Universidad Mariano Gálvez de Guatemala

Facultad de Ingeniería

Ingeniería en Sistemas y Ciencias de la Computación

Programación III

Sección ‘A’

Tik tak toe con Avl Trees

Alder Isaac Solis De Leon - 9490-22-227

Joshua Ivan Andree Mendez Vasquez - 9490-22-4032

Abner Salvador Cancinos - 9490-22-2101

Cristian Sebastian Rodas - 9490-22-523

Angel Emilio Méndez Muralles 9490-22-5851

Guatemala 28 de Mayo de 2024

Introducción

Propósito del Manual

El propósito de este manual es proporcionar instrucciones detalladas sobre cómo instalar, el juego de Tic-Tac-Toe con Machine Learning utilizando árboles AVL.

Alcance

Este manual cubre todos los aspectos relacionados con el sistema, incluyendo la instalación

Audiencia Prevista

Este manual está destinado a desarrolladores, ingenieros y cualquier persona interesada en aprender sobre la implementación de un juego de Tic-Tac-Toe con técnicas de Machine Learning.

Descripción General del Sistema

El sistema implementa un juego de Tic-Tac-Toe con inteligencia artificial que utiliza un árbol AVL para almacenar y gestionar los estados del juego y sus valores Q.

Descripción del Sistema

Componentes Principales

1. **BoardManager**: Clase principal que gestiona la interfaz gráfica del juego y la interacción del usuario.
2. **MachineIa**: Clase que maneja la lógica de la inteligencia artificial y el entrenamiento del modelo.
3. **AVLTree**: Clase que implementa el árbol AVL para almacenar los estados del juego.
4. **AVLNode**: Clase que representa un nodo del árbol AVL.
5. **GameUtilities**: Clase que proporciona utilidades adicionales como capturas de pantalla del tablero.
6. **main.py**: Script de inicio que ejecuta la aplicación.

Especificaciones Técnicas

- **Lenguaje de Programación**: Python
- **Librerías Principales**: **tkinter**, **numpy**, **Pillow**, **graphviz**
- **Requisitos de Sistema**: Python 3.x, pip, Graphviz

Instrucciones de Instalación

Requisitos Previos

Antes de comenzar la instalación, asegúrese de tener los siguientes requisitos cumplidos:

- Python 3.x instalado
- **pip** instalado
- Librerías necesarias: **tkinter**, **numpy**, **Pillow**, **graphviz**

Paso a Paso de la Instalación

1. **Instalar Python y pip**:
    - Descargue e instale Python desde python.org.
    - Asegúrese de que **pip** esté instalado junto con Python.
2. **Instalar las Librerías Necesarias**:

