# Práctica 1 Sistemas Inteligentes

## Algoritmo Minimax con poda α-β para el juego Conecta-4

### Introducción

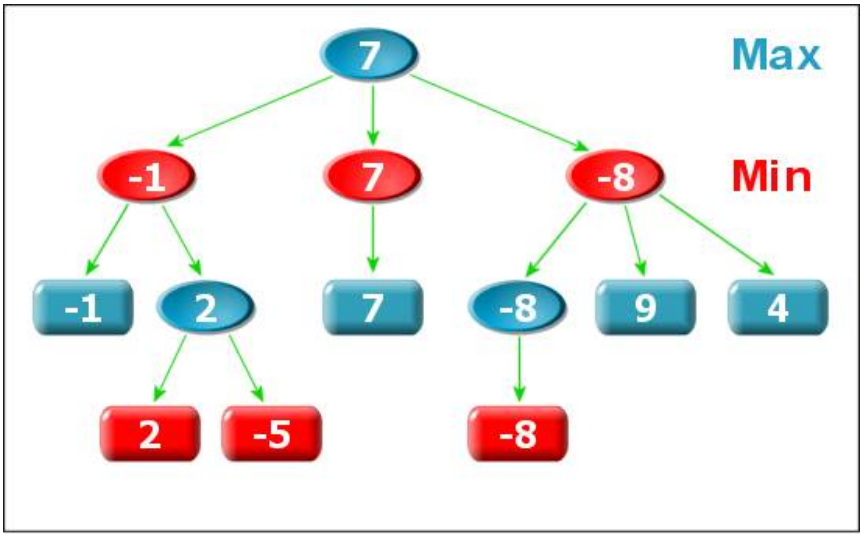
Se nos pide implementar el famoso algoritmo de búsqueda de juegos **Minimax** y una **función de evaluación** para el juego **Conecta-4**, también conocido como 4 en raya. Opcionalmente, se puede añadir una variante que incluya la **poda α-β** para optimizar el rendimiento.

Explicaremos lo que se ha realizado y cómo se ha implementado, analizando cada parte en la medida de lo posible.

### Algoritmo Minimax

Su funcionamiento es, por definición, bastante sencillo: **partiendo de un estado del juego inicial, simularemos posibles jugadas** que puedan realizar tanto el propio jugador como su contrincante, **evaluaremos cada una de estas** posibilidades **y seleccionaremos aquella que nos es más favorable** suponiendo que el otro hará también su mejor movimiento.

La implementación se basa en una estructura de **árbol** que genera todas las posibilidades, el cual recorreremos **en profundidad** hasta cierta altura (que tendremos que tener predefinida) para, **una vez llegado a un nodo terminal, establecerle un valor** numérico mediante una función que evalúa el estado del juego. **El padre** de los nodos terminales **seleccionará el más conveniente** según el nivel en el que se encuentre (los pares se corresponden con el nivel del jugador, MAX; los impares, con el del contrincante, MIN) y así **hasta que el nodo raíz disponga del mejor valor posible**.




En pseudo-código, el algoritmo es el siguiente:

```
algoritmo minimax. V(N)
Entrada: nodo N
Salida: valor de dicho nodo
Si N es nodo terminal entonces devolver f(N) fsi
sino
  generar todos los sucesores de N: N1, N2, …, Nb
  Si N es MAX entonces devolver max(V(N1), V(N2), …, V(Nb)) fsi
  Si N es MIN entonces devolver min(V(N1), V(N2), …, V(Nb)) fsi
fsi
falgoritmo
```

