Tarea 1: Busqueda Adversarial en Juegos

Universidad Andrés Bello®

Fundamentos de Inteligencia Artificial
CINF103.202220.7308.TR

Docente: Pablo Schwarzenberg
Universidad Nacional Andrés Bello

Excepto que se indique lo contrario, todo lo incluido en este informe es producto de mi trabajo e investigación.

Felipe Joaquín Pastén Cáceres
18 de septiembre de 2022

Contenidos

Capítulo 1

Introducción

1.1. Planteamiento del problema

En el contexto de comprender mejor el desarrollo de agentes de inteligencia artificial se requiere buscar posibles soluciones para resolver el juego Othello.

1.2. Base teórica

En este informe se trabajará con algoritmos de búsqueda

1.3. Objetivo

El objetivo general de esta tarea es comprender desempeño y complejidad de las distintas formas de resolver un problema computacional con inteligencia artificial, los objetivos especificos son,

1. Aplicar técnicas de búsqueda y razonamiento en Othello.
2. Diseñar sistemas computacionales basados en agentes inteligentes
3. Implementar sistemas inteligentes para resolver jugadas de Othello.

1.4. Esquema del informe

Este reporte continúa con 4 partes principales 2 Conceptos Teóricos, 3 Metodología, 4 Análisis y Resultados, 5 Conclusiones.

Capítulo 2

Conceptos Teoricos

2.1. Análisis del Problema

2.1.1. Othello

Othello es un juego que está dentro de la categoría de 'zero-sum' esto tiene implicancias en el modelamiento del problema ya que se pueden derivar ciertas condiciones. 'Zero-Sum' implica que la ganancia de un jugador es equivalente para el otro jugador pero negada, es decir

$$Ganancia_{Jugador1} = -Ganancia_{Jugador2} \tag{2.1}$$

Esto permite tener una única matriz de ganancia sobre la cual el jugador 1 maximiza y el jugador 2 minimiza, la sentencia anteriormente expresada permite realizar esto de forma matemática y así se pueden plantear algoritmos en base a esta sentencia que permitan resolver estados de juego maximizando ganancia dependeniendo del jugador o turno.

2.1.1.1. Reglas

- El jugador con piezas negras tiene el primer turno
- Si un jugador no tiene posibles formas de flanquear y dar vuelta al menos una pieza del oponente, debe ceder su turno, si tiene disponible una jugada no puede ceder su turno.
- Una pieza puede flanquear cualquier cantidad de piezas en una o más filas en dirección ortogonal y diagonal.
- No se pueden saltar piezas propias para flanquear piezas del oponente.
- Piezas solo se pueden flanquear y voltear como el directo resultado de un movimiento, si existen otras piezas flanqueadas no en linea directa pero como resultado del movimiento, estas no se voltean.

2.1.2. Requisitos funcionales

Para la implementación de este juego se requiere cumplir con algunos requisitos funcionales

- Permite elegir el tamaño del tablero entre 6x6 y 8x8.
- Permite elegir el nivel de dificultad entre al menos 3 niveles.
- Registrar e imprimir el numero de nodos explorados y el tiempo utilizado durante la selección de jugada de la computadora.
- Poseer una interfaz que sea de fácil uso, que permita ver el estado del juego en forma clara y seleccionar una jugada sin inducir a errores.
- Agregar una caracteristica adicional entre:

  - Sugerir una jugada al jugador humano si es que lo pide.

  - Ajustar el nivel de dificultad de acuerdo a la habilidad del jugador, recordando partidas anteriores

2.1.3. Algoritmos para Zero Sum

2.1.3.1. Minmax Theorem

Como indica Kjeldsen en 1928 Von Veumann junto con Morgenstern escriben la base teorica para el planteamiento de algoritmos en el marco de la teoría de juegos y comportamiento económico, En este establecen la relación entre teoría económica y de juegos. El teorema minmax plantea que para juegos finitos de dos participantes que sean del tipo zero sum, existe al menos una estategia mixta, por tanto existe un valor de juego que aplicando la estrategia optima garantice una ganancia no peor que P para si mismo, y a la vez garantice una ganancia no peor que -P para el adversario.

Esto se traduce en la siguiente sentencia,

$$min\{max\{F(x,y) : y \in Y\} : x \in X\} = max\{min\{F(x,y) : x \in X\} : y \in Y\} = F(x^*, y^*) \tag{2.2}$$

Existe un punto de silla $(x^*, y^*) \in X \times Y$ que satisface la ecuación anterior. Las demostraciones de esto se pueden encontrar en la publicación original por Von Neumann and Morgenstern, pero para mejor entendimiento las publicaciones de Kuhn and Tucker y Kjeldsen hacen una revisión más facil de digerir sobre todo la última.

Al llevar esto a un algoritmo se entiende que es necesario entonces recorrer la matriz de ganancia, buscando este punto de silla que garantiza utilidad P, esta función es recursiva ya que se deben explorar todos los movimientos legales, para encontrar la mejor utilidad, Como se indica en Aljubran [1] este algoritmo depende de la busqueda en profundida (DFS) con la variación de que asume cada nivel como un turno, por lo tanto, alterna la maximización y minimizacion para así obtener la mejor ganancia.

El pseudo código se muestra a continuación,

Algorithm 1 MiniMax

```
function MiniMax(node, depth, maximizingPlayer)
    if maximizingPlayer then
        bestValue ← −infinite
        for child of node do
            bestValue ← max(bestValue, minimax(child, depth − 1, FALSE))
        end for
        return bestValue
    else
        bestValue ← +infinite
        for child of node do
            bestValue ← min(bestValue, minimax(child, depth − 1, TRUE))
        end for
        return bestValue
    end if
end function
```

Al observar el detalle de del pseudocódigo, se puede apreciar que este algoritmo recorrerá todas las posibles jugadas, no tiene un mecanismo de descarte de jugadas que representen menor ganancia, este es el punto de optimización que se plantea para el siguiente algoritmo

2.1.3.2. Poda alfa beta

El algoritmo minimax recorre todos los nodos aunque sea obvio que una jugada puede ser ignorada porque la utilidad que representa es menor a alguna anteriormente analizada.

