INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO
INTELIGENCIA ARTIFICIAL
ING. EN SISTEMAS COMPUTACIONALES

Laboratorio 7: Minimax y Poda Alfa Beta

Alumno: Hurtado Morales Emiliano - 2021630390

Maestro: Andrés García Floriano

Grupo: 6CV3

Fecha de entrega: 24/10/2024

Ciclo Escolar: 2025 – 1

Introducción

El juego de Gato o Tic-Tac-Toe es un juego de estrategia que ha sido utilizado tradicionalmente como un desafío para desarrollar habilidades de razonamiento y análisis lógico. Su versión más común, jugada en un tablero de 3x3, es ampliamente conocida por ser un juego de suma cero y fácilmente resoluble. Sin embargo, al aumentar el tamaño del tablero a 4x4, la complejidad del juego se incrementa considerablemente, lo que requiere de estrategias más sofisticadas, especialmente para la implementación de una Inteligencia Artificial (IA).

En esta práctica, se ha desarrollado un juego de Gato 4x4 utilizando la biblioteca gráfica Tkinter de Python para crear una interfaz amigable e interactiva. El enfoque principal es implementar una IA que utiliza el algoritmo Minimax con poda Alfa-Beta para tomar decisiones estratégicas óptimas. Este algoritmo permite a la IA evaluar todas las posibles jugadas futuras y seleccionar la mejor opción, tomando en cuenta tanto sus propios movimientos como los de su oponente. El Minimax es un algoritmo clásico en el área de juegos de adversarios y, junto con la poda Alfa-Beta, se optimiza su tiempo de ejecución al reducir la cantidad de jugadas que se deben analizar.

Además, el programa ofrece tres modos de juego:

1. Humano vs Humano, donde dos jugadores interactúan entre sí.
2. Humano vs IA, en el que el jugador humano se enfrenta a la IA.
3. IA vs IA, en el cual se observa cómo dos IAs juegan entre sí, permitiendo analizar cómo la IA aplica su estrategia sin intervención humana.

El principal desafío de este laboratorio radica en la correcta implementación del algoritmo Minimax, optimizado mediante la poda Alfa-Beta, para garantizar que la IA pueda tomar decisiones en tiempo razonable sin sacrificar su capacidad para anticipar los movimientos de su oponente. Esta práctica no solo explora la construcción de interfaces gráficas y la lógica de juegos, sino que también introduce a los usuarios en el mundo de la inteligencia artificial aplicada a juegos, mostrando cómo se pueden utilizar técnicas de búsqueda y evaluación heurística para desarrollar agentes capaces de competir en escenarios de decisión secuencial.

Se buscó dar soporte a la conectividad a la red, pero no se pudo hacer en el mismo código de los modos. Sin embargo se realizó en dos códigos más debido a que se requiere que existe un juego servidor y un juego cliente.

Explicación de los Algoritmos

Link: https://github.com/EmilianoHM/Bender-IA/tree/main/Lab_7

En primer lugar, no es posible almacenar todos los estados del juego de Gato 4x4 en memoria debido al número exponencial de configuraciones posibles, alrededor de 43 millones, lo que sería inviable por las limitaciones de espacio y eficiencia.

Para manejar este problema, se implementan varias estrategias para limitar la búsqueda:

```
# Profundidad máxima permitida para el algoritmo Minimax
PROFUNDIDAD_MAX = 4  # Se ha reducido para acelerar el tiempo de respuesta
```

1. **Limitar la profundidad de Minimax**: El algoritmo se restringe a analizar solo unos pocos niveles (en este caso, 4 turnos), reduciendo la cantidad de estados que se evalúan.

```
def evaluar_linea(linea):
    """
    Evalúa una línea de 4 casillas (fila, columna o diagonal).
    La IA obtiene más puntaje por líneas que tiene casi completas.
    """
    puntaje = 0
    if linea.count(IA) == 3 and linea.count(VACIO) == 1:
        puntaje += 100  # 3 en línea para la IA
    elif linea.count(JUGADOR) == 3 and linea.count(VACIO) == 1:
        puntaje -= 100  # 3 en línea para el jugador (bloquear)
    return puntaje
```

