



Instituto Tecnológico de Costa Rica

IC-6200: Inteligencia Artificial

Trabajo Corto 1: Algoritmos A* y Minimax

Estudiantes:

Andrea María Li Hernández - 2021028783

Deyan Andrey Sanabria Fallas - 2021046131

Erick Fabián Madrigal Zavala - 2018146983

Profesor:

Kenneth Roberto Obando Rodriguez

Fecha de entrega:

22 de febrero del 2024

I Semestre, 2024

Búsqueda A*

1. Instala PyAmaze utilizando el siguiente comando: **pip install pyamaze**.
2. Crea un laberinto utilizando las funciones proporcionadas por PyAmaze. Puedes definir el tamaño del laberinto y colocar obstáculos de manera aleatoria.

```
#tamaño del Laberinto
x_inicial = random.randint(50,90)
y_inicial = random.randint(50,90)
#posicion de la meta
y_meta = random.randint(0,y_inicial)
x_meta = random.randint(0,x_inicial)

#se crea el laberinto
m = maze(y_inicial,x_inicial)
# goal en 1,1 inicio en y_inicial, x_inicial
m.CreateMaze(y_meta,x_meta,loopPercent= 100)

#otras formas de generar el laberinto
#m.CreateMaze(theme=COLOR.Light,pattern='v') #vertical
#m.CreateMaze(theme=COLOR.Light,pattern='h') #horizontal
agente=agent(m,inicio[0],inicio[1],footprints=True,filled=True)
# maze_map -> arreglo con dato de tipo
# {(y, x): {'E': 1, 'W': 0, 'N': 0, 'S': 0}} 1 = camino, 0 = pared
mapa = m.maze_map
```

3. Implementa el algoritmo A* para encontrar el camino más corto entre un punto de inicio y un punto de destino en el laberinto.

```
#algoritmo principal de A*
def a_estrella(mapa):
    x = inicio[1]
    y = inicio[0]

    lista_abierta = []
    heapq.heappush(lista_abierta, (0, y, x)) # se agrega un elemento inicial con un peso
0 y la posicion inicial
    # Lista de celdas visitadas
    lista_cerrada = [[False for _ in range((x_inicial + 1))] for _ in range((y_inicial +
1))]
    # detalles de las celdas
    detalles_celda = [[Celda() for _ in range((x_inicial + 1))] for _ in range((y_inicial
+ 1))]

    #se inicializa la celda de inicio
```

```

detalles_celda[y][x].f = 0
detalles_celda[y][x].g = 0
detalles_celda[y][x].h = 0
detalles_celda[y][x].padres = (y,x)

#flag para saber si se llego al destino
destino_alcanzado = False

#Logica princioal del A*
while len(lista_abierta) > 0:
    actual = heapq.heappop(lista_abierta)
    #se marca la celda como visitada
    x = actual[2]
    y = actual[1]
    lista_cerrada[y][x] = True

    #se buscan Los nodos adyacentes
    direcciones = [(y-1,x), (y+1,x), (y,x-1), (y,x+1)]
    for direccion in direcciones:
        nueva_y = direccion[0]
        nueva_x = direccion[1]

        #se revisa si el nodo es valido y desbloqueado
        if valido(mapa, direccion) and desbloqueado(mapa, actual, direccion):
            #Se revisa si el nodo es el destino
            if destino(direccion):
                detalles_celda[nueva_y][nueva_x].padres = (y,x)
                destino_alcanzado = True
                camino = armar_camino(detalles_celda, meta)
                mostrar_camino(camino)
                return
            else:
                #Segun Los nodos a los que se pueden llegar se generan g,h y f
                nueva_g = detalles_celda[y][x].g + 1.0
                nueva_h = calcular_heuristica(direccion, meta)
                nueva_f = nueva_g + nueva_h
                #Se revisa si el nodo ya fue visitado y si es asi se revisa si el
                nuevo camino es mejor
                if detalles_celda[nueva_y][nueva_x].f == float('inf') or
detalles_celda[nueva_y][nueva_x].f > nueva_f:
                    heapq.heappush(lista_abierta, (nueva_f, nueva_y, nueva_x))
                    detalles_celda[nueva_y][nueva_x].f = nueva_f
                    detalles_celda[nueva_y][nueva_x].g = nueva_g

```

```

        detalles_celda[nueva_y][nueva_x].h = nueva_h
        detalles_celda[nueva_y][nueva_x].padres = (y,x)

if not destino_alcanzado:
    print("No se encontro un camino al destino")

```

4. Utiliza las funciones proporcionadas por PyAmaze para visualizar el laberinto y el camino calculado por el algoritmo A*.

