Universidad del Valle de Guatemala
Inteligencia Artificial
Sección 30

Proyecto Final

Torneo Othello

Brian Carillo 21108

Guatemala, 02 de junio del 2024

Reporte escrito

Estrategias utilizadas

#1 Minimax con poda alpha-beta

Othello es un juego que cumple con las siguientes características:

1. Totalmente observable
2. Determinístico
3. Por Turnos
4. Estático
5. Zero Sum Game, es decir que las ganancias de un jugador se equilibran con las pérdidas de otro jugador.

Por lo tanto, inicialmente se planteó resolver el juego a través del concepto de simulación, visto en clase. Sin embargo, una simulación de todo el árbol de acciones supondría una amplia cantidad de estados, lo cual sería perjudicial para el tiempo de respuestas establecido en las instrucciones del torneo. Una mejor opción es juntar los conceptos de MaxValue y MinValue en el algoritmo Minimax, en donde consideramos que los dos jugadores juegan de manera óptima. En otras palabras, que el jugador manejado por la IA toma las decisiones que maximicen sus ganancias, mientras que el contrincante toma las decisiones que minimicen las ganancias de su oponente. Adicionalmente, se incluyeron los valores alfa-beta y profundidad, con el objetivo de optimizar el algoritmo MiniMax y retornar la mejor respuesta posible.

Para esto, se consideró alpha como la mejor alternativa para el jugador max en la rama, y beta como la mejor alternativa para el jugador min en la rama. Esto permitió tomar únicamente los valores importantes, y revisar las ramas indispensables.

Pseudocódigo

```
function make_move(board, player, row, col):
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1), (-1, -1), (-1, 1), (1, -1), (1, 1)]
    opponent = -player
    board[row][col] = player  # Coloca la ficha del jugador en la posición indicada

    for each direction in directions:
        dr, dc = direction
        r, c = row + dr, col + dc
        tiles_to_flip = []

        # Explora en la dirección actual
        while r is within board boundaries and c is within board boundaries and board[r][c] == opponent:
            add (r, c) to tiles_to_flip
            r = r + dr
            c = c + dc

        # Si se encuentra una ficha del jugador al final de la línea
        if r is within board boundaries and c is within board boundaries and board[r][c] == player:
            for each (flip_r, flip_c) in tiles_to_flip:
                board[flip_r][flip_c] = player  # Voltea las fichas del oponente
```

```
function minimax(board, player, depth, alpha, beta, maximizingPlayer):
    if depth == 0 or is_terminal_node(board):
        return evaluate_board(board, player)

    if maximizingPlayer:
        maxEval = -∞
        for each move in valid_moves(board, player):
            new_board = copy_board(board)
            make_move(new_board, player, move)
            eval = minimax(new_board, -player, depth - 1, alpha, beta, False)
            maxEval = max(maxEval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return maxEval
    else:
        minEval = ∞
        for each move in valid_moves(board, player):
            new_board = copy_board(board)
            make_move(new_board, player, move)
            eval = minimax(new_board, -player, depth - 1, alpha, beta, True)
            minEval = min(minEval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break
        return minEval

function AI_MOVE(board, player):
    bestMove = null
    bestScore = -∞
    depth = initial_depth  # Set the initial depth

    for each move in valid_moves(board, player):
        new_board = copy_board(board)
        make_move(new_board, player, move)
        moveScore = minimax(new_board, -player, depth, -∞, ∞, False)
        if moveScore > bestScore:
            bestScore = moveScore
            bestMove = move

    return bestMove
```

#2 Heurísticas para el tablero

Para la evaluación del tablero, se utilizaron diversas heurísticas planteadas según las características del juego, con el objetivo de establecer un score adecuado al escenario del juego tras realizar cierta acción.

1. **Paridad:** Este es un enfoque greedy en donde se sumó 1 punto por cada ficha del jugador en el tablero, así como también se restó 1 punto por cada ficha del contrincante en el tablero..
2. **Esquinas:** Se asignó 100 puntos al score por cada ficha del jugador en una esquina. Se restó la misma cantidad por cada ficha del contrincante en una esquina.

3. **Casillas X:** Estas son casillas en las posiciones (1, 1), (1, 6), (6, 1), (6, 6), que son consideradas perjudiciales para el jugador. Por lo tanto, se sumó 50 puntos por cada ficha del oponente en una casilla X, así como se restó dicha cantidad, en caso una ficha del jugador estuviera en dicha posición.
4. **Casillas C:** Estas son casillas en las posiciones (0, 1), (1, 0), (0, 6), (1, 7), (6, 0), (7, 1), (6, 7), (7, 6), consideradas menos perjudiciales que las casillas X, así que se sumó 20 puntos por cada ficha oponente en dicha situación, así como se restó dicha cantidad por las fichas del jugador en dicha posición.
5. **Movilidad:** Se sumaron 10 puntos por cada movimiento válido que tiene el jugador, y se restaron 10 puntos para cada movimiento válido que tiene el contrincante.

