Ki.

-Parámetros:

    - enemigo (Personaje): Instancia del personaje enemigo que recibirá el
daño.
    - Efectos:

    - Inflige daño basado en el doble del nivel de poder del Androide.
    - Reduce el Ki del Androide en 5000.
    - Restricciones:

    - Si el Ki del Androide es menor a 5000, no puede realizar el ataque.
```

Métodos Auxiliares

crear_arbol_transformaciones(self)

```
- Descripción:
    - Genera y devuelve un árbol de transformaciones que representa las fases
del Androide.

- Transformaciones Incluidas:

    - Base: Sin requisitos, forma inicial.
    - Fase 1: Requiere 4000 de Ki y estar en la forma base.
    - Fase 2: Requiere 5500 de Ki y estar en Fase 1.
    - Fase 3: Requiere 7000 de Ki y estar en Fase 2.
    - Fase 4: Requiere 8500 de Ki y estar en Fase 3.
- Estructura del Árbol:

    - Cada transformación es un nodo (NodoTransformacion) con relaciones
jerárquicas.
```

crear_arbol_habilidades(self,arbol_habilidades_data)

```
- Descripción:
    - Construye el árbol de habilidades del Androide utilizando datos
proporcionados.

- Parámetros:

    - arbol_habilidades_data (dict): Información estructurada para generar las
habilidades del personaje.
- Salida:

    - Un objeto ArbolHabilidades configurado según los datos recibidos.
```

Clases Torneo y Menu Pricipal

Introducción

Este sistema simula un torneo de artes marciales entre personajes utilizando estructuras de datos avanzadas y modos de juego diversos. El menú principal permite acceder a distintas modalidades como el torneo, batallas rápidas, exploración de esferas del dragón, y visualización del orden de habilidades.

Clases principales

Clase Torneo

```
- Descripción:
    - Simula un torneo donde los personajes luchan entre sí en enfrentamientos
1 contra 1 hasta que quede un único ganador.
```

```
- Atributos:
    - cola_prioridad: Instancia de una cola de prioridad para organizar los
enfrentamientos según el nivel de poder de los personajes.
    - ganador: Almacena el personaje que resulta ganador del torneo.
```

Métodos:

init(self, personajes)

```
- Descripción: Inicializa el torneo creando la cola de prioridad con los
personajes participantes.
- Parámetros:
    - personajes: Lista de personajes participantes.
```

iniciar_torneo(self)

```
- Descripción: Ejecuta el torneo, organizando enfrentamientos entre los
personajes hasta determinar un ganador.
- Flujo:
    - Si solo queda un personaje en la cola, se declara ganador.
    - Selecciona dos contrincantes para enfrentarse.
    - Simula el combate con el método simular_combate y retorna el ganador.
    - El ganador mejora sus estadísticas y vuelve a la cola.
```

simular_combate(self, juego)

```
- Descripción: Simula un combate entre dos personajes alternando turnos.
- Parámetros:
    - juego: Instancia de la clase Juego, que representa el enfrentamiento.
    - Retorno: Personaje ganador del combate.
```

Clase MenuPrincipal

```
- Descripción:
    - Proporciona la interfaz principal para interactuar con las
funcionalidades del sistema.

- Atributos:
    - personajes: Lista de personajes disponibles.
    - grafo: Grafo que representa conexiones entre ubicaciones o elementos
(como esferas del dragón).
    - habilidades: Conjunto de habilidades organizadas, utilizado para mostrar
su orden.
```

Métodos:

init(self, personajes, grafo, habilidades)

```
- Descripción: Inicializa el menú principal con los datos necesarios para
gestionar las funcionalidades del sistema.
- Parámetros:
    - personajes: Lista de personajes.
    - grafo: Grafo que conecta rutas o esferas.
    - habilidades: Estructura de datos para gestionar habilidades.
```

mostrar_menu(self)

```
- Descripción: Muestra las opciones del menú principal e invoca las
funcionalidades correspondientes según la elección del usuario.
- Opciones disponibles:
    - Modo Torneo: Simula un torneo completo.
    - Modo Batalla Rápida: Ejecuta un combate rápido entre dos personajes
seleccionados aleatoriamente.
    - Buscar Esferas del Dragón: Explora las esferas utilizando un personaje.
    - Ver Orden de Habilidades: Muestra un orden topológico de las habilidades.
    - Salir: Cierra el menú.
```

modo_batalla_rapida(self)

```
- Descripción: Selecciona dos personajes al azar y ejecuta un combate rápido
entre ellos.
- Flujo:
    - Selecciona al jugador y un contrincante diferente.
    - Inicia un juego de combate rápido.
```

buscar_esferas(self)

```
- Descripción: Permite a un personaje explorar un grafo para encontrar las
esferas del dragón.
- Flujo:
    - Selecciona un personaje al azar.
    - Inicia la exploración con la instancia Juego.
```

ver_habilidades(self)

```
- Descripción: Muestra las habilidades en un orden topológico.
- Flujo:
    - Realiza el ordenamiento topológico de las habilidades.
    - Imprime el resultado en pantalla.
    - Ejecución del Código
```

Modo Torneo:
Inicia con todos los personajes en una cola de prioridad.
Ejecuta enfrentamientos consecutivos hasta determinar un único ganador.

Batalla Rápida:
Selecciona dos personajes aleatoriamente.
Realiza un único combate entre ellos.

Búsqueda de Esferas:
Simula la exploración de un grafo con un personaje.
Puede expandirse para agregar misiones o desafíos.

Orden de Habilidades:
Utiliza un algoritmo de ordenamiento topológico para determinar dependencias entre habilidades.

Conclusión
Este sistema modular permite gestionar dinámicas de combate, exploración y habilidades de manera eficiente. Su diseño orientado a objetos facilita la extensibilidad, como la incorporación de nuevos modos de juego o personajes.

Juego de Combate y Exploración de Esferas del Dragón

Descripción General
Este código implementa un juego de combate basado en personajes del universo de Dragon Ball, con la posibilidad de explorar el mundo en busca de las Esferas del Dragón. El jugador puede recolectar las esferas a través de una exploración en un grafo de planetas, luchar contra un oponente, transformar su personaje y usar habilidades, y finalmente realizar un deseo al recolectar todas las esferas.

Clases

Clase Juego

La clase Juego gestiona los aspectos del combate entre el jugador y la máquina, la recolección de esferas y la ejecución de deseos. Incluye métodos para gestionar los turnos de combate, el estado del jugador y la máquina, y la búsqueda de las esferas.

Atributos:

jugador: Personaje