#funcion que arma el camino a seguir basado en las celdas visitadas

```

def armar_camino(celdas, destino):
    camino = []
    fila = destino[0]
    columna = destino[1]

    #se revisan y agregan los nodos padres al camino
    while not (celdas[fila][columna].padres[0] == fila and
               celdas[fila][columna].padres[1] == columna):
        camino.append((fila, columna))
        temp_fila = celdas[fila][columna].padres[0]
        temp_columna = celdas[fila][columna].padres[1]
        fila = temp_fila
        columna = temp_columna

    #se agrega el nodo objetivo
    camino.append((fila, columna))
    #se invierte el camino para que sea del origen al destino
    camino.reverse()
    return camino

```

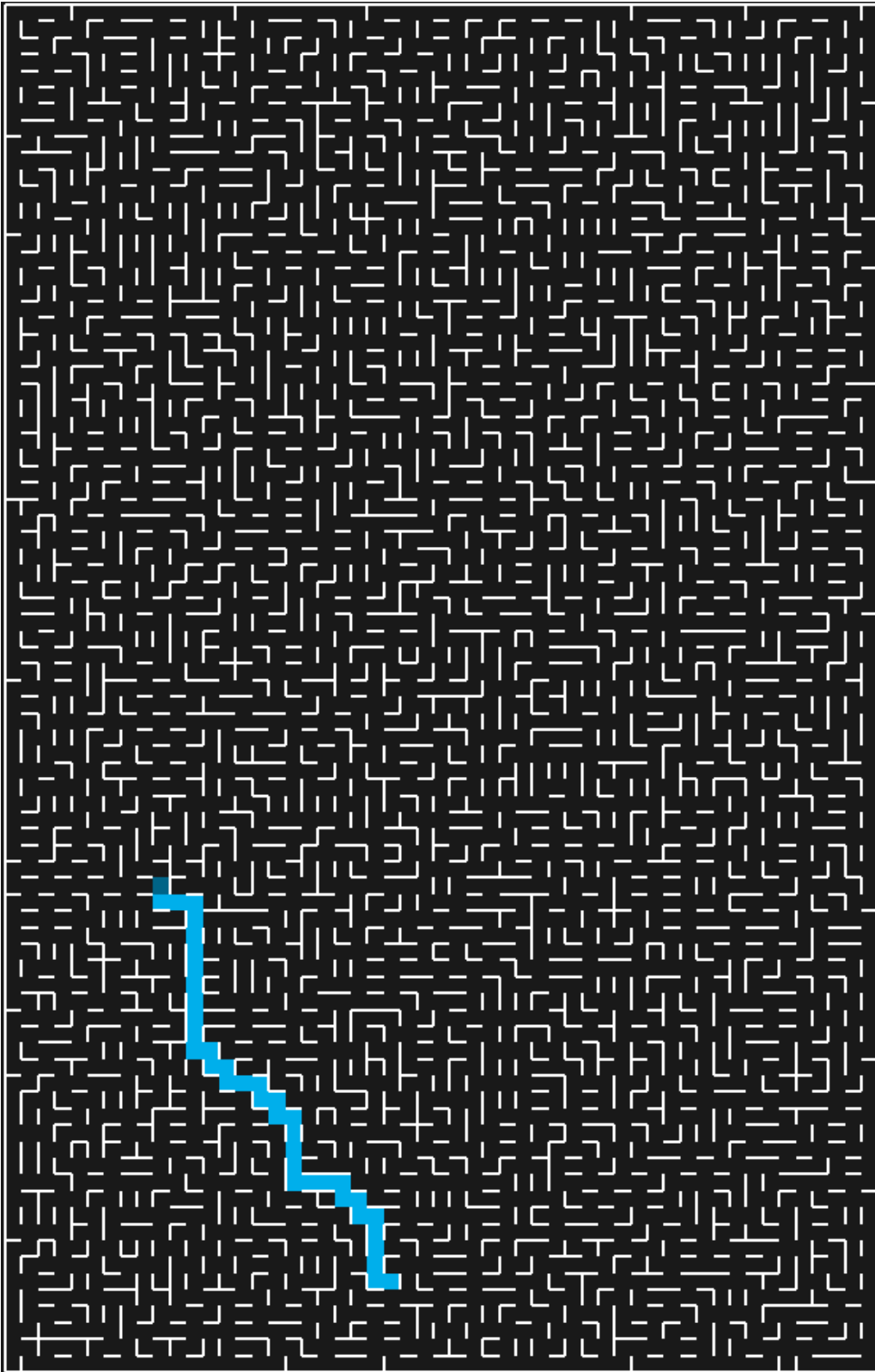
#funcion que mueve el agente por el laberinto para mostrar el camino

```

def mostrar_camino(camino):
    global longitud
    longitud = len(camino)
    for nodo in camino:
        agente.position = nodo

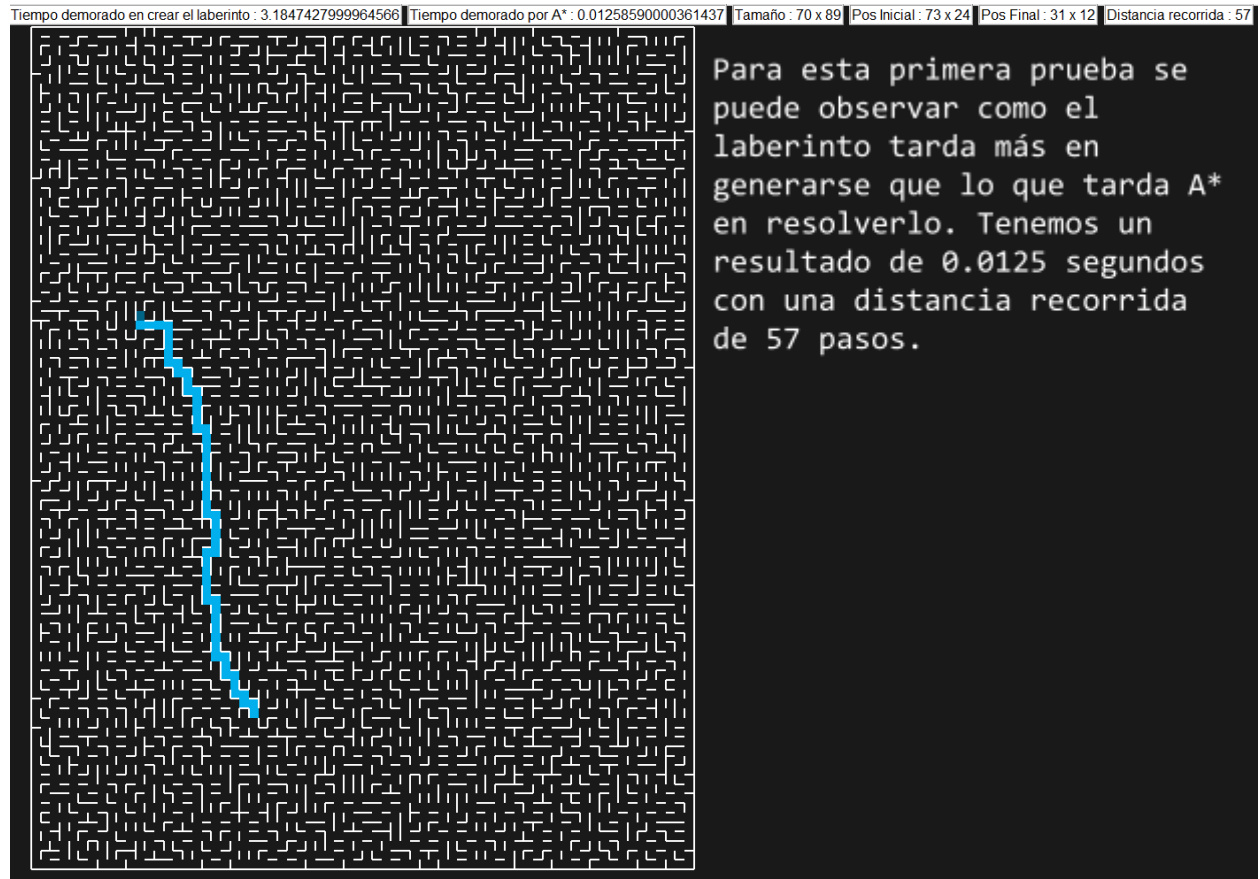
```