En la poda alfa beta se hace un analisis de donde alfa es la mejor utilidad que puede lograr Max en una evaluación pesimista y beta es la mejor utilidad que puede lograr Max en una evaluación optimista, Si una jugada muestra que beta es menor o igual que alfa entonces se descarta ya que no representa un escenario mejor, por lo tanto no vale la pena recorrer esa rama ya que otras representan mejor ganancia.

El pseudocódigo para poda alfa beta se muestra a continuación,

Algorithm 2 Poda Alfa Beta

```
function ALPHA_BETA_SEARCH(node, depth, alpha, beta, maximizingPlayer)
    if depth = 0 or node is a terminal then
        return calculated value of node
    end if
    if maximizingPlayer then
        for child of node do
            alpha ← max(alpha, alphabeta(child, depth − 1, alpha, beta, FALSE))
            if beta <= alpha then
                break
            end if
            return alpha
        end for
    else
        for child of node do
            beta ← min(beta, alphabeta(child, depth − 1, alpha, beta, FALSE))
            if beta <= alpha then
                break
            end if
            return beta
        end for
    end if
end function
```

Conla propiedad de que la ganancia de un jugador es la perdida del otro, el negar el valor de busqueda adversarial nos entregará la ganancia del adversario, por lo tanto se puede reescribir la función siempre maximizando de la siguiente forma,

Algorithm 3 Poda Alfa Beta Negada

```
function N_ALPHA_BETA(node, depth, alpha, beta)
    if depth = 0 or node is a terminal then
        return calculated value of node
    end if
    for child of node do
        if beta <= alpha then
            break
        end if
        val ← −negascout(child, depth − 1, −beta, −alpha)
        if val > alpha then
            alpha ← val
        end if
    end for
    return alpha
end function
```

Capítulo 3

Diseño

3.1. Backend: Elección y desarrollo del agente

3.1.1. Seleccionando un algoritmo

Considerando ambas alternativas para el agente AI en este juego, se debe tener en consideración que se debe priorizar la experiencia del usuario al jugar, por lo que tener tiempos de espera más largos no es óptimo, la poda alfa beta tendrá al menos en promedio un tiempo de ejecución menor que minimax, por lo tanto se seleccionará ese algoritmo.

3.1.2. Diseño del modelo

A continuación se hará la bajada desde le pseudocódigo a una implementación en python, primero se especifican los atributos y funciones necesarias,

3.1.2.1. Atributos

Los atributos ayudarán a saber el estado del juego, para esto es necesario,

- Color de ficha de jugador 1 y 2
- Puntajes de jugador 1 y 2
- Quien tiene la jugada actual
- Valor booleano de salto de jugada de jugador 1 y 2
- Cantidad de piezas de jugador 1 y 2
- Tamaño del tablero
- Nivel de dificultad del agente
- Movimientos analizados
- Matriz que representa tablero actual

La elección de representar el estado actual del juego en una matriz simplifica la aplicación de algoritmos de búsqueda, si bien existen otras posibles representaciones como una clase customizada de nodo con sus adyacencias que permitan recorrer directamente de forma diagonal y ortogonal, la implementación de esto requeriría reglas de recorrido especificas, que son mucho más fáciles de visualizar y pensar en representación matricial utilizando indices de columna y fila.

3.1.2.2. Funciones

Las funciones que controlan el flujo del juego serán,

- Obtener movimientos validos para jugador
- Obtener puntos obtenidos por movimiento
- Simular movimiento
- Calcular puntaje actual entregando una suma zero
- Calcular puntaje actual guardando puntaje de cada jugador
- Función auxiliar para obtener el simbolo/color del oponente
- Algoritmo de búsqueda adversarial, en este caso poda alfa beta.
- Función que use algoritmo de busqueda para devolver la mejor jugada.

3.2. Backend: Construcción del modelo

En base a la especificación anterior se generan las clases con atributos y funciones necesarias.

3.2.1. Clase Auxiliar Move

Para facilitar la escritura y acceso de un movimiento se genera una clase como estructura de datos,

```
class Move:
    def __init__(self, coordinates, points):
        self.coordinates = coordinates
        self.points = points
```

Además de esto se usará una función auxiliar para copiar matrices aprovechando la utilidad de que python slicing retorna copia de los objetos originales,

```
def copy_board(board):
    return [row[:] for row in board]
```

3.2.2. Clase Othello

La clase Othello contendrá todo el flujo y propiedades del juego, generamos el constructor de la clase,

```
    class Othello:
    def __init__(self, P1, P2, board_size, difficulty):
        self.P1 = P1
        self.P2 = P2

        self.P1_score = 2
        self.P2_score = 2
```

```
8
9           self.turn = None
10
11          self.P1_skip = False
12          self.P2_skip = False
13
14          self.P1_stones = int(((board_size**2)-4)/2)
15          self.P2_stones = int(((board_size**2)-4)/2)
16
17          self.space = 'space'
18          self.board_size = board_size
19          self.difficulty = difficulty
20          self.MAX_SCORE = board_size**2
21          self.MIN_SCORE = -self.MAX_SCORE
22
23          self.moves_analized = []
24
25          self.current_board = []
26          self.valid_moves = []
```

3.2.2.1. Funciones de poda alfa beta

A continuación se describen las funciones utilizada para la poda alfa beta,

La función de entrada *move* recibe el tablero a evaluar, el jugador a evaluar, y la profundidad, este ultimo elemento es importante ya que esta se puede utilizar como medidor de dificultad, una AI capaz de predecir mas jugadas en el futuro es más dificil de vencer.

```
1  def move(self, board, player, depth = None):
2      if not depth:
3          depth = self.difficulty
4      self.valid_moves = self.get_valid_moves(board, player
           )
5      if not self.valid_moves:
6          return None
7      move = self.alpha_beta_search(player, board, self.
           MIN_SCORE, self.MAX_SCORE, depth)
8      return move.coordinates
```