```
- Descripción: Este atributo contiene un objeto de la clase Personaje que
representa al jugador en el juego. El jugador puede realizar acciones como
atacar, usar habilidades, cargar ki, y transformarse.
- Tipo: Personaje
```

maquina: Personaje

```
- Descripción: Este atributo contiene un objeto de la clase Personaje que
representa al oponente en el combate. La máquina (oponente) sigue las mismas
reglas que el jugador, como cargar ki, usar habilidades, y transformarse.
- Tipo: Personaje
```

grafo: GrafoDragonBall

```
- Descripción: Este atributo contiene un objeto de la clase GrafoDragonBall,
que representa el mundo del juego mediante un grafo de planetas. Este grafo es
utilizado para la exploración de las esferas del dragón, donde cada nodo es un
planeta y las aristas son los caminos entre ellos.
- Tipo: GrafoDragonBall
```

transformaciones: ArbolTransformaciones

```
- Descripción: Este atributo contiene un objeto de la clase
ArbolTransformaciones, que almacena las posibles transformaciones que el
jugador y la máquina pueden utilizar en el combate.
- Tipo: ArbolTransformaciones
```

esferas_recolectadas: int

```
- Descripción: Este atributo lleva el conteo de las esferas del dragón que el
jugador ha recolectado durante el juego. El jugador debe recolectar las 7
esferas para poder hacer un deseo.
- Tipo: int
- Valor inicial: 0
```

Metodos

init(self, jugador: Personaje, maquina: Personaje, grafo: GrafoDragonBall, transformaciones: ArbolTransformaciones)

```
- Descripción: Constructor que inicializa un nuevo juego con un jugador, una máquina, un grafo de planetas y un árbol de transformaciones.
- Parámetros:
    - jugador: Objeto de la clase Personaje.
    - maquina: Objeto de la clase Personaje.
    - grafo: Objeto de la clase GrafoDragonBall.
    - transformaciones: Objeto de la clase ArbolTransformaciones.
    - Valor de retorno: Ninguno. Inicializa el estado del juego.
```

limpiar_pantalla(self)

```
- Descripción: Limpia la pantalla de la consola para mejorar la visualización del juego.
- Acción: Ejecuta un comando del sistema operativo (cls en Windows, clear en UNIX).
- Valor de retorno: Ninguno. Solo limpia la pantalla
```

iniciar_combate(self)

```
- Descripción: Inicia un combate entre el jugador y la máquina, gestionando los turnos de ambos hasta que uno de los dos sea derrotado.
- Acción: Alterna entre los turnos del jugador y la máquina, mostrando estadísticas y realizando acciones hasta que uno de los dos personajes pierda toda su vida.
- Valor de retorno: Devuelve las estadísticas del ganador cuando termina el combate (al derrotar al oponente).
```

turno_jugador(self)

```
- Descripción: Gestiona el turno del jugador, permitiéndole elegir entre usar una habilidad, cargar ki o transformarse.
- Acción:
    - Muestra las estadísticas actuales del jugador y el oponente.
    - Permite al jugador elegir entre usar una habilidad, cargar ki o transformarse. La acción se valida para asegurarse de que el jugador tenga suficiente ki o que la transformación sea válida.
- Valor de retorno: Ninguno. El turno del jugador se ejecuta y se muestra en pantalla.
```

turno_maquina(self)

```
- Descripción: Gestiona el turno de la máquina, eligiendo aleatoriamente entre transformar al oponente, usar una habilidad o cargar ki.
- Acción:
    - Si la máquina tiene menos de 100 ki, carga ki.
    - Si tiene suficiente ki y una transformación disponible, se transforma.
    - Si tiene suficiente ki, usa una habilidad aleatoria.
- Valor de retorno: Ninguno. El turno de la máquina se ejecuta y se muestra en pantalla.
```

busqueda_habilidades(self, hijos)

```
- Descripción: Realiza una búsqueda recursiva de habilidades dentro de los hijos de un nodo y las muestra.
- Parámetros:
    - hijos: Lista de nodos que representan las habilidades disponibles.
- Valor de retorno: Devuelve una lista de habilidades encontradas en la búsqueda.
```

explorar_esferas(self)

```
- Descripción: Permite al jugador buscar las Esferas del Dragón viajando entre planetas en el grafo. Utiliza los algoritmos DFS y Dijkstra para explorar.
- Acción:
    - El jugador elige un planeta de origen y busca una esfera utilizando DFS.
    - Luego, utiliza Dijkstra para encontrar el planeta más cercano con una esfera.
    - El jugador recoge las esferas hasta completar las 7.
- Valor de retorno: Ninguno. Una vez que el jugador recolecta todas las esferas, llama a hacer_deseo.
```

hacer_deseo(self)

```
- Descripción: Permite al jugador realizar un deseo una vez que haya recolectado las 7 Esferas del Dragón.
- Acción:
    - El jugador puede elegir entre aumentar su poder o incrementar sus atributos.
    - El jugador tiene hasta tres intentos para hacer un deseo válido.
- Valor de retorno: Ninguno. El deseo se cumple o se rechaza dependiendo de la entrada del jugador.
```

Función seleccionar_contrincante

seleccionar_contrincante(personajes)

```
- Descripción: Selecciona aleatoriamente un contrincante de una lista de
personajes.
- Parámetros:
    - personajes: Lista de personajes de tipo Personaje.
- Valor de retorno: Devuelve un objeto de tipo Personaje seleccionado
aleatoriamente de la lista de personajes.
```

Requisitos del Sistema

Este código depende de que las clases Personaje, GrafoDragonBall, ArbolTransformaciones, y otras sean definidas correctamente para que todo funcione correctamente. Además, los comandos del sistema (cls para Windows y clear para sistemas UNIX) deben ser ejecutados en un entorno de consola compatible.

Estructuras Recursivas

Concepto del Algoritmo

-Experiencia: Cada vez que un personaje gana una pelea, acumula experiencia. Al alcanzar un umbral de experiencia, puede subir de nivel.

- Nivel: El nivel del personaje determina su máximo de Ki y puede influir en el multiplicador de poder.
- Máximo Ki: El máximo de Ki aumenta con cada nivel, lo que permite al personaje realizar más ataques especiales o cargar Ki más rápidamente.

Estructura del Algoritmo

La clase implementa un sistema para gestionar las características y evolución de un personaje en un juego o simulador. Utiliza métodos recursivos para calcular la evolución del poder tras combates y permite incrementar atributos como velocidad, defensa y fuerza al subir de nivel. Los métodos están diseñados para ser claros y fáciles de entender, facilitando su uso en un contexto más amplio dentro del código.

Puede Realizar 2 situaciones, dependiendo el parametro que le pases:
Parametros

```
- combates_ganados (int, opcional): El número de combates ganados por el
personaje. Si es None, se asume que se trata de una transformación y se debe
pasar el multiplicador de dicha transformacion

- multiplicador (float): Un valor que multiplica el poder actual del personaje
durante las transformaciones (por defecto es 2 cuando se le pasa
combates_ganados).
```

ejemplo de usos

```
- nuevo_poder = personaje.evolucionar_poder(combates_ganados=3)
- tranformacion = self.evolucionar_poder(multiplicador =
nodo_transformacion.multiplicador_nivel_de_poder)
```

evolucionar_poder(combates_ganados=None, multiplicador=2)

```
    def evolucionar_poder(self, combates_ganados= None, multiplicador=2):
    """
    Método recursivo para calcular la evolución del poder tras cada combate.