5. El camino calculado debe ser resaltado de manera visual para que sea claramente visible en el laberinto.



Pruebas y experimentación

Para las pruebas del algoritmo de Búsqueda A*, se optó por hacer que el programa creara de forma aleatoria un laberinto entre tamaños de 50x50 hasta 90x90, con la posición inicial y final también aleatorizadas para poder visualizar el comportamiento del algoritmo.



Tiempo demorado en crear el laberinto : 3.268409099973705

Tiempo demorado por A* : 0.006184400001075119

Tamaño : 86 x 73

Pos Inicial : 11 x 42

Pos Final : 3 x 53

Distancia recorrida : 22



Para la segunda prueba se recorrió mucha menos distancia y el algoritmo tardo mucho menos en encontrar el camino de salida. El algoritmo tardó 0.00618 segundos y recorrió una distancia de 22 pasos.

Tiempo demorado en crear el laberinto : 3.1071382999944035

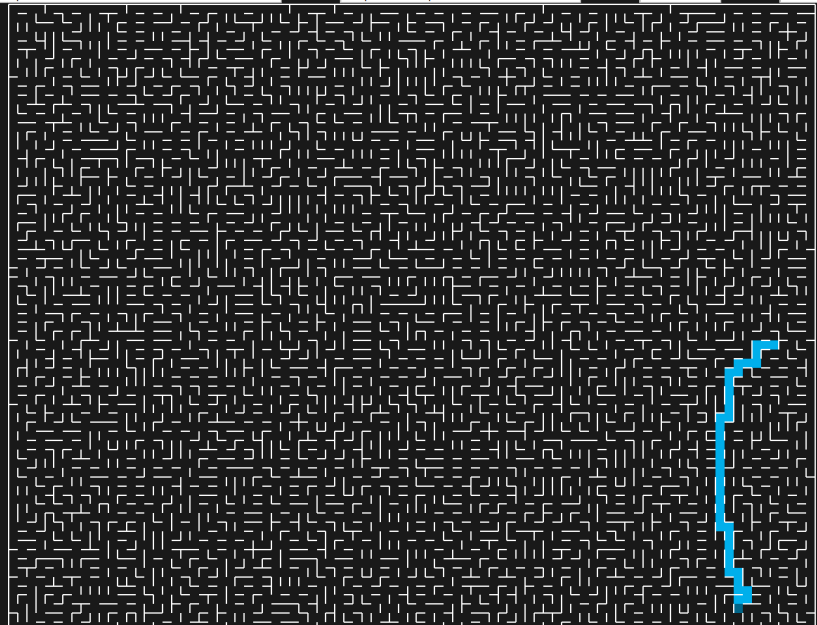
Tiempo demorado por A* : 0.009760699998878408