La función anterior llama dos funciones, *get_valid_moves* y *alpha_beta_search* la primera evalua el tablero y entrega todas las jugadas validas para el jugador evaluado, esta es la función que permite la generación de jugadas, ya que valida todos los posibles movimientos para un jugador en el tablero actual y recorrerlas con la búsqueda adversarial, también sirve para validar si un jugador debe saltar su jugada por no tener movimientos validos disponibles

```
def get_valid_moves(self, board, stone):
    valid_moves = []
    for i in range(self.board_size):
        for j in range(self.board_size):
            # explore free spaces only
            if board[i][j] == self.space:
                gain = self.check_move(board, i, j, stone)
                if gain > 0:
                    valid_moves.append(Move((i, j), gain))
    if not valid_moves:
        return None
    else:
        return valid_moves
```

Esta función al calcular las jugadas posibles, calcula inmediatamente los puntos obtenidos (piezas invertidas) de cada una y son almacenados en la clase auxiliar *Move*, la función que realiza el calculo es *check_move* que valida desde las casillas entorno a la ficha la cantidad de fichas que se pueden invertir con esa jugada,

```
def check_move(self, board, row, column, stone):
    points = 0
    directions = [-1, 0, 1]
    for hrzn in directions:
        for vrtc in directions:
            if vrtc != 0 or hrzn != 0:
                if in_bounds(row + hrzn, column + vrtc,
                   self.board_size):
                    examined = board[row + hrzn][column +
                        vrtc]
                    if examined != self.space and
                       examined != stone:
                        points += self.count_points(board
                            , row + hrzn, column + vrtc,
                           hrzn, vrtc)
                    # The move is valid if stones can be
                       flipped
                    if points > 0:
```

```
                return points
    return points
```

A su vez la función anterior llama a *count_points* que recibe las coordenadas de la jugada y las cordenadas de la casilla aledaña desde donde revisar, por ejemplo, si recibe $(hrzn, vrtc) = (1, 1)$ revisará en dirección diagonal inferior derecha, si recibe $(hrzn, vrtc) = (0, -1)$ revisará hacia la izquierda, esto con a multiplicación por un escalar incremental sumando 1 mientras se encuentre una pieza del mismo color, si encuentra un espacio o el borde del tablero entonces retorna 0 y si encuentra una pieza del color opuesto retorna el valor acumulado.

```
def count_points(self, board, row, column, hrzn, vrtc):
    stone = board[row][column]
    count = 1
    # scalar value allows iteration in (hrzn, vrtc)
       direction
    for scalar in range(1, self.board_size + 1):
        if in_bounds(row + scalar * hrzn, column + scalar
            * vrtc, self.board_size):
            if board[row + scalar * hrzn][column + scalar
                * vrtc] == self.space:
                return 0
            elif board[row + scalar * hrzn][column +
               scalar * vrtc] == stone:
                count += 1
            else:
                return count
        else:
            return 0
    return count
```

Luego el elemento principal del agente, el algoritmo de poda alfa beta, es recursivo por lo que debe tener un caso base para cerrar la busqueda DFS, en este caso cuando se llega al final del arbol (*depth = 0*), no se tienen más jugadas validas, o se encuentra la jugada optima,

```
def alpha_beta_search(self, stone, board, alpha, beta,
   depth):
    # recursion base case
    if depth == 0:
        return Move(None, self.calc_score(board, stone))

    valid_moves = self.get_valid_moves(board, stone)
    if not valid_moves:
        if not self.get_valid_moves(board, self.
           opponent_stone(stone)):
```

```
            # last round - return the last board score
            return Move(None, self.final_value(stone,
                board))
        # cannot play this round - return the board value
            unchanged
        val = -self.alpha_beta_search(self.opponent_stone
            (stone), board, -beta, -alpha, depth - 1).
            points
        return Move(None, val)

    best_move = valid_moves[0]
    best_move.points = alpha
    for move in valid_moves:
        if beta <= alpha:
            #prune nodes that aren't worth visiting
            break
        sim_move_board = self.simulate_move(board, move.
            coordinates, stone)
        val = -self.alpha_beta_search(self.opponent_stone
            (stone), sim_move_board, -beta, -alpha, depth
            - 1).points
        if val > alpha:
            # Save new maximum
            alpha = val
            best_move = move
            best_move.points = alpha
        self.moves_analized.append(move)
    return best_move
```

Para avanzar en el arbol se hace uso de dos funciones, una que simula la jugada que retorna un nuevo estado de tablero local (nodo) para utilizar en la busqueda y otra función que realiza el calculo de puntaje en suma cero, esto representa la utilidad de la jugada y es la base para la comparación alfa beta,

```
def simulate_move(self, board, move, stone):
    if move == None:
        return board
    row, col = move

    # Needed to not modify original board
    board_copy = copy_board(board)
    board_copy[row][col] = stone

    directions = [-1, 0, 1]
    # this combination checks surrounding cells
```

```
12      for vrtc in directions:
13          for hrzn in directions:
14              stones = []
15              if vrtc == hrzn == 0:
16                  continue
17              lrow = row
18              lcol = col
19              lrow += vrtc
20              lcol += hrzn
21              while in_bounds(lrow, lcol, self.board_size):
22                  # if stone is opponents color append to
                        flip later
23                  if board_copy[lrow][lcol] != stone and
                        board_copy[lrow][lcol] != self.space:
24                      stones.append((lrow, lcol))
25                  # space breaks direct line, no stones
                        flipped
26                  elif board_copy[lrow][lcol] == self.space
                        :
27                      break
28                  # if same color stone found, flip all
                        stones in between
29                  elif board_copy[lrow][lcol] == stone:
30                      for a, b in stones:
31                          board_copy[a][b] = stone
32                      break
33                  lrow += vrtc
34                  lcol += hrzn
35      return board_copy
```