    :param combates_ganados: Número de combates ganados.
    :param multiplicador: Multiplicador actual (por defecto es 2).
    :return: Poder total tras los combates.
    """

    if combates_ganados is not None:
        if combates_ganados <= 0:
            return self.nivel_de_poder

        # Calcular el nuevo poder
        nuevo_poder = self.nivel_de_poder * multiplicador
        nuevo_poder = round(nuevo_poder)
        # Actualizar el poder actual
        self.nivel_de_poder = nuevo_poder

        # Aumentar experiencia tras cada combate
        self.exp += 50  # Ejemplo: ganar 50 exp por combate

        # Verificar si se debe subir de nivel
        self.subir_nivel()

        # Llamada recursiva para el siguiente combate
        return self.evolucionar_poder(combates_ganados - 1, multiplicador)
    else:

        #Al tranformase se pasa como parametro el multiplicador de la
tranformacion y eleva el poder del personaje
        nuevo_poder = self.nivel_de_poder * multiplicador
        nuevo_poder = round(nuevo_poder)
        # Actualizar el poder actual en una transformacion
        self.nivel_de_poder = nuevo_poder

        self.max_ki = self.calcular_max_ki(multiplicador)
        self.vida *= multiplicador
        print("tranfomacion")
        return nuevo_poder
```

Árbol Binario en un Juego de Dragon Ball

Introduccion

En el desarrollo de un juego de rol basado en personajes, es esencial manejar información sobre cada personaje de forma eficiente, especialmente cuando se trata de acceder a ellos rápidamente o de mantenerlos organizados según algún atributo clave. Una estructura de datos adecuada para este propósito es el árbol binario, que es una estructura jerárquica que permite almacenar y organizar elementos de manera eficiente.
En este caso, utilizamos un árbol binario para organizar personajes del juego, basándonos en su nivel de poder, lo que facilita tanto la inserción, como la búsqueda de personajes en un entorno de combate o gestión.

¿Por qué utilizar un Árbol Binario?

Un árbol binario es una estructura de datos compuesta por nodos, donde cada nodo tiene como máximo dos hijos: izquierdo y derecho. En el contexto de un juego de rol, podemos usar esta estructura para organizar personajes según su nivel de poder de manera eficiente, permitiendo lo siguiente:

- Inserción ordenada: Los personajes se insertan en el árbol en orden según su nivel de poder. Esto asegura que el árbol esté equilibrado y que cada búsqueda se realice de manera rápida.
- Búsquedas rápidas: Dado que los nodos están organizados de forma ordenada, podemos buscar un personaje por su nivel de poder de manera eficiente, realizando comparaciones sucesivas a lo largo del árbol.
- Recorridos ordenados: Utilizando un recorrido en orden (inorden), se pueden obtener los personajes en orden ascendente según su nivel de poder.

Descripción de la Implementación

En la implementación proporcionada, hemos definido tres clases principales: Nodo y ArbolBinario.

- Clase Nodo: Cada nodo del árbol contiene una instancia de la clase Personaje, así como referencias a los hijos (izquierdo y derecho) que representan los subárboles izquierdo y derecho, respectivamente. Esta estructura permite almacenar de forma ordenada la información sobre los personajes.
- Clase ArbolBinario: Esta clase gestiona el árbol binario. Contiene métodos para insertar personajes en el árbol, realizar búsquedas de personajes por su nivel de poder y obtener la lista de personajes ordenados según su nivel de poder. El árbol utiliza una inserción recursiva, de modo que se asegura que los personajes se coloquen correctamente dependiendo de su nivel de poder. Además, el método de búsqueda recursiva permite encontrar un personaje de forma eficiente.

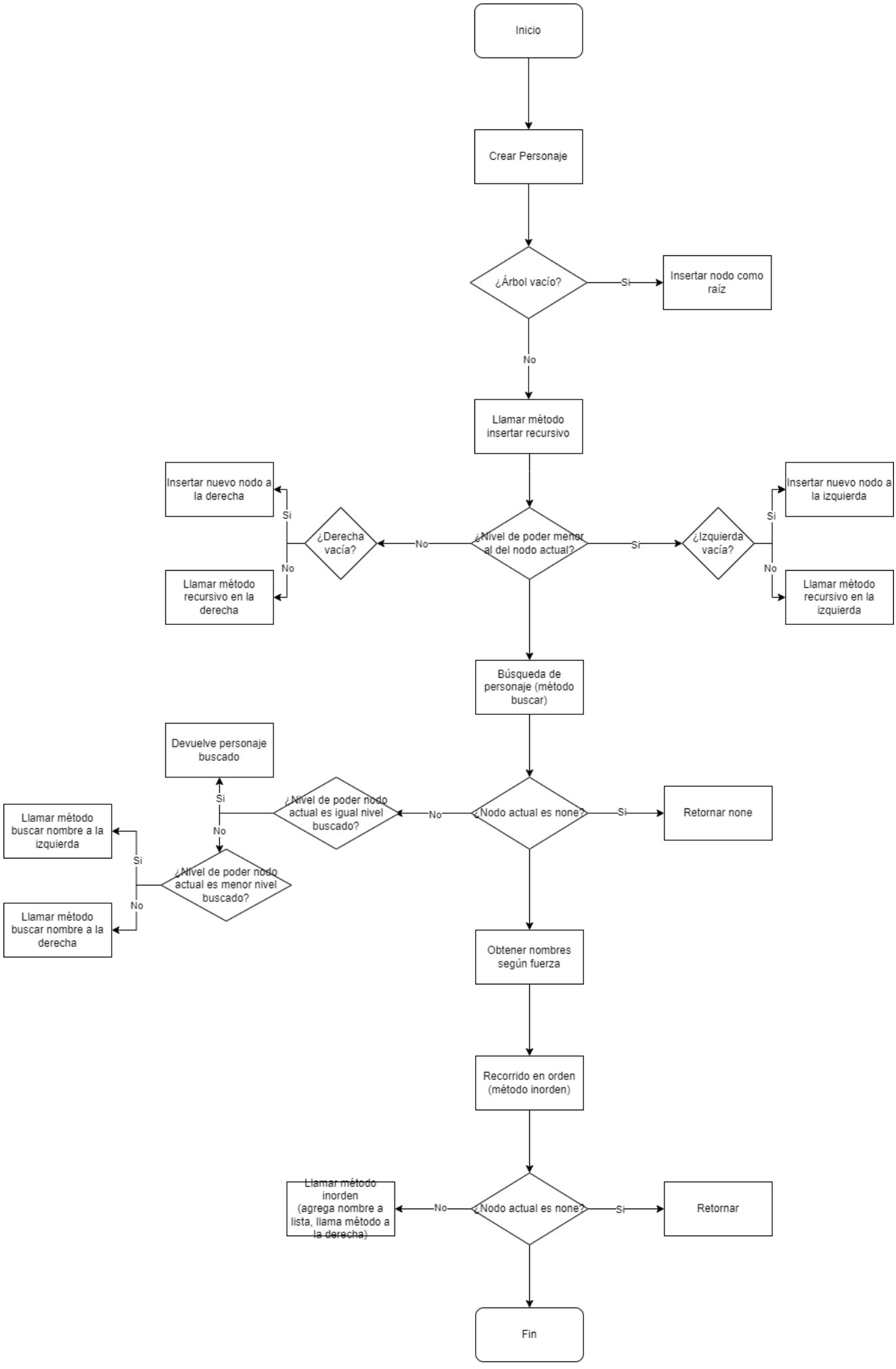
Metodos

- insertar(self, personaje): Inserta un personaje en el árbol en la posición correspondiente según su nivel de poder.

- _insertar_recursivo(self, nodo, personaje): Método auxiliar recursivo que busca la posición correcta para el nuevo nodo.
- buscar_personaje(self, nivel_poder): Busca un personaje en el árbol basado en su nivel de poder.
- _buscar_personaje_recursivo(self, nodo, nivel_poder): Método auxiliar recursivo para realizar la búsqueda.
- obtener_personajes_poder(self): Obtiene una lista de personajes ordenados por su nivel de poder.