Tamaño : 89 x 69

Pos Inicial : 38 x 85

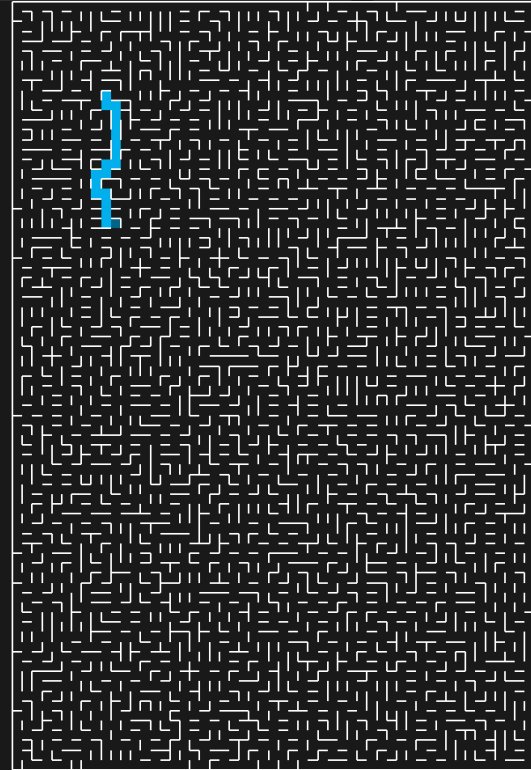
Pos Final : 67 x 81

Distancia recorrida : 40



Para la tercer prueba se obtuvo una cantidad de pasos intermedio de las últimas dos pruebas con un tiempo mismamente intermedio. Se tardó 0.00976 segundos y se dieron 40 pasos.

Tiempo demorado en crear el laberinto : 1.4483046999957878
Tiempo demorado por A* : 0.004872800003795419
Tamaño : 53 x 78
Pos Inicial : 10 x 10
Pos Final : 23 x 11
Distancia recorrida : 19



Para la cuarta y última prueba se tiene una cantidad de pasos ligeramente inferior a la segunda prueba pero se tarda significativamente menos. Se tardó 0.00487 segundos con 19 pasos recorridos.

Estas pruebas resultan en la siguiente tabla:

Tiempo tardado	Distancia Recorrida	# de prueba	Diff con la anterior
0.00487	19	4	N/A
0.00618	22	2	+0.00131
0.00976	40	3	+0.00358
0.0125	57	1	+0.00274

Según las pruebas anteriores se puede ver un comportamiento relativamente lineal, donde no aumenta de forma exagerada el tiempo del algoritmo de búsqueda, con forme la distancia del punto de inicio y de fin aumenta.

Búsquedas Adversiales: Minimax

1. Implementa las reglas y mecánicas del juego Tres en Raya. Necesitarás funciones para inicializar el estado del juego, realizar movimientos, verificar condiciones de fin de juego y determinar los movimientos legales disponibles.

```
class TicTacToe:
    """
    Esta clase implementa el algoritmo minimax para
    llevar a cabo un juego de tres en raya como matriz
    """
    # Siempre inicia el jugador que elija la "X"
    X = "X"      # Extremo max
    O = "O"      # Extremo min
    EMPTY = None

    def initial_state(self):
        """
        Inicializar el estado del juego
        """
        return [[self.EMPTY, self.EMPTY, self.EMPTY],
                [self.EMPTY, self.EMPTY, self.EMPTY],
                [self.EMPTY, self.EMPTY, self.EMPTY]]

    def player(self, board):
        """
        Retorna al jugador con el turno para jugar
        """
        x_count = sum(row.count(self.X) for row in board)
        o_count = sum(row.count(self.O) for row in board)

        if x_count > o_count: # Como 'X' siempre inicia entonces si lleva mayor cantidad
                               # de turnos, significa que va 'O'
            return self.O
        else:
            return self.X

    def actions(self, board):
        """
        Retorna un conjunto de posibles movimientos (fila,columna) según el estado actual
        del tablero
        """
        actions_set = set()

        for row in range(3):
```

```

        for col in range(3):
            if board[row][col] == self.EMPTY:
                actions_set.add((row, col))

        return actions_set

def result(self, board, action):
    """
    Retorna el tablero resultante del movimiento que se recibe
    """
    if board[action[0]][action[1]] is not self.EMPTY:
        raise Exception("Movimiento no es válido")

    new_board = copy.deepcopy(board) # Crear una copia independiente a la variable
de tablero original
    new_board[action[0]][action[1]] = self.player(board) # player va a retornar el
jugador del turno actual y se asignará a la casilla del tablero
    return new_board

def winner(self, board):
    """
    Retorna el ganador del juego si no hay empate
    """
    for player in [self.X, self.O]:
        # Revisar filas, columnas y diagonales
        if any(all(board[i][j] == player for j in range(3)) for i in range(3)) or \
            any(all(board[i][j] == player for i in range(3)) for j in range(3)) or \
            all(board[i][i] == player for i in range(3)) or \
            all(board[i][2 - i] == player for i in range(3)):
            return player
    return None

def terminal(self, board):
    """
    Retorna True si el juego terminó, False en caso contrario
    """
    # Un juego ha terminado si la función winner retorna a algún jugador o
    # si ninguna casilla está vacía (empate)
    return self.winner(board) is not None or all(board[i][j] is not self.EMPTY for i
in range(3) for j in range(3))

def utility(self, board):
    """