2. **Usar una función heurística**: Evalúa los estados intermedios en lugar de explorar hasta el final del juego, asignando puntajes sin necesidad de alcanzar un estado final.

```
# Minimax con poda Alfa-Beta
def minimax(tablero, profundidad, es_max, alpha, beta):
    """
    Implementa el algoritmo Minimax con poda Alfa-Beta, limitado por una
profundidad máxima.
    """
```

3. **Aplicar poda Alfa-Beta**: Elimina ramas innecesarias del árbol de decisiones, lo que reduce la cantidad de estados a explorar.

```
    if hay_ganador(tablero, IA):
        return 1000
    if hay_ganador(tablero, JUGADOR):
        return -1000
    if es_tablero_lleno(tablero) or profundidad == 0:
        return evaluar_tablero(tablero)
```

4. **Evaluación temprana**: Detiene la búsqueda cuando se detecta una victoria inminente, evitando explorar movimientos adicionales innecesarios.

1. Interfaz Gráfica (GUI)

El programa utiliza el módulo **Tkinter** de Python para crear la interfaz gráfica del juego. La ventana principal del juego (self.root) muestra un tablero 4x4 en forma de botones, los cuales representan las casillas del tablero. Los botones son interactivos, permitiendo a los jugadores humanos realizar sus movimientos al hacer clic en ellos.

- **Cuadrícula de botones**: Se crea una cuadrícula 4x4 de botones con self.crear_cuadricula(). Cada botón representa una casilla y su estado cambia al ser pulsado por el jugador.
- **Menú de opciones**: Se incluye un menú para seleccionar el modo de juego, reiniciar la partida o salir del programa. Los modos de juego son "Humano vs Humano", "Humano vs IA", e "IA vs IA".

```
class JuegoGato4x4:
    """Clase principal que implementa la interfaz gráfica y la lógica del juego
del gato 4x4."""

    def __init__(self, root):
        """Inicializa el tablero, los botones, el modo de juego y la interfaz
gráfica."""
        self.root = root
        self.root.title("Gato 4x4")

        # Crear el tablero 4x4 vacío
        self.tablero = crear_tablero()
        self.botones = [[None for _ in range(4)] for _ in range(4)]
        self.turno_humano = True  # El jugador humano comienza

        # Crear la cuadrícula de botones que representan el tablero en la GUI
        self.crear_cuadricula()

        # Etiqueta para mostrar mensajes sobre el estado del juego
        self.mensaje = tk.Label(self.root, text="Turno del Jugador X",
font=("Helvetica", 16))
        self.mensaje.grid(row=4, column=0, columnspan=4)

        # Variable que indica el modo de juego actual
        self.modo_juego = "Humano vs Humano"  # Modo de juego por defecto

        # Crear el menú para seleccionar los modos de juego
        self.crear_menu()

    def crear_cuadricula(self):
        """Crea una cuadrícula de botones que representan el tablero en la
interfaz gráfica."""
        for i in range(4):
            for j in range(4):
```

```
                boton = tk.Button(self.root, text="", font=("Helvetica", 20),
width=5, height=2,
                                  command=lambda i=i, j=j: self.click_boton(i,
j))
                boton.grid(row=i, column=j)
                self.botones[i][j] = boton
```

2. Lógica del Juego

El tablero está representado por una lista de listas (self.tablero), donde las casillas vacías se representan con VACIO, y las jugadas de los jugadores se representan con JUGADOR (X para el jugador humano) y IA (O para la IA).

