

Documentación de UltraProHexPlayer

Autor: Equipo de Desarrollo

12 de abril de 2025

Índice

1. Introducción	2
2. Visión general de la clase	2
2.1. Atributos principales	2
3. Hashing Zobrist	3
3.1. Inicialización	3
4. Selección de movimiento	3
5. Búsqueda <i>Alpha-Beta</i>	4
6. Heurísticas de evaluación	5
6.1. Prioridad de movimientos	5
6.2. Distancia mínima (Dijkstra)	5
6.3. Control de territorio	5
6.4. Detección de patrones	5
6.5. Evaluación rápida	5
7. Monte Carlo Tree Search	6
8. Aprendizaje por refuerzo	6
9. Conclusión	6

1. Introducción

La clase `UltraProHexPlayer` implementa un jugador para el juego de Hex que combina múltiples técnicas de inteligencia artificial:

- Hashing Zobrist para reconocimiento rápido de posiciones.
- Tabla de transposición para almacenar evaluaciones previas.
- Búsqueda minimax con poda *alpha-beta* y ordenamiento heurístico de movimientos.
- Evaluaciones heurísticas avanzadas (distancia mínima, control de territorio, detección de patrones).
- Monte Carlo Tree Search (MCTS) para fases de apertura.
- Aprendizaje por refuerzo simple basado en refuerzo de estadísticas de movimientos.

2. Visión general de la clase

Listing 1: Definición inicial y atributos de clase

```
'''
def __init__(self, player_id: int):
    super().__init__(player_id)
    self.player_id = player_id
    self.opponent_id = 3 - player_id
    self.transposition_table = {}
    self.neighbor_cache = {}
    self.move_history = []          # [(board_key, move)]
    self.move_stats = {}           # {(board_key, move): score}
    self.directions = [(-1, 0), (1, 0), (0, -1), (0, 1), (-1, 1), (1, -1)]
    self.max_depth = 3
'''
```

2.1. Atributos principales

- `zobrist_table`: Tabla estática de valores aleatorios para hashing Zobrist.
- `transposition_table`: Diccionario para almacenar evaluaciones de posiciones ya analizadas.
- `neighbor_cache`: Caché opcional para vecindad de casillas.
- `move_history`: Historial de movimientos jugados, usado para aprendizaje.
- `move_stats`: Estadísticas de refuerzo asociadas a movimientos y posiciones.
- `directions`: Lista de direcciones hexagonales para navegación.
- `max_depth`: Profundidad máxima para búsqueda alpha-beta.

3. Hashing Zobrist

3.1. Inicialización

Se generan valores aleatorios de 64 bits para cada casilla y cada posible estado (vacío, jugador 1, jugador 2):

Listing 2: Método `init_zobrist_table`

\subsection{Cálculo de la llave} Se recorre el tablero y se aplica XOR con los v

4. Selección de movimiento

El método `play` orquesta la decisión de jugada:

Listing 3: Método `play`

```
'''
# Limpiar transposición si crece mucho
if len(self.transposition_table) > 200000:
    self.transposition_table.clear()

possible_moves = board.get_possible_moves()
total_cells = board.size * board.size

# Apertura: usar MCTS si tablero muy vacío
if len(possible_moves) > total_cells * 0.8:
    return self._mcts_select_move(board, simulations=200, top_k=10)

# Primera jugada: centro
if len(possible_moves) == total_cells:
    return (board.size // 2, board.size // 2)

# Victoria o bloqueo inmediato
for move in possible_moves:
    b2 = board.clone()
    b2.place_piece(move[0], move[1], self.player_id)
    if b2.check_connection(self.player_id):
        return move
for move in possible_moves:
    b2 = board.clone()
    b2.place_piece(move[0], move[1], self.opponent_id)
    if b2.check_connection(self.opponent_id):
        return move

# Ajuste profundidad según fase
moves_played = total_cells - len(possible_moves)
ratio = moves_played / total_cells
if ratio < 0.2:
    self.max_depth = 2
elif ratio < 0.6:
    self.max_depth = 3
```

```

else:
    self.max_depth = 4

# Ordenar y limitar movimientos
move_limit = min(12, len(possible_moves))
ordered = sorted(
    possible_moves,
    key=lambda m: self._move_priority(board, m),
    reverse=True
)[:move_limit]

best_move, best_val = ordered[0], float('-inf')
alpha, beta = float('-inf'), float('inf')

for move in ordered:
    b2 = board.clone()
    b2.place_piece(move[0], move[1], self.player_id)
    val = -self.alpha_beta(b2, self.max_depth-1, -beta, -alpha, False)
    if val > best_val:
        best_val, best_move = val, move
        alpha = max(alpha, val)

# Guardar en historial
key = self._normalized_board_key(board)
self.move_history.append((key, best_move))
if len(self.move_history) > 200:
    self.move_history.pop(0)

return best_move
'''

```

5. Búsqueda *Alpha-Beta*

Implementa minimax con poda y transposición:

Listing 4: Método `alpha_beta`

```

'''
if board.check_connection(self.player_id):
    return 10000 + depth
if board.check_connection(self.opponent_id):
    return -10000 - depth
if depth == 0:
    score = self._fast_evaluation(board)
    self.transposition_table[key] = score
    return score

moves = board.get_possible_moves()
if not moves:
    return 0

```

```

    if maximizing:
        value = float('-inf')
        ordered = sorted(moves, key=lambda m: self._move_priority(board, m), reverse=True)
        for m in ordered:
            b2 = board.clone(); b2.place_piece(m[0], m[1], self.player_id)
            value = max(value, self.alpha_beta(b2, depth-1, alpha, beta, False))
            alpha = max(alpha, value)
            if alpha >= beta:
                break
    else:
        value = float('inf')
        ordered = sorted(moves, key=lambda m: self._move_priority_opponent(board, m), reverse=True)
        for m in ordered:
            b2 = board.clone(); b2.place_piece(m[0], m[1], self.opponent_id)
            value = min(value, self.alpha_beta(b2, depth-1, alpha, beta, True))
            beta = min(beta, value)
            if beta <= alpha:
                break

    self.transposition_table[key] = value
    return value
'''

```

6. Heurísticas de evaluación

6.1. Prioridad de movimientos

Explica la función `_move_priority` y `_move_priority_opponent` con bonificaciones por posición, proximidad y puentes.

6.2. Distancia mínima (Dijkstra)

Método `_calculate_min_distance` calcula la distancia mínima entre los bordes relevantes.

6.3. Control de territorio

Método `_calculate_territory_control` mide la influencia de casillas propias.

6.4. Detección de patrones

Método `_pattern_evaluation` detecta estructuras como puentes y penaliza fichas aisladas.

6.5. Evaluación rápida

Combina las anteriores en `_fast_evaluation`:

- Distancia: peso 60.
- Control de territorio: peso 40.
- Patrones: peso 25.

7. Monte Carlo Tree Search

Método `_mcts_select_move` realiza simulaciones aleatorias y elige el movimiento con más victorias.

8. Aprendizaje por refuerzo

Método `learn_from_game` ajusta `move_stats` según resultado de la partida.

9. Conclusión

La combinación de técnicas garantiza un jugador robusto en distintas fases de la partida.