```

```

Retorna 1 si X ganó, -1 si O ganó, 0 empate
"""
winner = self.winner(board)
if winner == self.X:
    return 1
elif winner == self.O:
    return -1
else:
    return 0

```

2. Implementa el algoritmo Minimax. Tu función de Minimax debe tomar como entrada el estado actual del juego y devolver el mejor movimiento para el jugador actual.

```

def minimax(self, board):
    """
    Retorna el movimiento óptimo para el jugador activo
    """
    if self.terminal(board):
        return None

    if self.player(board) == self.X:
        return self.max_value(board)[1]
    else:
        return self.min_value(board)[1]

def max_value(self, board):
    """
    Función recursiva que devuelve el valor máximo de una acción de
    todas las acciones posibles en el estado actual
    """
    if self.terminal(board): # Verificar si el juego ha terminado
        return self.utility(board), None

    max_value = float('-inf')
    max_actions = []

    for action in self.actions(board):
        value, _ = self.min_value(self.result(board, action))
        if value > max_value:
            max_value = value
            max_actions = [action]
        elif value == max_value: # Si hay movimientos con el mismo puntaje, se agrega a
            la lista de acciones
            max_actions.append(action)

```

```

# Se elige aleatoriamente alguna de las acciones con los mayores puntajes
# Si solo hay una con el valor máximo, solo esa se podrá elegir
return max_value, random.choice(max_actions)

def min_value(self, board):
    """
    Función recursiva que devuelve el valor mínimo de una acción de
    todas las acciones posibles en el estado actual
    """
    if self.terminal(board): # Verificar si el juego ha terminado
        return self.utility(board), None

    min_value = float('inf')
    min_actions = []

    for action in self.actions(board):
        value, _ = self.max_value(self.result(board, action))
        if value < min_value:
            min_value = value
            min_actions = [action]
        # Si hay movimientos con el mismo puntaje, se agrega a la lista de acciones
        elif value == min_value:
            min_actions.append(action)

    # Se elige aleatoriamente alguna de las acciones con los menores puntajes
    # Si solo hay una con el valor mínimo, solo esa se podrá elegir
    return min_value, random.choice(min_actions)

```

3. Implementa una función para evaluar el estado del juego. Esta función debe devolver un valor numérico que indique la deseabilidad del estado actual del juego para el jugador activo. Un valor positivo indica una ventaja para el jugador activo, mientras que un valor negativo indica una ventaja para el oponente.

```

def check_lines(self, board):
    """
    Revisa todas las líneas rectas verticales y horizontales
    retorna 1 si X tiene la ventaja, -1 si O tiene la ventaja
    y 0 si ninguno tiene la ventaja
    """
    x_winning = False
    o_winning = False

    # Revisar horizontalmente

```

```

for row in board:
    x_count = row.count(self.X)
    o_count = row.count(self.O)
    if x_count == 2 and o_count == 0:
        x_winning = True
    elif x_count == 0 and o_count == 2:
        o_winning = True

# Revisar verticalmente
t_board = list(map(list, zip(*board)))
for col in t_board:
    x_count = col.count(self.X)
    o_count = col.count(self.O)
    if x_count == 2 and o_count == 0:
        x_winning = True
    elif x_count == 0 and o_count == 2:
        o_winning = True

if(x_winning and not o_winning):
    return 1
elif(o_winning and not x_winning):
    return -1
else:
    return 0

def check_diag(self, board):
    """
    Revisa las diagonales y retorna 1 si X tiene la ventaja,
    -1 si O tiene la ventaja y 0 si ninguno tiene la ventaja
    """

    # Revisar la primer diagonal
    first_diag = [board[0][0], board[1][1], board[2][2]]
    x_count = first_diag.count(self.X)
    o_count = first_diag.count(self.O)
    if x_count == 2 and o_count == 0:
        return 1
    elif x_count == 0 and o_count == 2:
        return -1

    # Revisar la segunda diagonal
    second_diag = [board[0][2], board[1][1], board[2][0]]
    x_count = second_diag.count(self.X)
    o_count = second_diag.count(self.O)