Y ésta es la implementación realizada en Java:

```java
private int V(Tablero tablero, int jugador, int nivel) {
    //Casos base:
    if (tablero.cuatroEnRaya() == MAX) return Integer.MAX_VALUE;
    if (tablero.cuatroEnRaya() == MIN) return Integer.MIN_VALUE;
```

```
    if (tablero.tableroLleno()) return 0;
    if (nivel == NIVEL_DEFECTO) return f(tablero);

    //Caso general:
    int extremo = jugador == MAX ? Integer.MIN_VALUE : Integer.MAX_VALUE;
    for (int i = 0; i < tablero.numColumnas(); i++) {
        Tablero sucesor = new Tablero(tablero);
        if (sucesor.ponerFicha(i, jugador) == 0)
            if (jugador == MAX) extremo = Math.max(extremo, V(sucesor, MIN, nivel + 1));
            else extremo = Math.min(extremo, V(sucesor, MAX, nivel + 1));
    }
    return extremo;
}
```

El funcionamiento del código es el siguiente:

1. Primero, tenemos los **casos base**:
   - Si **he ganado**, no genero más jugadas y el nodo se queda con la **máxima puntuación**.
   - Si **he perdido**, igual que si ganara pero ahora la **puntuación** es la **mínima**.
   - Si hay un **empate**, no hay ganador y **la puntuación es 0**
   - Si estoy en un **nodo terminal**, no genero más hijos y **evalúo la jugada**.
2. Después, para el **caso recursivo, genero todas las jugadas posibles** mediante un bucle que, con una copia del tablero inicial, **inserto una ficha para cada columna y llamo de nuevo a la función quedándome siempre con el valor mínimo o máximo** de cada uno de los nodos generados dependiendo del nivel en el que estemos.

Hay que tener en cuenta que ésta es la función recursiva; sin embargo, hay que tener aislado este algoritmo para la raíz, el cual está implementado en la función `minimax()`, que es con la que seleccionamos realmente la mejor columna para hacer nuestra jugada:

```
public void minimax() {
  int max = Integer.MIN_VALUE, max_i = -1;
  for (int i = 0; i < m_tablero.numColumnas(); i++) {
      Tablero sucesor = new Tablero(m_tablero);
      if(sucesor.ponerFicha(i, MAX) == 0) {
          int actual = V(sucesor, MIN, 1);
          if (actual > max) {
              max = actual;
              max_i = i;
          }
      }
  }
  ...
}
```

## Función de evaluación

Para que Minimax funcione, **es necesario disponer de una función que nos indique**, mediante una puntuación, **qué tan bueno es el estado actual del juego** para el jugador. A partir de ahora, llamaremos a esta función **heurística**

Para este juego, hay muchas características que podemos tener en cuenta para determinar la favorabilidad de una jugada; algunas mejores y otras no tanto. Nosotros hemos optado por una cuyo fundamento es **encontrar casilla por casilla potenciales cuatros en raya** y asignarles una puntuación positiva o negativa dependiendo de para quién sea la jugada.

### Primera propuesta

Un primer acercamiento para plantear esta propuesta fue el de **asignar a cada casilla una puntuación que indicase las distintas combinaciones de 4 en raya que tiene cada una de ellas**. La matriz de puntuaciones era la siguiente:

```
private static int[][] evaluationTable = {
  { 3,  4,  5,  7,  5,  4,  3},
  { 4,  6,  8, 10,  8,  6,  4},
  { 5,  8, 11, 13, 11,  8,  5},
  { 5,  8, 11, 13, 11,  8,  5},
  { 4,  6,  8, 10,  8,  6,  4},
  { 3,  4,  5,  7,  5,  4,  3}
};
```

Entonces recorríamos el tablero y, **dependiendo de si la casilla era del jugador MIN o MAX, sumábamos o restábamos la puntuación correspondiente al acumulador**, el cual se sumaba finalmente a un valor constante de 138 (porque era la mitad de la puntuación máxima, 276 = 2 * 138; así, en caso de empate el acumulador daría 0):

```
private int f(Tablero t) {
int sum = 0;
for (int i = 0; i < t.numFilas(); i++)
  for (int j = 0; j < t.numColumnas; j++)
    if (t.obtenerCasilla(i, j)  == MAX) sum += evaluationTable[i][j];
    else if (t.obtenerCasilla(i, j)  == MIN) sum -= evaluationTable[i][j];
return 138 + sum;
}
```

Sin embargo, tras realizar un par de pruebas, se podía observar cómo **el único objetivo de la máquina era el de insertar las fichas en medio del tablero** (donde estaba la máxima puntuación), **pero no trataba de bloquearte los 4 en raya potenciales**; no tenía en cuenta el verdadero estado del tablero y, por tanto, **podías ganar en las primeras jugadas**. Sería necesario plantear una estrategia en la que las fichas que bloqueen a potenciales 4 en raya no valieran como buenas jugadas.

