# Informe del Proyecto de IA: HEX

Kendry Javier del Pino Barbosa C-312
MATCOM

11 de abril de 2025

## Índice

# 1. Visión General

Este proyecto desarrolla la clase `IAPlayer` para el juego Hex, implementando una estrategia basada en el algoritmo *Minimax* mejorado con poda alfa-beta y una función heurística especializada. La idea central es evaluar el estado actual del tablero para tomar decisiones que incrementen las posibilidades de conectar las piezas propias, mientras se disminuyen las oportunidades del oponente. Aunque la lógica del algoritmo minimax se mantiene sencilla, aquí se profundiza en el mecanismo del ordenamiento de movimientos y el diseño de la función heurística, dos elementos claves para el rendimiento estratégico del agente.

# 2. Componentes Fundamentales

El sistema está diseñado sobre tres pilares:

- **Selección y Ordenamiento de Movimientos**: Se generan todas las jugadas posibles y se les asigna un puntaje basado en su posición relativa al centro del tablero y su potencial estratégico en los bordes.
- **Algoritmo Minimax con Poda Alfa-Beta**: Se utiliza para simular el futuro juego, descartando ramas irrelevantes y optimizando la búsqueda para elegir la jugada que maximice las oportunidades de victoria.
- **Función Heurística de Evaluación**: Asigna un valor numérico al tablero, considerando tanto la centralidad de las piezas propias como la ocupación de zonas clave, a la vez que penaliza la posición adversaria.

# 3. Flujo y Lógica de Decisión

El agente sigue una serie de pasos que garantizan una selección informada:

1. **Generación de Movimientos:** Se obtienen todos los movimientos posibles del tablero.
2. **Ordenamiento Previo:** Se ordenan los movimientos según su cercanía al centro y la posibilidad de ocupar bordes estratégicos. Este proceso facilita que la poda alfa-beta sea más efectiva.
3. **Simulación y Evaluación:** Cada movimiento se evalúa mediante el algoritmo minimax, implementado inicialmente sin y luego con poda alfa-beta para optimizar el rendimiento.

4. **Selección de la Mejor Jugada:** Se elige el movimiento con el mayor puntaje, considerando tanto las oportunidades ofensivas como defensivas.

# 4. Detalles de la Implementación

## 4.1. Constructor y Método `play`

El constructor inicializa el agente estableciendo la profundidad máxima de búsqueda. El método `play` es responsable de procesar los movimientos posibles, ordenarlos y evaluar cada uno mediante el algoritmo minimax.

```
class IAPlayer(Player):
    def __init__(self, player_id: int):
        super().__init__(player_id)
        self.max_depth = 2  # Profundidad máxima de bú
            squeda

    def play(self, board: HexBoard) -> tuple:
        possible_moves = board.get_possible_moves()
        # Se ordenan los movimientos para priorizar
            jugadas con mayor potencial.
        possible_moves = self._sort_moves(board,
            possible_moves)
        best_score = float('-inf')
        best_move = None

        for move in possible_moves:
            new_board = board.clone()
            new_board.place_piece(move[0], move[1], self.
                player_id)
            # Se evalúa cada movimiento utilizando
                minimax sin alpha-beta.
            score = self._minimax(new_board, self.
                max_depth - 1, False)
            if score > best_score:
                best_score = score
                best_move = move

        return best_move if best_move else random.choice(
            possible_moves)
```

Listing 1: Constructor y método play sin poda alfa-beta

## 4.2. Ordenamiento de Movimientos (_sort_moves)

El ordenamiento de movimientos es esencial para reducir la cantidad de ramas evaluadas durante la búsqueda. Se asigna a cada movimiento un puntaje basado en dos criterios:

1. **Cercanía al Centro:** Se utiliza la fórmula

$$\text{score} = -\left(|x - \frac{n}{2}| + |y - \frac{n}{2}|\right)$$

donde $(x, y)$ es la posición del movimiento y $n$ es el tamaño del tablero. Este valor negativo favorece movimientos más próximos al centro, ya que se espera que estas posiciones ofrezcan mayor flexibilidad para crear conexiones.

2. **Bono por Bordes Estratégicos:** Dependiendo del jugador, se adiciona un bono a los movimientos que se encuentran en los bordes críticos: para el jugador 1 se premian posiciones en la primera y última columna, mientras que para el jugador 2 se priorizan la primera y última fila.

```
def _sort_moves(self, board: HexBoard, moves: list) ->
    list:
     move_scores = []
     for move in moves:
         % Se calcula la penalización por estar alejado
            del centro.
         score = - (abs(move[0] - board.size // 2) + abs(
            move[1] - board.size // 2))
         % Se aplica un bono en bordes estratégicos:
         if self.player_id == 1:
             if move[1] == 0 or move[1] == board.size - 1:
                 score += 5
         else:
             if move[0] == 0 or move[0] == board.size - 1:
                 score += 5
         move_scores.append((move, score))
     % Se ordenan los movimientos de mayor a menor puntaje
        .
     return [move for move, _ in sorted(move_scores, key=
        lambda x: x[1], reverse=True)]
```