```

```

        if x_count == 2 and o_count == 0:
            return 1
        elif x_count == 0 and o_count == 2:
            return -1

        return 0

def evaluate_game(self, board, invert=False):
    """
    Evalua el estado actual del juego y retorna un 1 si el estado es positivo para el
    jugador activo y un valor negativo si es ventaja para su oponente
    """
    diags = 0
    lines = self.check_lines(board)
    if invert:
        player = self.X if self.player(board) is self.O else self.O
    else:
        player = self.player(board)

    if not lines:
        diags = self.check_diag(board)
        if diags != 0:
            if player is self.X:
                print(f"Estado actual del juego para {player}: ", diags)
                return
            else:
                print(f"Estado actual del juego para {player}: ", diags*-1)
                return

    else:
        if player is self.X:
            print(f"Estado actual del juego para {player}: ", lines)
            return
        else:
            print(f"Estado actual del juego para {player}: ", lines*-1)
            return

    print(f"Estado actual del juego para {player}: ", 0)

```

La salida del juego se veria de la siguiente manera:

```
-----
Estado actual del juego para O:  0
X |  | 
-----
  | O | 
-----
  |  | X
-----
La IA va a hacer un movimiento

Estado actual del juego para O:  1
-----
Estado actual del juego para X: -1
X |  | 
-----
  | O | O
-----
  |  | X
-----
Es tu turno, X
Ingrese su movimiento (fila,col): 1,0
Estado actual del juego para X:  1
-----
Estado actual del juego para O: -1
X |  | 
-----
X | O | O
-----
  |  | X
-----
La IA va a hacer un movimiento

Estado actual del juego para O:  1
-----
Estado actual del juego para X: -1
X |  | 
-----
X | O | O
-----
O |  | X
-----
Es tu turno, X
Ingrese su movimiento (fila,col): 0,2
Estado actual del juego para X:  1
-----
```

Se le da un valor de 0 cuando no se presenta una ventaja real, 1 cuando el jugador activo tiene ventaja y -1 cuando el oponente tiene ventaja.

4. Agrega un factor de aleatoriedad al juego: si el algoritmo puede tener dos caminos con el mismo puntaje, escoger aleatoriamente entre cada uno.

```
def max_value(self, board):  
  
    .  
    .  
    .  
  
    # Se elige aleatoriamente alguna de las acciones con los mayores puntajes  
    # Si solo hay una con el valor máximo, solo esa se podrá elegir  
    return max_value, random.choice(max_actions)  
  
def min_value(self, board):  
  
    .  
    .  
    .  
  
    # Se elige aleatoriamente alguna de las acciones con los menores puntajes  
    # Si solo hay una con el valor mínimo, solo esa se podrá elegir  
    return min_value, random.choice(min_actions)
```

5. Investiga la variante Alpha-Beta para mejorar el rendimiento del algoritmo MinMax.

La eficiencia del algoritmo MinMax se ve limitada por la necesidad de analizar exhaustivamente todas las posibilidades, en juegos más complejos que el Tres en Raya puede llegar a ser computacionalmente costoso, por esto, surgió la variante o poda Alpha-Beta que consiste en eliminar las ramificaciones o los caminos donde se haya observado un movimiento con peor valor que algún movimiento analizado anteriormente.

El Alpha-Beta no evalúa a más profundidad ese camino y esto no influye en la decisión final, de esta forma se hace un mejor uso de los recursos y el minimax alcanza una mayor eficiencia. Donald Knuth y Ronald W. Moore refinaron el algoritmo en 1975; Judea Pearl exhibió la mejora en el tiempo de ejecución previsto en árboles con valores de hojas asignados de manera aleatoria en dos publicaciones. En 1986, Michael Saks y Avi Wigderson demostraron la mejora en la versión aleatoria de alfa-beta (Academia Lab, 2024).

Se utilizan los parámetros α y β para hacer el seguimiento de la mejor puntuación de cada jugador mientras se recorre el árbol, entonces Alpha representa el mejor resultado

para Max y Beta el de Min. Durante la exploración, cuando se encuentra un nodo que tiene un valor mayor que beta (en el caso del jugador maximizante) o menor que alfa (en el caso del jugador minimizante), se sabe que el resultado en ese nodo no influirá en la decisión final, por lo que se poda la búsqueda en ese nodo y sus descendientes. Primero se deberían analizar las ramas con mayor probabilidades de éxito y luego se configuran valores para alpha y beta. En la siguiente sección de pseudo-código (Aradhya, 2023):

function minimax(node, depth, isMaximizingPlayer, alpha, beta):

if node is a leaf node :

return value of the node

if isMaximizingPlayer :

 bestVal = -INFINITY

for each child node :

 value = minimax(node, depth+1, false, alpha, beta)

 bestVal = max(bestVal, value)

 alpha = max(alpha, bestVal)

if beta <= alpha:

break

return bestVal

else :

 bestVal = +INFINITY

for each child node :

 value = minimax(node, depth+1, true, alpha, beta)

 bestVal = min(bestVal, value)

 beta = min(beta, bestVal)

if beta <= alpha:

break

return bestVal

En un juego con tantas combinaciones de movimientos posibles por partida como el ajedrez, al algoritmo minimax representa gran uso de recursos computacionales al tener un árbol tan profundo y ancho; por esto se implementó la variante alpha-beta con la computadora Deep Blue. Deep Blue fue desarrollado por IBM y logró ganarle a Gari Kaspárov, conocido como el mejor jugador de ajedrez profesional del mundo en el año 1997, después de haber perdido contra Kaspárov en 1996, quedando el marcador 4-2. Cabe recalcar, que en 1997 Deep Blue ganó al tener una unidad de procesamiento más potente que le permitieron calcular millones de posiciones por segundo (Candial, 2021). Enfatizando así la importancia de variantes como estas que mejoran el rendimiento del minimax.

