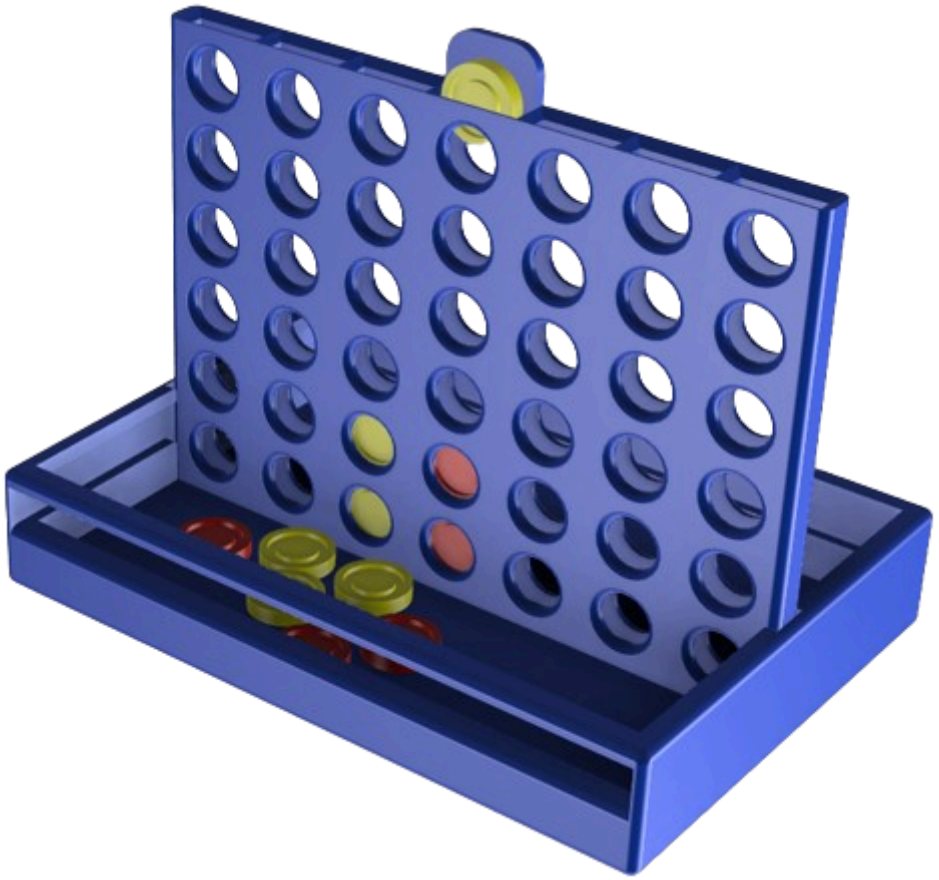
BÚSQUEDA CON ADVERSARIO



CONECTA 4

Adán González Rodríguez **adan.rozaguez@gmail.com**
Santiago Palomino Vazquez **santivzqzz@gmail.com**

ÍNDICE

Resumen del problema

Se busca desarrollar una estrategia para el CONECTA-4 , entrenaremos un algoritmo para que compita por formar una secuencia de 4 fichas. Debemos crear funciones que evalúen posiciones para conseguir nuestro objetivo.

Introducción

Modelaremos el juego CONECTA-4 y aplicaremos el algoritmo MINIMAX con poda ALFA-BETA, además buscaremos una buena función de evaluación. Para ello implementaremos el algoritmo, definiremos distintos evaluadores y diseñaremos un método de ajuste de pesos para finalmente comparar el funcionamiento de la heurística que hemos optimizado y de la aleatoria. Para la realización de los tests, hemos hecho los enfrentamientos con 1000 partidas cada uno para 7 profundidades.