Estas ideas fueron tomadas de diferentes fuentes en las que se describen diversas heurísticas, y la efectividad de las mismas. Se adjuntan las referencias al final del reporte.

Pseudocódigo

```
function evaluate_board(board, player):
    opponent = -player
    corners = [(0, 0), (0, 7), (7, 0), (7, 7)]
    x_squares = [(1, 1), (1, 6), (6, 1), (6, 6)]
    c_squares = [(0, 1), (1, 0), (0, 6), (1, 7), (6, 0), (7, 1), (6, 7), (7, 6)]

    score = 0
    player_moves = length(valid_moves(board, player))
    opponent_moves = length(valid_moves(board, opponent))

    for row from 0 to 7:
        for col from 0 to 7:
            if board[row][col] == player:
                score = score + 1
                if (row, col) is in corners:
                    score = score + 100
                elif (row, col) is in x_squares:
                    score = score - 50
                elif (row, col) is in c_squares:
                    score = score - 20

            elif board[row][col] == opponent:
                score = score - 1
                if (row, col) is in corners:
                    score = score - 100
                elif (row, col) is in x_squares:
                    score = score + 50
                elif (row, col) is in c_squares:
                    score = score + 20

    # Añadir la movilidad a la evaluación
    score = score + (player_moves - opponent_moves) * 10

    return score
```

#3 Elementos estocásticos

Dada las recomendaciones del catedrático, se consideró como una posible estrategia el incluir elementos estocásticos al modelo, es decir introducir aleatoriedad en las acciones, con el objetivo de que las acciones realizadas por la IA fueran menos predecibles y potencialmente más robustas. Para esto, los mejores movimientos con la misma puntuación fueron agregados a una lista, de la cual se escogió de manera aleatoria la que sería devuelta por la función.

Pseudocódigo

```
function AI_MOVE(board, player):
    best_moves = []
    best_score = -∞

    for each move in valid_moves(board, player):
        new_board = copy_board(board)
        make_move(new_board, player, move)
        move_score = minimax(new_board, -player, depth, -∞, ∞, False)

        if move_score > best_score:
            best_score = move_score
            best_moves = [move]
        elif move_score == best_score:
            best_moves.append(move)

    return random_choice(best_moves)
```

#4 Búsqueda adaptativa

Finalmente, según la cantidad de fichas en el tablero, se planteó una estrategia de búsqueda adaptativa, es decir, según la fase del juego se modificaron los valores de profundidades utilizados por el algoritmo minimax, así como también, se incluyó una variable booleana que determinaría si la decisión sobre la mejor acción pasaba a ser determinista.

Pseudocódigo

```
function AI_MOVE(board, player):
    valid_moves = get_valid_moves(board, player)
    if valid_moves is empty:
        return None

    best_moves = []
    best_score = -∞
    depth = 3  # Initial depth

    # Adjust depth based on the state of the game
    total_pieces = count_total_pieces(board)
    if total_pieces > 44:  # Endgame phase
        depth = 5
        deterministic = True
    else:
        deterministic = False
```

```
    if total_pieces < 12:  # Opening phase
        depth = 2

    for each move in valid_moves:
        row, col = move
        new_board = copy_board(board)
        make_move(new_board, player, row, col)
        move_score = minimax(new_board, -player, depth, -∞, ∞, False)

        if move_score > best_score:
            best_score = move_score
            best_moves = [move]  # Reset the list of best moves
        elif move_score == best_score:
            best_moves.append(move)  # Add the move to the list of best moves

    # Select deterministically in the endgame phase
    if deterministic:
        return best_moves[0]  # Select the first best move
    else:
        return random_choice(best_moves)  # Select randomly among the best moves
```

Posibles mejoras tras los resultados del Torneo

Tras analizar el desempeño en el torneo, algunas mejoras en las estrategias son las siguientes:

1. Mejorar la heurística del tablero: Considerar celdas como los costados, de manera que el score obtenido es más preciso.
2. Mejorar la aleatoriedad: Considerar no solamente los mejores movimientos, sino establecer un umbral de score, con el objetivo de hacer mucho menos predecibles las decisiones de la IA.

Referencias

Lung, J. (2022). How to write an Othello AI with Alpha-Beta search. Obtenido de https://medium.com/@gmu1233/how-to-write-an-othello-ai-with-alpha-beta-search-58131ffe67eb

Zhang, Z. y Che, Y. (2014). Searching Algorithms in Playing Othello. Obtenido de https://zzutk.github.io/docs/reports/2014.04%20-%20Searching%20Algorithms%20in%20Playing%20Othello.pdf

Sannidhanam, V. y Annamalai, M. (s.f.). An Analysis of Heuristics in Othello. Obtenido de https://courses.cs.washington.edu/courses/cse573/04au/Project/mini1/RUSSIA/Final_Paper.pdf