Connect 6

Javier Prada de Francisco

Universidad de Alicante, San Vicente del Raspeig, 03690, España
https://www.ua.es/es/

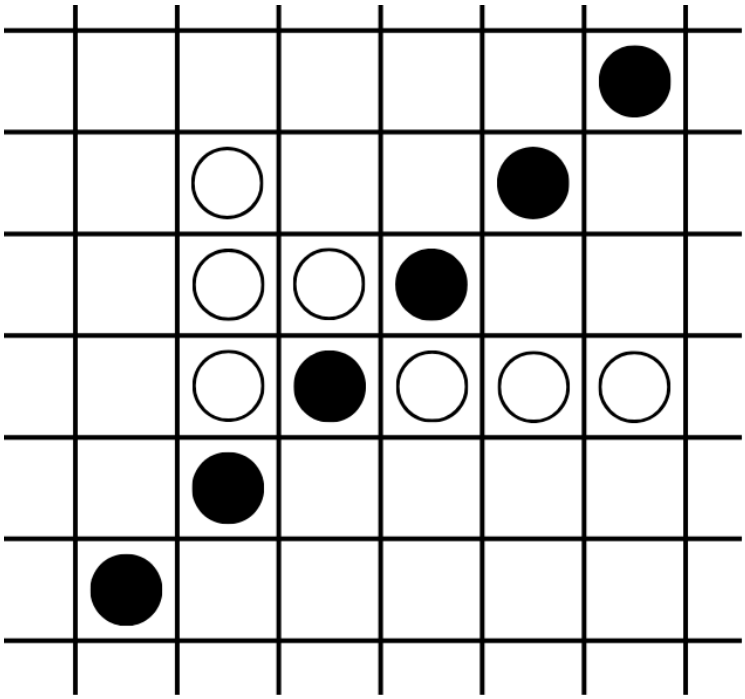

Table of Contents

1 Introduction

The Connect6 game, introduced by Wu and Huang in 2005 [1], represents an interesting evolution of the board alignment game. Unlike better known variants such as Tic-Tac-Toe or Go-Moku, Connect6 increases the strategic complexity by allowing players to place two pieces per turn, except for the first move. This seemingly simple change exponentially multiplies the possible combinations and tactical challenges, making it an attractive field of study for research in artificial intelligence and game theory.

In this project, an artificial intelligence has been developed to compete in Connect6, exploring various techniques and algorithms ranging from heuristic evaluation strategies to the implementation of a minimax algorithm with alpha-beta pruning. The main objective was to design a system capable of analyzing the board, predicting moves and making optimal decisions in a reasonable time, facing the challenges inherent to the enormous complexity of the game's search space.

In addition, the development of this project involved an iterative process in which different approaches were tested and discarded, including the use of genetic algorithms and concurrency techniques to optimize response times. The results obtained not only demonstrate the effectiveness of the strategies implemented, but also offer insight into the limitations and areas for improvement when applying computational methods to games with high levels of strategic complexity.

2 State of art

The state of the art of k-in-scratch type games, specifically 6-in-scratch, focuses on the advances and distinctive features of this type of games. This type of game is defined by the need to line up k consecutive checkers on a board to win. Among the best known variants are Tic-Tac-Toe, Go-Moku and Connect6. Connect6, introduced by Wu and Huang in 2005, presents an interesting variant where players place two checkers per turn after the first move, significantly increasing the strategic complexity.

In terms of complexity, Connect6 has a state space similar to the game of Go, with an estimated decision tree complexity of 10^{140} making it a challenge for both players and artificial intelligence researchers. Threat and evaluation strategies have been key tools in the development of AI programs for Connect6, allowing to optimize moves and respond to opponent's threats efficiently. Despite this, the fairness of the game remains a matter of debate, although the balance achieved by allowing two tokens per turn is considered to make it less likely to favor one player over another, as is the case in other variants such as Go-Moku.

Finally, recent studies have explored variants of the game on larger board sizes and modified rules to increase strategic diversity. Methodologies such as the use of threat space search and pattern-based strategies have also been investigated to improve the performance of automated systems in international tournaments. These advances consolidate Connect6 as a fruitful area for research in artificial intelligence and game theory. [2]

3 Development

3.1 Function ver_victoria and ver_empate

The first thing I did was to create a function that detects if any of the players have won. Unlike the is_win_by_premove method, the ver_victora method does not require the last move made to be passed to it; instead, it scours the entire board looking for a string of six consecutive tiles. This method, by needing to check the entire board each time it is invoked, generates a longer response time when processing moves. However, I chose to implement it this way because I found it easier to understand, but finally I opted to remove it and use the method is_win_by_premove.

```
def ver_victoria(board):
    direcciones = [(1, 0), (0, 1), (1, 1), (1, -1)]
    for fila in range(1, Defines.GRID_NUM - 1):
        for col in range(1, Defines.GRID_NUM - 1):
            color_actual = board[fila][col]
            if board[fila][col] == Defines.NOSTONE:
                continue
            for dx, dy in direcciones:
                count = 1
                longitud, _ = contar_cadena(board, fila, col,
 dx, dy, color_actual)
                count += longitud
                longitud, _ = contar_cadena(board, fila, col,
 -dx, -dy, color_actual)
                count += longitud
                if count >= 6:
                    return True
    return False
```