**Propuesta mejorada**

En este nuevo planteamiento, en vez de asignarle a cada posición un valor predeterminado, **buscaría en sus casillas vecinas posibles combinaciones de 4 en raya que sí se pudieran realizar** (esto es, que no quedasen bloqueadas), **y asignaría una puntuación dependiendo de la cantidad de fichas del jugador consecutivas que encontrase**.

```
for (int i = 0; i < t.numFilas(); i++) {
  for (int j = 0; j < t.numColumnas(); j++) {
    int puntos = 0;
    jugador = t.obtenerCasilla(i, j);
    if (jugador == 0) jugador = MAX; //Si la casilla está vacía, simulo ser el jugador MAX
    //3 en raya potencial
    puntos += 3 * comprobarFichas(i, j,  0, jugador, jugador, t);
    puntos += 3 * comprobarFichas(i, j, jugador,  0, jugador, t);
    puntos += 3 * comprobarFichas(i, j, jugador, jugador,  0, t);
    //2 en raya potencial
    puntos += 2 * comprobarFichas(i, j, jugador,  0, 0, t);
    puntos += 2 * comprobarFichas(i, j,  0, jugador, 0, t);
    puntos += 2 * comprobarFichas(i, j,  0, 0, jugador, t);
    //1 en raya potencial -> Sólo si no es casilla vacía
    if (t.obtenerCasilla(i, j)  != 0) puntos += 1 * comprobarFichas(i, j,  0, 0, 0, t);
    if(t.obtenerCasilla(i, j)  == 0) puntos /= 2; //Si la casilla está vacía, la puntuación es la mitad
    if(jugador == MIN) puntos = -puntos; //Si el jugador es el oponente; la puntuación es negativa
    total += puntos;
  }
}
```

El funcionamiento a simple vista es simple: **para cada ficha del tablero, compruebo que las vecinas a ésta sean las que especifico** en la función `comprobarFichas` (hablaremos de ella a continuación), **tratando de encontrar todas las combinaciones posibles** de 3 en raya junto a 1 casilla vacía, 2 en raya junto a 2 casillas vacías, y 1 en raya junto a 3 casillas vacías. **Después, dependiendo del tipo de ficha** que haya en la posición que estoy evaluando, el comportamiento cambia ligeramente:

- Si la ficha es **MAX**, el **funcionamiento** es el **normal**.
- Si la ficha es **MIN**, la **puntuación** será **negativa**.
- Si la **casilla** está **vacía**, **simulamos** que buscamos posibles buenas jugadas para **el jugador MAX** (no tenemos en cuenta a MIN para este caso, cosa que sería interesante) y finalmente **la puntuación** acumulada al total **será la mitad** de la obtenida. La posibilidad de 1 en raya no cuenta para este caso.

La función `comprobarFichas` **comprueba que las fichas que recibe** por argumentos **aparezcan en todas las direcciones**: horizontal, vertical, diagonal y diagonal inversa **y devuelve un recuento de cuántas han coincidido**. Luego, `vertical`, `horizontal`, `diagonal` y `diagonalInv` se aseguran de que existan al menos tres fichas en la dirección correspondiente para así compararlas con las que recibe mediante `comprobarVecinos`:

```
private static int comprobarFichas( int i, int j, int uno, int dos, int tres, Tablero t) {
  int posibilidades = 0;
  if (vertical(i, j, uno, dos, tres, t)) posibilidades ++;
```