Implementación Poda ALFA-BETA

```
public class EstrategiaAlfaBeta extends Estrategia {
    private Evaluador _evaluador;
    private int _capaMaxima;
    private int _jugadorMAX;

    public EstrategiaAlfaBeta() {
    }
    public EstrategiaAlfaBeta(Evaluador evaluador, int capaMaxima) {
       this.establecerEvaluador(evaluador);
       this.establecerCapaMaxima(capaMaxima);
    }

    public int buscarMovimiento(Tablero tablero, int jugador) {
        num_movimientos++;
        double tiempo_inicio = System.currentTimeMillis();
        boolean movimientosPosibles[] = tablero.columnasLibres();
        Tablero nuevoTablero;
        int col,valorSucesor;
        int mejorPosicion=-1;  // Movimiento nulo
        int mejorValor=_evaluador.MINIMO; // Minimo  valor posible

        _jugadorMAX = jugador; // - anota el identificador del jugador que
                               //   tiene el papel de MAX
                               // - necesario para evaluar posiciones finales
        for (col=0; col<Tablero.NCOLUMNAS; col++) {
            if (movimientosPosibles[col]) { //se puede añadir ficha en columna
                // crear nuevo tablero y comprobar ganador
                nuevoTablero = (Tablero) tablero.clone();
                nuevoTablero.anadirFicha(col,jugador);
                nuevoTablero.obtenerGanador();

                // evaluarlo (OJO: cambiar jugador, establecer capa a 1)
                valorSucesor = ALFABETA(nuevoTablero,Jugador.alternarJugador(jugador), _evaluador.MINIMO,
_evaluador.MAXIMO, 1);
                nuevoTablero = null; // Ya no se necesita

                // tomar mejor valor
                if (valorSucesor >= mejorValor) {
                    mejorValor = valorSucesor;
                    mejorPosicion = col;
                }
            }
        }
        tiempo_total += System.currentTimeMillis() - tiempo_inicio;
        return(mejorPosicion);
    }


    public int ALFABETA(Tablero tablero, int jugador, int alfa, int beta, int capa) {
        num_nodos++;
        // Casos base
        if (tablero.hayEmpate()) {
            return(0);
        }
```

```
        if (tablero.esGanador(_jugadorMAX)){ // gana MAX
            return(_evaluador.MAXIMO);
        }
        if (tablero.esGanador(Jugador.alternarJugador(_jugadorMAX))){ // gana el otro
            return(_evaluador.MINIMO);
        }
        if (capa == (_capaMaxima)) { // alcanza nivel maximo
            return(_evaluador.valoracion(tablero, _jugadorMAX));
        }

        // Recursividad sobre los sucesores
        boolean movimientosPosibles[] = tablero.columnasLibres();
        Tablero nuevoTablero;
        int col, alfa_actual, beta_actual;
        int valor = 0;

        if (esCapaMAX(capa)) {
            alfa_actual = alfa;
            valor = _evaluador.MINIMO;
            for (col=0; col<Tablero.NCOLUMNAS; col++) {
                nuevoTablero = (Tablero) tablero.clone();
                nuevoTablero.anadirFicha(col, jugador);
                nuevoTablero.obtenerGanador();

                if (alfa_actual >= beta) {
                    break; //poda BETA
                }
                else {
                    valor = maximo2(valor, ALFABETA(nuevoTablero, Jugador.alternarJugador(jugador),
alfa_actual, beta, capa + 1));
                    alfa_actual = maximo2(alfa_actual, valor);
                }

            }
        }
        else if (esCapaMIN(capa)) {
            beta_actual = beta;
            valor = _evaluador.MAXIMO; // valor máximo
            for (col=0; col<Tablero.NCOLUMNAS; col++) {
                if (movimientosPosibles[col]) { //se puede añadir ficha en columna
                    // crear nuevo tablero y comprobar ganador
                    nuevoTablero = (Tablero) tablero.clone();
                    nuevoTablero.anadirFicha(col,jugador);
                    nuevoTablero.obtenerGanador();

                    if (beta_actual <= alfa) {
                        break; // poda ALFA
                    }
                    else {
                        valor = minimo2(valor, ALFABETA(nuevoTablero,Jugador.alternarJugador(jugador), alfa,
beta_actual, capa + 1));
                        beta_actual = minimo2(beta_actual, valor);
                    }
                }
            }
        }
        return(valor);
    }
```

```
   public void establecerCapaMaxima(int capaMaxima) {
      _capaMaxima = capaMaxima;
   }

   public void establecerEvaluador(Evaluador evaluador) {
      _evaluador = evaluador;
   }
    private static final boolean esCapaMIN(int capa) {
        return((capa % 2)==1); // es impar
    }

    private static final boolean esCapaMAX(int capa) {
        return((capa % 2)==0); // es par
    }

    private static final int maximo2(int v1, int v2) {
        if (v1 > v2)
            return(v1);
        else
            return(v2);
    }

    private static final int minimo2(int v1, int v2) {
        if (v1 < v2)
            return(v1);
        else
            return(v2);
    }
}
```