Along with this function, I created another method to detect a tie in case the board is full. This method scans the entire board looking for empty squares; if it finds any, it returns False, indicating that there is no tie.

```
def ver_empate(board):
for fila in board:
    if Defines.NOSTONE in fila:
        return False
return True
```

3.2 Find possible moves

To find possible moves for the AI, I started by creating a simple function that generates a list of all possible moves around the tiles it already owns. The first version of this function is as follows:

```
    def buscar_candidatos(self, board):
        candidatos = set()
        direcciones = [(1, 0), (0, 1), (1, 1), (1, -1)]
        max_candidatos = 10
        for fila in range(Defines.GRID_NUM):
            for col in range(Defines.GRID_NUM):
                if board[fila][col] == self.color_ia:
                    for dx, dy in direcciones:
                        nueva_fila = fila + dx
                        nueva_col = col + dy
                        if (0 <= nueva_fila < Defines.
GRID_NUM) and (0 <= nueva_col < Defines.GRID_NUM):
                            if board[nueva_fila][nueva_col]
== Defines.NOSTONE:
                                candidatos.add((nueva_fila,
nueva_col))
        candidatos = list(candidatos)
        if len(candidatos) > max_candidatos:
            candidatos = random.sample(candidatos,
max_candidatos)
        return candidatos
```

The problem with this version was that it only worked if the AI played with the black pieces. In order for the AI to find a move, it must look for an already placed checker of its own; in the case of black, an initial checker was placed by default in the JJ position, which allowed the code to work correctly. However, the same was not true for white checkers, which generated an error. To solve this, I added a condition at the end that places a checker in a random position on the board.

```
    while len(candidatos) <= 0:
            nueva_fila = random.randint(0,Defines.GRID_NUM-1)
            nueva_col = random.randint(0, Defines.GRID_NUM-1)
            if board[nueva_fila][nueva_col] == Defines.
NOSTONE:
                candidatos.add((nueva_fila, nueva_col))
```

This function was very simple and only generated moves next to the ia's pieces without blocking any of mine. For this I decided to return a move to continue with the ia's pieces and another one to block mine. In addition, here I assign a score to each pair of moves and when limiting the list of candidates I order the best moves according to the score and return only the 10 best ones. This can be seen in the following code:

```
def buscar_candidatos(self):
    candidatos_bloqueo = set()
    candidatos_cadena = set()
    candidatos = set()
    direcciones = [(1, 0), (0, 1), (1, 1), (1, -1)]
    mejor_movimiento = StoneMove()
    mejor_movimiento.score = Defines.MININT
    max_candidatos = 10

    for move in self.movimientos_ia:
        for pos in move.positions:
            for dx, dy in direcciones:
                nueva_fila = pos.x + dx
                nueva_col = pos.y + dy
                if (0 <= nueva_fila < Defines.GRID_NUM) and (0 <= nueva_col < Defines.GRID_NUM):
                    if self.m_board[nueva_fila][nueva_col] == Defines.NOSTONE:
                        movimiento = StonePosition(nueva_fila, nueva_col)
                        candidatos_cadena.add((movimiento))

    for move in self.movimientos_jugador:
        for pos in move.positions:
            for dx, dy in direcciones:
                nueva_fila = pos.x + dx
                nueva_col = pos.y + dy
                if (0 <= nueva_fila < Defines.GRID_NUM) and (0 <= nueva_col < Defines.GRID_NUM):
                    if self.m_board[nueva_fila][nueva_col] == Defines.NOSTONE:
                        movimiento = StonePosition(nueva_fila, nueva_col)
                        candidatos_bloqueo.add((movimiento))

    for movimiento_cadena in candidatos_cadena:
        for movimiento_bloqueo in candidatos_bloqueo:
            movimiento = StoneMove()
            movimiento.positions = [movimiento_cadena, movimiento_bloqueo]
            movimiento.score = evaluar_movimiento(movimiento, self.movimientos_ia, self.movimientos_jugador)
            candidatos.add(movimiento)

    if len(candidatos) > max_candidatos:
        candidatos = sorted(candidatos, key=lambda movimiento: movimiento.score, reverse=True)[:max_candidatos]

    while len(candidatos) <= 0:
        movimiento = StoneMove()
```

```
42          movimiento.positions[0].x = random.randint(0,Defines.
    GRID_NUM-1)
43          movimiento.positions[0].y = random.randint(0,Defines.
    GRID_NUM-1)
44          movimiento.positions[1].x = random.randint(0,Defines.
    GRID_NUM-1)
45          movimiento.positions[1].y = random.randint(0,Defines.
    GRID_NUM-1)
46          movimiento.score = evaluar_movimiento(movimiento,
    self.movimientos_ia, self.movimientos_jugador)
47          if self.m_board[nueva_fila][nueva_col] == Defines.
    NOSTONE:
48              candidatos.add(movimiento)
49      return candidatos
```

However, I had a problem: the code was unable to run in the GUI, as I will explain later in the "errors" section. This forced me to start practically from scratch, starting from the first version of the code I had developed. This setback led me to try different approaches to candidate selection.

