

Inteligencia Artificial

Juegos

Guión 4 - 2019/2020

Alejandro Arroyo Loaisa

Manuel Germán Morales

Grupo 1 de Prácticas

Profesor: Cristóbal J. Carmona

En esta última práctica se pide de la implementación del algoritmo MINIMAX aplicado al juego Conecta 4. En este documento describiremos paso a paso como hemos ido programando tal algoritmo. Hemos dividido el informe en 3 partes, una por cada objetivo.

1. El algoritmo MINIMAX. Construyendo el árbol.

El algoritmo MINIMAX nos permite dotar a la máquina de una estrategia para que pueda jugar de manera racional contra un humano (en este caso, un juego de 2 jugadores). Para aplicar este algoritmo se necesita generar un árbol de juego, formado por nodos. Cada nivel corresponderá a un jugador al que llamaremos MIN o MAX dependiendo en cual nivel nos encontremos. Los nodos terminales de este árbol poseerán un valor de utilidad, que será propagado hacia arriba en función del jugador que lo genere, ya que si el nivel pertenece al jugador MAX le interesará quedarse con la máxima utilidad de sus hijos y viceversa para el jugador MIN. En nuestro algoritmo, MAX será la máquina (PLAYER2) y MIN el jugador humano (PLAYER1).

La generación del árbol es recursiva siguiendo el esquema del recorrido preorden. Primero se generará el padre y después sus hijos de izquierda a derecha.

```
función construyeMiniMax(Nodo: padre, Entero: nivel)
  variables locales: Entero: jugador, Entero: columna, Nodo: hijo
  Si nivel es par entonces
    jugador ← PLAYER2
  si no
    jugador ← PLAYER1
  fin si

  columna ← 0
  Desde columna hasta MAX_HIJOS
    si puedoColocar(columna) entonces
      hijo ← crearNodoHijo(columna)
      si esTerminal(hijo) entonces
        calculaUtilidad(hijo)
      si no
        construyeMiniMax(hijo, nivel + 1)
      fin si
      AddHijo(padre, columna, hijo)
      si jugador = PLAYER2 entonces
        si Utilidad(padre) < Utilidad(hijo) ActualizaUtilidad(padre, Utilidad(hijo))
      si no
        si Utilidad(padre) > Utilidad(hijo) ActualizaUtilidad(padre, Utilidad(hijo))
      fin si
    fin si
  fin mientras
fin función
```

1.1 Representando los estados del árbol.

Teniendo una idea sobre la construcción del árbol, describimos la representación de lo que sería un nodo en nuestro árbol. Hemos creado una clase llamada **Jugada** para esto. Podemos destacar los siguientes atributos y métodos de esta clase:

- **ArrayList<Jugada>**: Lista usada para guardar los hijos que se generan a partir de un nodo.
- **int utilidad**: Entero que representa el valor de utilidad de un nodo. Si el nodo es hoja, su valor será el dado por la función de utilidad. Si no, se calcula como se muestra arriba.
- **int columna**: Guarda la columna en la que se ha insertado la ficha que ha generado este nodo.
- **void addHijo(Jugada j)**: Método para insertar un hijo. Si el número de hijos ya es máximo, no inserta nada para evitar la excepción *IndexOutOfBounds*.

1.2 Construyendo el árbol. Métodos y atributos útiles.

Con esta primera clase **Jugada**, podemos construir nuestro árbol perfectamente. Podríamos haber guardado la matriz de enteros que representa el tablero pero teníamos un problema: El consumo de memoria era muy grande impidiendo este la generación completa del árbol (a no ser que tuvieses mucha RAM, claro). El ahorro en memoria conlleva la implementación de otros métodos y la adicción a la clase **IAPlayer** de algunos atributos esenciales para su correcto funcionamiento:

- **Stack<Integer> historialMovimientos**: Pila que permite guardar un historial en qué columnas se ha ido poniendo en cada jugada. Cada vez que podamos construir un hijo añadimos a la pila la columna en la que echa ficha para luego deshacer la jugada, gracias a esquema LIFO. Cuando se vuelva al padre de la jugada generada, se deshace su movimiento.
- **int[][] estado**: Matriz usada para representar los diferentes estados del árbol, simula el tablero.
- **int[][] anterior**: Matriz que guarda cómo estaba el tablero la jugada anterior al tablero representado por estado. Sirve para hallar cuál ha sido la última columna jugada.
- **int filas**: Número de filas que posee el tablero.
- **int columnas**: Número de columnas que posee el tablero.

