Inteligencia Artificial 2026

Lab 07

27.abril.2026

Búsqueda Adversaria. En este laboratorio vamos a implementar los algoritmos de búsqueda adversaria: minimax, minimax con horizonte limitado, α-β prunning, y Montecarlo Tree Search (MCTS) en juegos de suma cero, analizando el impacto de la profundidad de búsqueda y el azar estadístico.

1. **Tic-Tac-Toe**:
   Implementar una clase `TicTacToeEngine` para realizar y visualizar un juego de Tic-Tac-Toe de 3×3 ó 4×4.

   Objetivo: Desarrollar un motor de juegos que compare la eficiencia de los algoritmos deterministas (Minimax/Alfa-Beta) frente a probabilísticos (MCTS) bajo distintas configuraciones de tablero.

   Construir dos clases principales:

   - Clase `TicTacToeEngine`: Esta clase representa el "cerebro" y el estado del juego. No debe contener lógica de interfaz de usuario (inputs), solo procesamiento de datos. Métodos obligatorios a implementar:
     - `is_empty(row, col)`: Retorna si una celda está disponible.
     - `get_moves()`: Retorna lista de coordenadas (r, c) disponibles.
     - `is_winner(player)`: Verifica si el jugador ha ganado.
     - `evaluate()`: Función heurística para tableros no terminados (asigna puntajes basados en proximidad a ganar).
     - `minimax_pure()`: Implementación exhaustiva (solo recomendada para 3×3).
     - `minimax_limit(depth)`: Minimax que se detiene en un horizonte fijo y usa evaluate().
     - `alpha_beta(depth, alpha, beta)`: Minimax optimizado con poda.
     - `mcts(iterations, C)`: Monte Carlo Tree Search utilizando la fórmula UCT para selección. Fórmula UCT:

$$\text{Score} = \text{mean\_win\_rate} + C \times \sqrt{\frac{\ln(\text{ParentVisits})}{\text{NodeVisits}}}.$$

   - Clase `GameLoop`: Esta clase orquesta el flujo de la partida. Debe ser capaz de configurar una partida con los siguientes parámetros en su constructor.
     - `size`: 3 ó 4.
     - `mode`: "H-H" (Humano vs Humano), "H-IA" (Humano vs IA), "IA-IA" (IA contra IA).
     - `starting_player`: Quien realiza el primer movimiento ('H' o 'IA'). Asumiremos que El primero el humano o la IA1 siempre juegan con 'X', y el adversario con 'O'.
     - `ia_configs`: Un diccionario o estructura que defina para cada IA:
       - Algoritmo a usar (minimax, alpha_beta, mcts).
       - depth: Horizonte para algoritmos de límite (default 4 para el tablero de 4×4).
       - N: Número de simulaciones para MCTS.
       - C: Constante de exploración para UCT (default $\sqrt{2}$).

2. **Explosión Combinatoria**:
   (a) En el Tic-Tac-Toe de 3×3, realizar lo siguiente:

   - Una búsqueda minimax, desde el tablero vacío, variando el depth desde 1 a 9. Registrar el número de nodos visitados y el tiempo de ejecución.

- Repetir lo anterior, pero ahora implementanto la poda α-β dentro del minimax.

(b) En el Tic-Tac-Toe de 4×4, realizar una búsqueda desde el tablero vacío con α-β variando el depth de 1 a 6. Registrar el número de nodos visitados, el tiempo de ejecución y el factor de ramificación efectivo $\sqrt[depth]{nodos}$.

3. **Duelo de Algoritmos (IA-IA).**
   Enfrenten dos configuraciones en 20 partidas:

   - IA-1: MCTS con $N = 500$ y $C = \sqrt{2}$.
   - IA-2: Minimax limitado a depth=4 y poda α-β.

   Pregunta: ¿Cuál es más "inteligente" en términos de ganar y cuál es más "eficiente" en términos de tiempo por jugada?

4. **Para pensar.**
   Imagine que la IA no debe tardar más de 1 segundo por jugada. Si el algoritmo es lento, ¿qué tipo de implementación o modificación a la estructura anterior debe hacerse para devolver la mejor jugada encontrada considerando la restricción de tiempo? Proponga sus ideas.

Recomendaciones

- No usar nunca minimax puro en un tablero de 4×4, si no quiere que su PC se quede sin memoria. Se sugiere usar una versión minimax limitada siempre. En el de 3×3 no debería haber problemas.
- En MCTS, asegúrense de no mantener en memoria árboles que no se estén utilizando para evitar fugas de memoria en partidas largas.
- La clase **GameLoop** debe imprimir el tablero tras cada movimiento, indicando cuántos nodos exploró la IA y cuánto tiempo le tomó.