- **Control de turnos**: El programa alterna entre los turnos del jugador humano y la IA. Los turnos se manejan mediante la función click_boton para el jugador humano y turno_ia para la IA. También hay un modo donde dos IAs juegan entre sí (turno_ia_vs_ia).
- **Detección de victoria**: La función hay_ganador comprueba si un jugador ha ganado revisando filas, columnas y diagonales en busca de 4 símbolos iguales.
- **Empate**: La función es_tablero_lleno verifica si el tablero está completamente lleno, lo que indicaría un empate si no hay un ganador.

```
def crear_tablero():
    """Crea un tablero vacío de 4x4, representado por una lista de listas."""
    return [[VACIO for _ in range(4)] for _ in range(4)]

def hay_ganador(tablero, jugador):
    """
    Verifica si un jugador ha ganado el juego al conseguir 4 en línea.
    Revisa filas, columnas y diagonales.
    """
    for fila in tablero:
        if fila.count(jugador) == 4:
            return True
    for col in range(4):
        if [fila[col] for fila in tablero].count(jugador) == 4:
            return True
    if [tablero[i][i] for i in range(4)].count(jugador) == 4:
        return True
    if [tablero[i][3 - i] for i in range(4)].count(jugador) == 4:
        return True
    return False

def es_tablero_lleno(tablero):
    """Verifica si el tablero está lleno (empate)."""
    return all(VACIO not in fila for fila in tablero)
```

3. Algoritmo Minimax con poda Alfa-Beta

El **algoritmo Minimax** es una técnica de búsqueda utilizada en juegos de adversarios, como el Gato, donde dos jugadores se alternan en realizar movimientos. El objetivo del Minimax es maximizar el beneficio del jugador que controla la IA mientras se minimizan las posibles ganancias del oponente. El proceso básico es el siguiente:

```
    if es_max:
        max_eval = -math.inf
        for i in range(4):
            for j in range(4):
                if tablero[i][j] == VACIO:
                    tablero[i][j] = IA
                    eval = minimax(tablero, profundidad - 1, False, alpha, beta)
                    tablero[i][j] = VACIO
                    max_eval = max(max_eval, eval)
                    alpha = max(alpha, eval)
                    if beta <= alpha:
                        break
        return max_eval
    else:
        min_eval = math.inf
        for i in range(4):
            for j in range(4):
                if tablero[i][j] == VACIO:
                    tablero[i][j] = JUGADOR
                    eval = minimax(tablero, profundidad - 1, True, alpha, beta)
                    tablero[i][j] = VACIO
                    min_eval = min(min_eval, eval)
                    beta = min(beta, eval)
                    if beta <= alpha:
                        break
        return min_eval
```

1. **Exploración del árbol de decisiones**: Minimax genera un árbol de decisiones donde cada nodo representa un posible estado del tablero. Cada vez que la IA hace una jugada, el algoritmo considera todas las posibles respuestas del jugador humano, y así sucesivamente, hasta una profundidad límite o hasta que se alcance un estado terminal (victoria, derrota o empate).

```
    if hay_ganador(tablero, IA):
        return 1000
    if hay_ganador(tablero, JUGADOR):
        return -1000
    if es_tablero_lleno(tablero) or profundidad == 0:
        return evaluar_tablero(tablero)
```

2. **Evaluación de los estados**: En los nodos terminales, el algoritmo evalúa el estado del tablero utilizando una **función heurística**. Esta función asigna un valor numérico a cada estado, donde valores positivos favorecen a la IA y valores negativos favorecen al jugador humano. Un valor de +1000 indica que la IA ha ganado, mientras que un valor de -1000 significa que el jugador humano ha ganado.

```
                max_eval = max(max_eval, eval)
                alpha = max(alpha, eval)
                if beta <= alpha:
                    break
        return max_eval
```

```
                min_eval = min(min_eval, eval)
                beta = min(beta, eval)
                if beta <= alpha:
                    break
        return min_eval
```

3. **Maximización y minimización**: El algoritmo alterna entre **niveles de maximización** (jugadas de la IA) y **niveles de minimización** (jugadas del jugador humano). En el nivel de maximización, la IA selecciona la jugada con el puntaje más alto posible, mientras que en el nivel de minimización, el algoritmo simula el peor escenario, suponiendo que el oponente humano seleccionará la jugada que minimice el puntaje de la IA.

```
                alpha = max(alpha, eval)
                if beta <= alpha:
```

```
                if beta <= alpha:
                    break
```

4. **Poda Alfa-Beta**: El principal problema con Minimax es que explorar todas las posibles jugadas hasta el final del juego puede ser muy costoso en términos de tiempo y recursos, especialmente en un tablero de 4x4. Aquí es donde entra la **poda Alfa-Beta**, que mejora la eficiencia al evitar evaluar ramas innecesarias del árbol. Cuando la IA ya sabe que un movimiento es mejor que otro en una rama particular, puede "podar" o descartar otras jugadas que no ofrezcan una mejor ventaja, reduciendo así el número de nodos explorados.