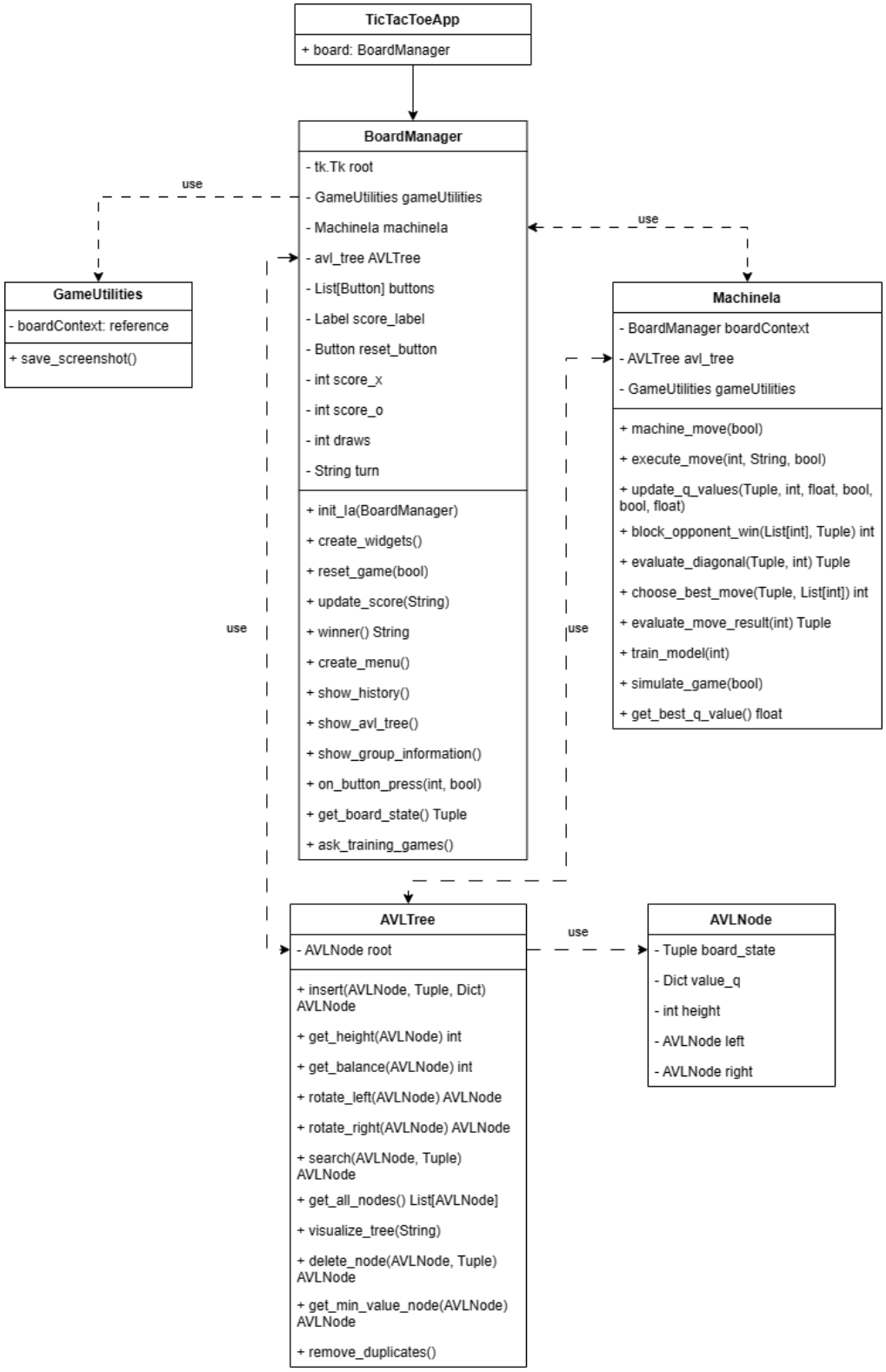
    pip install numpy Pillow graphviz

3. **Configurar Graphviz:**
   Instale Graphviz desde graphviz.org. Asegúrese de que el comando dot esté disponible en el PATH del sistema.
4. **Ejecutar código fuente**
   Ejecute el código fuente usando Python con el comando py main.py

Código disponible en el siguiente repositorio:

https://github.com/aisolis/tik-tak-toe

Diagrama de clases



Componentes y Relaciones

1. **TicTacToeApp**
    - **Propósito**: Clase principal de la aplicación que se encarga de iniciar la ejecución del juego.
    - **Relación**:
        - **Asociación con BoardManager**: La clase **TicTacToeApp** tiene un atributo **board** que es una instancia de **BoardManager**. Esto significa que **TicTacToeApp** utiliza **BoardManager** para gestionar la interfaz gráfica y la lógica del juego.
2. **BoardManager**
    - **Propósito**: Maneja la interfaz gráfica del usuario (GUI), la lógica del juego, y coordina las acciones entre el usuario, la IA, y el árbol AVL.
    - **Relaciones**:
        - **Asociación con GameUtilities**: **BoardManager** utiliza **GameUtilities** para funcionalidades auxiliares como guardar capturas de pantalla.
        - **Asociación con Machinela**: **BoardManager** crea y utiliza una instancia de **Machinela** para manejar los movimientos y el entrenamiento de la IA.
        - **Asociación con AVLTree**: **BoardManager** utiliza **AVLTree** para almacenar y gestionar los estados del juego.
        - **Dependencia de tk.Tk**: **BoardManager** depende de la biblioteca **tkinter** para crear y gestionar la GUI.
3. **GameUtilities**
    - **Propósito**: Proporciona utilidades adicionales para el juego, como capturar y guardar capturas de pantalla del tablero.
    - **Relación**:
        - **Dependencia de BoardManager**: **GameUtilities** tiene una referencia a **BoardManager** para acceder al contexto del tablero.

