Algoritmo Minimax y Alpha-Beta prunning

Minimax El algoritmo de minimax en simples palabras consiste en la elección del mejor movimiento para el computador, suponiendo que el contrincante escogerá uno que lo pueda perjudicar, para escoger la mejor opción este algoritmo realiza un árbol de búsqueda con todos los posibles movimientos, luego recorre todo el árbol de soluciones del juego a partir de un estado dado, es decir, según las casillas que ya han sido rellenadas.

Alpha-Beta prunning

Es una versión modificada del algoritmo minimax. Es una técnica de optimización para el algoritmo minimax.

Los dos parámetros se pueden definir como: Alfa: la mejor opción (de mayor valor) que hemos encontrado hasta ahora en cualquier punto del camino de Maximizer. El valor inicial de alfa es $-\infty$.

Beta: la mejor opción (valor más bajo) que hemos encontrado hasta ahora en cualquier punto del camino de Minimizer. El valor inicial de beta es $+\infty$. 1.1

Minimax Alan Turing La meta es construir un algoritmo (Algoritmo de minimax) que a partir de una posición en que juega el computador pueda escoger entre las diversas alternativas a su disposición.

John Von Newmann y Osear Morgenstern (1945) propusieron la idea básica de Minimax con la suposición fundamental de que la misma función de evaluación está disponible para ambos jugadores. La idea de función de evaluación es perfeccionada por Claude Shannon (1950). Alan Turíng en 1951 encuentra que la profundidad del árbol no necesariamente debe ser homogénea. Shanon lo sugirió originalmente basado en teoría de juegos de Von Neumann y O. Morgenstern, aunque Turing presentó el primer programa La idea: Maximizar mis tiradas considerando que el oponente va a minimizar. Para decidir qué jugada hacer, el árbol se divide por niveles:

• Max: el primer jugador (nivel) y todas las posiciones (niveles) donde juega.

• Min: el oponente y todas las posiciones en donde juega

Alpha-Beta prunning

Donald Knuth y Ronald Moore (1975) fueron los primeros en analizar a fondo la eficiencia de la poda alfa-beta, en su libro “An analysis of alpha-beta pruning”. Es posible calcular en un determinado juego una decisión correcta sin tener que explorar todos los nodos de un árbol de búsqueda. A la técnica que consideramos en particular se le conoce como poda alfa-beta. Aplicada a un árbol minimax estándar, produce la misma jugada que se obtendría con minimax, pero elimina todas las ramas que posiblemente no influirán en la decisión final. La eficiencia de alfa-beta dependerá del orden como se exploren los sucesores dentro de un árbol, es mejor primero explorar aquellos sucesores que aparentemente tienen más posibilidades de ser los mejores.

Ejemplo juego Tres en Raya

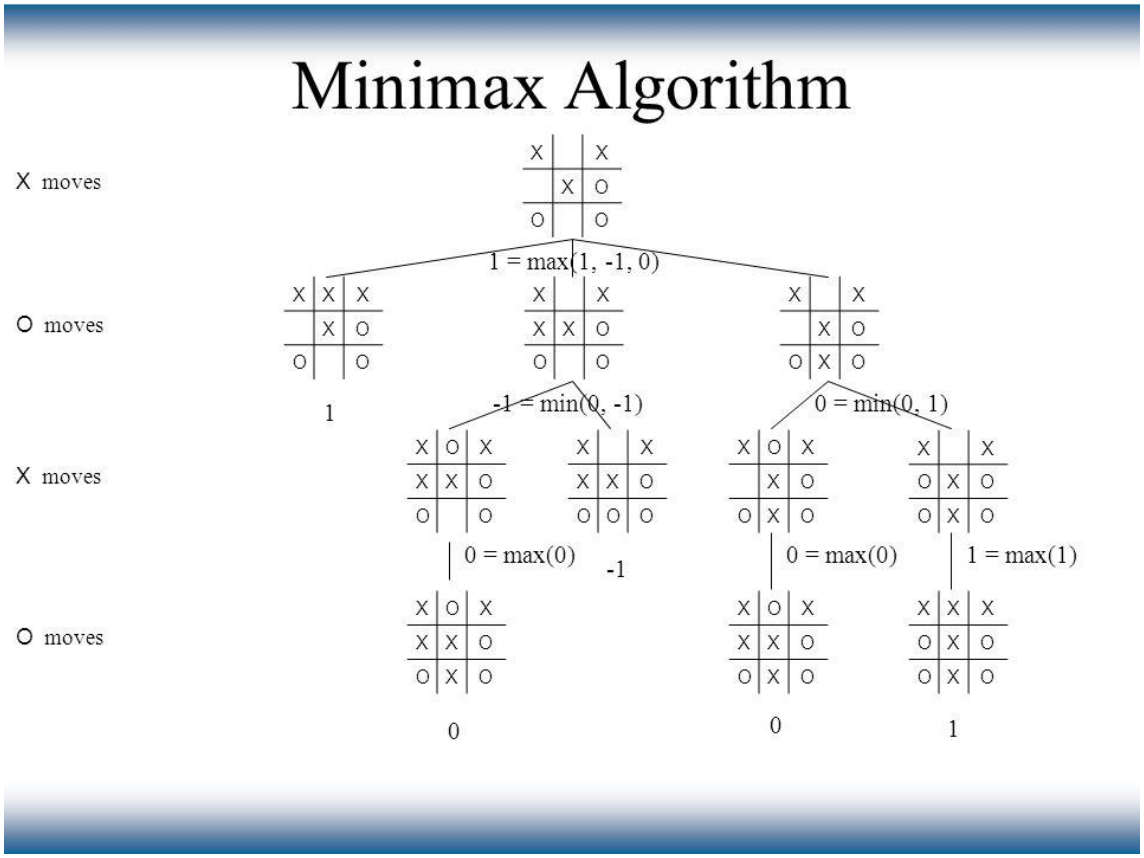
Minimax Para resolver este juego utilizaremos la siguiente heurística:

E(n)=X(n) - O(n)

donde: E(n) es la función.

