# Practica 4. Búsqueda Adversaria

La **búsqueda adversaria** es un tipo de algoritmo utilizado principalmente en inteligencia artificial (IA) para resolver problemas donde dos o más agentes (jugadores) compiten entre sí, y las acciones de un jugador afectan las decisiones del otro. Este tipo de enfoque es muy común en juegos de dos jugadores como el ajedrez, las damas o el Go, donde cada jugador tiene objetivos opuestos.

## Principales conceptos de la búsqueda adversaria:

1. **Juego de suma cero**: Los juegos en los que la ganancia de un jugador es la pérdida del otro (como el ajedrez) son llamados juegos de suma cero. En este tipo de juegos, el objetivo de un jugador es maximizar su propia ganancia minimizando la del oponente.
2. **Árbol de juego**: En la búsqueda adversaria, el estado del juego se representa mediante un árbol, donde cada nodo corresponde a un estado del juego y las aristas (conexiones entre nodos) representan las acciones o movimientos posibles.
3. **Jugador MAX y jugador MIN**: Estos dos roles representan los jugadores opuestos en un juego. El jugador **MAX** busca maximizar su ganancia, mientras que el jugador **MIN** busca minimizar la ganancia de MAX (o, lo que es lo mismo, maximizar su propia ganancia a costa de MAX). El árbol de búsqueda alterna entre nodos donde MAX mueve y nodos donde MIN mueve.
4. **Algoritmo Minimax**: Este es el algoritmo más básico para la búsqueda adversaria. Funciona calculando el valor mínimo o máximo que un jugador puede obtener en cada nodo del árbol, dependiendo de si es el turno de MIN o de MAX. El objetivo de MIN es minimizar las ganancias de MAX, y viceversa.

   > En cada turno, el algoritmo evalúa todos los posibles movimientos futuros para ambos jugadores y selecciona el mejor movimiento basándose en los valores de utilidad de los nodos hoja (es decir, los nodos que representan el fin del juego o una evaluación de una posición).

5. **Poda alfa-beta**: Es una mejora del algoritmo Minimax. La poda alfa-beta reduce el número de nodos evaluados en el árbol de búsqueda eliminando aquellas ramas que no afectan la decisión final. Esto se logra manteniendo dos límites, alfa (el valor máximo que puede garantizar MAX) y beta (el valor mínimo que puede garantizar MIN). Si un nodo no puede mejorar el valor de alfa o beta, se "poda", es decir, se descarta sin necesidad de ser evaluado.
6. **Evaluación heurística**: En juegos complejos como el ajedrez, no es posible evaluar todas las posiciones posibles hasta el final del juego debido al tamaño del árbol de búsqueda. Por lo tanto, se utilizan funciones heurísticas que proporcionan una estimación del valor de una posición intermedia.

## Aplicaciones:

La búsqueda adversaria se usa principalmente en la resolución de problemas en juegos de dos jugadores. Sin embargo, también tiene aplicaciones en la toma de decisiones en ambientes competitivos o de conflicto en el mundo real, como la negociación, la planificación estratégica y la economía.

## Desafíos:

- **Profundidad del árbol**: En juegos complejos, el número de posibles estados del juego es inmenso, por lo que la búsqueda completa de todos los estados es computacionalmente inviable.
- **Tiempo de procesamiento**: Evaluar demasiados movimientos posibles puede requerir mucho tiempo y recursos. Aquí es donde técnicas como la poda alfa-beta y las funciones de evaluación heurística juegan un papel crucial para hacer la búsqueda más eficiente.