Listing 2: Método _sort_moves

## 4.3. Algoritmo Minimax con Poda Alfa-Beta

La versión mejorada del algoritmo minimax incorpora poda alfa-beta para evitar evaluar ramas que no influyen en la decisión final. Se actualizan dinámicamente los valores de $\alpha$ y $\beta$, lo que permite cortar la búsqueda cuando la evaluación ya no puede mejorar.

```
def _minimax(self, board: HexBoard, depth: int,
    is_maximizing: bool, alpha: float, beta: float) ->
    float:
     opponent = 2 if self.player_id == 1 else 1

     % Condiciones terminales: conexión ganadora o
        profundidad máxima alcanzada.
     if board.check_connection(self.player_id):
         return 1000
     if board.check_connection(opponent):
         return -1000
     if depth == 0 or not board.get_possible_moves():
         return self._evaluate_board(board)

     possible_moves = board.get_possible_moves()
     if is_maximizing:
         max_eval = float('-inf')
         for move in possible_moves:
             new_board = board.clone()
             new_board.place_piece(move[0], move[1], self.
                player_id)
             eval = self._minimax(new_board, depth - 1,
                False, alpha, beta)
             max_eval = max(max_eval, eval)
             alpha = max(alpha, eval)
             if beta <= alpha:
                 break  % Poda beta: descarta ramas
                    irrelevantes.
         return max_eval
     else:
         min_eval = float('inf')
         for move in possible_moves:
             new_board = board.clone()
             new_board.place_piece(move[0], move[1],
                opponent)
             eval = self._minimax(new_board, depth - 1,
                True, alpha, beta)
```

```
30                min_eval = min(min_eval, eval)
31                beta = min(beta, eval)
32                if beta <= alpha:
33                    break  % Poda alfa: evita caminos sin
                          impacto.
34            return min_eval
```

Listing 3: Algoritmo minimax con poda alfa-beta

## 4.4. Función Heurística de Evaluación del Tablero

La función `_evaluate_board` es el núcleo para determinar la calidad de una posición. Se basa en tres componentes clave:

1. **Evaluación de la Centralidad:** Cada pieza del jugador suma un valor proporcional a su cercanía al centro, calculado mediante:

$$\text{contribución} = 10 - \left(|x - \frac{n}{2}| + |y - \frac{n}{2}|\right)$$

   Esto favorece posiciones centrales, fundamentales para formar rutas de conexión.

2. **Bonos por Ocupación de Bordes Estratégicos:** Las ubicaciones en los bordes críticos reciben un bono adicional. Por ejemplo, el jugador 1 recibe un bono en la primera y última columna, mientras que el jugador 2 en la primera y última fila. Este bono incentiva la formación de rutas en los sectores decisivos del tablero.

3. **Penalización por Posiciones del Oponente:** Se resta el mismo valor que se utiliza para evaluar la centralidad en las posiciones del oponente. Esto reduce el puntaje del tablero cuando las piezas rivales se aproximan al centro, orientando la estrategia hacia la contención.

```
1  def _evaluate_board(self, board: HexBoard) -> float:
2      opponent = 2 if self.player_id == 1 else 1
3      score = 0
4
5      % Obtener las posiciones de ambos jugadores
6      my_positions = board.player_positions[self.player_id]
7      opp_positions = board.player_positions[opponent]
8
9      % Aporte por centralidad: mayor valor para piezas más
           cerca del centro.
```

```
10      for pos in my_positions:
11          score += 10 - (abs(pos[0] - board.size // 2) +
               abs(pos[1] - board.size // 2))
12      for pos in opp_positions:
13          score -= 10 - (abs(pos[0] - board.size // 2) +
               abs(pos[1] - board.size // 2))
14
15      % Bono por posiciones en bordes estratégicos
16      if self.player_id == 1:  % Estrategia para jugador 1
           (izquierda-derecha)
17          for pos in my_positions:
18              if pos[1] == 0:
19                  score += 15
20              if pos[1] == board.size - 1:
21                  score += 15
22      else:  % Estrategia para jugador 2 (arriba-abajo)
23          for pos in my_positions:
24              if pos[0] == 0:
25                  score += 15
26              if pos[0] == board.size - 1:
27                  score += 15
28
29      return score
```

Listing 4: Función Heurística _evaluate_board

#### 4.4.1. Análisis de la Función Heurística

- **Centralidad:** Favorecer las piezas que se ubican en el centro es vital para una mayor flexibilidad, ya que permiten conexiones en cualquier dirección.
- **Bonos Estratégicos:** Incentivar las posiciones en los bordes críticos ayuda a formar rutas de conexión que, además, dificultan la intervención del oponente.
- **Penalización del Oponente:** Al restar el valor derivado de la centralidad en función de las piezas rivales, la heurística no solo busca aumentar la fortaleza ofensiva, sino también mitigar la amenaza adversaria.

# 5. Ventajas del Enfoque Implementado

- **Precisión en la Evaluación:** La combinación de centralidad, bonos estratégicos y penalizaciones ofrece una valoración realista del tablero.
- **Eficiencia:** El ordenamiento previo y la poda alfa-beta optimizan la búsqueda, permitiendo una toma de decisiones rápida.
- **Adaptabilidad:** El diseño modular permite ajustar y perfeccionar tanto la estrategia como la función heurística en futuras iteraciones.

# 6. Conclusión

El proyecto IAPlayer para el juego Hex utiliza una estrategia inteligente que combina un algoritmo minimax optimizado con poda alfa-beta y una función heurística detallada. La habilidad de ordenar los movimientos de manera eficiente y valorar el tablero a través de criterios como la centralidad, los bonos estratégicos y la penalización del oponente, garantiza que la IA tome decisiones informadas para maximizar sus oportunidades de victoria y minimizar las amenazas adversarias.