X(n) son el número de movimientos posibles para el Max, en este caso el jugador de las X.

O(x) son el número de movimientos posibles para el Min, en este caso el jugador de las O.



Alpha-Beta Prunning Este algoritmo es una version mejorada de Minimax.

Implementación Juego con EasyAI

juego Connect Four (también conocido como Four Up , Plot Four , Find Four , Four in a Row , Four in a Line , Drop Four y Gravitrips (en la Unión Soviética))

es un juego de tablero de conexión para dos jugadores en el que los jugadores primero eligen un color y luego se turnan para dejar caer un disco de color desde la parte superior en una cuadrícula suspendida verticalmente de siete columnas y seis filas. Las piezas caen hacia abajo, ocupando el espacio más bajo disponible dentro de la columna. El objetivo del juego es ser el primero en formar una línea horizontal, vertical o diagonal de

cuatro discos propios. Connect Four está resueltojuego. El primer jugador siempre puede ganar jugando los movimientos correctos.

Objetivo:

conecta cuatro de tus fichas seguidas mientras evitas que tu oponente haga lo mismo. Pero ten cuidado: ¡tu oponente puede acercarte sigilosamente y ganar el juego!

In [2]:
```
import numpy as np
```

In [5]:
```
from easyAI import TwoPlayersGame


class ConnectFour(TwoPlayersGame):
    """
    The game of Connect Four, as described here:
    http://en.wikipedia.org/wiki/Connect_Four
    """

    def __init__(self, players, board = None):
        self.players = players
        self.board = board if (board != None) else (
            np.array([[0 for i in range(7)] for j in range(6)]))
        self.nplayer = 1 # player 1 starts.

    def possible_moves(self):
        return [i for i in range(7) if (self.board[:, i].min() == 0)]

    def make_move(self, column):
        line = np.argmin(self.board[:, column] != 0)
        self.board[line, column] = self.nplayer

    def show(self):
        print('\n' + '\n'.join(
                        ['0 1 2 3 4 5 6', 13 * '-'] +
                        [' '.join([['.', 'O', 'X'][self.board[5 - j][i]]
                        for i in range(7)]) for j in range(6)]))

    def lose(self):
        return find_four(self.board, self.nopponent)

    def is_over(self):
        return (self.board.min() > 0) or self.lose()

    def scoring(self):
        return -100 if self.lose() else 0


def find_four(board, nplayer):
    """
    Returns True iff the player has connected  4 (or more)
    This is much faster if written in C or Cython
    """
    for pos, direction in POS_DIR:
        streak = 0
        while (0 <= pos[0] <= 5) and (0 <= pos[1] <= 6):
            if board[pos[0], pos[1]] == nplayer:
                streak += 1
                if streak == 4:
                    return True
            else:
                streak = 0
            pos = pos + direction
    return False


POS_DIR = np.array([[[i, 0], [0, 1]] for i in range(6)] +
```

```
        while (0 <= pos[0] <= 5) and (0 <= pos[1] <= 6):
            if board[pos[0], pos[1]] == nplayer:
                streak += 1
                if streak == 4:
                    return True
            else:
                streak = 0
            pos = pos + direction
    return False


POS_DIR = np.array([[[i, 0], [0, 1]] for i in range(6)] +
                   [[[0, i], [1, 0]] for i in range(7)] +
                   [[[i, 0], [1, 1]] for i in range(1, 3)] +
                   [[[0, i], [1, 1]] for i in range(4)] +
                   [[[i, 6], [1, -1]] for i in range(1, 3)] +
                   [[[0, i], [1, -1]] for i in range(3, 7)])

if __name__ == '__main__':
    # LET'S PLAY !

    from easyAI import Human_Player, AI_Player, Negamax, SSS, DUAL

    ai_algo_neg = Negamax(5)
    ai_algo_sss = SSS(5)
    game = ConnectFour([AI_Player(ai_algo_neg), AI_Player(ai_algo_sss)])
    game.play()
    if game.lose():
        print("Player %d wins." % (game.nopponent))
    else:
        print("Looks like we have a draw.")
```

```
O . . . . . .
X . . . . . .
O . . . . . .

Move #6: player 2 plays 0 :

0 1 2 3 4 5 6
-------------
X . . . . . .
O . . . . . .
X . . . . . .
O . . . . . .
X . . . . . .
O . . . . . .

Move #7: player 1 plays 1 :

0 1 2 3 4 5 6
-------------
X . . . . . .
O . . . . . .
```

Conclusiones

Gym nos permite elaborar juegos de manera muy fácil con el uso mediante el uso de aprendizaje continuo para poder hacer más efectivas las decisiones que debe de tomar para resolver un problema en este caso ganar un juego mediante el uso de Alpha-Beta Prunning.