Referencias:

Academia Lab. (2024). Poda alfa-beta. Enciclopedia. Revisado el 21 de febrero del 2024.
<https://academia-lab.com/enciclopedia/poda-alfa-beta/>

Candial, A. F. (2021, February 10). Deep Blue-Kaspárov: cuando la máquina venció al hombre.

La

Vanguardia.

<https://www.lavanguardia.com/vida/junior-report/20210210/6234712/kasparov-deep-blue-maquina-vencio-hombre.html>

GfG. (2023, January 16). *Minimax Algorithm in Game Theory Set 4 (Alpha-Beta Pruning)*.

GeeksforGeeks.

<https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/>

Pruebas de algoritmo Minimax

```
| | |
-----
| | |
-----
| | |
-----
Tres en raya
Elija un jugador:
1. X
2. O
Ingrese su respuesta: 1
| | |
-----
| | |
-----
| | |
-----
Es tu turno, X
Ingrese su movimiento (fila,col): 1,1
| | |
-----
| X |
-----
| | |
-----
La IA va a hacer un movimiento
| | O
-----
| X |
-----
| | |
-----
Es tu turno, X
Ingrese su movimiento (fila,col): 2,1
| | O
-----
| X |
-----
| X |
-----
```

```
La IA va a hacer un movimiento
| O | O
-----
| X |
-----
| X |
-----
Es tu turno, X
Ingrese su movimiento (fila,col): 0,0
X | O | O
-----
| X |
-----
| X |
-----
La IA va a hacer un movimiento
X | O | O
-----
| X |
-----
| X | O
-----
Es tu turno, X
Ingrese su movimiento (fila,col): 1,2
X | O | O
-----
| X | X
-----
| X | O
-----
La IA va a hacer un movimiento
X | O | O
-----
O | X | X
-----
| X | O
-----
```

```
Es tu turno, X
Ingrese su movimiento (fila,col): 2,0
X | O | O
-----
O | X | X
-----
X | X | O
-----
Game over. Es un empate!
```

- El usuario elige ser el jugador 'X', por lo que tiene el primer turno.
- El usuario juega óptimamente.
- Termina en empate.

```

|  |
-----
|  |
-----
|  |
-----
Tres en raya
Elija un jugador:
1. X
2. O
Ingrese su respuesta: 1
|  |
-----
|  |
-----
|  |
-----
Es tu turno, X
Ingrese su movimiento (fila,col): 2,1
|  |
-----
|  |
-----
| X |
-----
La IA va a hacer un movimiento
|  |
-----
| O |
-----
| X |
-----
Es tu turno, X
Ingrese su movimiento (fila,col): 2,2
|  |
-----
| O |
-----
| X | X
-----

```

```

La IA va a hacer un movimiento
|  |
-----
| O |
-----
O | X | X
-----
Es tu turno, X
Ingrese su movimiento (fila,col): 1,2
|  |
-----
| O | X
-----
O | X | X
-----
La IA va a hacer un movimiento
|  | O
-----
| O | X
-----
O | X | X
-----
Game over. O es el ganador!

```

- El usuario elige ser el jugador 'X', por lo que tiene el primer turno.
- El usuario NO juega óptimamente.
- Gana el jugador 'O' (IA).

```

  | |
-----
  | |
-----
  | |
-----
Tres en raya
Elija un jugador:
1. X
2. O
Ingrese su respuesta: 2
  | |
-----
  | |
-----
  | |
-----
La IA va a hacer un movimiento
  | |
-----
  | | X
-----
  | |
-----
Es tu turno, O
Ingrese su movimiento (fila,col): 1,1
  | |
-----
  | O | X
-----
  | |
-----
La IA va a hacer un movimiento
  | |
-----
  | O | X
-----
  | X |
-----

```

```

Es tu turno, O
Ingrese su movimiento (fila,col): 2,2
  | |
-----
  | O | X
-----
  | X | O
-----
La IA va a hacer un movimiento
X | |
-----
  | O | X
-----
  | X | O
-----
Es tu turno, O
Ingrese su movimiento (fila,col): 0,2
X | | O
-----
  | O | X
-----
  | X | O
-----
La IA va a hacer un movimiento
X | | O
-----
  | O | X
-----
X | X | O
-----
Es tu turno, O
Ingrese su movimiento (fila,col): 1,1
Esta celda ya está ocupada. Intente de nuevo.
Ingrese su movimiento (fila,col): 1,0
X | | O
-----
O | O | X
-----
X | X | O
-----

```

```

La IA va a hacer un movimiento
X | X | O
-----
O | O | X
-----
X | X | O
-----
Game over. Es un empate!
  | |
-----
  | |
-----
  | |
-----

```

- El usuario elige ser el jugador 'O', por lo que tiene el segundo turno.
- El usuario juega óptimamente.
- Termina en empate.