First, I made a significant change: now, with all the moves stored in a dictionary, I use a 'for' loop to analyze the positions. For each position, I identify the player to which it belongs and, with this information, I add the corresponding positions to a list of offensive or defensive candidates. In addition, I only consider close candidates within a range of one to three spaces away.

Finally, I evaluate the candidates, order both lists according to the highest scores and select the top four from each list. All this can be seen in the following code:

```
1  def buscar_candidatos(self, board, movimientos):
2      ofensivos = set()
3      defensivos = set()
4      direcciones = [(1, 0), (0, 1), (1, 1), (1, -1)]
5
6      for movimiento, jugador in movimientos.items():
7          for dx, dy in direcciones:
8              for dist in range(1, 3):
9                  nueva_fila = movimiento.x + dx * dist
10                 nueva_col = movimiento.y + dy * dist
11                 if isValidPos(nueva_fila, nueva_col) and
    board[nueva_fila][nueva_col] == Defines.NOSTONE:
12                     if jugador == self.color_jugador:
13                         defensivos.add((nueva_fila,nueva_col))
14                     if jugador == self.color_ia:
15                         ofensivos.add((nueva_fila,nueva_col))
16                 nueva_fila = movimiento.x + -dx * dist
17                 nueva_col = movimiento.y + -dy * dist
18                 if isValidPos(nueva_fila, nueva_col) and
    board[nueva_fila][nueva_col] == Defines.NOSTONE:
19                     if jugador == self.color_jugador:
```

```
20                      defensivos.add((nueva_fila,nueva_col))
21                  if jugador == self.color_ia:
22                      ofensivos.add((nueva_fila,nueva_col))
23
24      ofensivos = sorted(ofensivos, key=lambda pos: self.
        evaluar_posicion(pos,board,self.color_ia),reverse=True)
25      defensivos = sorted(defensivos, key=lambda pos: self.
        evaluar_posicion(pos,board,self.color_jugador),reverse=
        True)
26      candidatos_finales = ofensivos[:4] + defensivos[:4]
27      return candidatos_finales
```

3.3 Evaluation

For score evaluation, I created a simple function that receives three parameters: ganador (it is None if there is none), a boolean indicating whether the game is over, and the current state of the board. I use the variable win only to prevent the ver_empate method from executing on each call to the evaluation function. Once it is determined if there is a winner, the score is calculated by counting the tiles and chains of each player.

```
def evaluacion(self, ganador=None, victoria=False, board=None
    ):
    if victoria:
        if ganador == self.color_ia:
            return Defines.MAXINT
        elif ganador == self.color_jugador:
            return Defines.MININT
        elif ver_empate(board):
            return self.DRAW_SCORE

    puntuacion = 0

    puntuacion += contar_piedras(self.color_ia, board, self.
    color_jugador) * 10
    puntuacion -= contar_piedras(self.color_jugador, board,
    self.color_jugador) * 10

    return puntuacion
```

To calculate the tiles of each player, I implemented two methods: one to count the individual tiles and one to count the chains. The function contar_piedras counts the tiles of each player, and then, to calculate the score, I apply the formula $2^{chain\_lenght-1}$. The contar_cadena method checks, starting from a specific token, if there are more consecutive tokens of the same color, and returns the number of consecutive tokens found.

```
def contar_piedras(color, board, jugador):
    piedras = 0
    direcciones = [(1, 0), (0, 1), (1, 1), (1, -1)]

    for fila in range(1, Defines.GRID_NUM - 1):
        for col in range(1, Defines.GRID_NUM - 1):
            if board[fila][col] == color:
                for dx, dy in direcciones:
                    longitud_cadena, es_cadena_viva =
    contar_cadena(board, fila, col, dx, dy, color)
                    if es_cadena_viva:
                        piedras += 2 ** (longitud_cadena - 1)

    return piedras

```

```
def contar_cadena(board, fila, col, dx, dy, color):
    consecutivas = 0
    n = len(board)

    for i in range(1, 6):
        nueva_fila = fila + i * dx
        nueva_col = col + i * dy

        if nueva_fila < 0 or nueva_fila >= n or nueva_col < 0 or nueva_col >= n:
            break

        if board[nueva_fila][nueva_col] == color:
            consecutivas += 1
        else:
            break

    return consecutivas
```

The initial evaluation method seemed too simple, so I designed a new system that does not only consider the tiles and chains of each player. In this method, I go around the board and start by evaluating the distance of each checker to the center, assigning it a maximum score of 10 depending on the number of squares separating it from the center.