- _inorden(self, nodo, lista): Método auxiliar recursivo para recorrer el árbol en orden ascendente.

Conclusión

El uso de un árbol binario es una estrategia efectiva para manejar grandes cantidades de personajes en un juego de rol. La eficiencia en las operaciones de inserción, búsqueda y recorrido garantiza que el juego pueda escalar bien a medida que se agregan más personajes, sin que se vea comprometida la experiencia de usuario. Al estructurar los personajes de esta manera, se asegura que el rendimiento del sistema se mantenga optimizado.
Esta implementación también es fácilmente extensible, permitiendo agregar nuevasfuncionalidades, como la eliminación de personajes o la actualización de sus atributos, sin perder la eficiencia en las operaciones.


Inicio
Crear Personaje
¿Árbol vacío?
Si
Insertar nodo como raíz
No
Llamar método insertar recursivo
¿Nivel de poder menor al del nodo actual?
No
¿Derecha vacía?
Si
Insertar nuevo nodo a la derecha
No
Llamar método recursivo en la derecha
Si
¿Izquierda vacía?
Si
Insertar nuevo nodo a la izquierda
No
Llamar método recursivo en la izquierda
Búsqueda de personaje (método buscar)
¿Nodo actual es none?
Si
Retornar none
No
¿Nivel de poder nodo actual es igual nivel buscado?
Si
Devuelve personaje buscado
No
¿Nivel de poder nodo actual es menor nivel buscado?
Si
Llamar método buscar nombre a la izquierda
No
Llamar método buscar nombre a la derecha
Obtener nombres según fuerza
Recorrido en orden (método inorden)
¿Nodo actual es none?
Si
Retornar
No
Llamar método inorden (agrega nombre a lista, llama método a la derecha)
Fin

Uso de Árboles Generales para Habilidades y Transformaciones

Introducción

En el contexto de este juego, se utilizarán árboles generales para representar las habilidades y transformaciones de los personajes. Un árbol general es una estructura de datos en la que cada nodo puede tener un número ilimitado de hijos, lo cual lo hace adecuado para modelar jerarquías como las transformaciones de los personajes y sus habilidades. Cada nodo del árbol representará una habilidad o transformación, y las ramas entre los nodos representarán la relación entre habilidades o transformaciones en diferentes niveles.

¿Por que utilizar Arboles Generales?

Las habilidades y transformaciones en un sistema de juego como el de Dragon Ball siguen una estructura jerárquica de progresión. Por ejemplo, para alcanzar una habilidad avanzada o una forma transformada, a menudo es necesario primero adquirir habilidades o formas iniciales. Esta relación jerárquica de "maduración" es perfectamente modelada por un árbol general, donde:

El nodo raíz representa la habilidad o transformación base.
Los nodos hijos representan habilidades derivadas o transformaciones más avanzadas que dependen de la habilidad o transformación anterior.
Los árboles permiten capturar este tipo de dependencias de manera natural, asegurando que se puedan modelar las relaciones entre habilidades y transformaciones de manera coherente y fácil de seguir.

Objetivos

- Organizar y gestionar las habilidades y transformaciones de los personajes mediante un árbol.
- Permitir la expansión de un personaje a través de las transformaciones que se desbloquean o mejoran según el progreso del juego.
- Cada habilidad tendrá un nodo en el árbol, y puede depender de otras habilidades, lo que se reflejará en la estructura jerárquica

Estructura del Árbol General

Un árbol general se compone de nodos y relaciones jerárquicas entre ellos. En este caso, los nodos representarán las habilidades y transformaciones.

- **NodoHabilidad**
  Un nodo en el árbol que representa una habilidad específica. Cada nodo incluye información detallada de la habilidad, como el costo de Ki, el daño que inflige, y si requiere una transformación específica para ser desbloqueada.

  Atributos:

  ```
  - nombre (str): Nombre de la habilidad.
  ```

```
  - costo_ki (int): Cantidad de Ki requerida para usar la habilidad.

  - daño (int): Cantidad de daño que la habilidad inflige.

  - transformacion_requerida (list): Lista de transformaciones necesarias
para desbloquear esta habilidad.

  - descripcion (str): Descripción breve de la habilidad.

  - hijo (NodoHabilidad): Primer hijo del nodo, que representa la
habilidad derivada más directa.