Durante la exploración de las ramas del árbol, si el valor de alfa es mayor o igual que beta en un nodo MAX, no es necesario seguir explorando esa rama, ya que el oponente no elegiría esa opción.

De manera similar, si el valor de beta es menor o igual que alfa en un nodo MIN, tampoco es necesario seguir explorando esa rama, ya que el oponente de MAX no elegiría esa opción.

Estas podas ayudan a reducir el número de nodos que deben ser evaluados, lo que resulta en una mejora significativa en la eficiencia del algoritmo. Esto lo podemos ver más adelante en las tablas de los enfrentamientos utilizando el algoritmo MiniMax con y sin la poda AlfaBeta.

Modificaciones

Tablero.java

```
    private static final String RESET = "\u001B[0m";
    private static final String RED = "\u001B[31m";
    private static final String BLUE = "\u001B[34m";
    private static final String MARCA_J1 = BLUE + "X" + RESET;
    private static final String MARCA_J2 = RED + "O" + RESET;
```

Modificado el inicio de esta clase para que las fichas tengan colores.

Estrategia.java

```java
public abstract class Estrategia {
    public int num_nodos = 0;
    public int num_movimientos = 0;
    public double tiempo_total = 0;

    public Estrategia() {
    }
    public abstract int buscarMovimiento(Tablero tablero, int jugador);
}
```

Definiciones de variables para recopilar datos.

Jugador.java

```java
public class Jugador {
    private Estrategia _estrategia;
    private int _identificador;
    public Jugador() {
    }
    public Jugador(int identificador) {
        _identificador = identificador;
    }
    public void establecerEstrategia(Estrategia estrategia) {
        _estrategia = estrategia;
    }
    public int obtenerJugada(Tablero tablero) {
        return(_estrategia.buscarMovimiento(tablero, _identificador));
    }
    public int getIdentificador() {
        return(_identificador);
    }

    public double[] getDatos() {
        double[] datos = new double[3];
        datos[0] = _estrategia.num_nodos;
        datos[1] = _estrategia.num_movimientos;
        datos[2] = _estrategia.tiempo_total;
        return datos;
    }
    public static final int alternarJugador(int jugadorActual) {
        return(((jugadorActual%2)+1));  // Alterna entre jugador 1 y 2
    }
}
```

Añadida la función para obtener los datos (definidos antes) de cada jugador con una estrategia en específico.

Descripción de Evaluadores

Evaluador Fichas Consecutivas

```
public static int evaluarFichasConsecutivas(Tablero tablero, int jugador) {
    int valoracion = 0;
    for (int fila = 0; fila < Tablero.NFILAS; fila++) {
        for (int columna = 0; columna <= Tablero.NCOLUMNAS - Tablero.NOBJETIVO; columna++) {
            int contador = 0;
            for (int k = 0; k < Tablero.NOBJETIVO; k++) {
                if (tablero.getCasilla(columna + k, fila) == jugador) {
                    contador++;
                }
            }
            if (contador == Tablero.NOBJETIVO) {
                valoracion += 1;
            }
        }
    }
    return valoracion;
}
```

