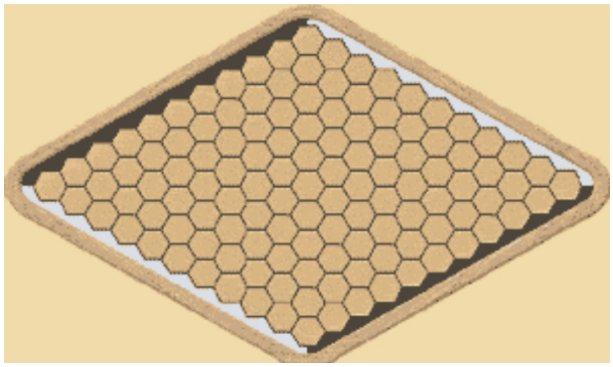
Informe

HEX AI Player



Richard Alejandro Matos Arderí

Grupo 311, Ciencia de la Computación

Facultad de Matemática y Computación

Universidad de La Habana

MATCOM

2025

Índice

1 Implementación del Jugador Inteligente

1.1 Enfoque basado en Monte Carlo Tree Search

Esta implementación de jugador inteligente para HEX utiliza una versión mejorada del algoritmo **Monte Carlo Tree Search (MCTS)**, siguiendo la estructura general del pseudocódigo presentado en el libro "Inteligencia Artificial: Un enfoque moderno" . El algoritmo se compone de cuatro fases principales:

1. **Selección (SELECT)**: Recorre el árbol desde la raíz seleccionando nodos hijos según el criterio UCB1 hasta llegar a un nodo hoja.
2. **Expansión (EXPAND)**: Crea uno o más nodos hijos a partir del nodo hoja si el juego no ha terminado.
3. **Simulación (SIMULATE)**: Juega una partida aleatoria o semi-aleatoria desde el nuevo nodo hasta alcanzar un estado terminal.
4. **Retropropagación (BACK-PROPAGATE)**: Actualiza las estadísticas de victorias y visitas en todos los nodos del camino recorrido.

```
function MONTE-CARLO-TREE-SEARCH(state) returns an action
tree ← NODE(state)
while IS-TIME-REMAINING() do
leaf ← SELECT(tree)
child ← EXPAND(leaf)
result ← SIMULATE(child)
BACK-PROPAGATE(result, child)
return the move in ACTIONS(state) whose node has highest number of playouts
```

Figura 1: Pseudocódigo de MCTS adaptado de "Inteligencia Artificial: Un enfoque moderno"

2 Mejoras Implementadas

La implementación extiende el MCTS básico con varias mejoras clave:

2.1 Selección Heurística de Movimientos

En la fase de expansión, en lugar de seleccionar movimientos puramente al azar, es utilizada una función heurística combinada que evalúa:

- Distancia del camino más corto a la victoria (Dijkstra)
- Patrones estratégicos (puentes)
- Control del centro del tablero
- Conectividad de grupos
- Movilidad potencial
- Proximidad a bordes estratégicos

```
def select_heuristic_move(self) -> tuple:
    moves = self.untried_moves
    move_scores = [(self._heuristic_score(move), move) for move in moves
    ]
    move_scores.sort(reverse=True, key=lambda x: x[0])
    return move_scores[0][1]
```

Listing 1: Función de expansión heurística

2.2 Simulación Informada

La fase de simulación utiliza políticas heurísticas en lugar de movimientos aleatorios, lo que produce estimaciones de valor más precisas:

```
def _simulate(self, node: EnhancedMCTSNode) -> float:
temp_board = node.board.clone()
current_player = node.player_id
steps = 0
max_steps = temp_board.size * 2

while steps < max_steps:
    if temp_board.check_connection(current_player):
        return 1.0 if current_player == self.player_id else 0.0
    moves = temp_board.get_possible_moves()
    move_scores = [(node._heuristic_score(move), move) for move in moves
    ]
    best_move = max(move_scores, key=lambda x: x[0])[1]
    temp_board.place_piece(*best_move, current_player)
    current_player = 3 - current_player
    steps += 1
return 0.5
```

Listing 2: Simulación basada en heurísticas

2.3 Estructura de Datos

El árbol MCTS se implementa mediante la clase `EnhancedMCTSNode`, que almacena:

- Estado del tablero (clonable)
- Movimiento que llevó a este nodo
- Estadísticas de visitas y victorias
- Hijos expandidos
- Movimientos no probados
- ID del jugador actual

```
class EnhancedMCTSNode:
  def __init__(self, board: HexBoard, parent=None, move=None,
 player_id: int = 1):
    self.board = board.clone()
    self.parent = parent
    self.move = move
    self.children: List[EnhancedMCTSNode] = []
    self.wins = 0
    self.visits = 0
    self.untried_moves = board.get_possible_moves()
    self.player_id = player_id
```

Listing 3: Estructura del nodo MCTS

2.4 Control de Tiempo

El algoritmo ejecuta iteraciones MCTS dentro del límite de tiempo asignado (por defecto 9 segundos), reservando 0.1 segundos para la selección final del mejor movimiento:

```
while time.time() - start_time < self.time_limit - 0.1:
  node = self._select(root)
  if not node.is_terminal():
    node = node.expand()
  result = self._simulate(node)
  self._backpropagate(node, result)
  iterations += 1
```

Listing 4: Bucle principal de búsqueda