    - **Alfa** representa el mejor valor que el jugador maximizador (la IA) puede asegurar.
    - **Beta** representa el mejor valor que el jugador minimizador (el humano) puede asegurar.
    - Si en algún momento, el valor de Beta es menor o igual que Alfa, se puede podar esa rama porque no puede producir un mejor resultado.

4. Heurística

La función evaluar_linea otorga puntajes basados en cuántas casillas tiene un jugador en una fila, columna o diagonal. Por ejemplo, si la IA tiene 3 símbolos consecutivos y una casilla vacía, la función otorga un puntaje alto, ya que está cerca de ganar. De manera similar, se resta puntaje si el jugador humano está a punto de ganar.

```
def evaluar_linea(linea):
    """
    Evalúa una línea de 4 casillas (fila, columna o diagonal).
    La IA obtiene más puntaje por líneas que tiene casi completas.
    """
    puntaje = 0
    if linea.count(IA) == 3 and linea.count(VACIO) == 1:
        puntaje += 100  # 3 en línea para la IA
    elif linea.count(JUGADOR) == 3 and linea.count(VACIO) == 1:
        puntaje -= 100  # 3 en línea para el jugador (bloquear)
    return puntaje
```

5. Mejor movimiento

La función mejor_movimiento utiliza Minimax para encontrar el movimiento óptimo de la IA. Recorre todas las casillas vacías del tablero, simula una jugada en cada una, y selecciona la que tenga el puntaje más alto de acuerdo con la evaluación del Minimax.

```
# Encuentra el mejor movimiento para la IA
def mejor_movimiento(tablero, profundidad):
    """Encuentra el mejor movimiento posible para la IA usando Minimax con Poda
Alfa-Beta."""
    mejor_valor = -math.inf
    mejor_mov = (-1, -1)
    for i in range(4):
        for j in range(4):
            if tablero[i][j] == VACIO:
                tablero[i][j] = IA
                mov_valor = minimax(tablero, profundidad, False, -math.inf,
math.inf)
                tablero[i][j] = VACIO
                if mov_valor > mejor_valor:
                    mejor_valor = mov_valor
                    mejor_mov = (i, j)
    return mejor_mov
```

Soporte para la conectividad en red

Servidor (Lab_7S.py)

```
def juego_servidor():
    servidor = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    servidor.bind(('localhost', 12345))
    servidor.listen(2)

    print("Esperando a los jugadores...")
    conexion_jugador1, _ = servidor.accept()
    print("Jugador 1 (X) conectado.")
    conexion_jugador2, _ = servidor.accept()
    print("Jugador 2 (O) conectado.")
```

1. **Inicialización y espera de conexiones:** El servidor utiliza socket para abrir una conexión en localhost y escucha en el puerto 12345. Acepta dos conexiones: un jugador que será asignado como "X" y otro como "O". Ambos jugadores reciben su rol (X u O) cuando se conectan.

```
    while True:
        tablero = inicializar_tablero()
        turno = JUGADOR_X

        # Enviar información inicial a ambos jugadores (quién es X y quién es O)
        enviar_datos(conexion_jugador1, JUGADOR_X)
        enviar_datos(conexion_jugador2, JUGADOR_O)

        juego_activo = True

        while juego_activo:
            # Enviar el tablero actual y el turno actual a ambos jugadores
            enviar_datos(conexion_jugador1, (tablero, turno))
            enviar_datos(conexion_jugador2, (tablero, turno))

            if turno == JUGADOR_X:
                print("Esperando movimiento del Jugador X...")
                movimiento = recibir_datos(conexion_jugador1)
            else:
                print("Esperando movimiento del Jugador O...")
                movimiento = recibir_datos(conexion_jugador2)

            fila, col = movimiento
            tablero[fila][col] = turno
```

