MCTS aplicado al juego de Hex

Pablo Gómez Vidal

1. Introducción

Monte Carlo Tree Search (MCTS) es un enfoque poderoso para diseñar bots que jueguen juegos o para resolver problemas de decisión secuenciales. El método se basa en una búsqueda en árbol que equilibra exploración y explotación. MCTS realiza muestreos aleatorios en forma de simulaciones y almacena estadísticas de acciones para tomar decisiones más informadas en cada iteración subsiguiente.

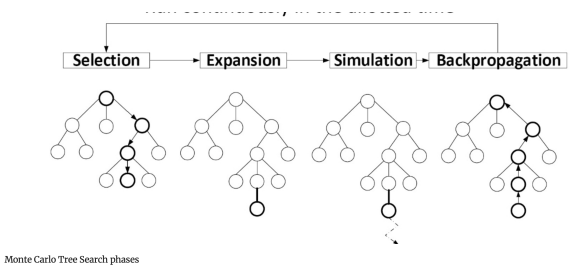
En problemas no triviales, el espacio de estados (por ejemplo, un árbol del juego de Hex) no puede ser explorado por completo. En aplicaciones prácticas, a MCTS se le asigna un presupuesto computacional que puede especificarse por el número de iteraciones(es lo que se implementó en este caso) o el tiempo disponible para tomar una decisión. MCTS es un algoritmo *anytime*. Esta propiedad significa que puede detenerse en cualquier momento y proporcionar la mejor acción actual.

1.1. ¿Por qué este algoritmo?

La principal causa por la que se decide usar **MCTS** en vez del Minimax con poda alfa-beta, es porque computacionalmente es menos costoso y al comparar una versión **mvp** de ambos algoritmos, este resultó no solo ser más rápido sino también más efectivo, lo cual introduce la segunda razón por la que se decide usar MCTS, la falta de una heurística fuerte para este juego, no me fue posible encontrar una heurística .ªbsoluta"debido a la naturaleza abstracta del juego, por lo que al usar Minimax era necesario unir diferentes heurísticas en diferentes etapas del juego para su mejora, mientras que MCTS no necesita una heurística explícita, ya que evalúa movimientos mediante simulaciones aleatorias hasta el final del juego. Esto evita sesgos o errores derivados de una heurística mal diseñada.

2. Fases del Algoritmo

El algoritmo MCTS consta de cuatro fases principales:



Monte Carlo Tree Search phases

2.1. 1. Selección

El algoritmo busca la parte del árbol que ya ha sido representada en la memoria. La selección siempre comienza desde el nodo raíz y, en cada nivel, selecciona el siguiente nodo según la

política de selección. Esta fase termina cuando se visita un nodo hoja y el siguiente nodo no está representado en la memoria aún o no se ha alcanzado un estado terminal del juego.

```
 while node.children and node.is_fully_expanded():
            node = node.best_child()
```

2.2. 2. Expansión

A menos que la selección haya alcanzado un estado terminal, la expansión agrega al menos un nuevo nodo hijo al árbol representado en la memoria. El nuevo nodo corresponde a un estado que se alcanza realizando la última acción en la fase de selección. Si la expansión alcanza un estado terminal, la iteración actual pasa directamente a la retropropagación.

```
possible_moves = node.board.get_possible_moves()
        if possible_moves:

            tried_moves = []
            for child in node.children:
                tried_moves.append(child.move)

            untried_moves = []
            for move in possible_moves:
                if move not in tried_moves:
                    untried_moves.append(move)

            if untried_moves:
                move = random.choice(untried_moves)
                new_board = node.board.clone()

                # Escogemos el jugador actual, si no tengo padre
                   ↪ entonces soy la raiz por tanto es el
                   ↪ simulation_player
                # En otro caso vamos switcheando
                current_player = 0
                if node.parent is None:
                    current_player = simulation_player
                else:
                    current_player = 3 - node.parent.player_id

                new_board.place_piece(move[0], move[1], current_player)
                child_node = MCTSNode(new_board, move, node,
                   ↪ current_player)
                node.children.append(child_node)
                node = child_node
```

2.3. 3. Simulación

Realiza una simulación aleatoria completa del juego, es decir, alcanza un estado terminal y obtiene las recompensas. Esta es la parte "Monte Carlo"del algoritmo.

```
def simulate_random_playout(board: HexBoard, current_player: int):
    board_copy = board.clone()
    player = current_player
    while True:
        possible_moves = board_copy.get_possible_moves()
        if not possible_moves:
```

```
            return 0  # empate, creo que esto no puede pasar pero bueno
                ↪ por si acaso
        move = random.choice(possible_moves)
        board_copy.place_piece(move[0], move[1], player)
        if board_copy.check_connection(player):
            return player  # devuelvo el player que haya ganado
        player = 3 - player  # si aun no gana nadie vamos switchendo y
            ↪ seguimos en el while
```

2.4. 4. Retropropagación

Actualiza las recompensas (+1 en caso de victoria), para todos los agentes modelados en el juego, de regreso a todos los nodos a lo largo de la ruta desde el último nodo visitado en el árbol (la hoja) hasta la raíz.

```
while node is not None:
            node.visits += 1
            # Si ganamos entonces cuenta como victoria y le sumamos 1 a
                ↪ las wins
            if simulation_result == node.player_id:
                node.wins += 1
            node = node.parent
```

3. Política de Selección UCT

El objetivo de la política de selección es mantener un equilibrio adecuado entre la **exploración** (acciones no bien probadas) y la **explotación** (mejores acciones identificadas hasta el momento). Es el algoritmo más común que se implementa en MCTS

$$UCT = \frac{Q_i}{N_i} + C\sqrt{\frac{\ln N_p}{N_i}}$$

Donde:

- Q_i: recompensa total del nodo i
- N_i: número de visitas al nodo i
- N_p: número de visitas al nodo padre
- C: constante de exploración (valor recomendado: $\sqrt{2}$ aunque termino usando 1,4)

```
def best_child(self, c_param=1.4):
        # Formula UCT
        choices_weights = [
            (child.wins / child.visits) + c_param * math.sqrt(math.log(
                ↪ self.visits) / child.visits)
            for child in self.children
        ]
        return self.children[choices_weights.index(max(choices_weights))
            ↪ ]
```

4. ¿Cómo escogemos el nodo (jugada) que vamos a usar?

Se probaron dos métodos, luego de realizar las n iteraciones se probó devolver el nodo más visitado y devolver el nodo con más victorias acumuladas, se desarrolló una simulación donde se puso ambos agentes a competir y la mayoría de las veces el ganador fue el que devolvía el nodo con más visitas.

5. Conclusión

MCTS es un algoritmo versátil que ha demostrado gran eficacia en diversos dominios, especialmente en juegos. Su capacidad para equilibrar exploración y explotación, junto con su naturaleza *anytime*, lo hacen particularmente útil en problemas con espacios de estado grandes y complejos como el caso del juego de Hex.