  - hermano (NodoHabilidad): Nodo hermano, que representa otra habilidad
del mismo nivel jerárquico.
```

Métodos:

```
  - str: Retorna una representación en texto del nodo, incluyendo el
nombre, costo de Ki, daño y descripción.

  - str(self): Representa la habilidad en un formato legible.
```

- **ArbolHabilidades**

Clase que representa el árbol general de habilidades. Gestiona la estructura jerárquica de las habilidades, permitiendo agregar nuevas habilidades, buscarlas y mostrar el árbol completo.

Atributos:

```
- raiz (NodoHabilidad): Nodo raíz del árbol, que representa la habilidad base
inicial.
```

Métodos:

mostrar_arbol(self,nodo=None, nivel=0)

```
- Descripción: Imprime el árbol en un formato jerárquico.
- Parámetros:
    - nodo (NodoHabilidad, opcional): Nodo desde donde empezar a mostrar. Por
defecto, es la raíz.
    - nivel (int): Nivel de profundidad para indentar la visualización.
- Salida: Representación visual del árbol en la consola.
```

agregar_hijo(self,padre, hijo)

```
- Descripción: Agrega un nodo hijo a un nodo padre.
- Parámetros:
    - padre (NodoHabilidad): Nodo al que se le agregará el hijo.
    - hijo (NodoHabilidad): Nodo hijo que se agregará.
- Salida: Enlaza el hijo al padre. Si el padre ya tiene hijos, se agrega como
último hermano.
```

ordenamiento_topologico(self)

```
- Descripción: Realiza un ordenamiento topológico de las habilidades usando un
recorrido DFS.
- Parámetros: Ninguno.
- Salida: Lista de nodos en el orden en que deben adquirirse según la
jerarquía.

- Detalle del funcionamiento interno:

    1- Define una función auxiliar dfs para recorrer los nodos recursivamente.
    2- Usa un conjunto visitado para evitar procesar nodos repetidos.
    3- Agrega los nodos procesados a una pila en orden inverso.
    4- Devuelve la pila invertida para obtener el orden correcto.
```

buscar_habilidad(self,nombre, nodo=None)

```
- Descripción: Busca una habilidad específica por su nombre.
- Parámetros:
    - nombre (str): Nombre de la habilidad a buscar.
    - nodo (NodoHabilidad, opcional): Nodo desde donde empezar la búsqueda. Por
defecto, es la raíz.
- Salida: Devuelve el nodo de la habilidad si se encuentra, o None si no
existe.
```

Arbol de Habilidades

Introducción

La función *crear_arbol_habilidades* permite construir un árbol de habilidades dinámicamente a partir de un conjunto de datos estructurados en forma de diccionario. Este enfoque facilita la generación automática de árboles de habilidades complejos, definiendo nodos, relaciones jerárquicas y atributos directamente en los datos iniciales.

Descripción de la Función

- Función Principal: crear_arbol_habilidades

- Crea y devuelve una instancia de ArbolHabilidades, con las habilidades organizadas en una jerarquía basada en los datos proporcionados.
- Argumentos:
  - datos (dict): Un diccionario con los detalles de la habilidad raíz y sus sub-habilidades. El formato esperado es:

```
{
    "nombre": "Habilidad raíz",
    "costo": int,
    "poder": int,
    "transformacion_requerida": list,
    "descripcion": str,
    "hijos": {
        "Habilidad hija 1": {
            "costo": int,
            "poder": int,
            "transformacion_requerida": list,
            "descripcion": str,
            "hijos": { ... }  # Subhijos
        },
        ...
    }
}
```

- Retorno:
  - (ArbolHabilidades): Una instancia del árbol, con todos los nodos organizados jerárquicamente.

- Componentes Internos
  - Subfunción: construir_nodo
    - Crea una instancia de NodoHabilidad basada en los datos proporcionados.
  - Argumentos:
    - nombre (str): Nombre de la habilidad.
    - info (dict): Diccionario con atributos como costo de Ki, poder, transformaciones requeridas y descripción.
  - Retorno:
    - (NodoHabilidad): Nodo creado a partir de los datos proporcionados.
    - Subfunción: construir_arbol_recursivo
      - Construye recursivamente un subárbol a partir de un nodo y sus hijos.
      - Argumentos:

        - data (dict): Diccionario con los datos del nodo actual y sus hijos.
    - Retorno:
        - (NodoHabilidad): Nodo raíz del subárbol construido

Ventajas de la Implementación

- Modularidad: Cada subfunción se encarga de una tarea específica (crear nodos, construir subárboles), lo que facilita la comprensión y el mantenimiento del código.
- Flexibilidad: Permite construir árboles de cualquier complejidad simplemente ajustando el diccionario de entrada.
- Escalabilidad: La construcción recursiva asegura que el árbol pueda manejar estructuras anidadas de manera eficiente

Analisis Algoritmico

- En términos de complejidad temporal, el tiempo de ejecución está directamente relacionado con el número de nodos en el árbol. Dado que cada nodo se procesa una vez y la recursión se aplica a cada nivel de profundidad del árbol, la complejidad temporal es O(n), donde n es el número total de nodos en el árbol. Esto se debe a que cada nodo y cada hijo son visitados una única vez durante el proceso de construcción.
- En cuanto a la complejidad espacial, el uso de recursión implica que cada llamada a la función recursiva ocupa un espacio en la pila. Por lo tanto, la complejidad espacial se ve afectada por la profundidad máxima del árbol, lo que puede ser O(d), donde d es la profundidad del árbol. Además, se necesita O(n) espacio para almacenar los nodos del árbol, ya que cada nodo tiene que ser creado y almacenado en memoria. En total, la complejidad espacial es O(n + d).
- En conclusión, el algoritmo es eficiente en términos de tiempo, ya que visita cada nodo una sola vez, lo que lo hace lineal con respecto al número de nodos. Sin embargo, la eficiencia espacial puede verse afectada por la profundidad del árbol y el tamaño de la estructura de datos. Para árboles no muy profundos, la solución es adecuada, pero si el árbol árbol es grande o profundo, podría haber limitaciones de espacio, debido al uso de la recursión.

Arbol de Transformaciones

Introducción

Este módulo define una estructura basada en árboles generales para modelar transformaciones jerárquicas en un sistema de personajes. Las transformaciones están organizadas como nodos, con información sobre el ki necesario, transformaciones previas requeridas y un multiplicador que afecta el nivel de poder del personaje.

seleccionar_habilidad_aleatoria(self)

```
- Descripción: Selecciona aleatoriamente una habilidad del árbol.
- Parámetros: Ninguno.
```

```
- Salida: Nodo de la habilidad seleccionada al azar.