Then, for each piece I analyze if it is blocking an opponent's chain, if there are empty squares around it, if it contributes to continue a row of its own and how many pieces the AI has on the board. Finally, with all this data, I calculate the score based on the chains of each player and awarding higher scores to positions with free squares. This can be seen here:

```
def evaluar(board, color_ia, color_jugador):
    puntuacion = 0
    centro = (Defines.GRID_NUM // 2, Defines.GRID_NUM // 2)
    for fila in range(Defines.GRID_NUM):
        for col in range(Defines.GRID_NUM):
            if board[fila][col] == color_ia:
                distancia_al_centro = abs(fila - centro[0]) + abs(col - centro[1])
                puntuacion += 10 - distancia_al_centro
            if board[fila][col] == color_jugador:
                distancia_al_centro = abs(fila - centro[0]) + abs(col - centro[1])
                puntuacion -= 10 - distancia_al_centro
            for dx, dy in [(1, 0), (0, 1), (1, 1), (1, -1)]:
                count_fichas = 0
                count_vacio = 0
                count_bloqueo = 0
                consecutivas_ia = 0
                consecutivas_jugador = 0
```

```
                for i in range(6):
                    nueva_fila = fila + i * dx
                    nueva_col = col + i * dy
                    if  0 <= nueva_fila < Defines.GRID_NUM
    and 0 <= nueva_col < Defines.GRID_NUM:
                        if board[nueva_fila][nueva_col] ==
    color_ia:
                            count_fichas += 1
                            count_bloqueo += 1
                            consecutivas_ia += 1
                        if board[nueva_fila][nueva_col] ==
    color_jugador:
                            count_bloqueo -= 1
                            consecutivas_jugador += 1
                        if board[nueva_fila][nueva_col] ==
    Defines.NOSTONE:
                            count_vacio += 1

                puntuacion += ((2 ** (consecutivas_ia - 1)) -
     (2 ** (consecutivas_jugador - 1))) * 10
                if count_bloqueo == 5:
                    if board[nueva_fila][nueva_col] ==
    color_ia:
                        puntuacion += 100
                    if board[nueva_fila][nueva_col] ==
    color_jugador:
                        puntuacion -= 100
                elif count_bloqueo == 4:
                    if board[nueva_fila][nueva_col] ==
    color_ia:
                        puntuacion += 50
                    if board[nueva_fila][nueva_col] ==
    color_jugador:
                        puntuacion -= 50
                if count_fichas > 0:
                    puntuacion += (count_vacio * 10) if
    count_vacio > 0 else -10
    return puntuacion
```

In addition, to achieve a more complex evaluation, I implemented another function that analyzes each move individually. This function is invoked within the candidate search process and consists of three steps: in the first, I evaluate whether the move blocks an opponent's chain; in the second, I check whether it extends a chain of my own; and finally, I check whether the move would complete a chain of six tokens.

```
def evaluar_movimiento(movimiento, movimientos_ia,
    movimientos_jugador):
    puntuacion = 0
    puntuaciones = [1, 25, 50, 100, 250, Defines.MAXINT]
```

```
4       for pos in movimiento.positions:
5           cadena = es_bloqueo(pos, movimientos_jugador)
6           puntuacion += puntuaciones[cadena-1]
7       for pos in movimiento.positions:
8           cadena = extiende_cadena_propia(pos, movimientos_ia)
9           puntuacion += puntuaciones[cadena-1]
10      centro_x, centro_y = Defines.GRID_NUM // 2, Defines.
        GRID_NUM // 2
11      for pos in movimiento.positions:
12          distancia_al_centro = abs(pos.x - centro_x) + abs(pos
        .y - centro_y)
13          puntuacion += 10 / (distancia_al_centro + 1)
14      if completa_cadena_ganadora(movimiento, movimientos_ia):
15          return Defines.MAXINT
16      return puntuacion
17
18      def es_bloqueo(pos, movimientos_jugador):
19      direcciones = [(1, 0), (0, 1), (1, 1), (1, -1)]
20      for dx, dy in direcciones:
21          conteo = 1
22          x, y = pos.x + dx, pos.y + dy
23          while isValidPos(x, y) and (StonePosition(x, y)  in
        movimientos_jugador):
24              conteo += 1
25              x += dx
26              y += dy
27          x, y = pos.x - dx, pos.y - dy
28          while isValidPos(x, y) and (StonePosition(x, y)  in
        movimientos_jugador):
29              conteo += 1
30              x -= dx
31              y -= dy
32      return conteo
33
34  def extiende_cadena_propia(pos, movimientos_ia):
35      direcciones = [(1, 0), (0, 1), (1, 1), (1, -1)]
36      for dx, dy in direcciones:
37          conteo = 1
38          x, y = pos.x + dx, pos.y + dy
39          while isValidPos(x, y) and (StonePosition(x, y) in
        movimientos_ia):
40              conteo += 1
41              x += dx
42              y += dy
43          x, y = pos.x - dx, pos.y - dy
44          while isValidPos(x, y) and (StonePosition(x, y) in
        movimientos_ia):
45              conteo += 1
46              x -= dx
47              y -= dy
```

```
    return conteo

def completa_cadena_ganadora(movimiento, movimientos_ia):
    direcciones = [(1, 0), (0, 1), (1, 1), (1, -1)]
    longitud_cadena = 6
    movimientos_ia.update(movimiento.positions)
    for pos in movimiento.positions:
        for dx, dy in direcciones:
            conteo = 1
            x, y = pos.x + dx, pos.y + dy
            while StonePosition(x, y) in movimientos_ia:
                conteo += 1
                x += dx
                y += dy
            x, y = pos.x - dx, pos.y - dy
            while StonePosition(x, y) in movimientos_ia:
                conteo += 1
                x -= dx
                y -= dy
            if conteo >= longitud_cadena:
                return True
    return False
```

Finally, in the minmax algorithm I combine the two evaluations as follows:

```
def combinar_evaluaciones(puntuacion_minimax,
    movimiento_score):
    ponderacion_minimax = 0.5
    ponderacion_movimiento = 0.5
    return (ponderacion_minimax * puntuacion_minimax) + (
    ponderacion_movimiento * movimiento_score)
```