4. **Machinela**
    - **Propósito**: Maneja la lógica de la IA, incluyendo la selección de movimientos y el entrenamiento del modelo utilizando aprendizaje por refuerzo.
    - **Relaciones**:
        - **Dependencia de BoardManager**: **Machinela** necesita el contexto del tablero proporcionado por **BoardManager** para realizar movimientos y evaluar estados.
        - **Dependencia de AVLTree**: **Machinela** utiliza **AVLTree** para buscar y actualizar valores Q asociados a diferentes estados del juego.
        - **Dependencia de GameUtilities**: **Machinela** puede utilizar utilidades adicionales proporcionadas por **GameUtilities**.
5. **AVLTree**
    - **Propósito**: Implementa la estructura de árbol AVL para almacenar y gestionar los estados del juego y sus valores Q, manteniendo balance y eficiencia.
    - **Relaciones**:
        - **Asociación con AVLNode**: **AVLTree** consiste en múltiples instancias de **AVLNode**, donde cada nodo representa un estado del tablero y sus valores Q.
        - **Dependencia de AVLNode**: **AVLTree** depende de **AVLNode** para crear y gestionar los nodos del árbol.
6. **AVLNode**
    - **Propósito**: Representa un nodo en el árbol AVL, almacenando un estado del tablero y sus valores Q asociados.
    - **Relación**:
        - **Dependencia de AVLTree**: **AVLNode** es utilizado por **AVLTree** para construir la estructura del árbol.

Resumen de Funcionalidades y Relaciones

- **Iniciación y GUI**: **TicTacToeApp** inicia la aplicación y crea una instancia de **BoardManager**, que maneja la GUI y la lógica del juego.
- **Utilidades**: **GameUtilities** proporciona funciones auxiliares como capturar y guardar capturas de pantalla del tablero.
- **IA y Lógica del Juego**: **Machinela** maneja la lógica de la IA y el entrenamiento del modelo utilizando el árbol AVL.
- **Estructura de Datos**: **AVLTree** y **AVLNode** implementan una estructura de datos eficiente para almacenar y gestionar los estados del juego y sus valores Q.

Machine IA – Q learning

Porque un árbol AVL frente a otras estructuras de datos

Árbol AVL vs. Otras Estructuras de Datos

1. **Árbol AVL vs. Árbol Binario de Búsqueda (BST)**
    - **Ventajas del AVL**:
        - **Balanceo Automático**: Los árboles AVL son auto-balanceados, lo que significa que las operaciones de inserción, eliminación y búsqueda se mantienen en tiempo logarítmico $O(\log n)$. Esto es crucial para mantener un rendimiento eficiente a medida que aumenta el número de estados del juego.
        - **Rendimiento Consistente**: Dado que el árbol AVL está balanceado, evita los peores casos de rendimiento que pueden ocurrir en un BST no balanceado, donde la estructura del árbol se puede degradar a una lista enlazada, resultando en tiempos de operación $O(n)$.
2. **Árbol AVL vs. Árbol B**
    - **Complejidad y Overhead**: Los árboles B están diseñados para sistemas que requieren operaciones de entrada/salida eficientes, como bases de datos y sistemas de archivos. Aunque los árboles B también mantienen operaciones en tiempo logarítmico, son más complejos de implementar y pueden tener un overhead adicional debido a la gestión de nodos con múltiples hijos.
    - **Propósito del Sistema**: En un entorno de IA para juegos, donde las operaciones de memoria y procesamiento son críticas, la simplicidad y

eficiencia del árbol AVL lo hacen más adecuado en comparación con la complejidad de un árbol B.

3. **Árbol AVL vs. Pilas y Colas**
    - **Estructura Lineal vs. Estructura Jerárquica**: Pilas y colas son estructuras lineales que no son adecuadas para la búsqueda y recuperación eficiente de estados del juego. Las operaciones de búsqueda en pilas y colas tienen un tiempo $O(n)$, lo que no es eficiente para manejar grandes conjuntos de datos.
    - **Acceso Directo**: Los árboles AVL permiten acceso directo y eficiente a cualquier nodo, mientras que en pilas y colas solo se puede acceder al elemento en la parte superior (pila) o al frente (cola).
4. **Árbol AVL vs. Listas Enlazadas**
    - **Tiempo de Búsqueda**: Las listas enlazadas, ya sean simples o dobles, tienen un tiempo de búsqueda lineal $O(n)$, lo cual es ineficiente para operaciones frecuentes de búsqueda y actualización de estados del juego.
    - **Estructura de Datos**: Los árboles AVL proporcionan una estructura jerárquica que es más adecuada para representar relaciones entre diferentes estados del juego y realizar operaciones de búsqueda y actualización de manera eficiente.