La función de utilidad revisa cada celda del tablero y calcula la utilidad sumando uno si se encuentra el mismo color y restando 1 si se encuentra un color distinto, es importante que esta función trabaja en suma cero, ya que el algoritmo de alfa beta (minimax también) funciona en base a que la ganancia de un jugador es exactamente igual a la perdida del otro,

```
1  def calc_score(self, board, stone):
2      score = 0
3      for i in range(self.board_size):
4          for j in range(self.board_size):
5              if board[i][j] == stone:
6                  score += 1
7              elif board[i][j] == self.opponent_stone(stone
                   ):
8                  score -= 1
```

```
9       return score
```

3.2.3. Backend: Implementando función especial 'Sugerir Jugada'

Ya que se tiene al función move que se utiliza para que la AI haga su mejor jugada, se puede utilizar esta misma función para sugerir al jugador humano, como está implementada con profunidad, se puede además en el frontend agregar la posibilidad de seleccionar el nivel de ayuda requerida, donde mientras más ayuda, más profundidad recorrerá el algoritmo de busqueda adversarial.

3.3. Front End

Para el desarrollo del front end se utilizará la librería PyQt5, esto ya que se tiene experiencia previa con esta.

En el front end será necesario representar dos pestañas, en una estará el juego y en el otro las configuraciones del juego. Las configuraciones de juego permiten seleccionar el tamaño del tablero con checkboxes, la dificultad se elegirá con combobox para la IA y las sugerencias al jugador, además se podrá seleccionar el color de ficha.

La pestaña de juego tendrá una cabecera con los puntajes de cada jugador, luego se tendrá el tablero con las fichas por jugador a cada lado, izquierdo para las fichas de la computadora y derecho para las fichas del jugador. El tablero como tal debe tener una matriz $N \times N$ de botones que actuarán como casillas que al ser seleccionadas ejecutarán una función de movimiento conectada con el backend, estas estarán contenidas en una grilla. En la parte inferior habrá una caja de texto que servirá para mostrar turnos, condiciones de juego, cantidad de jugadas analizadas (nodos) por la computadora y resultado final.

3.3.1. Creación del tablero

La parte más 'compleja' del front end, o la más interesante de revisar, es la generación del tablero, debido a que se sacó provecho a caracteristicas que ofrece PyQt en la generación de sus layouts y la posibilidad de acceder a elementos dentro de ellos, especialmente en QGridLayout, donde se puede especificar columna, fila y uso de espacio en ambas dimensiones. Este QGridlayout es poblado con una clase especial que se creó y se detalla a continuación.

3.3.1.1. Subclase Board Cell

Para los botones de la grilla del tablero se creó una subclase de QPushbuttons, esto nos permite agregar atributos al botón que facilitan la programación y ciertas funciones que cambian su propio estado, se tiene el status de esa casilla (negro, blanco, espacio) sus coordenadas, y la conexión a la función de front end que valida las condiciones de juego, dentro de las funciones tiene *setStone* para poner una pieza, esto cambia su estado, pone una imagen sobre el fondo correspondiente al a ficha y deshabilita el botón. *flipStone* sirve para dar vuelta una ficha en caso de que se haya realizado una jugada que flanquee fichas.

```python
class BoardCell(QPushButton):

    def __init__(self, coordinates, func, *args, **kwargs):
        super(QPushButton, self).__init__(*args, **kwargs)
        self.setStyleSheet('width: 100%; height: 100%; background-color: #407936 !important;')
        self.setStyleSheet(self.styleSheet() + ' color: rgba(0, 0, 0, 0);')
        self.stone = 'space'
        self.coordinates = coordinates
        self.clicked.connect(lambda: func(self))

    def setStone(self, color):
        if color == 'black':
            self.setStyleSheet(self.styleSheet() + f'border-image: url(./sprites/{color}.svg);')
        elif color == 'white':
            self.setStyleSheet(self.styleSheet() + f'border-image: url(./sprites/{color}.svg);')
        self.stone = color
        self.setDisabled(True)

    def flipStone(self):
        if self.stone == 'black':
            self.setStyleSheet(self.styleSheet().replace('black', 'white'))
            self.stone = 'white'
        elif self.stone == 'white':
            self.setStyleSheet(self.styleSheet().replace('white', 'black'))
            self.stone = 'black'
```

3.3.1.2. Construcción de la matriz/tablero

Con la subclase de botones creados se puede crear la grilla utilizando QGridLayout, este layout es particularmente ventajoso en la creación del juego ya que tiene una función que permite obtener los widgets en una posición especifica, como el estado del juego está representado en una matriz, esto se puede traducir directamente en llamar a los botones con la función *itemAtPosition* enviando el valor de coordenadas

obtenido de la matriz o a futuro, de la sugerencia de jugada para el jugador humano o jugada del agente AI.

Al lanzar un nuevo juego se limpia el tablero y las fichas de cada jugador, se llama la función para obtener el tamaño del tablero y la posición de fichas iniciales, y luego se inicializa la clase Othello del backend, enviando la asignación de colores por cada jugador, tamaño de tablero y dificultad seleccionada, después se llama la función *build_board* enviando tamaño dle tablero y fichas iniciales, el tablero construido se guarda en el backend, se rellenan las fichas que corresponden a cada jugador y se inicia el primer turno.

```
def new_game(self):
    clearLayout(self.board.layout)
    clearLayout(self.P1stones.layout)
    clearLayout(self.P2stones.layout)

    n, starting_stones = self.set_initial_stones()

    if n:
        stones = ['black','white']
        if self.player_stone_cbox.currentIndex() == 1:
            stones = stones[::-1]
        self.game = Othello(*stones, n, self.
           difficulty_cbox.currentIndex()+1)

        self.game.current_board = self.build_board(n,
           starting_stones)

        self.fill_player_stones(n)

        self.P1score.setText('Jugador 1 | 2')
        self.P2score.setText('2 | Jugador 2')

        self.switch_menu_items()
        self.tabs.setCurrentIndex(0)
        self.first_turn()
```