2. **Gestión del juego:** El servidor mantiene el estado del tablero en una matriz 4x4 y alterna los turnos entre los jugadores. En cada turno, envía el estado del tablero y el jugador actual a ambos clientes. Recibe el movimiento del jugador correspondiente, actualiza el tablero y verifica si alguien ha ganado o si hay un empate.

```
            # Verificar si hay ganador
            if verificar_ganador(tablero, turno):
                enviar_datos(conexion_jugador1, f"{turno} ha ganado!")
                enviar_datos(conexion_jugador2, f"{turno} ha ganado!")
                juego_activo = False
                break

            # Verificar si el tablero está lleno
            if tablero_lleno(tablero):
                enviar_datos(conexion_jugador1, "Empate")
                enviar_datos(conexion_jugador2, "Empate")
                juego_activo = False
                break
```

3. **Verificación de victoria o empate:** El servidor revisa si el jugador actual ha ganado verificando filas, columnas y diagonales. Si alguien gana o el tablero está lleno (empate), el servidor envía el resultado a ambos jugadores y finaliza el juego.

```
        # Preguntar si los jugadores desean jugar nuevamente
        enviar_datos(conexion_jugador1, "¿Quieres jugar de nuevo? (s/n)")
        enviar_datos(conexion_jugador2, "¿Quieres jugar de nuevo? (s/n)")
        respuesta_jugador1 = recibir_datos(conexion_jugador1)
        respuesta_jugador2 = recibir_datos(conexion_jugador2)

        if respuesta_jugador1.lower() != 's' or respuesta_jugador2.lower() !=
's':
            enviar_datos(conexion_jugador1, "Gracias por jugar!")
            enviar_datos(conexion_jugador2, "Gracias por jugar!")
            break
```

4. **Reinicio del juego:** Cuando una partida termina, el servidor pregunta a ambos jugadores si desean jugar otra vez. Si ambos responden "sí", el tablero se reinicia y comienza una nueva partida. Si alguno responde "no", el servidor envía un mensaje de despedida y cierra la conexión.

Cliente (Lab_7C.py)

```
# --- Ejecución del Cliente ---
if __name__ == "__main__":
    cliente = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    cliente.connect(('localhost', 12345))

    root = tk.Tk()
```

```
    jugador = recibir_datos(cliente)  # El servidor asigna quién es X o O
    juego = ClienteGato4x4(root, cliente, jugador)
    root.mainloop()
    cliente.close()
```

1. **Conexión con el servidor:** El cliente se conecta al servidor y recibe su rol (X u O). Utiliza Tkinter para crear una interfaz gráfica que muestra el tablero como una cuadrícula de botones, donde cada botón representa una casilla del tablero.

```
    def recibir_actualizacion(self):
        while True:
            mensaje = recibir_datos(self.cliente)
            if isinstance(mensaje, str):
                if mensaje.startswith("¿Quieres jugar de nuevo?"):
                    self.preguntar_reiniciar()
                else:
                    messagebox.showinfo("Resultado", mensaje)
                    self.reiniciar_juego()
            else:
                self.tablero, turno = mensaje
                self.mi_turno = (turno == self.jugador)  # Solo puedo jugar si es
mi turno
                self.actualizar_tablero()
```

2. **Gestión de turnos:** El cliente recibe continuamente el estado del tablero y el jugador actual del servidor en un hilo separado usando threading. Esto permite que la interfaz gráfica permanezca receptiva mientras espera los datos del servidor.

```
   def click_boton(self, i, j):
        if self.tablero[i][j] == VACIO and self.mi_turno:
            self.tablero[i][j] = self.jugador
            self.botones[i][j]['text'] = self.jugador
            enviar_datos(self.cliente, (i, j))
            self.mi_turno = False  # Desactivar el turno hasta que recibamos la
siguiente actualización
```

3. **Interacción del jugador:** Cuando es el turno del jugador, el cliente permite que el jugador haga clic en una casilla vacía. Una vez seleccionado, el cliente envía las coordenadas del movimiento al servidor y desactiva su turno hasta que reciba la próxima actualización del servidor.