Justificación Especifica para el proyecto

1. **Frecuencia de Operaciones de Búsqueda y Actualización**:
    - En el contexto del Q-learning, es necesario buscar frecuentemente los estados del juego y actualizar los valores Q asociados. El tiempo logarítmico de las operaciones en un árbol AVL asegura que estas operaciones se realicen de manera eficiente incluso a medida que el número de estados crece.
2. **Balanceo Automático**:
    - El balanceo automático del árbol AVL garantiza que el rendimiento del sistema se mantenga estable y predecible, evitando los peores casos que pueden ocurrir en árboles binarios de búsqueda no balanceados.

3. **Eficiencia en Memoria y Tiempo**:
    - Los árboles AVL son relativamente simples de implementar y mantienen un equilibrio entre complejidad y eficiencia, proporcionando tiempos de operación logarítmicos con un overhead manejable.
4. **Adecuación para Aprendizaje por Refuerzo**:
    - En el aprendizaje por refuerzo, donde se necesita acceder y actualizar estados del juego frecuentemente, la estructura auto-balanceada del árbol AVL asegura que estas operaciones se mantengan eficientes.

Modelo de aprendizaje reforzado

Q-Learning

El Q-learning es un algoritmo de aprendizaje por refuerzo que busca aprender una política óptima de acciones para maximizar una recompensa acumulada. En este sistema, el Q-learning se utiliza para que la IA aprenda a jugar al Tic-Tac-Toe mejorando sus decisiones de juego con el tiempo.

Componentes Clave

1. **Estados del Juego**: Representados por la disposición del tablero en un momento dado.
2. **Acciones**: Movimientos posibles que la IA puede realizar en el tablero.
3. **Valores Q**: Representan la "calidad" de una acción en un estado dado, indicando la recompensa esperada por realizar esa acción desde ese estado.

Implementación en el Código

Clases Involucradas

- **MachineIa**: Contiene la lógica de la IA y las funciones de Q-learning.
- **AVLTree**: Estructura de datos que almacena los estados del juego y sus valores Q.
- **AVLNode**: Representa un nodo en el árbol AVL, almacenando un estado del juego y sus valores Q asociados.

Proceso de Q-Learning

Inicialización del Estado: La IA comienza en un estado dado del tablero, representado como una tupla.

```
current_state = self.boardContext.get_board_state()
```

Selección de Acción: La IA selecciona una acción basada en una estrategia epsilon-greedy. Puede explorar (elegir una acción aleatoria) o explotar (elegir la mejor acción conocida).

```
move_index = self.choose_best_move(current_state, empty_indices)
```

Ejecución de la Acción: La IA realiza el movimiento seleccionado y observa el resultado (estado siguiente y recompensa).

```
self.execute_move(move_index, self.boardContext.turn, False)
```

Evaluación del Movimiento: Después de ejecutar el movimiento, se evalúa la recompensa recibida y si el movimiento fue diagonal o bloqueó al oponente.

```
reward, is_diagonal, blocked_opponent = self.evaluate_move_result(index)
```

Actualización de Valores Q: Se actualiza el valor Q para el par (estado, acción) utilizando la fórmula de Q-learning:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max \mathrm{a}'\ Q(s',a') - Q(s,a))$$

Donde:

$\boldsymbol{Q}(\boldsymbol{s},\boldsymbol{a})$ *es el valor Q actual.*

$\boldsymbol{r}$ *es la recompensa recibida.*

$\boldsymbol{\gamma}$ *es el factor de descuento.*

$\mathbf{max}\,\mathbf{a}'\ \boldsymbol{Q}(\boldsymbol{s}',\boldsymbol{a}')$ *es el valor de descuento*