La función auxiliar *set_initial_stones* entrega el tamaño del tablero y las coordenadas de las fichas iniciales,

```
def set_initial_stones(self):
    n, starting_stones = None, {}
    if self.cb1.isChecked():
        n = 6
        starting_stones = {
```

```
            '14': 'black',
            '15': 'white',
            '20': 'white',
            '21': 'black',
        }

    elif self.cb2.isChecked():
        n = 8
        starting_stones = {
            '27': 'black',
            '28': 'white',
            '35': 'white',
            '36': 'black',
        }
    return n, starting_stones
```

La función *build_board* recibe el tamaño del tablero y coordenadas de piezas iniciales, en base a esto agrega los botones a la matriz del frontend y a su vez construye la matriz que se utilizará en el backend, los botones se inicializan con su posición y si corresponde su color,

```
def build_board(self, n, starting_stones):
    board_matrix = []
    for i in range(n):
        temp = []
        for j in range(n):
            # cell = str(n*i+j)
            button = BoardCell((i, j), self.try_move,
               cell, self.tab1)
            if cell in starting_stones:
                color = starting_stones[cell]
                button.setStone(color)
                temp.append(color)
            else:
                temp.append('space')

            self.board.layout.addWidget(button, i, j, 1,
               1)
            self.board.layout.setSpacing(2)
        board_matrix.append(temp)

    return board_matrix
```

El primer turno depende de la elección del jugador sobre el color de pieza a

utilizar, si selecciona blanco, entonces la computadora parte, con la función *singleShot* de QTimer se puede programar una ejecución para que el jugador humano pueda visualizar la jugada de la AI luego de 2 segundos de dar click en iniciar Juego

```
def first_turn(self):
    self.send_output('### Inicia nueva partida ###')
    if self.player_stone_cbox.currentIndex() == 1:
        self.send_output('Computadora tiene el primer
           turno')
        self.game.turn = self.game.P2
        QTimer.singleShot(2000, lambda: self.ai_turn())
    else:
        self.send_output('Jugador tiene el primer turno')
        self.game.turn = self.game.P1
```

3.3.2. Frontend: Implementando función especial 'Sugerir Jugada'

Como se explicó en el apartado de backend se hará uso de la función move, que se utiliza para que la AI explore las jugadas, en el frontend se llamará una función que verifica si existe una jugada disponible, si existe la mejor jugada entonces obtendrá el botón de la grilla que corresponde a la mejor jugada y llamará la función *suggestMove*, si la jugada no existe quiere decir que no existen jugadas validas, por lo que le dará la indicación al jugador de saltar su turno. Es importante recalcar que el nivel de la jugada sugerida va a depender del nivel de ayuda seleccionada.

```
def suggest_moves(self):
    suggested_move= self.game.move(self.game.
       current_board, self.game.P1, self.ai_help_cbox.
       currentIndex()+1)
    if suggested_move:
        i, j = suggested_move
        self.board.layout.itemAtPosition(i,j).widget().
           suggestMove()
    else:
        self.send_output('No hay jugadas disponibles,
           debes saltar el turno')
```

La función *suggestMove* hará que el botón se deshabilite temporalmente, y agregará una estrella en la casilla para indicar que esa es la mejor jugada, luego de 1 segundo se desaparecera la estrella y se habilitará el botón para que el jugador pueda seleccionarla.

```
def suggestMove(self):
    self.setDisabled(True)
    old_style = self.styleSheet()
```

```
4       self.setStyleSheet(old_style + ' border-image: url(./
            sprites/star.svg);')
5       QTimer.singleShot(1000, lambda: self.setStyleSheet(
            old_style))
6       QTimer.singleShot(1000, lambda: self.setEnabled(True)
            )
```

Testing

4.1. Backend: Probando el algoritmo de búsqueda

Para realizar pruebas del algoritmo de búsqueda se entregará un tablero que podría ser considerado de la etapa media del juego y será sometido a distintos niveles de profunidad para identificar cuanto más tarda y cuanto crece el arbol si se aumenta un nivel de profundidad, para esto se utiliza la librería timeit con un setup para medir tableros 6x6 y 8x8.

```
import timeit
from matplotlib import pyplot as plt

def test_suite(max_n):
    numbers, othello6, othello8 = [], [], [],
    for i in range(1, max_n+1):
        setup6 = f'''
from game import Othello

sample_board = [
    ['space', 'space', 'black', 'space', 'space', 'white
        '],
    ['space', 'space', 'space', 'black', 'white', 'black
        '],
    ['space', 'space', 'white', 'white', 'black', 'space
        '],
    ['space', 'white', 'white', 'white', 'black', 'space
        '],
    ['space', 'white', 'white', 'white', 'space', 'space
        '],
    ['space', 'white', 'space', 'white', 'space', 'space
        '],
]
player = Othello('black', 'white', 6, int({i}))
        '''

```

```
20
21          setup8 = f'''
22 from game import Othello
23
24 sample_board = [
25     ['space', 'space', 'space', 'space', 'space', 'space
       ', 'space', 'white'],
26     ['space', 'space', 'space', 'black', 'space', 'space
       ', 'white', 'space'],
27     ['space', 'space', 'space', 'space', 'black', 'white
       ', 'black', 'black'],
28     ['space', 'space', 'space', 'white', 'white', 'black
       ', 'space', 'space'],
29     ['space', 'space', 'white', 'white', 'white', 'black
       ', 'space', 'space'],
30     ['space', 'space', 'white', 'white', 'white', 'space
       ', 'space', 'space'],
31     ['space', 'space', 'white', 'space', 'white', 'space
       ', 'space', 'space'],
32     ['space', 'space', 'space', 'space', 'space', 'space
       ', 'space', 'space'],
33 ]
34 player = Othello('black', 'white', 8, int({i}))
35          '''
36          numbers.append(i)
37
38          othello6.append(timeit.timeit("player.move(
               sample_board,'black')", setup=setup6, number
               =1))
39          othello8.append(timeit.timeit("player.move(
               sample_board,'black')", setup=setup8, number
               =1))
40
41          print(numbers, othello6, othello8)
42      return numbers, othello6, othello8
```