## Ejemplo: Algoritmo Minimax en MATLAB para tres en raya

En este juego, tenemos dos jugadores: el jugador 1 (MAX, "X") y el jugador 2 (MIN, "O"). El objetivo del jugador MAX es maximizar el resultado, mientras que el jugador MIN busca minimizarlo. El algoritmo Minimax evaluará todas las posibles jugadas para ambos jugadores y determinará el mejor movimiento.

```
% Minimax algorithm for Tic-Tac-Toe (Tres en Raya)
function bestMove = minimax_example()

    % Inicializa el tablero (0 = vacío, 1 = X, -1 = O)
    board = [0 0 0; 0 0 0; 0 0 0];

    % Llama a la función minimax para encontrar el mejor movimiento
    [~, bestMove] = minimax(board, 1);

end

% Función Minimax
% board: el estado actual del tablero
% player: 1 (MAX, jugador X) o -1 (MIN, jugador O)
function [score, move] = minimax(board, player)

    % Revisa si hay un ganador o si es empate
    winner = check_winner(board);

    if winner ~= 0
        % Retorna la evaluación: 1 si X gana, -1 si O gana, 0 si es empate
        score = winner;
        move = [];
        return;
    elseif isempty(find(board == 0, 1))
        % Si no quedan movimientos posibles, es empate
```

```
        score = 0;
        move = [];
        return;
    end

    % Inicialización
    bestScore = -inf * player;
    bestMove = [];

    % Bucle para probar cada posible movimiento
    for i = 1:3
        for j = 1:3
            if board(i, j) == 0
                % Simula el movimiento
                board(i, j) = player;
                % Evalúa el tablero después de ese movimiento
                [nextScore, ~] = minimax(board, -player);
                % Deshace el movimiento
                board(i, j) = 0;

                % Actualiza el mejor movimiento
                if (player == 1 && nextScore > bestScore) || (player == -1 &&
nextScore < bestScore)
                    bestScore = nextScore;
                    bestMove = [i, j];
                end
            end
        end
    end

    % Retorna la mejor puntuación y el mejor movimiento
    score = bestScore;
    move = bestMove;
end

% Función para revisar si hay un ganador
function winner = check_winner(board)
    % Comprobación de filas, columnas y diagonales
    for i = 1:3
        if abs(sum(board(i, :))) == 3  % Revisa filas
            winner = sign(sum(board(i, :)));
            return;
        elseif abs(sum(board(:, i))) == 3  % Revisa columnas
            winner = sign(sum(board(:, i)));
            return;
        end
    end

    % Revisa diagonales
    if abs(sum(diag(board))) == 3
        winner = sign(sum(diag(board)));
        return;
    elseif abs(sum(diag(flipud(board)))) == 3
        winner = sign(sum(diag(flipud(board))));
        return;
```

```
    end

    % Si no hay ganador
    winner = 0;
end
```

## Explicación del código:

1. **`minimax_example`**: Esta función inicializa el tablero vacío y llama al algoritmo Minimax para que determine el mejor movimiento para el jugador MAX (jugador 1 o "X").
2. **`minimax`**: Esta es la función principal que implementa el algoritmo Minimax. Evalúa todos los posibles movimientos futuros y retorna el mejor movimiento y la evaluación de la posición. Aquí, el parámetro `player` indica de quién es el turno (1 para MAX, -1 para MIN).
3. **`check_winner`**: Esta función revisa si hay un ganador verificando las filas, columnas y diagonales. Si uno de los jugadores ha ganado, devuelve 1 para MAX o -1 para MIN. Si no hay ganador, devuelve 0.

## Cómo funciona el algoritmo:

- En cada turno, el algoritmo explora todos los posibles movimientos y, usando el Minimax, evalúa el mejor resultado para el jugador actual.
- MAX (jugador 1) intenta maximizar su puntaje y MIN (jugador 2) intenta minimizar el puntaje.
- El algoritmo simula los movimientos de ambos jugadores hasta que se alcanza un estado terminal (victoria, derrota o empate).

Este código puede usarse para jugar automáticamente tres en raya con el algoritmo Minimax.

## El Dilema del Prisionero

El **Dilema del Prisionero** es un problema clásico en la teoría de juegos que se utiliza para estudiar la cooperación y la competencia entre dos jugadores. Cada jugador puede elegir entre cooperar o traicionar al otro, y la decisión de cada uno influye en el resultado para ambos.