Además, necesitamos los siguientes métodos para realizar una correcta construcción del árbol:

- **int colocar(int[][] estado, int col, int jugador):** Coloca en la matriz estado una ficha del jugador jugador en la columna col. Devuelve en qué fila se ha colocado la ficha.
- **void deshacer():** Deshace la última jugada que se ha hecho de la matriz estado.
- Los métodos **int checkWinMatriz(...)** y **boolean fullColumn(...)** son adaptaciones de los métodos de la clase Grid para que solo use la matriz estado.

Con esto, ya podemos generar perfectamente nuestro árbol. En la clase `IAPlayer` tendremos un atributo que referencia al nodo raíz del árbol.

- **Jugada padre:** Atributo que referencia al nodo raíz del árbol.

1.3 Moviéndose por el árbol. Propagación de las utilidades.

La propagación de la utilidad de nodos hoja a padre se hace también en el método `construyeMiniMax(...)` como se puede ver en el pseudocódigo de arriba. En esta etapa se han implementado los métodos para moverse por el árbol:

- **Jugada seleccionaTirada():** Devuelve la jugada que almacena la columna donde debe de colocar `IAPlayer`. Se selecciona de los nodos hijo pertenecientes al padre, aquel que tenga la misma utilidad que este. Si no se encuentra (o no tiene hijos) se devuelve `null`.
- **int getUltimaColumna():** Devuelve la columna donde se ha colocado la ficha más recientemente colocada.
- **Jugada buscaSiguientePadre(int columna):** Devuelve, sabiendo la última columna donde se ha jugado, el nodo correspondiente a esta última jugada. Usada para cambiar el padre cuando le toca jugar a `IAPlayer` después del jugador Humano. No podemos obtener directamente el padre, porque puede ser que `padre.hijos.get(columna)` nos de una jugada cuya columna no se corresponda con la última usada (ya que las jugadas se guardan en la lista contiguamente, puede ser que la jugada con el índice 0 en la lista, sea de la columna 2 por ejemplo).

Excepto la primera vez (que se construye el árbol), en cada ejecución del método `turnoJugada(...)` se buscará el siguiente nodo raíz correspondiente al nuevo estado del tablero (es decir, el nodo correspondiente a la jugada del humano), y se volverá al cambiar el nodo raíz después de que `IAPlayer` coloque ficha para asegurar la integridad del árbol y que en todo momento se mantenga coherente con el desarrollo de la partida (es decir, que el árbol no vaya niveles atrasados, ni adelantados).

2. Restringiendo el árbol. Combatiendo el efecto horizonte con una función heurística.

Evidentemente, realizar el cálculo del 100% del árbol de juego para saber exactamente cuál es la mejor solución, de entre absolutamente todas las jugadas posibles, es una tarea ineficiente tanto en tiempo como en memoria.

La primera solución en la que podemos pensar, es en limitar estos cálculos a cierto nivel del árbol, y tomar la decisión a partir de la situación que encontremos en dicho nivel. En otras palabras, tendremos una aproximación bastante acertada (mediante el cálculo de la heurística) de la mejor solución sin necesidad de calcular el árbol completo. La heurística se implementa para contrarrestar lo máximo posible el efecto horizonte que se describe en la parte teórica de la asignatura, ocasionado por la restricción de nivel aplicada al árbol.

2.1 Sobre la restricción de nivel.

Para poder restringir bien nuestro árbol necesitamos añadir a la clase `IAPlayer` los siguientes atributos:

- `static int MAX_LV`: Número máximo de niveles que generará el árbol.
- `int supnivel`: Nivel a partir del cual restringimos el árbol.
- `int infnivel`: Nivel del padre del árbol.