Luego usamos matplotlib para graficar los resultados de ejecución, en este caso se seleccionó 9 como profunidad maxima para evaluar,

```
numbers, othello6, othello8 = test_suite(9)

plt.plot(numbers, othello6, 'b', label="Othello 6x6")
plt.plot(numbers, othello8, 'g', label="Othello 8x8")
plt.legend()
plt.show()
```

Eso nos entrega la siguiente gráfica, donde se puede identificar un comportamiento exponencial.

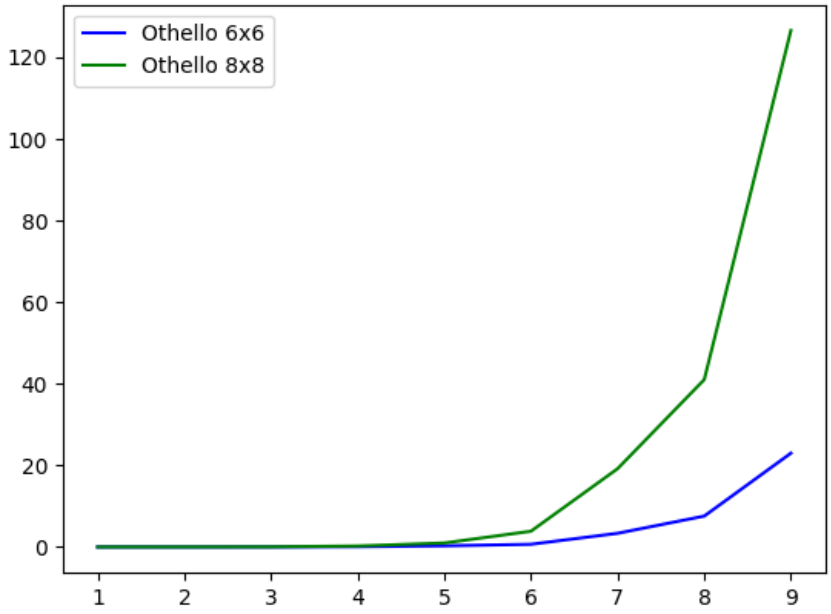

Figura 4.1: Tiempo de ejecución v/s profundidad (jugadas en el futuro) del arbol de búsqueda

Esto coincide con lo indicado por Aljubran en cuanto a los tiempos de ejecución, en este caso el valor variable era el exponente, ya que los tableros a pesar de que son de distinta dimensión eran comparables, y se utilizó el mismo para cada prueba. Conn estos resultados se puede identificar un valor maximo de profunidad para que el juego sea utilizable, ya que esperar 120 segundos por turno de computadora no es la mejor experiencia para el jugador, el máximo aceptable sería 6 niveles de profundidad, posiblemente 7 para un tablero de 6x6.

4.2. Frontend: Probando la interfaz gráfica

Se prueba la interfaz, lo primero que se prueba el lanzamiento del juego y el bloqueo de botones, al iniciar el programa aparece el menú de opciones para seleccionar la dificultad,