En el código:

```python
def update_q_values(self, state, action_index, reward, is_diagonal_move=False, blocked_opponent=False, gamma=0.9):
    # Busca el nodo con el estado actual del tablero
    node = self.avl_tree.search(self.avl_tree.root, state)

    # Si no existe un nodo, entonces crea uno nuevo y lo inicializa con valores Q
    if not node:
        new_q_values = {i: 0 for i in range(9) if self.boardContext.buttons[i]['text'] == ''}
        node = AVLNode(state, new_q_values)
        self.avl_tree.root = self.avl_tree.insert(self.avl_tree.root, state, new_q_values)
        node.value_q[action_index] = reward  # Se brinda una recompensa inicial

    # Calcula la recompensa, premiando si son movimientos dificiles de bloquear
    adjusted_reward = reward
    if is_diagonal_move:
        adjusted_reward += 0.5
    if blocked_opponent:
        adjusted_reward += 0.3

    # Calcula el mejor valor Q futuro para este estado
    if node.value_q:
        future_q = max(node.value_q.values())
    else:
        future_q = 0

    # Actualiza el valor q, usando la formula de recompensas para valores Q
    updated_q = adjusted_reward + gamma * future_q
    print(updated_q)
    # Actualiza el valor del nodo en base al valor q actualizado
    node.value_q[action_index] = updated_q
```

Detalles del Código

Selección de Acción (choose_best_move): La IA busca el mejor movimiento posible basado en los valores Q almacenados en el árbol AVL.

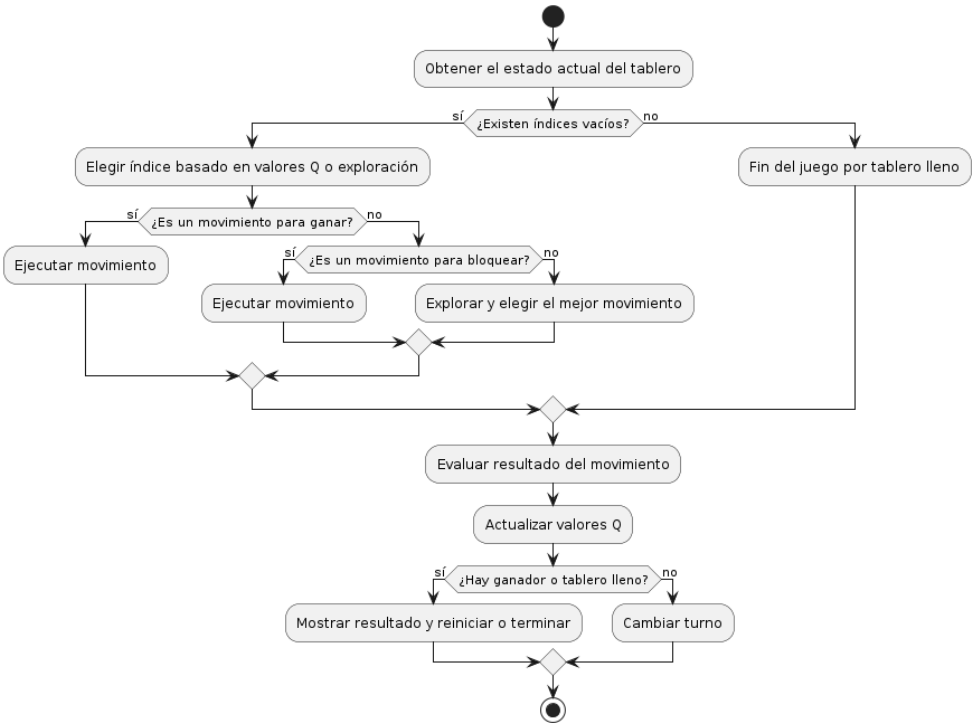
```python
def choose_best_move(self, state, possible_moves):
    node = self.avl_tree.search(self.avl_tree.root, state)
    if node and node.value_q:
        # Incorporar una pequeña probabilidad de elegir un movimiento aleatorio incluso durante la explotación
        if np.random.random() < 0.05:  # 5% de probabilidad de movimiento aleatorio
            return random.choice(possible_moves)

        # Elegir el índice con el máximo valor Q entre los posibles movimientos
        max_q_value = max(node.value_q.get(index, 0) for index in possible_moves)
        best_moves = [index for index in possible_moves if node.value_q.get(index, 0) == max_q_value]
        print("machine choose a best movement")
        return random.choice(best_moves)  # Para evitar sesgos si hay múltiples mejores movimientos
    return random.choice(possible_moves)
```