Cuando se de la siguiente condición en la construcción (método `construyeMiniMax(...)`) de nuestro árbol `(nivel+1) >= supnivel` (o que el nodo sea ya un estado terminal de por sí) se dejarán de generar hijos y el nodo pasará a ser un nodo terminal para nuestro árbol, por lo tanto calcularemos su valor heurístico y se lo asignaremos.

En el método `turnoJugada(...)`, si a medida que vayamos jugando se da la condición `infnivel == supnivel` tendremos que generar el árbol desde el nodo padre en el que nos encontremos, ya que no tendremos hijos. La variable `supnivel` se incrementará `MAX_LV` niveles para así marcar en qué nivel debemos de volver a generar el árbol hasta finalizar la partida.

2.2 Sobre la función heurística.

Esta “heurística” no es otra cosa que una función que nos ayuda a darle un valor a las jugadas, teniendo en cuenta las fichas de ambos jugadores y sus posiciones, para saber exactamente cuál es la mejor solución en ese punto de la partida a ese nivel del árbol. Se puede calcular de diversas maneras, pero nosotros hemos optado por la que se explica a continuación.

La heurística la hemos descompuesto en cuatro funciones, que reciben como parámetro *conecta*, entero que representa cuántas fichas necesitan ser alineadas para ganar:

```
int heuristicaUtilidad(int conecta) {  
    //Entero dónde se acumulará el valor heurístico de cada función  
    int cont = 0;  
  
    //Función heurística que evalúa verticales  
    cont += heuristicaVerticales(conecta);  
  
    //Función heurística que evalúa horizontales  
    cont += heuristicaHorizontales(conecta);  
  
    //Funciones heurísticas que evalúan las diagonales  
    cont += heuristicaDiagonalesIzqDer(conecta);  
    cont += heuristicaDiagonalesDerIzq(conecta);  
  
    return cont;  
}
```

Para el cálculo de la función heurística, debemos comprobar de TODAS las fichas colocadas: sus verticales, sus horizontales y sus diagonales, siguiendo el siguiente patrón:

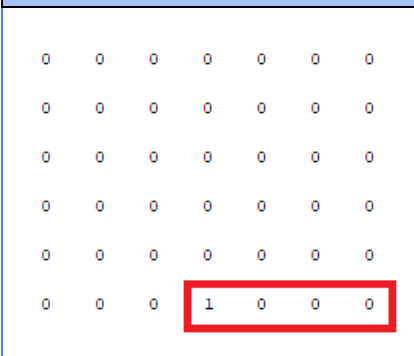
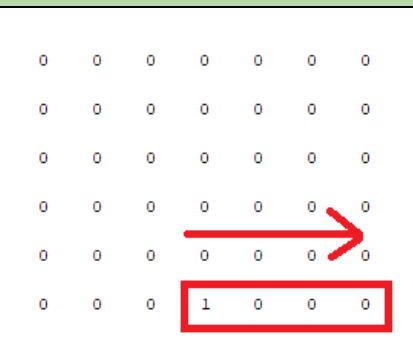
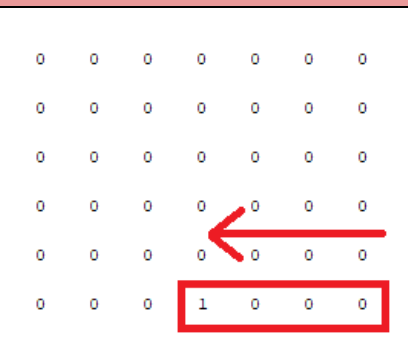
1. Agruparemos por grupos de *conecta* casillas, partiendo de cada ficha colocada, en todas las direcciones.
2. Por cada grupo de *conecta* casillas, comprobaremos si hay solo fichas de un jugador. Si esto es así, restaremos a nuestra heurística 10^n , si son fichas del jugador 1 (ya que es el jugador MIN), o sumaremos 10^n si son del jugador 2 (ya que es el jugador MAX), siendo n el número de fichas en el grupo. Si hay fichas de ambos jugadores, pasamos a comprobar el siguiente grupo de cuatro casillas.