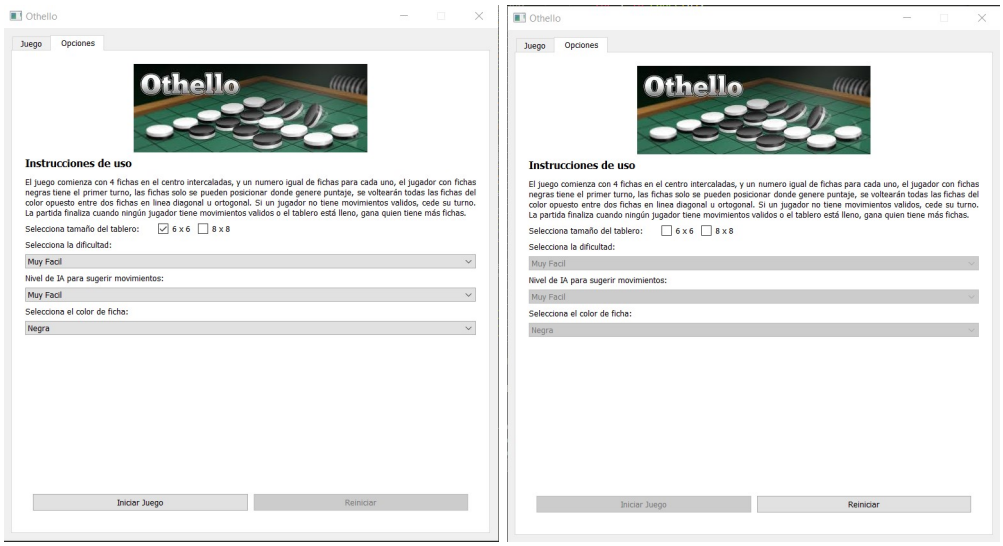

Figura 4.2: Menú principal

Luego de lanzar se ve lo siguiente para la versión de 6x6 y 8x8,

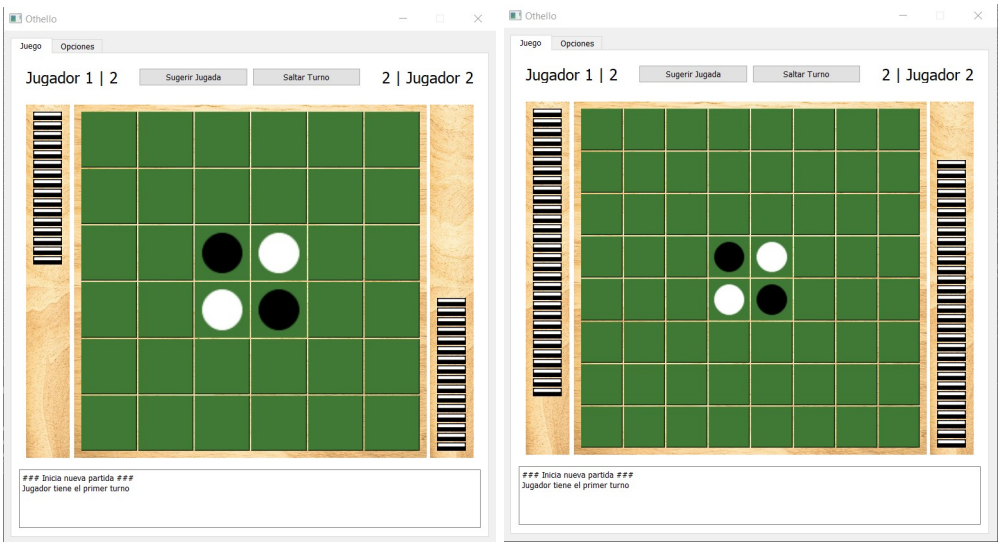

Figura 4.3: Tablero de 6x6 y 8x8 recién iniciado

Además se valida jugando que el puntaje se actualiza correctamente y que el botón sugerir jugada está funcionando correctamente,

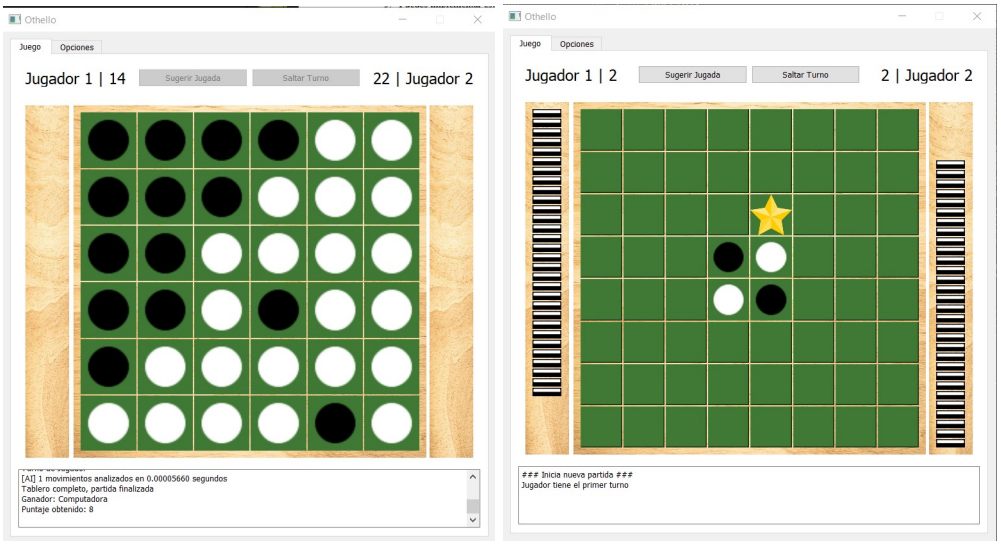

Figura 4.4: Tablero completo y jugada sugerida en otro tablero

Se validan los estados finales posibles,

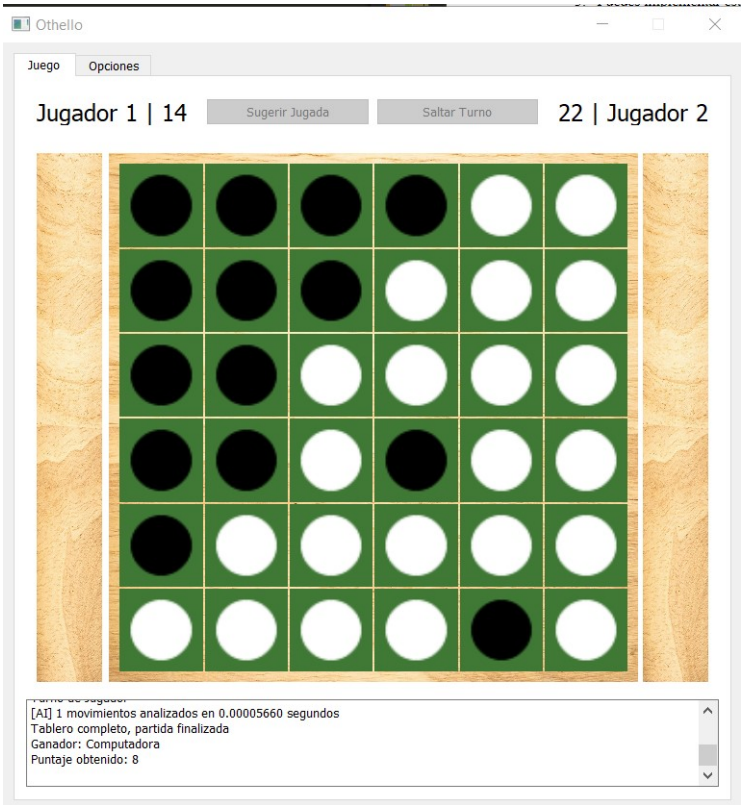

Figura 4.5: Tablero completo indicando que ese es el estado de juego, el ganador y el puntaje obtenido

