SEGUNDO PROYECTO

JUAN CAMILO AZCARATE CÁRDENAS

ESTEFANY CASTRO AGUDELO

DANIEL GÓMEZ SUÁREZ

VALENTINA HURTADO GARCÍA

JOSHUA DAVID TRIANA MADRID

INTRODUCCION A LA INTELIGENCIA ARTIFICIAL

UNIVERSIDAD DEL VALLE

TULUÁ

2023

Tabla de contenido

INTRODUCCION

Un juego de estrategia, como lo es el ajedrez, cuenta con una alta complejidad al contar con 32 piezas divididas en dos bandos: las blancas y las negras. El objetivo del juego tradicional es capturar el rey del oponente, evitando que el propio rey sea capturado. Teniendo en cuenta las reglas, movimiento específicos y restricciones por pieza que requieren una cuidadosa planificación.

Con esto en mente se hace uso del algoritmo Minimax para tomar las decisiones óptimas al contar con ese grado de complejidad, asumiendo que el otro jugador también va a escoger la mejor decisión, sin embargo, el tamaño exponencial de un árbol de búsqueda hace que sea poco óptimo y eficiente evaluar todas las posiciones en el juego. Por lo tanto, se utilizará la técnica de poda alfa-beta para reducir la cantidad de nodos a evaluar en la exportación, sin afectar el resultado final.

Para concluir el documento contará la implementación del código, su comportamiento y la complejidad que se presenta en las partidas.

ALGORITMO MINIMAX

El algoritmo Minimax es ampliamente utilizado en inteligencia artificial y juegos estratégicos para tomar decisiones óptimas en situaciones competitivas de suma cero. Su principal objetivo es encontrar el mejor movimiento posible en un juego de dos jugadores, asumiendo que ambos jugadores juegan de manera óptima.

Minimax en los juegos es ampliamente utilizado para tomar decisiones óptimas, encontrando el mejor movimiento para una partida en los juegos que requieren estrategia, maximizando así la ganancia propia, mientras que minimiza las del oponente, al hacer una exploración exhaustiva en el árbol de búsqueda asumiendo que su contrincante tiene el mismo objetivo.

PODA ALPHA-BETA

El algoritmo Minimax utiliza la poda alfa-beta como técnica de optimización para reducir la cantidad de nodos evaluados en un árbol de búsqueda sin alterar el resultado final. Esta técnica ayuda a ahorrar tiempo y recursos computacionales significativos al permitir una exploración del árbol de búsqueda más eficiente.

El algoritmo Minimax analiza todos los nodos de un árbol de búsqueda, pero con frecuencia encuentra ramas que no son relevantes para la toma de decisiones óptimas, por lo cual Interrumpe la búsqueda en algún nivel y aplica evaluaciones heurísticas a las hojas (profundidad limitada). Si el valor del nodo MAX (alfa) es menor que el más alto hasta este momento, entonces omitir nodo. Si el valor del nodo MIN (beta) es mayor que el nodo más bajo hasta el momento, entonces omitir nodo.

Esta búsqueda alfa-beta va actualizando el valor de los parámetros según se recorre el árbol. El método realizará la poda de las ramas restantes cuando el valor actual que se está examinando sea peor que el valor actual de MAX o MIN.

¿CÓMO FUNCIONA?

Tablero:

Hacemos declaraciones iniciales y le decimos las piezas que contendrá

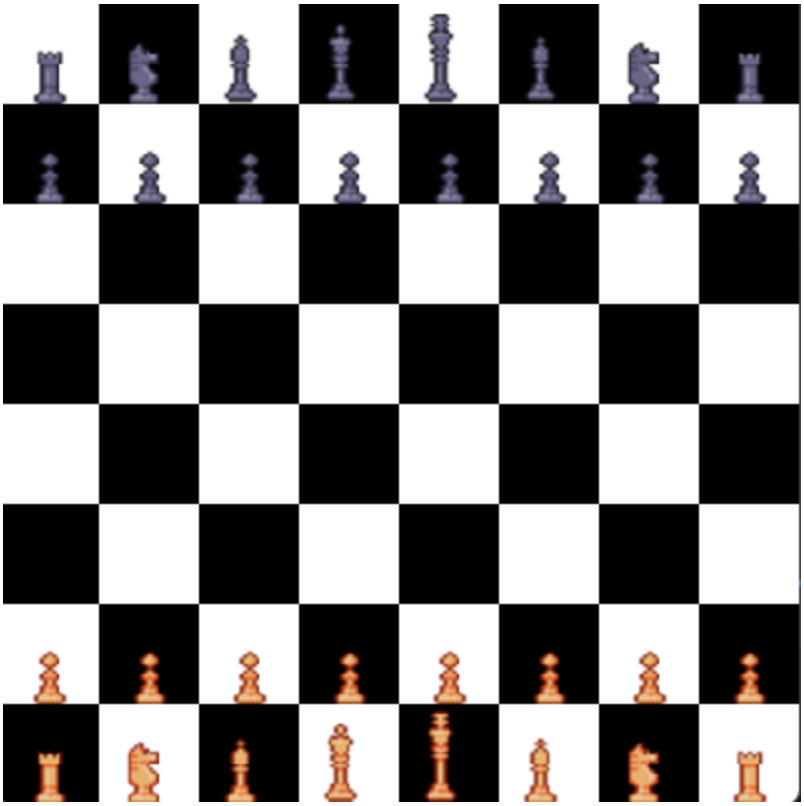
```
import ChessAlgoritmo
import pygame as p
import math


ancho = alto = 400
dim = 8
sqsize = alto // dim
frames = 10
images = {}

def loadImages():
    pieces = ["wP","wR","wN","wB","wQ","wK","bP","bR","bN","bB","bQ","bK"]
    for piece in pieces:
        images[piece] = p.transform.scale(p.image.load("images/" + piece +".png"),(sqsize,sqsize))

def main():
    p.init()
    screen = p.display.set_mode((ancho,alto))
    clock = p.time.Clock()
    screen.fill(p.Color("White"))
```