Para no tener solapamiento de datos, es decir, que parte de la heurística se calcule varias veces para un mismo conjunto de casillas, la función heurística solo se calculará en una dirección (en verde, como se hace. En rojo como no):

- Si se están calculando las verticales, solo de abajo a arriba:

GRUPO VERTICAL CASILLAS	DE ABAJO A ARRIBA	DE ARRIBA A ABAJO
		

- Si se están calculando las horizontales, solo de izquierda a derecha:

GRUPO HORIZONTAL CASILLAS	DE IZQUIERDA A DERECHA	DE DERECHA A IZQUIERDA
		

- Si se están calculando las diagonales, solo las diagonales hacia arriba:

GRUPO DIAGONAL CASILLAS	HACIA ARRIBA	

Además, no hará falta comprobar la heurística en los tres métodos para TODAS las casillas del tablero, solo para las casillas que estén comprendidas entre los siguientes rangos:

- **Para las verticales:** sólo si la casilla más baja del grupo de casillas está **entre la fila 0 y la fila FILAS - CONECTA**.
- **Para las horizontales:** sólo si la casilla más a la izquierda del grupo de casillas está **entre la columna 0 y la columna COLUMNAS - CONECTA**.
- **Para las diagonales:** sólo si la casilla más a la izquierda del grupo de casillas está **entre la fila 0 y la fila FILAS - CONECTA y entre la columna 0 y la columna COLUMNAS - CONECTA** (para las diagonales de izquierda a derecha), y sólo si la ficha más a la derecha del grupo de casillas está **entre la fila FILAS - CONECTA y la fila FILAS - 1 y entre la columna 0 y la columna COLUMNAS - CONECTA** (para las diagonales de derecha a izquierda).