Since I had to restart the whole development, I took the opportunity to design a more optimized evaluation function, although I was looking for similar results. In this function, I analyze the strings generated by each move and assign a value depending on whether the move belongs to the player or the AI. To encourage the AI to prioritize defensive moves, I give a higher score to the player's moves. Additionally, the AI receives a bonus score for proximity to the center of the board.

```
def evaluacion(self, ganador=None, victoria=False, board=None
    , movimientos={}):
    if victoria:
        if ganador == self.color_ia:
            return Defines.MAXINT
        elif ganador == self.color_jugador:
            return Defines.MININT
        elif ver_empate(board):
            return 0
    puntuacion = 0
```

```
    direcciones = [(1, 0), (0, 1), (1, 1), (1, -1)]
    for movimiento, color in movimientos.items():
        for dx, dy in direcciones:
            longitud = contar_cadena(board, movimiento.x,
movimiento.y, dx, dy, color)
            if color == self.color_ia:
                puntuacion += 2 ** longitud
            else:
                puntuacion -= 3 ** longitud
            if longitud == 4:
                if color == self.color_ia:
                    puntuacion += 500
                else:
                    puntuacion -= 800
            if longitud == 5:
                if color == self.color_ia:
                    puntuacion += 2000
                else:
                    puntuacion -= 5000
    centro = Defines.GRID_NUM // 2
    for fila in range(1, Defines.GRID_NUM - 1):
        for col in range(1, Defines.GRID_NUM - 1):
            if board[fila][col] == self.color_ia:
                puntuacion += max(0, 10 - abs(fila - centro)
- abs(col - centro))
            elif board[fila][col] == self.color_jugador:
                puntuacion -= max(0, 10 - abs(fila - centro)
- abs(col - centro))
    return puntuacion
```

The function of evaluating the movement is also quite simple, simply looking at whether a chain continues and its length.

```
def evaluar_posicion(self, posicion, board, color):
    direcciones = [(1, 0), (0, 1), (1, 1), (1, -1)]
    puntuacion = 0
    for dx, dy in direcciones:
        longitud = contar_cadena(board, posicion[0], posicion
[1], dx, dy, color)
        puntuacion += 2 ** longitud
    return puntuacion
```

3.4 Minmax

The minimax algorithm I have developed includes directly the alpha-beta pruning. First, I check for a win, and if there is no win, I check for a tie. Also, so that the execution time is not too long I set a maximum depth of 2. Then, I define the stopping conditions of the minimax, since it is a recursive function. These conditions are met when the depth reaches 0, one of the players has won or there is a draw. In each case, the function returns the corresponding evaluation, considering whether the winner is the player or the AI, depending on whether the algorithm is maximizing or minimizing.

Next, I declare two variables: one to store the best score and one to store a list of possible moves generated by the buscar_candidatos function. Then, I go through this list, taking pairs of moves and making sure that both selected moves are distinct. For each pair of moves, I call the simular_movimiento function, which creates an auxiliary board where I apply these moves. This board is passed to the next minimax call, which will calculate the score if depth 0, win or draw has been reached, or continue adding moves until the stop conditions are met.

The call to minimax returns a score that I use to determine the best combination of moves. If we are maximizing, I compare the score returned to the best score recorded and save the higher value of the two. I also compare the alpha value to the score and update alpha if it is greater. If the beta value is less than or equal to alpha, I prune this branch of the minimax tree. In the minimax case, the process is similar, only I keep the smaller values and compare the score to beta instead of alpha. Finally, I return the best score.

```
def min_max(self, board, depth, alpha, beta, maximizando):
    victoria = ver_victoria(board)
    if not victoria:
        empate = ver_empate(board)
    if depth > 2:
        depth = 2
    if depth == 0 or victoria or empate:
        if victoria:
            ganador = self.color_jugador if not maximizando
    else self.color_ia
            return self.evaluacion(ganador=ganador, victoria=
    True, board=board)
        elif empate:
            return self.evaluacion(ganador=None, victoria=
    True, board=board)
        return self.evaluacion(ganador=None, victoria=False,
    board=board)
    mejor_puntuacion = Defines.MININT if maximizando else
    Defines.MAXINT
    movimientos = self.buscar_candidatos(board)
    for i, movimiento1 in enumerate(movimientos):
        for j, movimiento2 in enumerate(movimientos):
            if i == j:
                continue
            tablero_aux = self.simular_movimiento(board,
    movimiento1, self.color_ia if maximizando else self.
    color_jugador)
            tablero_aux = self.simular_movimiento(tablero_aux
    , movimiento2, self.color_ia if maximizando else self.
    color_jugador)
            puntuacion = self.min_max(tablero_aux, depth - 1,
     alpha, beta, not maximizando)
            if maximizando:
                mejor_puntuacion = max(mejor_puntuacion,
    puntuacion)
                alpha = max(alpha, puntuacion)
                if beta <= alpha:
                    break
            else:
                mejor_puntuacion = min(mejor_puntuacion,
    puntuacion)
                beta = min(beta, puntuacion)
                if beta <= alpha:
                    break
    return mejor_puntuacion
```