Supongamos que tenemos dos prisioneros, A y B, y ambos enfrentan la misma decisión: **cooperar** (no delatar al otro) o **traicionar** (delatar al otro). Las recompensas o castigos dependen de lo que ambos decidan:

- Si ambos **cooperan**, ambos reciben una recompensa moderada (ambos obtienen 3 años de cárcel).
- Si uno **traiciona** y el otro **coopera**, el traidor queda libre y el cooperador recibe un castigo fuerte (10 años de cárcel).
- Si ambos **traicionan**, ambos reciben un castigo menos severo (5 años de cárcel).

Esto puede representarse en una tabla de pagos, donde los valores negativos indican los años de cárcel:

| | Prisionero B Copera | Prisionero B traiciona |
|---|---|---|
| Prisionero A Copera | (-3, -3) | (-10, 0) |
| Prisionero A Traiciona | (0, -10) | (-5, -5) |

## Implementación del Dilema del Prisionero en MATLAB

El siguiente código implementa el dilema del prisionero en MATLAB, donde los jugadores pueden elegir entre cooperar o traicionar, y se calculan sus pagos según la tabla.

```
% Dilema del Prisionero en MATLAB

% Definir los pagos para las posibles acciones
% Matriz de pagos: [pago_A, pago_B]
payoffs = [
    -3, -3;   % Ambos cooperan
    0, -10;   % A traiciona, B coopera
    -10, 0;   % A coopera, B traiciona
    -5, -5    % Ambos traicionan
];

% Acciones posibles
actions = {'Cooperar', 'Traicionar'};

% Elección de los jugadores (1 = Cooperar, 2 = Traicionar)
% Puedes cambiar estas elecciones para ver distintos resultados
choice_A = 1; % Elección del prisionero A
choice_B = 2; % Elección del prisionero B

% Determinar el pago según las elecciones
if choice_A == 1 && choice_B == 1
    outcome = payoffs(1, :);  % Ambos cooperan
elseif choice_A == 2 && choice_B == 1
    outcome = payoffs(2, :);  % A traiciona, B coopera
elseif choice_A == 1 && choice_B == 2
    outcome = payoffs(3, :);  % A coopera, B traiciona
elseif choice_A == 2 && choice_B == 2
    outcome = payoffs(4, :);  % Ambos traicionan
end

% Mostrar los resultados
disp(['Prisionero A elige: ', actions{choice_A}]);
disp(['Prisionero B elige: ', actions{choice_B}]);
disp(['Resultado (Años de cárcel para A, Años de cárcel para B): ',
num2str(outcome)]);
```

## Explicación del código:

1. **`payoffs`**: La matriz `payoffs` contiene las recompensas para cada combinación de elecciones. Cada fila de la matriz representa un resultado diferente:
   - Fila 1: Ambos cooperan (-3, -3).
   - Fila 2: A traiciona, B coopera (0, -10).
   - Fila 3: A coopera, B traiciona (-10, 0).
   - Fila 4: Ambos traicionan (-5, -5).
2. **`actions`**: Las posibles acciones que pueden tomar los prisioneros: cooperar o traicionar.
3. **`choice_A` y `choice_B`**: Estas variables representan las decisiones de los prisioneros A y B, donde 1 significa cooperar y 2 significa traicionar. Puedes cambiar estos valores para simular diferentes elecciones.
4. **Determinación del resultado**: El código evalúa las elecciones de los jugadores y selecciona la fila adecuada de la matriz `payoffs` para determinar los años de cárcel correspondientes a ambos jugadores.

## Ejemplo de ejecución:

Si el prisionero A coopera (`choice_A = 1`) y el prisionero B traiciona (`choice_B = 2`), el programa devolverá:

```
Prisionero A elige: Cooperar
Prisionero B elige: Traicionar
Resultado (Años de cárcel para A, Años de cárcel para B): -10  0
```

Esto significa que el prisionero A coopera y recibe 10 años de cárcel, mientras que el prisionero B traiciona y queda libre (0 años de cárcel).

## Experimentación:

Puedes modificar las decisiones de los prisioneros (`choice_A` y `choice_B`) para ver cómo cambian los resultados, así como cambiar los valores de la matriz `payoffs` para representar diferentes versiones del dilema.