Game over. Es un empate!

```
| |  
-----
```

```
| |  
-----
```

```
| |  
-----
```

```
| |  
-----
```

Tres en raya

Elija un jugador:

1. X

2. O

Ingrese su respuesta: 2

```
| |  
-----
```

```
| |  
-----
```

```
| |  
-----
```

```
| |  
-----
```

La IA va a hacer un movimiento

```
| |  
-----
```

```
| X |  
-----
```

```
| |  
-----
```

```
| |  
-----
```

Es tu turno, O

Ingrese su movimiento (fila,col): 0,2

```
| | O  
-----
```

```
| X |  
-----
```

```
| |  
-----
```

```
| |  
-----
```

La IA va a hacer un movimiento

```
| | O  
-----
```

```
X | X |  
-----
```

```
| |  
-----
```

```
| |  
-----
```

Es tu turno, O

Ingrese su movimiento (fila,col): 1,2

```
| | O  
-----
```

```
X | X | O  
-----
```

```
| |  
-----
```

```
| |  
-----
```

La IA va a hacer un movimiento

```
| | O  
-----
```

```
X | X | O  
-----
```

```
| | X  
-----
```

```
| |  
-----
```

Es tu turno, O

Ingrese su movimiento (fila,col): 0,1

```
| O | O  
-----
```

```
X | X | O  
-----
```

```
| | X  
-----
```

```
| |  
-----
```

La IA va a hacer un movimiento

```
X | O | O  
-----
```

```
X | X | O  
-----
```

```
| | X  
-----
```

```
| |  
-----
```

Game over. X es el ganador!

- El usuario elige ser el jugador 'O', por lo que tiene el segundo turno.
- El usuario NO juega óptimamente.
- Gana el jugador 'X' (IA).

Prueba contra un Oponente Aleatorio

1. Implementa un oponente aleatorio. El oponente aleatorio debe realizar movimientos legales al azar sin considerar el estado del juego.

```
def get_random_move(self):
    print(f"El oponente aleatorio va hacer su movimiento")

    empty_spaces = []

    for i,row in enumerate(self.board):
        for j,element in enumerate(row):
            if element is None:
                empty_spaces.append((i,j))

    random_pos = random.choice(empty_spaces)

    self.board = self.tic_tac_toe.result(self.board, random_pos)
```

2. Configura un bucle de simulación para jugar varios juegos. En cada juego, alterna entre el algoritmo Minimax y el oponente aleatorio para realizar movimientos.

```
def random_oponent_test(self):
    test_scenarios=[self.tic_tac_toe.X,self.tic_tac_toe.O,None]
    msg="\nEjecutando 500 pruebas del escenario: "
    # Las columnas son: P1, P2, Empates
    # Las Filas: Random - Minmax
    #             Minmax - Random
    #             Minmax - Minmax
    scenario_statistics=[["Random - Minmax", 0,0,0],
                        ["Minmax - Random", 0,0,0],
                        ["Minmax - Minmax", 0,0,0]]

    winner = None

    for i,test in enumerate(test_scenarios):
        scenario_func = self.play_game_AI if test is None else self.play_game_random
        match i:
            case 0:
                print(msg, "Random - Minmax")
            case 1:
                print(msg, "Minmax - Random")
            case 2:
                print(msg, "Minmax - Minmax")
        for j in range(500):
            print(f"\n----- Prueba #{j+1} -----")
```

```

self.user = self.tic_tac_toe.X if test is None else test
while True:
    if self.user is None:
        match(winner):
            case self.tic_tac_toe.X:
                scenario_statistics[i][1] += 1
            case self.tic_tac_toe.O:
                scenario_statistics[i][2] += 1
            case None:
                scenario_statistics[i][3] += 1
        break
    #self.print_board()
    winner = scenario_func()

self.test_statistics(scenario_statistics)

```

- Realiza un seguimiento de los resultados de los juegos (ganar/perder/empatar) para el algoritmo Minimax. Prueba generando 500 juegos y obtenga estadísticas generales sobre el funcionamiento del algoritmo.

```

Pruebas terminadas... Imprimiendo resultados...

Estilo de juego   P1 Gana   P2 Gana   Empate
Random - Minmax   0         383       117
Minmax - Random   480       0         20
Minmax - Minmax   0         0         500
Tres en raya
Elija un jugador:

```

Esto lleva a la siguiente tabla:

Estilo de Juego	P1 Gana	P2 Gana	Empate
Random - Minmax	0%	76.6%	23.4%
Minxmax - Random	96%	0%	4%
Minmax - Minmax	0%	0%	100%

El comportamiento parece indicar que contra un oponente aleatorio, en el caso de empezar el oponente aleatorio, es más complicado que el algoritmo Minmax gane. Sin embargo, si el algoritmo Minxmax empieza, tiene más probabilidades de ganar.

Para el caso de Minmax vs Minmax, cancelan sus jugadas causando que siempre se genere un empate.