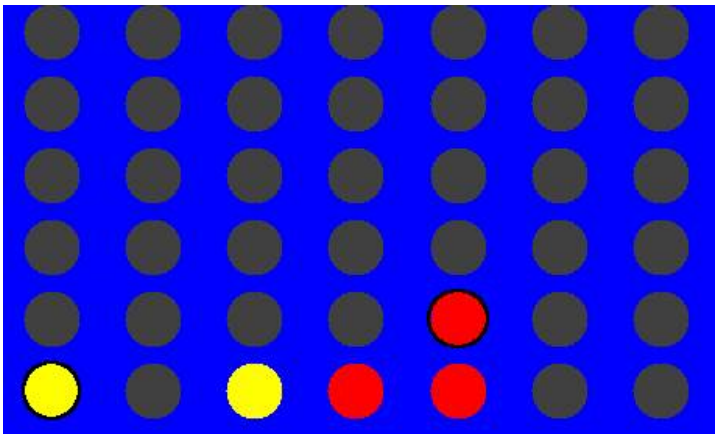
```
  if (horizontal(i, j, uno, dos, tres, t)) posibilidades ++;
  if (diagonal(i, j, uno, dos, tres, t)) posibilidades ++;
  if (diagonalInv(i, j, uno, dos, tres, t)) posibilidades ++;
  return posibilidades;
}

private static boolean vertical(int i, int j, int segunda, int tercera, int cuarta, Tablero t) {
  return i+3 < t.numFilas() ? comprobarVecinos(t.obtenerCasilla(i+1, j), t.obtenerCasilla(i+2, j), t.obtenerC
}

private static boolean izquierda(int i, int j, int segunda, int tercera, int cuarta, Tablero t) {
  return j-3 >= 0 ? comprobarVecinos(t.obtenerCasilla(i, j-1), t.obtenerCasilla(i, j-2), t.obtenerCasilla(i,
}

private static boolean horizontal(int i, int j, int segunda, int tercera, int cuarta, Tablero t) {
  return j+3 < t.numColumnas() ? comprobarVecinos(t.obtenerCasilla(i, j+1), t.obtenerCasilla(i, j+2), t.obten
}

private static boolean diagonal(int i, int j, int segunda, int tercera, int cuarta, Tablero t) {
  return i+3 < t.numFilas() && j+3 < t.numColumnas() ? comprobarVecinos(t.obtenerCasilla(i+1, j+1), t.obtener
}

private static boolean diagonalInv(int i, int j, int segunda, int tercera, int cuarta, Tablero t) {
  return i-3 >= 0 && j+3 < t.numColumnas() ? comprobarVecinos(t.obtenerCasilla(i-1, j+1), t.obtenerCasilla(i-
}

private static boolean comprobarVecinos(int v1, int v2, int v3, int c1, int c2, int c3) {
  return v1 == c1 && v2 == c2 && v3 == c3;
}
```

**No realizamos las comprobaciones opuestas porque al avanzar en el tablero las contaríamos por doble**: hacer un cuatro en raya de izquierda a derecha es igual que hacerlo de derecha a izquierda; así como de arriba a abajo y de abajo a arriba (aunque no lo parezca, las diagonales no son una excepción).

**Pruebas de funcionamiento**

Para comprobar el funcionamiento de la propuesta, aparte de probarlo de forma empírica, **comprobaremos cómo asigna las puntuaciones a un tablero** con un sencillo caso. Para ello, justo antes de terminar la función de evalución, imprimiremos el tablero que ha analizado y la puntuación que ha obtenido para comprobar si los cálculos realizados los hace correctamente.