se insertan las piezas y se dibuja el tablero.

```
def drawEstado(screen, gs):
    drawBoard(screen)
    drawPieces(screen, gs.board)

def drawBoard(screen):
    colours = [p.Color("white"),p.Color("black")]
    for r in range(dim):
        for c in range(dim):
            colour = colours[((r + c) % 2)]
            p.draw.rect(screen, colour, p.Rect(c*sqsize, r*sqsize, sqsize, sqsize))



def drawPieces(screen,board):
    for r in range(dim):
        for c in range(dim):
            piece = board[r][c]
            if piece != "--":
                screen.blit(images[piece], p.Rect(c*sqsize, r*sqsize, sqsize, sqsize))
```

Después declaramos las posiciones de las piezas.

```
class Estado():
    def __init__(self):

        self.board = [
            ["bR","bN","bB","bQ","bK","bB","bN","bR"],
            ["bP","bP","bP","bP","bP","bP","bP","bP"],
            ["--","--","--","--","--","--","--","--"],
            ["--","--","--","--","--","--","--","--"],
            ["--","--","--","--","--","--","--","--"],
            ["--","--","--","--","--","--","--","--"],
            ["wP","wP","wP","wP","wP","wP","wP","wP"],
            ["wR","wN","wB","wQ","wK","wB","wN","wR"]]
        self.moveFunctions = {'P': self.MovimientosPeon, 'R': self.MovimientosTorre, 'N': self.MovimientosCaball
                              'B': self.MovimientoAlfil, 'Q': self.MovimientosReyna, 'K': self.MovimientosRey}
```



Interfaz gráfica del tablero y sus piezas

Implementación:

Minimax y Poda alpha-beta: mediante el uso de math y profundidad rectificamos las jugadas y aplicamos minimax, evaluando en max y en min por forma separada.

```
def minimax(self, profundidad, alpha, beta, Max):
    possibleMoves = self.getJugadasV()
    value = self.MejoresJugadas()
    if profundidad == 0 or value == -math.inf or value == math.inf:
            return value
    if Max:
        maxEval = -math.inf
        for move in possibleMoves:
            self.HacerMovimiento(move)
            evaluation = self.minimax(profundidad - 1, alpha, beta, False)
            self.deshacer()
            if (evaluation > maxEval):
                maxEval = evaluation
            alpha = max(alpha, evaluation)
            if beta <= alpha:
                break
        return maxEval
```

```
        else:
            minEval = math.inf
            for move in possibleMoves:
                self.HacerMovimiento(move)
                evaluation = self.minimax(profundidad - 1, alpha, beta, True)
                self.deshacer()
                minEval = min(evaluation, minEval)
                beta = min(evaluation, beta)
                if beta <= alpha:
                    break
            return minEval
```

Evaluamos las diferentes posibilidades y retornamos la mejor.

```
def Mejorjugada(self, profundidad, Max):
    mateTemp = copy.deepcopy(self.mate)
    tablasTemp = copy.deepcopy(self.tablas)
    torreTemp = copy.deepcopy((self.enroqueDerecho.wks, self.enroqueDerecho.bks,
                               self.enroqueDerecho.wqs, self.enroqueDerecho.bqs))

    self.AIturn = True
    moves = self.getJugadasV()
    bestValue = -math.inf if Max else math.inf
    bestMove = None
    for move in moves:
        self.HacerMovimiento(move)
        value = self.minimax(profundidad - 1, -math.inf, math.inf, not Max)
        self.deshacer()
        if value == -math.inf and not Max:
            return move
        elif value == math.inf and Max:
            return move
```

Heurística: Mediante un sistema de puntos evaluamos la heurística y encontramos la jugada más rentable, ejemplo de uno de ellos:

```
#Pawns
wPjugada = [[ 0,  0,  0,  0,  0,  0,  0,  0],
            [50, 50, 50, 50, 50, 50, 50, 50],
            [10, 10, 20, 30, 30, 20, 10, 10],
            [5,  5, 10, 25, 25, 10,  5, 5],
            [0,  0,  0, 20, 20,  0,  0, 0],
            [5, -5,-10,  0,  0,-10, -5,  5],
            [5, 10, 10,-20,-20, 10, 10,  5],
            [0,  0,  0,  0,  0,  0,  0,  0]]
```

```
values = {'wP': 100, 'wR':500, 'wN':300, 'wB':300, 'wQ':900, 'wK':20000,
          'bP': -100, 'bR':-500, 'bN':-300, 'bB':-300, 'bQ':-900, 'bK':-20000}
```

CONCLUSIONES

Mediante la implementación de algoritmos como Minimax o alpha-beta pudimos mejorar nuestra comprensión en el uso de un sistema de IA a partir de del uso de heurísticas para buscar la evaluación de un objetivo y así escoger la solución más óptima para el mismo, además de ello pudimos profundizar nuestra comprensión de la creación de aplicaciones de escritorio mediante diferentes herramientas y/o librerías como Tkinter y Chess para el área gráfica del proyecto, así como métodos matemáticos como lo es math, también cabe mencionar que mediante el uso de la aplicación logramos ver la abrumadora diferencia en la capacidad de una máquina a la hora de jugar, denotando de forma masiva que esta está mejor capacitada que nosotros para este juego en particular teniendo en cuenta la cantidad de cálculos que debe hacer en el instante, lo que terminó sorprendiéndonos gratamente con el resultado.