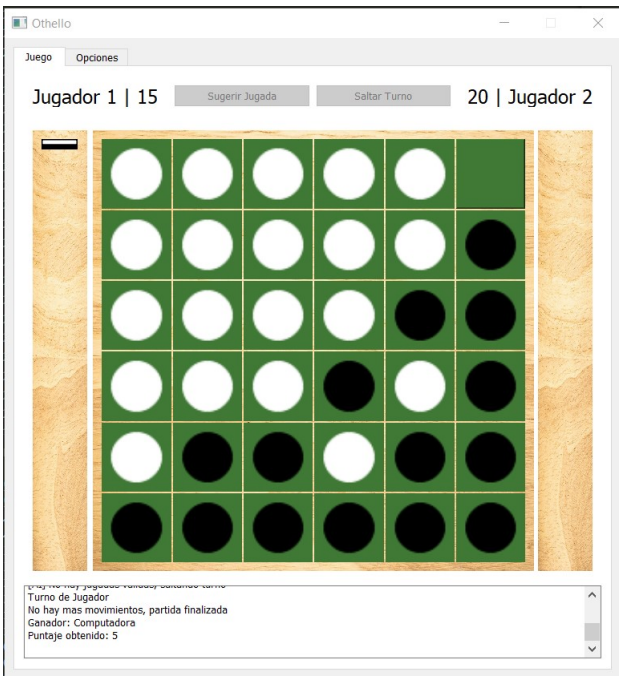

Figura 4.6: Tablero incompleto sin jugadas validas indicando que ese es el estado de juego, el ganador y el puntaje obtenido

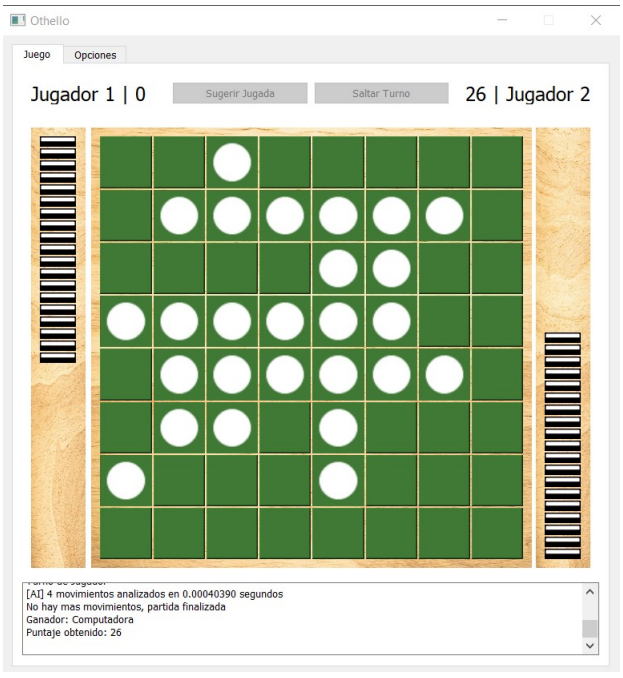

Figura 4.7: Tablero incompleto sin jugaas validas para un jugador indicando que ese es el estado de juego, el ganador y el puntaje obtenido

Se valida la diferencia de cantidad de jugadas y tiempo de computo,

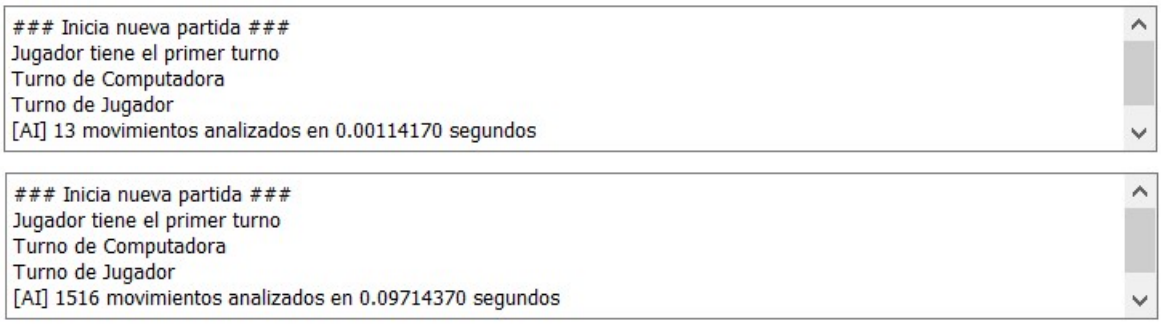

Figura 4.8: Output obtenido en nivel facil y T800 (máximo nivel)

Capítulo 5

Conclusiones

En el ejercicio de esta tarea se investigó sobre la naturaleza de los juegos suma cero, las distintas formas desarrollar un agente con inteligencia artificial, además de implementar estos algoritmos en un backend que se conecte con un frontend facil de utilizar que sea capaz de sacar provecho a las funciones y estados que ofrece el backend desarrollado.

En la investigación tanto de los juegos tipo suma cero, la teoría de juegos planteada por Von Neumann en 1928 y el desarrollo de distintos algoritmos en base a esa base teorica planteada, sirvió para tener un entendimiento más profundo de todos estos elementos, teniendo una muy buena base para poder ahondar más en el desarrollo de agentes y la aplicación de inteligencia artificial en general.

Referencias

[1] Aljubran, A., 2014. Othello. http://cs.indstate.edu/~aaljubran/paper.pdf. (cited on pages 4 and 23)

[2] Kjeldsen, T. H., 2001. John von neumann's conception of the minimax theorem: A journey through different mathematical contexts. *Archive for history of exact sciences*, 56, 1 (2001), 39–68. (cited on page 3)

[3] Kuhn, H. W. and Tucker, A. W., 1958. John von neumann's work in the theory of games and mathematical economics. *Bulletin of the American Mathematical Society*, 64, 3 (1958), 100–122. (cited on page 3)

[4] Russell, S. J., 2010. *Artificial intelligence a modern approach*. Pearson Education, Inc. https://zoo.cs.yale.edu/classes/cs470/materials/aima2010.pdf.

[5] Von Neumann, J. and Morgenstern, O., 1928. https://uvammm.github.io/docs/theoryofgames.pdf. (cited on page 3)