Partiremos del siguiente tablero, en el que hemos supuesto que **el jugador MIN ha puesto una ficha en la columna 4 y MAX en la 0**:



Las **posibles jugadas** son las siguientes:

- Para **MAX**:
  i. En la **posición (0,0)** tiene un 1 en raya vertical y otro en diagonal hacia arriba: **2 puntos**.
  ii. En la **posición (0,2)** tiene otra vez un 1 en raya vertical y otro en diagonal hacia arriba: **2 puntos**.
- Para **MIN**:
  i. En la **posición (0,3)** tiene un 1 en raya vertical, un 2 en raya horizontal y un 2 en raya en diagonal hacia arriba: **-5 puntos**.
  ii. En la **posición (0,4)** tiene un 2 en raya vertical: **-2 puntos**.
  iii. En la **posición (1,4)** tiene un 1 en raya vertical: **-1 punto**.

Si le pasamos una traza a la función de evaluación, el resultado que mostrará es el siguiente:

```
2 0 2 1 1 0 0
0 0 0 0 1 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
ha obtenido -4 puntos
```

Podemos comprobar que **no es un caso muy favorable**. Seguramente hayan otros estados del juego que nos proporcionen mejores puntuaciones (de hecho, ese movimiento no es el escoge para esta jugada concreta).

**Reglas de libro: refinamineto de la función de evaluación**

**Hay un caso en el que nuestra propuesta no funciona** del todo bien: si la profundidad del árbol es 2, el jugador MIN a veces puede hacer un 4 en raya en la primera fila. Esto no pasa si aumentamos el nivel de profundidad y tenemos en cuenta más posibles jugadas, pero deberíamos de conseguir que incluso sin preveer tanto el futuro nuestra función fuese eficaz. Para ello, **introducimos un pequeño caso particular para éstas primera jugadas** el cual funciona bastante bien:

```
Si el tablero tiene más de tres fichas:
  caso general
Sino
  Si MIN ha puesto una ficha en medio
    Si yo coloco la ficha en una posición contigua a la del centro -> 2 puntos Fin Si
  Si no
    Si yo coloco la ficha en medio -> 3 puntos Fin Si
  Fin Si
Fin Si
```

La implementación de esta idea sería la siguiente:

```
if (numFichasMayorQue(t,  3)) {
  ... //funcionamiento normal de f
} else {
  if(t.obtenerCasilla( 0, 3) == MIN) {
    if (t.obtenerCasilla( 0, 2) == MAX || t.obtenerCasilla( 0, 4) == MAX) total = 2;
  } else if (t.obtenerCasilla( 0, 3) == MAX) total = 3;
}
```

La función `numFichasMayorQue` devuelve true si en el tablero t hay más de n fichas (en este caso, 3):

```
private boolean numFichasMayorQue( Tablero t, int n) {
  int count = 0;
  for (int i = 0; i < t.numFilas(); i++)
    for (int j = 0; j < t.numColumnas(); j++)
      if (t.obtenerCasilla(i, j)  != 0 && ++count > n) return true;
  return false;
}
```