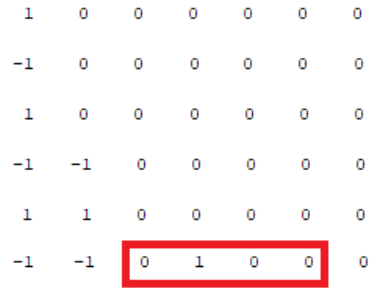
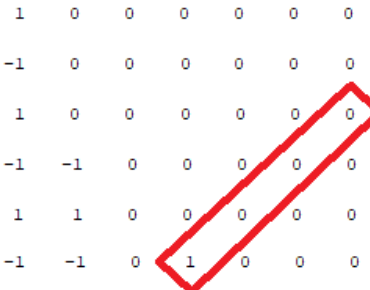
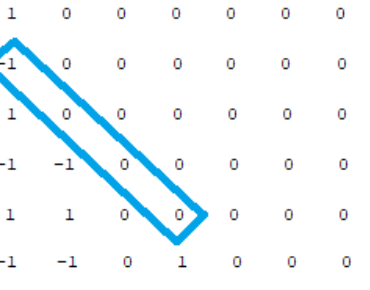
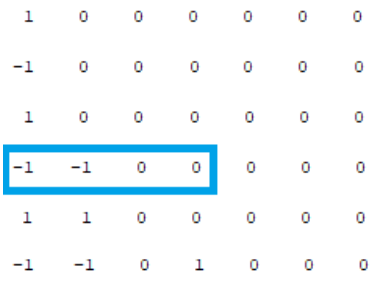
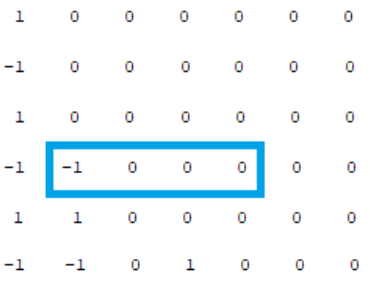
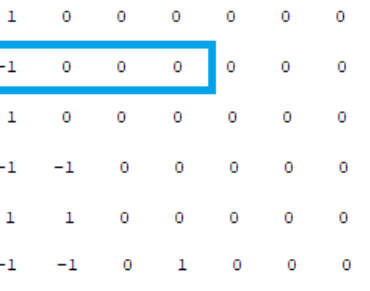
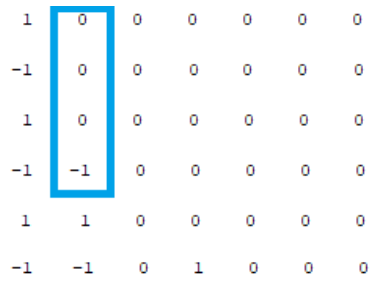
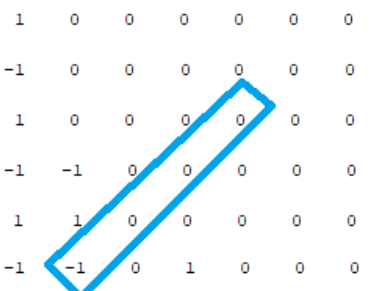
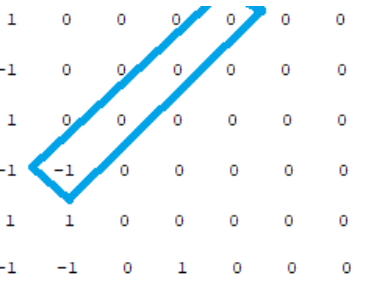
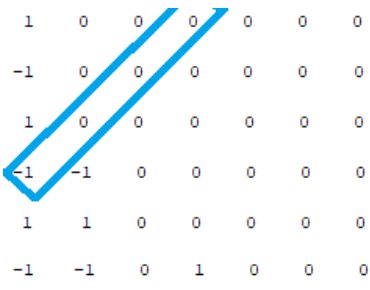
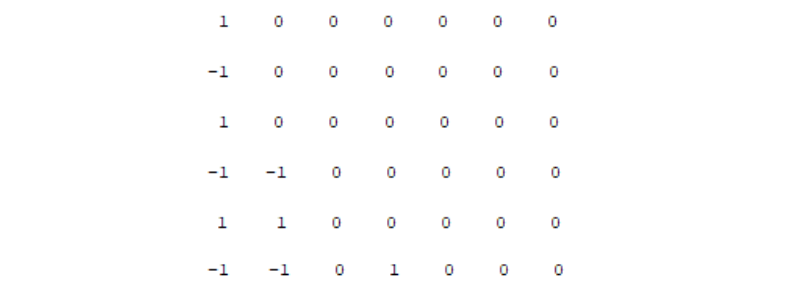
siendo *FILAS*, el número de filas del tablero, *COLUMNAS*, el número de columnas del tablero, y *CONECTA*, el número de fichas que se deben conectar para ganar.

Esto lo hacemos así por una sencilla razón: una ficha fuera de esos rangos en las heurísticas indicadas, no podría conectar 4, así que no se calcula. Con esto y calculando la heurística solo en un sentido (como se explicaba arriba), nos ahorramos gran parte de los cálculos, evitando considerar todos estos casos.

Para ilustrar mejor el funcionamiento de nuestra función heurística, calcularemos el valor del siguiente ejemplo:

[**1 = JUGADOR 1** **-1 = JUGADOR 2**]