Evaluación del Movimiento (evaluate_move_result): Evalúa el resultado del movimiento realizado para ajustar la recompensa, a su vez identifica índices que detonarían una potencial derrota y reajusta el movimiento de la IA para que bloquee la victoria del jugador.

```python
def evaluate_move_result(self, index):
    # Establece recompensas base
    reward = 0
    is_diagonal = False
    blocked_opponent = False

    # Verifica si el movimiento es diagonal
    is_diagonal = index in [0, 2, 4, 6, 8]

    # Guarda el estado original del botón para restaurarlo después de la evaluación
    original_text = self.boardContext.buttons[index]['text']

    # Actualiza el tablero temporalmente para la evaluación
    self.boardContext.buttons[index]['text'] = self.boardContext.turn
    winner = self.boardContext.winner()

    if winner:
        # Si el jugador actual gana con este movimiento
        reward = 1 if winner == self.boardContext.turn else -1
    else:
        # Evaluar si el movimiento bloquea al oponente
        opponent = 'X' if self.boardContext.turn == 'O' else 'O'
        for a, b, c in [(0, 1, 2), (3, 4, 5), (6, 7, 8),
                        (0, 3, 6), (1, 4, 7), (2, 5, 8),
                        (0, 4, 8), (2, 4, 6)]:
            if {self.boardContext.buttons[a]['text'], self.boardContext.buttons[b]['text'], self.boardContext.buttons[c]['text']} == {opponent, self.boardContext.turn, ''}:
                blocked_opponent = True
                reward += 0.3  # Agrega una recompensa pequeña por bloquear una jugada ganadora

    # Restaura el estado original del tablero
    self.boardContext.buttons[index]['text'] = original_text
```

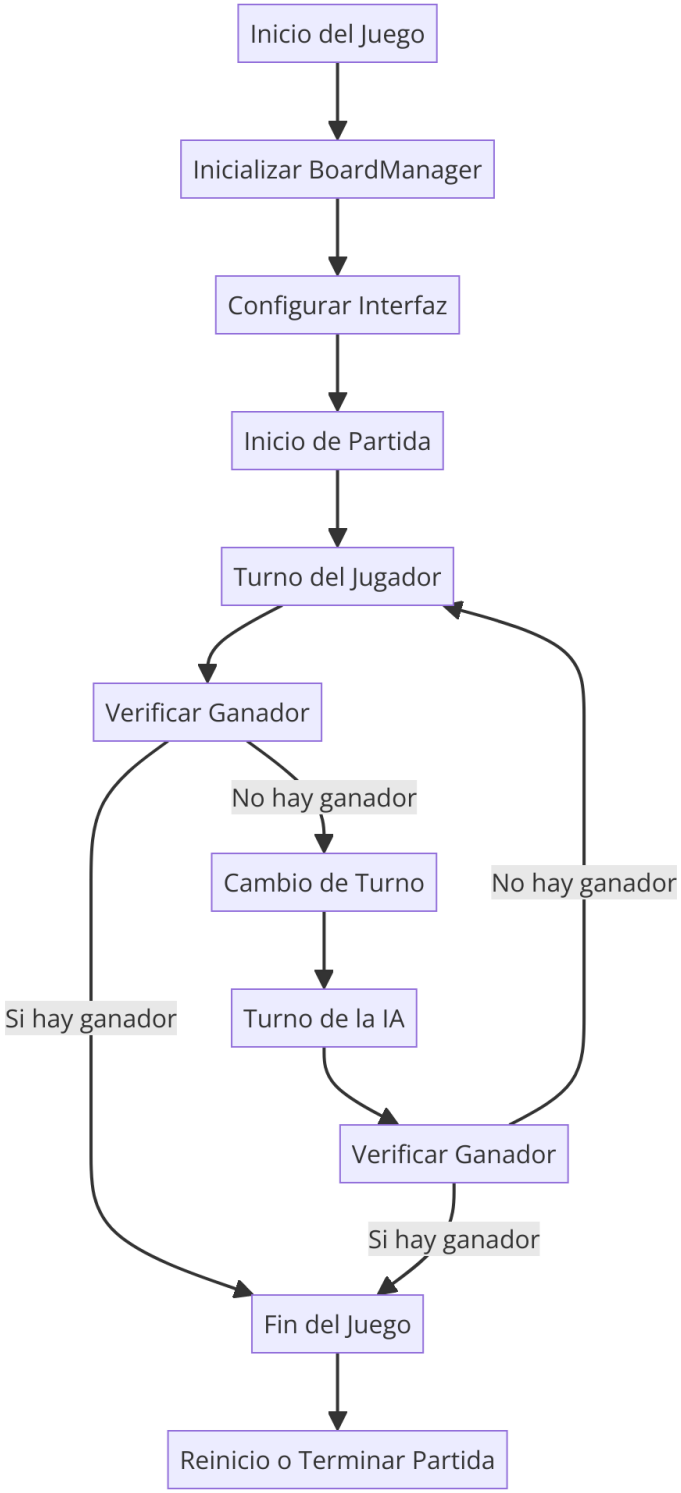
Diagrama de flujo para la ejecución de un movimiento basado en tablas Q