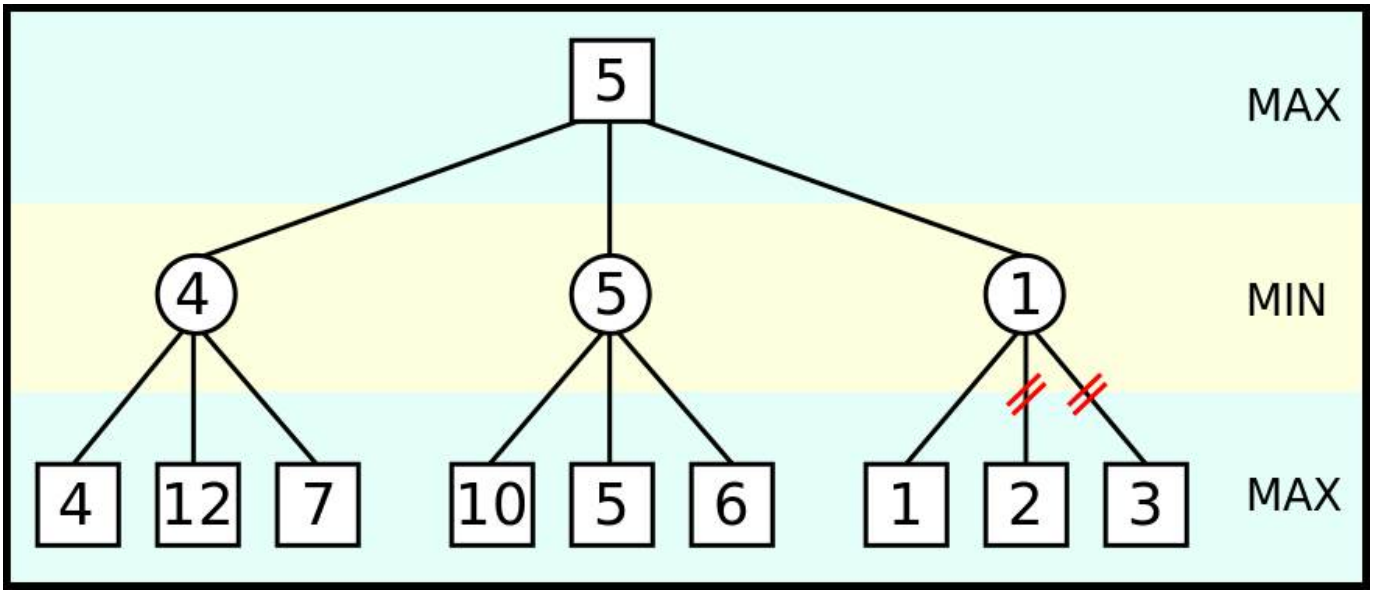
Es preciso recordar que **esta excepción no se utiliza para niveles de profundidad del árbol de nivel superior 2**; en esos casos, el algoritmo puede valerse por sí mismo sin problemas.

## Poda α-β

**La complejidad del algoritmo minimax es exponencial** y, por tanto, **hay que intentar reducir el número de nodos** lo máximo posible. Para conseguirlo, Introducimos esta poda.

**α y β se corresponden con el mejor y el peor valor hasta el momento** a lo largo del recorrido que realizamos en el árbol de las posibles jugadas. **Se debe de cumplir en todo momento que α > β**; **en caso contrario, habremos encontrado el valor**

**extremo** (mínimo o máximo dependiendo del nivel) de ese nodo **y no hará falta generar más hijos** (podamos el árbol).




El algoritmo en pseudo-código **tan sólo trae un par de cambios respecto al del de Minimax**:

```
algoritmo alpha-beta. V(N, alpha, beta)
Entrada: nodo N, valores alpha y beta
Salida: valor de dicho nodo
Si N es nodo terminal entonces devolver f(N) fsi
sino
  Si N es MAX entonces
    Para k = 1 hasta b hacer
      alpha = max[alpha, V(N_k, alpha, beta)]
      Si alpha >= beta entonces devolver beta fsi
      Si k = b entonces devolver alpha fsi
    FPara
  Sino //MIN
    Para k = 1 hasta b hacer
      beta = min[beta, V(N_k, alpha, beta)]
      si alpha >= beta entonces devolver alpha fsi
      Si k = b entonces devolver beta fsi
    fpara
  fsi
fsi
falgoritmo
```

Esta vez es más sencillo verlo implementado. Tan sólo **comprobamos que el extremo (ahora también alpha/beta) cumple siempre la condición; si no lo hace, devolver el otro valor**.

```
private int alphaBetaV( Tablero tablero, int jugador, int nivel, int alpha, int beta) {
  //Casos base:
  if (tablero.cuatroEnRaya() == MAX) return Integer.MAX_VALUE;
  if (tablero.cuatroEnRaya() == MIN) return Integer.MIN_VALUE;
  if (tablero.tableroLleno()) return 0;
  if (nivel == NIVEL_DEFECTO ) return f(tablero);

  //Caso general:
  int extremo = jugador == MAX ? Integer.MIN_VALUE : Integer.MAX_VALUE;
  for(int i = 0; i < tablero.numColumnas(); i ++) {
    Tablero sucesor = new Tablero(tablero);
    if(sucesor.ponerFicha(i, jugador) == 0) {
      if(jugador == MAX) {
        alpha = Math.max(alpha, alphaBetaV(sucesor,  MIN, nivel + 1, alpha, beta));
        if(alpha >= beta) return beta;
        extremo = alpha;
      } else { //MIN
        beta = Math.min(beta, alphaBetaV(sucesor,  MAX, nivel + 1, alpha, beta));
        if(alpha >= beta) return alpha;
        extremo = beta;
      }
    }
  }
```

```
  return extremo;
}
```