<pre> 1 0 0 0 0 0 0 -1 0 0 0 0 0 0 1 0 0 0 0 0 0 -1 -1 0 0 0 0 0 1 1 0 0 0 0 0 -1 -1 0 1 0 0 0 </pre>	<pre> 1 0 0 0 0 0 0 -1 0 0 0 0 0 0 1 0 0 0 0 0 0 -1 -1 0 0 0 0 0 1 1 0 0 0 0 0 -1 -1 0 1 0 0 0 </pre>	<pre> 1 0 0 0 0 0 0 -1 0 0 0 0 0 0 1 0 0 0 0 0 0 -1 -1 0 0 0 0 0 1 1 0 0 0 0 0 -1 -1 0 1 0 0 0 </pre>
PUNTO DE PARTIDA: $v = 0$	$v = 0 - 10^1 = -10$	$v = -10 - 10^1 = -20$
<pre> 1 0 0 0 0 0 0 -1 0 0 0 0 0 0 1 0 0 0 0 0 0 -1 -1 0 0 0 0 0 1 1 0 0 0 0 0 -1 -1 0 1 0 0 0 </pre>	<pre> 1 0 0 0 0 0 0 -1 0 0 0 0 0 0 1 0 0 0 0 0 0 -1 -1 0 0 0 0 0 1 1 0 0 0 0 0 -1 -1 0 1 0 0 0 </pre>	<pre> 1 0 0 0 0 0 0 -1 0 0 0 0 0 0 1 0 0 0 0 0 0 -1 -1 0 0 0 0 0 1 1 0 0 0 0 0 -1 -1 0 1 0 0 0 </pre>
$v = -20 - 10^2 = -120$	$v = -120 - 10^1 = -130$	$v = -130 - 10^1 = -140$
<pre> 1 0 0 0 0 0 0 -1 0 0 0 0 0 0 1 0 0 0 0 0 0 -1 -1 0 0 0 0 0 1 1 0 0 0 0 0 -1 -1 0 1 0 0 0 </pre>	<pre> 1 0 0 0 0 0 0 -1 0 0 0 0 0 0 1 0 0 0 0 0 0 -1 -1 0 0 0 0 0 1 1 0 0 0 0 0 -1 -1 0 1 0 0 0 </pre>	<pre> 1 0 0 0 0 0 0 -1 0 0 0 0 0 0 1 0 0 0 0 0 0 -1 -1 0 0 0 0 0 1 1 0 0 0 0 0 -1 -1 0 1 0 0 0 </pre>
$v = -140 - 10^1 = -150$	$v = -150 - 10^1 = -160$	$v = -160 - 10^1 = -170$

		
$v = -170 - 10^1 = -180$	$v = -180 - 10^1 = -190$	$v = -190 + 10^1 = -180$
		
$v = -180 + 10^2 = -80$	$v = -80 + 10^1 = -70$	$v = -70 + 10^1 = -60$
		
$v = -60 + 10^1 = -50$	$v = -50 + 10^1 = -40$	$v = -40 + 10^1 = -30$
		
$v = -30 + 10^1 = -20$	El valor de la heurística para este punto del juego es: -20	

3. Optimizando el árbol. Poda alfa-beta. Análisis de ejecuciones.

Por último, implementaremos la poda alfa-beta en nuestro algoritmo. La poda alfa-beta solo mejora la eficiencia del algoritmo, pero no debe cambiar el comportamiento del agente. Esta poda permite (sumada a la restricción de niveles) explorar hasta una profundidad considerable en poco tiempo, haciendo que el tiempo desde que pone el humano hasta que le vuelva a tocar sea muy pequeño.

3.1 Implementación de la poda.

Para la poda alfa-beta hemos añadido a la clase **Jugada** dos atributos, estos son:

- **int alfa**: Valor alfa del nodo. Es el valor que se usará en los nodos pertenecientes a niveles MAX (PLAYER2) para comprobar si se tienen que podar o no.
- **int beta**: Valor beta del nodo. Es el valor que se usará en los nodos pertenecientes a niveles MAX (PLAYER1) para comprobar si se tienen que podar o no.

La única modificación para la implementación de la poda (además de estos atributos) se da a la hora de propagar el valor de utilidad hacia arriba. Ahora también tenemos que propagar los valores alfa-beta a cada nodo. La poda funciona de la siguiente manera:

- Para todos los niveles se va actualizando la utilidad de la manera usual.
- Si estamos en un nivel **MAX**, tenemos que ver si el valor del hijo que acabamos de generar (este valor será nuestro hipotético valor alfa) es peor que nuestro valor alfa actual, esto es, si el valor beta es mayor o igual que la utilidad del hijo, podemos.
- Si estamos en un nivel **MIN**, el razonamiento es similar que el de los niveles MAX. Lo único que cambia es que ahora trabajamos como hipotético valor beta la utilidad del nodo hijo. Si esta utilidad es menor o igual que alfa, podemos.
- Es importante recalcar que **o podemos, o propagamos valores alfa-beta** pero no se pueden hacer las dos cosas ya que puede llevar a errores en la poda.