Once I made all the improvements that I will talk about in the section 3.7, I slightly changed the minmax algorithm to optimize it and adapt it to these improvements. The final function is as follows:

```
def min_max(self, depth, alpha, beta, maximizando, movimientos, move):
    victoria = is_win_by_premove(self.m_board, move)
    empate = ver_empate(self.m_board)
    if depth > 2:
        depth = 2
    if depth == 0 or victoria or empate:
        if victoria:
            ganador = self.color_jugador if not maximizando else self.color_ia
            return self.evaluacion(ganador=ganador, victoria=True, board=self.m_board, movimientos=movimientos)
        elif empate:
            return self.evaluacion(ganador=None, victoria=True, board=self.m_board, movimientos=movimientos)
        return self.evaluacion(ganador=None, victoria=False, board=self.m_board, movimientos=movimientos)
    mejor_puntuacion = Defines.MININT if maximizando else Defines.MAXINT
    candidatos = self.buscar_candidatos(self.m_board, movimientos)
    for i, movimiento1 in enumerate(candidatos):
        for j, movimiento2 in enumerate(candidatos):
            if i == j:
                continue
            movimiento = StoneMove()
            movimiento.positions[0].x = movimiento1[0]
            movimiento.positions[0].y = movimiento1[1]
            movimiento.positions[1].x = movimiento2[0]
            movimiento.positions[1].y = movimiento2[1]
            make_move(self.m_board, movimiento, self.color_ia if maximizando else self.color_jugador, movimientos)
            puntuacion = self.min_max(depth - 1, alpha, beta, not maximizando, movimientos, movimiento)
            unmake_move(self.m_board, movimiento, movimientos)
            if maximizando:
                mejor_puntuacion = max(mejor_puntuacion, puntuacion)
                alpha = max(alpha, puntuacion)
                if beta <= alpha:
                    break
            else:
                mejor_puntuacion = min(mejor_puntuacion, puntuacion)
                beta = min(beta, puntuacion)
                if beta <= alpha:
                    break
    return mejor_puntuacion
```

3.5 Function simular_movimiento

I created a simple function to check movements on the board in an auxiliary way, without the need to modify the real board or undo changes afterwards. The function copies the actual board and adds the move you want to simulate.

```
def simular_movimiento(self, board, movimiento, color):
    fila, col = movimiento
    tablero_aux = [fila[:] for fila in board]
    tablero_aux[fila][col] = color
    return tablero_aux
```

To this function I applied an improvement in which instead of calling this function twice in the minmax I call it once and send it the two movements at the same time:

```
def simular_movimientos(self, board, movimiento1, movimiento2
    , color):
    fila1, col1 = movimiento1
    fila2, col2 = movimiento2
    tablero_aux = [fila[:] for fila in board]
    tablero_aux[fila1][col1] = color
    tablero_aux[fila2][col2] = color
    return tablero_aux
```

3.6 Changes in predefined functions

In order for all my code to integrate well with the provided code I had to make some changes, which, along with all the code were varying but here I will only put the final changes, since these changes have only been to make my code usable and not to optimize or improve the results.

The first thing is that in order to create a StonePosition object dictionary, I had to modify this class to be compatible with a Python dictionary. For this I had to add an __eq__ method with which to make comparisons between StonePosition objects, a __hash__ method to make StonePosition objects can be keys of a dictionary and the __repr__ method that I used to test the code. It should be noted that at the beginning I tried to make the dictionary of StoneMove objects so they also have these methods done:

```
class StonePosition:
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def __eq__(self, other):
        return isinstance(other, StonePosition) and self.x ==
     other.x and self.y == other.y
    def __hash__(self):
        return hash((self.x, self.y))
    def __repr__(self):
        return f"StonePosition(x={self.x}, y={self.y})"
class StoneMove:
    def __init__(self):
        self.positions = [StonePosition(0,0),StonePosition
    (0,0)]
        self.score = 0
    def __eq__(self, other):
        return (
            isinstance(other, StoneMove)
            and self.positions == other.positions
            and self.score == other.score
        )
    def __hash__(self):
        return hash((tuple(self.positions), self.score))
    def __repr__(self):
        return f"StoneMove(positions={self.positions}, score
    ={self.score})"
```

To add the moves to the dictionary I also had to modify the make_move and unmake_move methods as follows:

```
def make_move(board, move, color, movimientos):
    board[move.positions[0].x][move.positions[0].y] = color
    board[move.positions[1].x][move.positions[1].y] = color
    movimiento = copy.deepcopy(move)
```

```
    movimientos[movimiento.positions[0]] = color
    movimientos[movimiento.positions[1]] = color

def unmake_move(board, move, movimientos):
    board[move.positions[0].x][move.positions[0].y] = Defines
    .NOSTONE
    board[move.positions[1].x][move.positions[1].y] = Defines
    .NOSTONE
    if move.positions[0] in movimientos:
        del movimientos[move.positions[0]]
    if move.positions[1] in movimientos:
        del movimientos[move.positions[1]]
```