- Detalle del funcionamiento interno:

    - Calcula el número total de nodos con _contar_nodos.
    - Genera un índice aleatorio.
    - Usa _encontrar_nodo_por_indice para recorrer el árbol y encontrar el nodo
correspondiente al índice.
```

_contar_nodos(nodo)

```
- Descripción: Cuenta el número total de nodos en el subárbol desde un nodo
dado.
- Parámetros:
    - nodo (NodoHabilidad): Nodo desde donde contar.
- Salida: Número entero representando el total de nodos
```

_encontrar_nodo_por_indice(nodo, indice)

```
- Descripción: Encuentra un nodo basado en un índice específico.
- Parámetros:
    - nodo (NodoHabilidad): Nodo desde donde buscar.
    - indice (int): Índice del nodo buscado.
- Salida: Nodo correspondiente al índice.
```

Descripción de las Clases

Clase: NodoTransformacion
Representa un nodo del árbol, el cual modela una transformación específica.

Atributos:

```
- nombre (str): El nombre de la transformación.

- ki_necesario (int): Cantidad de ki requerida para acceder a esta
transformación.

- transformacion_requerida (str): El nombre de la transformación previa
requerida.

- multiplicador_nivel_de_poder (float): Factor multiplicativo aplicado al nivel
de poder del personaje al usar esta transformación.

- hijo (NodoTransformacion): Primer nodo hijo que representa una transformación
más avanzada.
```

```
- hermano (NodoTransformacion): Nodo hermano al mismo nivel jerárquico
```

Métodos:

init(self, nombre, ki_necesario, transformacion_requerida, multiplicador_nivel_de_poder)

```
- Constructor de la clase.

- Inicializa el nodo con los valores proporcionados y establece los enlaces de
hijo y hermano como None.
```

agregar_hijo(self, nodo_hijo)

```
- Agrega un nodo como hijo del nodo actual. Si ya existe un hijo, el nuevo nodo
se agrega al final de la lista de hermanos.

- Argumentos:
    - nodo_hijo (NodoTransformacion): Nodo a agregar como hijo.
- Retorno: Ninguno.
```

Clase: ArbolTransformaciones
Representa el árbol general de transformaciones, organizando jerárquicamente las distintas etapas que puede alcanzar un personaje.

Atributos:

```
- raiz (NodoTransformacion): Nodo raíz del árbol, que representa la
transformación base.
```

Métodos:

init(self, nodo_raiz)

```
- Constructor de la clase. Inicializa el árbol con un nodo raíz.

- Argumentos:
    - nodo_raiz (NodoTransformacion): Nodo raíz del árbol.
    - Retorno: Ninguno.
```

mostrar_arbol(self, nodo=None, nivel=0)

```
- Muestra el árbol completo en una vista jerárquica, indicando nombre, ki
necesario y multiplicador de nivel de poder.

- Argumentos:
    - nodo (NodoTransformacion, opcional): Nodo desde el cual comenzar a
mostrar. Por defecto, se toma la raíz.
    - nivel (int, opcional): Nivel de indentación para la representación
jerárquica. Por defecto, 0.
- Retorno: Ninguno.
```

buscar_nodo(self, nombre, nodo=None)

```
- Busca un nodo por su nombre dentro del árbol, recorriendo hijos y hermanos.

- Argumentos:
    - nombre (str): Nombre de la transformación a buscar.
    - nodo (NodoTransformacion, opcional): Nodo desde el cual comenzar la
búsqueda. Por defecto, se toma la raíz.
- Retorno:
    - (NodoTransformacion o None): Nodo encontrado o None si no se encuentra.
```

obtener_proxima_transformacion(self, personaje)

```
- Encuentra la próxima transformación disponible para un personaje.

- Parámetros:
    - personaje (Personaje): Personaje que busca su próxima transformación.

- Retorno:
    - El nodo de la siguiente transformación o None si no hay disponible.
```

_obtener_proxima_transformacion(self, nodo, personaje)

```
- Método recursivo para encontrar la próxima transformación en el árbol.

- Parámetros:
    - nodo (NodoTransformacion): Nodo actual de la búsqueda.
    - personaje (Personaje): Personaje que busca transformar.
- Retorno:
    - El nodo correspondiente a la próxima transformación disponible.
```

Beneficios de la Implementación

- Flexibilidad: Permite modelar relaciones complejas entre transformaciones.
- Escalabilidad: Puede manejar grandes jerarquías de transformaciones gracias a su estructura enlazada.
- Facilidad de navegación: Métodos como mostrar_arbol y buscar_nodo simplifican la visualización y consulta del árbol.

Cola de Prioridades y Heap Binaria

Cola de Prioridad para Gestión de Combates

Introducción

La cola de prioridad es una estructura de datos que permite gestionar y organizar elementos en función de su prioridad, de manera eficiente. En este proyecto, se utiliza una cola de prioridad para gestionar los combates de un torneo, donde los personajes con el mayor nivel de poder tienen mayor prioridad para participar en los enfrentamientos.

Objetivo

El objetivo de esta implementación es gestionar los personajes del torneo de tal forma que los personajes más poderosos sean seleccionados primero para los combates, asegurando que el flujo de combate siga un orden lógico y justo basado en la fuerza de los luchadores.

Tecnología Utilizada

Para implementar la cola de prioridad, se utiliza el módulo heapq de Python. Este módulo proporciona una implementación eficiente de un heap binario, que permite organizar los elementos de manera que el elemento con la mayor prioridad (en este caso, el nivel de poder más alto) siempre se encuentra en la raíz del heap.

Estructura de la Cola de Prioridad

La clase principal que gestiona la cola de prioridad es *ColaDePrioridad*. Esta clase utiliza internamente el heap proporcionado por heapq para garantizar que el elemento con mayor prioridad siempre sea el primero en ser extraído. Los personajes son insertados en el heap de acuerdo con su nivel de poder.

Clase:

- ColaDePrioridad

Atributos

```
- heap:
    - Tipo: list
    - Descripción: Es una lista que almacena los personajes. Cada personaje se
representa como una tupla de dos elementos: el primer elemento es el nivel de
poder negativo del personaje (para invertir la ordenación del heap de modo que
el nivel de poder más alto tenga mayor prioridad), y el segundo elemento es el
objeto Personaje que contiene los datos del personaje.
```

Métodos

init(self):

```
- Descripción: Constructor de la clase. Inicializa el heap como una lista
vacía.
```

agregar_personaje(self, personaje):

```
- Descripción: Agrega un personaje al heap. Se utiliza heapq.heappush para
insertar el personaje en el heap de acuerdo con su nivel de poder (invertido
para priorizar los niveles más altos).
```

ejemplo de uso:

```
- goku = Personaje("goku",1000) #se ponen los atributos mas importantes en este
ejemplo
- freezer = Personaje("Freezer", 10000)
- cola_prioridad = ColaPrioridad()
- #agregar personajes a la cola
- cola_prioridad.agregar_personaje(goku)
- cola_prioridad.agregar_personaje(freezer)
```

obtener_siguiente_personaje(self):

```
- Descripción: Extrae y devuelve el personaje con el mayor nivel de poder (el
que está en la raíz del heap). Si el heap está vacío, devuelve None.
```

Ejemplo de uso:

```
- #Obtener y mostrar los primeros dos personajes para el combate
- personaje1 = cola_prioridad.obtener_siguiente_personaje()
- personaje2 = cola_prioridad.obtener_siguiente_personaje()
```

personajes_restantes(self):

```
- Descripción: Devuelve una lista de los personajes restantes en el heap,
ordenados por su nivel de poder en orden descendente.
```

Ejemplo de uso:

```
- restantes = cola_prioridad.personajes_restantes()
```

Detalles Técnicos de la Implementación

- Uso de heapq
    - La implementación de la cola de prioridad en este proyecto se basa en el uso del módulo heapq de Python, que maneja internamente un heap binario.
- Heap binario: Un heap es una estructura de datos en la que cada elemento tiene una relación de orden con sus hijos. En un max-heap (que es lo que necesitamos), el valor del nodo raíz es el mayor, y el valor de cada nodo padre es mayor que el valor de sus nodos hijos.
  heapq maneja un min-heap por defecto (el valor más pequeño está en la raíz). Para lograr un max-heap (donde los valores más altos estén en la raíz), se utiliza el nivel de poder negativo de los personajes, de manera que los valores mayores se convierten en menores cuando se insertan en el heap.

Conclusión
La cola de prioridad es una herramienta eficiente para gestionar los combates en un torneo de personajes, donde la prioridad está basada en el nivel de poder de los luchadores. Gracias al uso de heapq, las operaciones de inserción y extracción se realizan de manera eficiente, lo que permite que el sistema escale bien a medida que el número de personajes en el torneo crece.

Este sistema de gestión de combates garantiza que los personajes más fuertes sean seleccionados primero, asegurando que los enfrentamientos se desarrollen de forma justa y organizada.

Análisis de Algoritmos

- En cuanto a las batallas entre personajes, es importante notar cómo se gestionan los turnos de forma dinámica. Cada vez que un personaje ataca o carga su Ki, se realiza una serie de verificaciones que determinan si el combate continúa o si uno de los personajes ha sido derrotado. Este proceso es bastante eficiente, ya que, en general, la cantidad de tiempo que toma cada turno es constante. Sin embargo, el número total de turnos dependerá de cuánto dure la pelea, lo que hace que la complejidad sea lineal. Es decir, si el combate se extiende, el tiempo total se incrementará proporcionalmente.
- La evolución del poder de los personajes representa otra característica del juego. Cada vez que un personaje gana un combate, acumula experiencia que puede llevarlo a subir de nivel. Este proceso es bastante rápido, ya que verificar si se ha alcanzado la experiencia necesaria para ascender es una operación sencilla y rápida. Sin embargo, si un personaje tiene que subir varios niveles, el proceso puede llevar un poco más de tiempo.
- En lo que refiere a la organización de los personajes, se utilizan estructuras de datos como árboles binarios y colas de prioridad. Los árboles permiten insertar y buscar personajes de manera eficiente

según su nivel de poder. En promedio, estas operaciones son rápidas, lo que significa que, a medida que se añaden más personajes, el juego sigue funcionando sin problemas. La cola de prioridad es especialmente útil en este contexto, ya que permite acceder rápidamente al personaje más fuerte, lo que es fundamental para gestionar los combates de manera efectiva.

- En términos de espacio, es importante mencionar que tanto los árboles como las colas de prioridad requieren un poco de memoria adicional para almacenar toda la información. Un árbol binario, por ejemplo, puede ocupar espacio proporcional al número de personajes que contiene, lo cual se debe tener en cuenta, pero no es un problema mayor en la mayoría de los casos. Las estructuras que almacenan habilidades y transformaciones también añaden algo de carga, aunque su impacto es menor en comparación con la cantidad de personajes.

- A modo de conclusión, los algoritmos y estructuras de datos que se utilizan en este juego son bastante eficientes. Las batallas se desarrollan de manera fluida, la evolución del poder es rápida y sencilla, y la organización de los personajes se maneja de forma efectiva gracias a las estructuras elegidas. Sin embargo, siempre es bueno destacar que el equilibrio y la profundidad de estas estructuras son clave para mantener la eficiencia, especialmente cuando el número de personajes aumenta.

Grafos

Introducción

Este grafo representa el universo de Dragon Ball en el que los planetas son nodos y las rutas espaciales entre ellos son aristas. Los planetas en este grafo son: Tierra, Namek y Vegeta. Las conexiones entre los planetas están representadas por aristas que pueden tener un peso (como distancia, tiempo de viaje, etc.).

Estructura del Grafo
El grafo está representado utilizando una matriz de adyacencia, donde, cada fila y columna representa un planeta.
Los valores dentro de la matriz indican si hay una conexión entre dos planetas y, si existe, el peso asociado a esa conexión.

Clase GrafoDragonBall

La clase GrafoDragonBall permite crear un grafo con planetas (nodos) y rutas espaciales (aristas). Esta clase incluye métodos para agregar rutas entre planetas con pesos, mostrar las rutas y obtener el peso de una ruta específica.

Atributos

- planetas: Lista de planetas representados como nodos en el grafo.

- num_planetas: Número de planetas (nodos) en el grafo.

- matriz_adyacencia: Matriz de adyacencia que almacena las conexiones entre planetas y los pesos de estas conexiones.

- dirido: Indica si el grafo es dirigido o no.
  Métodos

Metodos

-**init(self, planetas)**

-Descripción: Inicializa la clase con una lista de planetas. Crea una matriz de adyacencia con tamaño num_planetas x num_planetas, inicializada en 0 (sin rutas entre los planetas).

- **Parámetros**:
  -planetas: Lista de planetas (nombres) que serán los nodos del grafo.

-Ejemplo de uso:

```
- planetas = ["Tierra", "Namek", "Vegeta"]
- grafo = GrafoDragonBall(planetas)
```

- **agregar_ruta(self, origen, destino, peso)**
- Descripción: Este método permite agregar una ruta entre dos planetas en un grafo representado por una matriz de adyacencia. La ruta se define mediante un peso que puede representar diferentes métricas, como distancia o tiempo de viaje. Si el grafo es no dirigido, se establece la conexión en ambas direcciones.
- **Parámetros**:
  - origen: El nombre del planeta de origen.
  - destino: El nombre del planeta de destino.
  - peso: El peso de la ruta (por ejemplo, distancia o tiempo de viaje).

efecto

```
- Si ambos planetas (origen y destino) están presentes en la lista de planetas
(self.planetas), se actualiza la matriz de adyacencia (self.matriz_adyacencia)
para reflejar el peso de la ruta entre ellos.
- Si el grafo es no dirigido (self.dirigido es False), se establece también el
peso en dirección opuesta, asegurando que la conexión sea bidireccional.
```

-Ejemplo de uso:

```
- grafo.agregar_ruta("Tierra", "Marte", 225000)  # Agrega una ruta de Tierra a
Marte con un peso de 225000 km
```

-**mostrar_rutas(self)**

- Descripción: Muestra todas las rutas existentes entre planetas y sus respectivos pesos.

-Ejemplo de uso:

```
- grafo.mostrar_rutas()
```

-**obtener_peso(self, origen, destino)**

- Descripción: Obtiene el peso de la ruta entre dos planetas.
- Parámetros:
    - origen: El nombre del planeta de origen.
    - destino: El nombre del planeta de destino.

-Ejemplo de uso:

```
-peso = grafo.obtener_peso("Tierra", "Namek")

-print(peso)
```

armar_grafo(self)

- Descripción: Configura el grafo con planetas y rutas predefinidos.
- Funcionalidad:Define una lista de planetas del universo Dragon Ball.
  Añade rutas con pesos específicos mediante el método agregar_ruta.

Planetas y Rutas Predefinidos

Planetas:
- Tierra
- Namek
- Vegeta
- Planeta Kaio
- Reino de los Demonios
- Planeta de Bills
- La Habitación del Tiempo
- Planeta Yadarat

Conclusión

Este grafo proporciona una representación simple y eficiente del universo de Dragon Ball, permitiendo agregar rutas entre planetas con pesos, mostrar las rutas y consultar el peso de una ruta específica. Es una herramienta útil para modelar las conexiones entre diferentes ubicaciones en un sistema de mapas o redes de transporte.

Recorridos DFS y BFS

Introduccion
En este proyecto, se implementan dos algoritmos clásicos para encontrar caminos entre planetas en un grafo dirigido: DFS (Búsqueda en Profundidad) y BFS (Búsqueda en Anchura). A continuación, se describe cada uno de ellos.

Algoritmo DFS (Búsqueda en Profundidad)
El DFS (Depth First Search) es un algoritmo utilizado para explorar un grafo. A diferencia de BFS, que explora nivel por nivel, DFS sigue un camino desde un nodo hasta llegar al final antes de retroceder.

Descripción

- Objetivo: Encontrar un camino entre dos planetas en el grafo.
- Método: Se utiliza un enfoque recursivo, donde el algoritmo explora profundamente cada nodo hasta que llega al destino o no hay más caminos a seguir.

Parámetros

- origen: El planeta de inicio.
- destino: El planeta de destino.
- Retorno: Devuelve una lista con los nombres de los planetas en el camino encontrado. Si no se encuentra un camino, devuelve None.

Funcionamiento Interno

- dfs_recursivo: Función recursiva que realiza la búsqueda en profundidad desde un nodo dado. Se marca cada nodo como visitado para evitar ciclos y redundancias.
- Recursión: Se exploran los vecinos del nodo actual. Si se llega al destino, el algoritmo termina y devuelve el camino encontrado.
- Exploración de vecinos: Si un vecino no ha sido visitado y existe una conexión (peso > 0), se explora ese vecino.

Algoritmo BFS (Búsqueda en Anchura)

El BFS (Breadth First Search) es otro algoritmo utilizado para explorar un grafo, pero a diferencia de DFS, BFS explora todos los nodos en el nivel actual antes de pasar al siguiente nivel.

Descripción

- Objetivo: Encontrar el camino más corto entre dos planetas en el grafo.
- Método: Utiliza una cola para almacenar los caminos que se van explorando. En cada paso, se expande el camino desde el nodo actual hacia sus vecinos.

Parámetros

- origen: El planeta de inicio.
- destino: El planeta de destino.

- Retorno: Devuelve una lista con los nombres de los planetas en el camino más corto entre el origen y el destino. Si no se encuentra un camino, devuelve None.

Funcionamiento Interno

- Inicialización de la Cola: Se usa una cola (FIFO) para explorar los nodos de manera iterativa, comenzando desde el nodo de origen.
- Expansión de Caminos: Se exploran todos los caminos posibles hacia los nodos vecinos hasta encontrar el destino.
- Nivel por Nivel: Se expande cada camino, asegurando que el primer camino encontrado al destino sea el más corto.

Conclusión

DFS es útil cuando se quiere explorar un grafo profundamente desde un nodo inicial, pero no garantiza el camino más corto.
BFS es preferible cuando se busca el camino más corto entre dos nodos, ya que explora de manera sistemática todos los nodos a nivel de profundidad.

Ordenamiento Topológico

Problemas NP y Camino Mínimo

Algoritmo de Dijkstra para el Camino Mínimo

Introducción

En el universo de Dragon Ball, un problema interesante puede ser encontrar el camino más rápido entre planetas para recolectar las Esferas del Dragón. Este es un problema clásico de camino mínimo, el cual se puede resolver utilizando el algoritmo de Dijkstra.

El algoritmo de Dijkstra encuentra el camino más corto desde un nodo de inicio a todos los demás nodos en un grafo ponderado. Para este problema, los nodos son planetas y las aristas son las rutas espaciales entre ellos, con un peso asociado que representa la distancia o el tiempo de viaje.

Anexo

	TIERRA	NAMEK	VEGETA	PLANETA KAIO	REINO DE LOS DEMONIOS	PLANETA DE BILLS	LA HABITACION DEL TIEMPO	PLANETA YADARAT
TIERRA	0	40	0	0	0	180	10	0
NAMEK	40	0	90	0	70	0	0	0
VEGETA	0	90	0	0	0	0	0	50
PLANETA KAIO	0	0	0	0	90	0	0	0
REINO DE LOS DEMONIOS	0	70	0	90	0	0	0	0
PLANETA DE BILLS	180	0	0	0	0	0	0	0
LA HABITACION DEL TIEMPO	10	0	0	0	0	0	0	0
PLANETA YADARAT	0	0	90	0	0	0	0	0