Con los valores alfa y beta definimos el intervalo (α, β) . El intervalo es útil para definir cuándo tenemos que podar. Si la utilidad de un nodo no pertenece a este intervalo, será podado. α es el extremo inferior porque nos interesa tener valores mayores a este, es decir: **como mínimo la utilidad de mi nodo tiene que ser α** . Similar para beta: **como máximo la utilidad de mi nodo tiene que ser β** . Por ello, cuando se cruzan ambos valores ($\alpha \geq \beta$) significa que hemos encontrado un nodo óptimo y no seguimos explorando. Aclaremos que este es el motivo por el que no se

propagan valores cuando se poda, porque llevaría a podar todo el árbol a partir de ese momento. Por ello, nos hemos referido antes como “hipotético valor alfa (o beta)” a la utilidad del hijo.

3.2 Experimentos

Se presentan ahora una serie de experimentos clasificados según el nivel de profundidad que tenía el árbol en su ejecución. Se ha intentado que la partida sea lo más natural posible. Estos son los resultados:

Niveles	Ejecución	Movimientos	Ganador
4	1	26	ALFABETA
	2	20	
	3	18	
5	1	22	
	2	18	
	3	18	
6	1	30	
	2	16	
	3	16	
7	1	36	
	2	12	
	3	24	
8	1	16	
	2	32	
	3	32	
9	1	26	
	2	22	
	3	24	

Podemos observar que la Inteligencia Artificial se proclama vencedora después de una victoria aplastante en absolutamente todas las partidas. Pese a nuestras excelentes habilidades en el *Conecta 4*, nos ha resultado imposible ganarle, perdiendo incluso por Conecta 4 dobles o conectando hasta seis fichas seguidas como se ve a continuación (en partidas en las que solo tenía que conectar cuatro). Aquí un ejemplo de una *victoria implacable* de IAPlayer (partida independiente a las partidas de los experimentos) :



La IA ha sido capaz de ganarnos CASI SIEMPRE, puede ser por dos cosas:

- 1) Que funcione como debe de funcionar y esté **bien implementada**.
- 2) Que nuestra **habilidades** en este tipo de juegos sean **nulas**, y por eso siempre gane.

Solo hemos conseguido ganar a la IA usando el debugger de Netbeans para ver las utilidades de los nodos que nos pertenecen (como jugadores MIN) e ir colocando fichas en aquellas posiciones que mejoren o mantengan nuestra utilidad.

Niveles	Movimientos Medios
4	21
5	19
6	21
7	24
8	27
9	24

Realizados los experimentos, hemos hecho esta tabla de medias para poder comparar resultados y realizar algunas conclusiones. Se aprecia que a medida que se va subiendo de nivel, el número de movimientos medios aumenta. **¿Pero esto quiere decir que el algoritmo funciona peor con niveles más profundos? No tiene por qué.** Como hemos dicho, estos experimentos han intentado ser los más naturales posibles, intentando no anticiparse al resultado de la máquina por su implementación.

Al principio el jugador humano no sabía cómo podía ser la estrategia de la máquina, no tenía una idea sobre cómo iba a jugar por lo que a medida que va jugando **el humano aprende** de su contrincante, sin embargo **la máquina siempre sigue el mismo algoritmo**, lo único que cambian son sus niveles de profundidad para poder afinar mejor su decisión. Si el humano llegase a encontrar una **combinación** que siempre le otorgase la victoria podría usar esta **para ganar en cualquier partida**, o adaptarla para generar otra parecida ya que la máquina no va a cambiar de estrategia porque no aprende.

Si hubiésemos dotado de **aprendizaje** a `IAPlayer` entrenándolo a base de jugar una gran cantidad de partidas contra diferentes humanos, podría adaptarse a la situación y cambiar de estrategia a mitad del juego haciendo sus **movimientos impredecibles** para el humano.