I also had to change the function find_possible_move. This function is now quite similar to the minmax, but as this one does not return a movement, I implemented it again here so that if it stays with the best movement. As in this function the AI already simulates a move that it wants to maximize, in the call to the minmax algorithm the maximizing value is passed as False so that it simulates the next move of the player.

```
def find_possible_move(self, depth, movimientos):
    mejor_puntuacion = Defines.MININT
    mejor_movimiento1 = None
    mejor_movimiento2 = None
    alpha = Defines.MININT
    beta = Defines.MAXINT
    candidatos = self.buscar_candidatos(self.m_board,
    movimientos)
    for i, movimiento1 in enumerate(candidatos):
        for j, movimiento2 in enumerate(candidatos):
            if i == j:
                continue
            movimiento = StoneMove()
            movimiento.positions[0].x = movimiento1[0]
            movimiento.positions[0].y = movimiento1[1]
            movimiento.positions[1].x = movimiento2[0]
            movimiento.positions[1].y = movimiento2[1]
            make_move(self.m_board, movimiento, self.color_ia
    , movimientos)
            if is_win_by_premove(self.m_board, movimiento):
                unmake_move(self.m_board, movimiento,
    movimientos)
                return movimiento1, movimiento2
            puntuacion = self.min_max(depth - 1, alpha, beta,
     False, movimientos, movimiento)
            unmake_move(self.m_board, movimiento, movimientos
    )
            if puntuacion > mejor_puntuacion:
                mejor_puntuacion = puntuacion
                mejor_movimiento1 = movimiento1
```

```
                mejor_movimiento2 = movimiento2
            alpha = max(alpha, puntuacion)
            if beta <= alpha:
                break
    return mejor_movimiento1, mejor_movimiento2
```

Finally, in the GameEngine and SearchEngine classes, I created a variable where I save the player's color apart from the AI color that is saved in m_chess_type. With this, in some of the game commands I had to add a line like self.m_chess_type = Defines.BLACK or self.color_jugador = Defines.WHITE. In addition I also added in the new command the line self.movimientos = {}, because otherwise when starting a new game the dictionary would keep the moves of the previous game.

3.7 General improvements

One improvement I made throughout the code was to change the structure of the moves from using lists to using the StoneMove and StonePosition structures. For example, in the simular_movimiento function, the moves are now instances of the StonePosition structure instead of an array of two positions. With this change I have also eliminated the score variables that I had before and I have changed them for the score value of the movements.

```
def simular_movimientos(self, board, movimiento1, movimiento2, color):
    tablero_aux = [fila[:] for fila in board]
    tablero_aux[movimiento1.x][movimiento1.y] = color
    tablero_aux[movimiento2.x][movimiento2.y] = color
    return tablero_aux
```

Another improvement I added was to dispense with the simulate motion function, as this required creating an auxiliary board continuously, generating a longer response time. I changed this by using the functions make_move and unmake_move.

I also created two lists that store the moves of each player so that instead of having to go through the whole board looking for the pieces, I had to iterate over these lists and it would be faster. This caused me problems at the beginning because I did it with a python array, which are not very optimized and it took me more than a minute to make the first move. To fix it I changed the arrays by set() objects and later by a dictionary in which I save as a key each movement and as a value I save the color of the token. You can see an example in the code of buscar_candidatos:

```
def buscar_candidatos(self):
    candidatos = set()
    direcciones = [(1, 0), (0, 1), (1, 1), (1, -1)]
    max_candidatos = 10
    movimientos_totales = self.movimientos_ia | self.movimientos_jugador

    for move in movimientos_totales:
        for pos in move.positions:
            for dx, dy in direcciones:
                nueva_fila = pos.x + dx
                nueva_col = pos.y + dy
                if (0 <= nueva_fila < Defines.GRID_NUM) and (0 <= nueva_col < Defines.GRID_NUM):
                    if self.m_board[nueva_fila][nueva_col] == Defines.NOSTONE:
                        movimiento = StonePosition(nueva_fila, nueva_col)
                        candidatos.add((movimiento))
    candidatos = list(candidatos)
```

```
17      if len(candidatos) > max_candidatos:
18          candidatos = random.sample(candidatos,max_candidatos)
19      return candidatos
```

An improvement that reduced quite a lot the response time was to change in all the code the way in which it looked for new movements. Before it returned an array with positions of the board, so to go through this I had to do in each function two for loops, one for the row and another for the column. To fix this I changed in the function search_candidates the movements in format board positions ([row][column]) to movements in format StoneMove(), which saves me the second for loop in all the functions.

3.8 Errors occurred

The development of this project has had quite a few drawbacks. The main one is the response time of the movements. As at the beginning I had to go through the whole board this generated a lot of delay in the time it took the ia to make the moves, so I tried to change it first to use lists, then sets and finally a dictionary but using these structures made the code not work in the GUI. This in turn meant that I could not implement a very complex evaluation function or much depth in the minmax algorithm as it would then take too long to put in a token.

```
if msg[4:] == "black":
    self.m_best_move = msg2move("JJ")
    move = copy.deepcopy(self.m_best_move)
    self.movimientos[move.positions[0]] = 1
    make_move(self.m_board, self.m_best_move, Defines.BLACK,
    self.movimientos, self.m_chess_type)
    self.m_chess_type = Defines.BLACK
        self.color_jugador = Defines.WHITE
```

As I have already mentioned, having changed several times the way I saved the movements made the code not work in the GUI and many times also in the terminal. This is due to changing such a fundamental structure several times as the errors accumulate. To fix this I went back to a much earlier version of the code and started changing it until I was sure that everything worked correctly and I could start implementing things again.