Para poder seleccionar si queremos jugar con poda o sin ella, hemos añadido un Toggle a la interfaz al que le hemos asignado un evento de activación/desactivación:

```
private javax.swing.JToggleButton jToggleButton1;
public static boolean ALPHABETA_TOGGLE = false;

private void jToggleButton1ActionPerformed ( java.awt.event.ActionEvent evt) {
  ALPHABETA_TOGGLE = jToggleButton1.isSelected();
  System.out.println("Alpha-Beta -> " + ALPHABETA_TOGGLE );
}
```

Y luego desde `minimax()` comprobamos el valor de la variable para llamar a una función u otra:

```
...
int actual = Interfaz.ALPHABETA_TOGGLE ? alphaBetaV(sucesor, MIN, 1, Integer.MIN_VALUE, Integer.MAX_VALUE) :
...
```

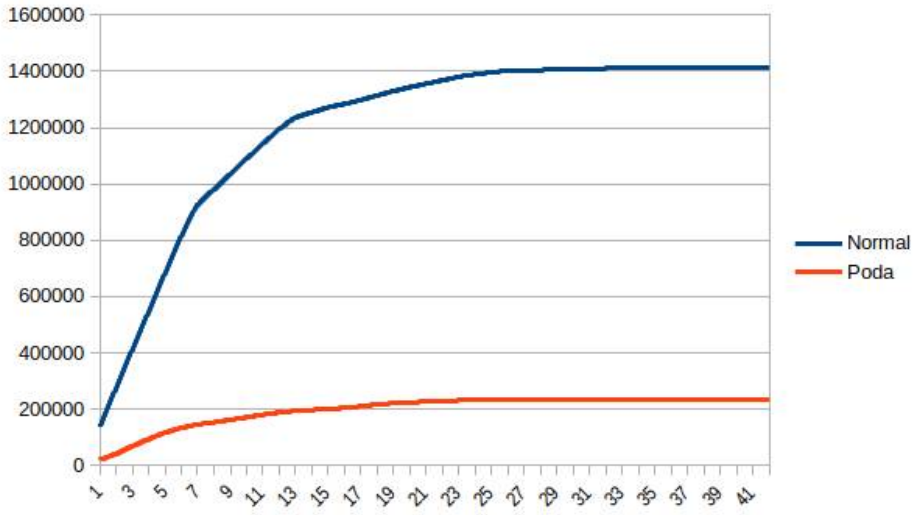
**Prueba de rendimiento**

Para comprobar la potencia de esta poda, tendremos un contador de llamadas que se incrementará cada vez que entre en la función `V`. Al finalizar cada jugada mostraremos el total de estas llamadas. Para que se note la diferencia, fijaremos el nivel de profundidad del árbol en 6.

Los resultados que hemos obtenido son los siguientes:

| Normal | Poda | Diferencia |
|---|---|---|
| 137.256 | 20.567 | 116.689 |
| 274.511 | 41.981 | 232.530 |
| 411.513 | 68.306 | 343.207 |
| 546.469 | 94.276 | 452.193 |
| 682.647 | 116.943 | 565.704 |
| 814.337 | 133.278 | 681.059 |
| 924.208 | 146.054 | 778.154 |
| 980.043 | 153.565 | 826.478 |
| 1.035.703 | 162.817 | 872.886 |
| 1.090.078 | 171.517 | 918.561 |
| 1.143.365 | 181.110 | 962.255 |
| 1.194.731 | 188.429 | 1.006.302 |
| 1.235.409 | 194.113 | 1.041.296 |
| 1.254.128 | 197.704 | 1.056.424 |
| 1.271.996 | 201.759 | 1.070.237 |
| 1.284.291 | 204.647 | 1.079.644 |
| 1.298.417 | 210.556 | 1.087.861 |
| 1.314.735 | 216.968 | 1.097.767 |
| 1.330.119 | 221.543 | 1.108.576 |
| 1.343.694 | 224.487 | 1.119.207 |

| Normal | Poda | Diferencia |
|---|---|---|
| 1.356.192 | 227.058 | 1.129.134 |
| 1.368.246 | 228.906 | 1.139.340 |
| 1.380.431 | 231.779 | 1.148.652 |
| 1.389.334 | 233.783 | 1.155.551 |
| 1.396.686 | 234.797 | 1.161.889 |
| 1.401.748 | 235.697 | 1.166.051 |
| 1.403.399 | 235.812 | 1.167.587 |
| 1.404.778 | 236.077 | 1.168.701 |
| 1.406.375 | 236.117 | 1.170.258 |
| 1.407.600 | 236.224 | 1.171.376 |
| 1.408.652 | 236.290 | 1.172.362 |
| 1.410.536 | 236.389 | 1.174.147 |
| 1.411.892 | 236.489 | 1.175.403 |
| 1.412.210 | 236.554 | 1.175.656 |
| 1.412.398 | 236.570 | 1.175.828 |
| 1.412.489 | 236.591 | 1.175.898 |
| 1.412.511 | 236.608 | 1.175.903 |
| 1.412.527 | 236.615 | 1.175.912 |
| 1.412.534 | 236.617 | 1.175.917 |
| 1.412.540 | 236.620 | 1.175.920 |
| 1.412.542 | 236.621 | 1.175.921 |
| 1.412.543 | 236.622 | 1.175.921 |




(El eje X son las llamadas y el eje Y el número de jugada).

En estas dos partidas se han eliminado 42.100.357 nodos; un promedio de 1.002.389 (83'49%) por jugada.

## Conclusiones

El algoritmo **Minimax es una herramienta muy potente** para los juegos de decisión de jugada. **Sin embargo, debido a su complejidad computacional, a veces resulta impracticable** para aquellos en los que existen muchas jugadas posibles, o incluso cuando se establece un nivel de profundidad de árbol elevado.

Para que éste funcione correctamente, debemos de establecer una buena **función de evaluación** para otorgarle a cada posible jugada una puntuación **que represente *de la manera más realista posible* el estado** que tiene. También es cierto que, incluso **usando solamente los casos base** (ganas, pierdes, empatas), la máquina **ya es bastante "inteligente"**. Después de implementar la segunda propuesta en la que se buscan aquellos potenciales 4 en raya, es prácticamente imbatible, sin contar aquel caso inicial, pero **con el añadido de las reglas de libro todavía no he conseguido ganarlo** (ni creo que pueda).

Después, gracias a la poda de los nodos, el tiempo de respuesta se ve drásticamente reducido, y eso que tan sólo son un par de líneas de código extra. El algoritmo se vuelve muy eficiente con muy poco esfuerzo.

## Referencias


- Generación de árbol minimax:
  i. Apuntes de clase, explicaciones y ayuda del profesor en prácticas
  ii. Wikipedia
  iii. Conecta-4 en Ruby
- Función de evaluación:
  i. Primera propuesta de función de evaluación
  ii. Cómo diseñar una buena función de evalución para el 4 en raya
  iii. Discusión sobre función de evaluación para una jugada perfecta
- Reglas de libro: apuntes de clase, explicaciones y ayuda del profesor en prácticas
- Poda Alpha-Beta:
  i. Apuntes de clase
  ii. Wikipedia



**Pavel Razgovorov (pr18@alu.ua.es)**