A continuación se presenta un diagrama de flujo sobre el razonamiento de la IA para tomar decisiones sobre su siguiente movimiento:



Game-flow

A continuación, se presenta un diagrama del game-flow:



Descripción del Flujo del Juego

1. **Inicio del Juego**
    - El juego comienza y se inicializa la instancia principal **TicTacToeApp**, que a su vez crea una instancia de **BoardManager**.
2. **Inicializar BoardManager**
    - **BoardManager** configura todos los componentes necesarios, incluyendo la interfaz gráfica del usuario (GUI) y las instancias de las clases auxiliares (**GameUtilities**, **Machinela**, **AVLTree**).
3. **Configurar Interfaz**
    - La interfaz gráfica se configura mediante la creación de widgets, botones, y menús. Este paso se realiza en el método **create_widgets** de **BoardManager**.
4. **Inicio de Partida**
    - Una vez que la interfaz está configurada, se inicia una nueva partida. Esto implica reiniciar el estado del juego y preparar el tablero para un nuevo juego. El método **reset_game** de **BoardManager** se encarga de esta tarea.
5. **Turno del Jugador**
    - El juego comienza con el turno del jugador humano, que puede hacer un movimiento presionando uno de los botones del tablero.
6. **Verificar Ganador**
    - Después de cada movimiento del jugador, se verifica si hay un ganador. El método **winner** de **BoardManager** revisa todas las posibles combinaciones ganadoras en el tablero.
        - **No hay ganador**: Si no hay un ganador, el flujo del juego continúa al siguiente paso.
        - **Si hay ganador**: Si se detecta un ganador, el juego termina y se muestra un mensaje al jugador. Luego, se procede al reinicio o término de la partida.

7. **Cambio de Turno**
    - Si no hay un ganador, el turno cambia a la IA. El método **on_button_press** de **BoardManager** se encarga de alternar el turno.
8. **Turno de la IA**
    - La IA analiza el estado actual del tablero y selecciona el mejor movimiento posible basado en los valores Q almacenados en el árbol AVL. El método **machine_move** de **Machinela** gestiona este proceso.
9. **Verificar Ganador (IA)**
    - Después de que la IA realiza su movimiento, se verifica nuevamente si hay un ganador.
        - **No hay ganador**: Si no hay un ganador, el turno regresa al jugador humano y el flujo del juego vuelve al paso 5.
        - **Si hay ganador**: Si se detecta un ganador, el juego termina y se muestra un mensaje al jugador. Luego, se procede al reinicio o término de la partida.
10. **Fin del Juego**
    - Si se detecta un ganador (jugador humano o IA) o si el tablero está lleno sin ganador (empate), el juego termina.
11. **Reinicio o Terminar Partida**
    - Una vez que el juego ha terminado, el jugador tiene la opción de reiniciar el juego para una nueva partida o cerrar la aplicación. El método **reset_game** de **BoardManager** maneja el reinicio del juego, mientras que la finalización de la aplicación es gestionada por la lógica de cierre de la interfaz gráfica.