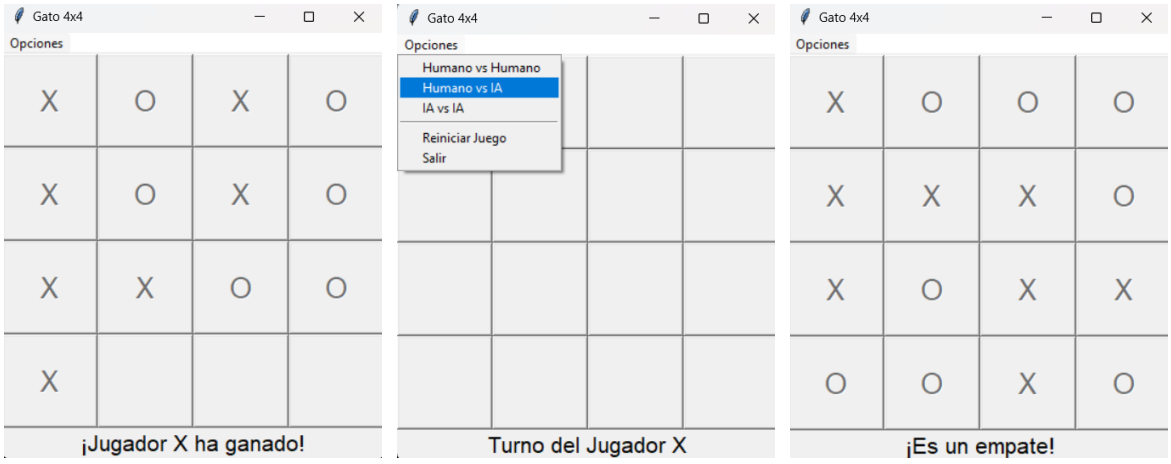
```
    def actualizar_tablero(self):
        for i in range(TAMAÑO):
            for j in range(TAMAÑO):
                self.botones[i][j]['text'] = self.tablero[i][j]
```

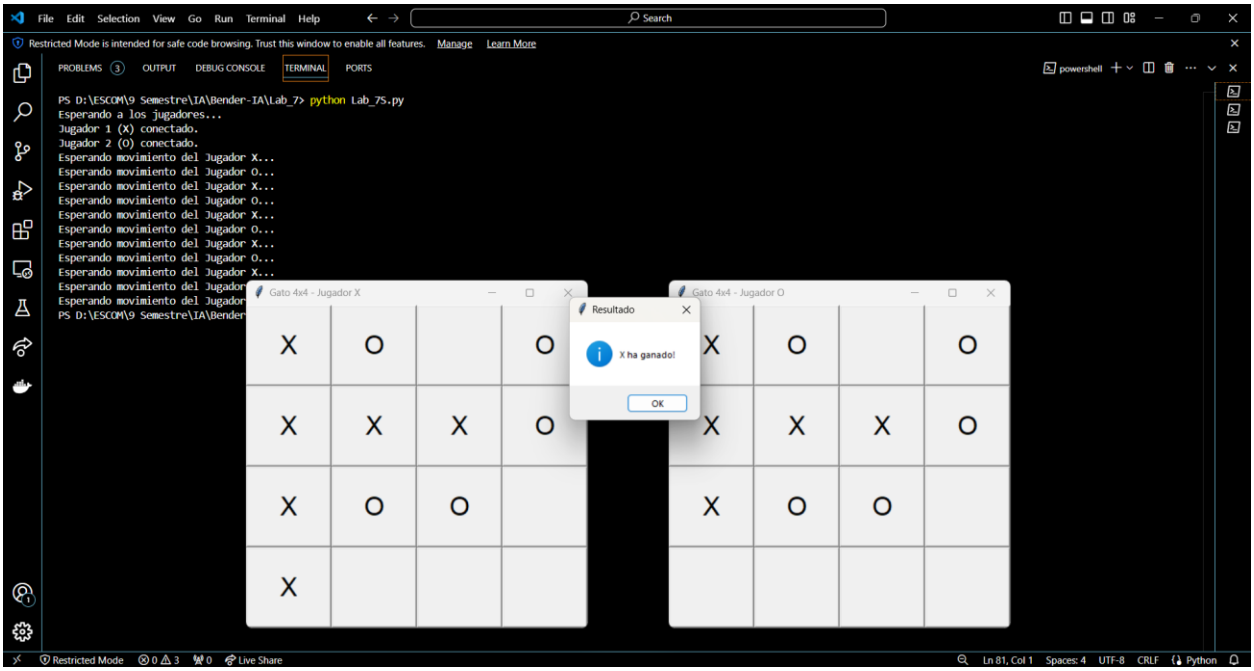
4. **Actualización de la interfaz:** El cliente actualiza la visualización del tablero en la interfaz gráfica con los movimientos más recientes recibidos del servidor y continúa el ciclo hasta que se detecta una victoria o empate, en cuyo caso se muestra un mensaje y se cierra la aplicación.