Another error I got when saving moves is that the player's moves that I saved were replaced by the AI's moves. This is because I saved a reference of the movement when I did it and later when the AI did his, it changed. To fix it I imported the copy library and I make a copy of the move before saving it in the dictionary to make sure that it is not going to change.

4 Experimentation

I have been experimenting with many things throughout the development. To see some of them I have been creating some print to see the values of some variable or the simulations of possible movements. In addition I created three terminal commands that I have used a lot, one to see the movements of the ia, another to see the movements of the player and another to see the longest chain of each player, which have helped me to adjust several functions.

Something that I tried to do to reduce the waiting time was to put threading and make the search of movements to be concurrent with the Python concurrent library. This worked quite well in the console, reducing from 3 and a half seconds per movement to 1.1. The problem is that in the GUI I was unable to make it work so I had to remove it.

I also tried to make a version of the code in which the candidate search was done by a genetic algorithm, but, although I was able to develop the algorithm, I was unable to implement it in the code, which made it futile to develop it. The code I developed is as follows:

```
class GeneticAlgorithm:
    def __init__(self, board, movimientos, color_ia,
    color_jugador, tam_poblacion=50, generaciones=50,
    ratio_mutacion=0.1):
        self.board = board
        self.movimientos = movimientos
        self.color_ia = color_ia
        self.color_jugador = color_jugador
        self.tam_poblacion = tam_poblacion
        self.generaciones = generaciones
        self.ratio_mutacion = ratio_mutacion
        self.poblacion = []
    def iniciar_poblacion(self):
        for _ in range(self.tam_poblacion):
            candidate = self.generar_candidato()
            self.poblacion.append(candidate)
    def generar_candidato(self):
        search_engine = SearchEngine()
        candidatos = search_engine.buscar_candidatos(self.
    board, self.movimientos)
        if len(candidatos) < 2:
            return None
        return random.sample(candidatos, 2)
    def fitness(self, candidate):
        move = StoneMove()
        search_engine = SearchEngine()
        move.positions[0] = StonePosition(candidate[0][0],
    candidate[0][1])
        move.positions[1] = StonePosition(candidate[1][0],
    candidate[1][1])
```

```
26          make_move(self.board, move, self.color_ia, self.
    movimientos)
27          score = search_engine.evaluacion(None, False, self.
    board, self.movimientos)
28          unmake_move(self.board, move, self.movimientos)
29          return score
30      def seleccionar_parientes(self):
31          self.poblacion.sort(key=self.fitness, reverse=True)
32          return self.poblacion[:2]
33      def crossover(self, padre1, padre2):
34          crossover_point = 1
35          hijo1 = padre1[:crossover_point] + padre2[
    crossover_point:]
36          hijo2 = padre2[:crossover_point] + padre1[
    crossover_point:]
37          return hijo1, hijo2
38      def mutar(self, candidate):
39          search_engine = SearchEngine()
40          if random.random() < self.ratio_mutacion:
41              possible_positions = search_engine.
    buscar_candidatos(self.board, self.movimientos)
42              candidate[random.randint(0, 1)] = random.choice(
    possible_positions)
43          return candidate
44      def run(self):
45          self.iniciar_poblacion()
46          for _ in range(self.generaciones):
47              new_poblacion = []
48              padres = self.seleccionar_parientes()
49              for _ in range(self.tam_poblacion // 2):
50                  hijo1, hijo2 = self.crossover(padres[0],
    padres[1])
51                  new_poblacion.extend([self.mutar(hijo1), self
    .mutar(hijo2)])
52              self.poblacion = new_poblacion
53          mejor_movimiento = max(self.poblacion, key=self.
    fitness)
54          return mejor_movimiento
```

5 Conclusions

The development of this artificial intelligence for Connect6 has been an interesting challenge combining programming with strategic analysis. Throughout the project, various techniques were implemented, such as the minimax algorithm with alpha-beta pruning, which allowed the AI to evaluate moves efficiently within the inherent complexity of the game. Although satisfactory results were achieved, such as the ability of the AI to compete at a reasonable strategic level, limitations related to response time and depth of analysis were also identified.

Attempts to overcome these limitations included tests with genetic algorithms to optimize move selection and the use of concurrent programming to reduce waiting times. However, these approaches could not be fully integrated into the system, due to the technical complexity and restrictions imposed by the code architecture. These experiences, although not successful in terms of implementation, provided valuable lessons on the applicability and difficulties of these techniques in specific contexts.

In conclusion, this project not only allowed the development of a functional AI for a complex game, but also opened the door to future improvements and explorations. The implementation of more advanced methods, such as neural networks or hybrid systems combining heuristic and evolutionary strategies, could represent a promising path to optimize AI performance. This work reinforces the importance of experimentation and continuous learning in the search for solutions to complex problems in artificial intelligence.

References

1. I.-C. Wu and D.-Y. Huang, "A new family of k-in-a-row games," 12 2006, pp. 180–194.
2. Y.-C. Liu, "A defensive strategy combined with threat-space search for connect6," Ph.D. dissertation, University of INSERT SCHOOL NAME HERE, 2009, phD Thesis.