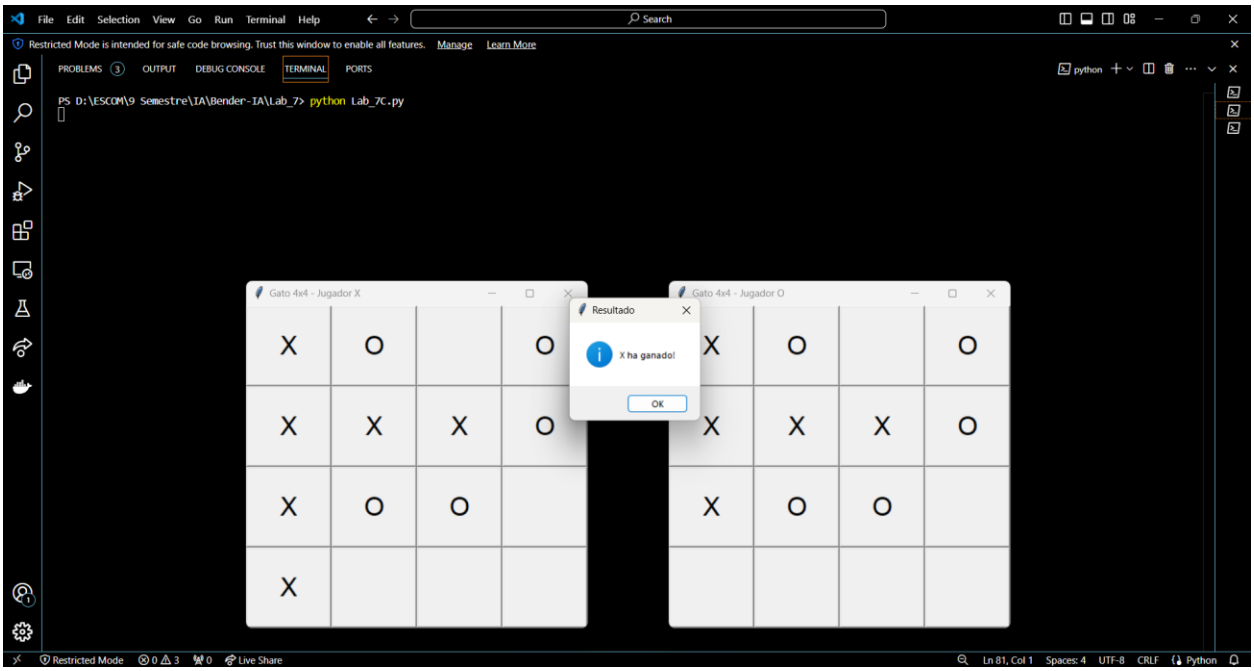
Resultados Esperados

Se espera que, al ejecutar el programa, se observe lo siguiente:

1. **Interacción fluida**: La interfaz gráfica debe ser interactiva y permitir al jugador humano seleccionar casillas con facilidad. La IA debe tomar decisiones rápidas, aunque la profundidad de Minimax está limitada para reducir el tiempo de espera.
2. **Modos de juego diversos**: El programa debe permitir jugar en tres modalidades: Humano vs Humano, Humano vs IA, e IA vs IA. En el modo IA vs IA, los jugadores pueden observar cómo la IA toma decisiones óptimas.
3. **Comportamiento estratégico de la IA**: Gracias al algoritmo Minimax, la IA será capaz de identificar las mejores jugadas y bloquear los intentos del jugador humano de completar una línea de 4 casillas. Si es posible, la IA tratará de ganar o al menos forzar un empate.




File Edit Selection View Go Run Terminal Help
Search
Restricted Mode is intended for safe code browsing. Trust this window to enable all features. Manage Learn More
PROBLEMS 3 OUTPUT DEBUG CONSOLE TERMINAL PORTS
powershell
PS D:\ESCOM\9 Semestre\IA\Bender-IA\Lab_7> python Lab_7S.py
Esperando a los jugadores...
Jugador 1 (X) conectado.
Jugador 2 (O) conectado.
Esperando movimiento del Jugador X...
Esperando movimiento del Jugador O...
Esperando movimiento del Jugador X...
Esperando movimiento del Jugador O...
Esperando movimiento del Jugador X...
Esperando movimiento del Jugador O...
Esperando movimiento del Jugador X...
Esperando movimiento del Jugador O...
Esperando movimiento del Jugador X...
Esperando movimiento del Jugador
Esperando movimiento del Jugador
PS D:\ESCOM\9 Semestre\IA\Bender
Gato 4x4 - Jugador X
Resultado
X ha ganado!
OK
Gato 4x4 - Jugador O
Restricted Mode
Live Share
Ln 81, Col 1
Spaces: 4
UTF-8
CRLF
Python


File Edit Selection View Go Run Terminal Help
Search
Restricted Mode is intended for safe code browsing. Trust this window to enable all features. Manage Learn More
PROBLEMS 3 OUTPUT DEBUG CONSOLE TERMINAL PORTS
python
PS D:\ESCOM\9 Semestre\IA\Bender-IA\Lab_7> python Lab_7C.py
Gato 4x4 - Jugador X
Resultado
X ha ganado!
OK
Gato 4x4 - Jugador O
Restricted Mode
Live Share
Ln 81, Col 1
Spaces: 4
UTF-8
CRLF
Python

Conclusión

El desarrollo de este juego de Gato 4x4 con una IA competitiva proporciona una sólida aplicación de los conceptos de inteligencia artificial y teoría de juegos. El uso del algoritmo Minimax junto con la poda Alfa-Beta no solo permite a la IA realizar jugadas estratégicamente sólidas, sino que también mejora su tiempo de respuesta, lo cual es crucial para mantener una experiencia de usuario fluida en la interfaz gráfica.

A lo largo del desarrollo de la práctica, se ha explorado cómo la combinación de una interfaz gráfica intuitiva y una IA optimizada puede ofrecer una experiencia de juego desafiante para los usuarios. La implementación de diferentes modos de juego, incluyendo IA vs IA, proporciona una oportunidad única para observar cómo dos estrategias óptimas interactúan entre sí, permitiendo un análisis más profundo del comportamiento de los algoritmos en juegos de adversarios.

Este laboratorio demuestra cómo técnicas avanzadas como el Minimax con poda Alfa-Beta pueden ser aplicadas en problemas cotidianos como los juegos de mesa, destacando la importancia de la optimización algorítmica en aplicaciones interactivas. Al permitir que la IA identifique las mejores jugadas de manera eficiente, este programa no solo asegura una experiencia de juego equilibrada, sino que también sirve como una introducción accesible a temas complejos de la inteligencia artificial y la toma de decisiones automáticas.