Evaluador Posibles Alineaciones

```
public static int evaluarPosiblesAlineaciones(Tablero tablero, int jugador) {
    int valoracion = 0;
    for (int columna = 0; columna < Tablero.NCOLUMNAS; columna++) {
        for (int fila = 0; fila < Tablero.NFILAS; fila++) {
            if (tablero.getCasilla(columna, fila) == 0) {
                int contador = 0;
                for (int k = 0; k < Tablero.NOBJETIVO; k++) {
                    if (tablero.esCasillaValida(columna + k, fila)) {
                        if (tablero.getCasilla(columna + k, fila) == jugador) {
                            contador++;
                        }
                    }
                }
                if (contador == Tablero.NOBJETIVO - 1) {
                    valoracion -= 1;
                }
            }
        }
    }
    return valoracion;
}
```

Evaluador Fichas Centrales

```
public static int evaluarFichasCentrales(Tablero tablero, int jugador) {
    int valoracion = 0;
    for (int columna = 0; columna < Tablero.NCOLUMNAS; columna++) {
        for (int fila = 0; fila < Tablero.NFILAS; fila++) {
            if (fila >= 2 && fila <= 3) {
                if (tablero.getCasilla(columna, fila) == jugador) {
                    valoracion += 2;
                }
            }
        }
    }
    return valoracion;
}
```

Evaluador Diagonalmente (Ascendente)

```
public static int evaluarFichasDiagonalAscendente(Tablero tablero, int jugador) {
    int valoracion = 0;
    for (int columna = 0; columna <= Tablero.NCOLUMNAS - Tablero.NOBJETIVO; columna++) {
        for (int fila = 0; fila <= Tablero.NFILAS - Tablero.NOBJETIVO; fila++) {
            int contador = 0;
            for (int k = 0; k < Tablero.NOBJETIVO; k++) {
                if (tablero.getCasilla(columna + k, fila + k) == jugador) {
                    contador++;}
            }
            if (contador == Tablero.NOBJETIVO) {
                valoracion -= 2;
            }
        }
    }
    return valoracion;
}
```

Función de Ajuste de Pesos

Hemos empleado una estrategia de evaluación ponderada, en el que el programa asigna valores a posiciones en el tablero según diversas heurísticas, lo que le permite realizar elecciones estratégicas más informadas y ajustar pesos.

Además, estos pesos se ajustan de manera iterativa mediante un algoritmo que busca mejorar el rendimiento del evaluador. Este proceso de ajuste se basa en la simulación de múltiples partidas utilizando los pesos actuales y comparando el rendimiento obtenido con diferentes configuraciones de pesos. Así, el sistema evoluciona gradualmente hacia una configuración óptima que maximiza la capacidad del evaluador para identificar las jugadas más favorables.

Por último, el código es capaz de leer y escribir estos pesos ajustados en un archivo *"pesos_ajustados.txt"* lo que nos permite que el ajuste se realice de manera automática.

```
    public static final int NUMERO_ITERACIONES = 100; // Número de iteraciones para ajustar los pesos
    public static final double FACTOR_AJUSTE = 0.1; // Factor de ajuste para modificar los pesos

    public static void ajustarPesos() {
        leerPesosDesdeArchivo("pesos_ajustados.txt");
        float mejorValoracion = Float.MIN_VALUE;
              float[]  mejoresPesos  =  {PONDERACION_FICHAS_CONSECUTIVAS,  PONDERACION_POSIBLES_ALINEACIONES,
PONDERACION_FICHAS_CENTRALES, PONDERACION_FICHAS_DIAGONALES_ASCENDENTES};

        for (int i = 0; i < NUMERO_ITERACIONES; i++) {
            float[] nuevosPesos = generarNuevosPesos(mejoresPesos);
            setPonderaciones(nuevosPesos);
            // Evaluar el desempeño con los nuevos pesos
            float valoracion = evaluarDesempeno();
            // Comparar con el mejor desempeño hasta ahora
            if (valoracion > mejorValoracion) {
                mejorValoracion = valoracion;
                mejoresPesos = nuevosPesos;
            }
        }
        // Establecer los mejores pesos
        setPonderaciones(mejoresPesos);
        // Guardar los pesos ajustados en un archivo
        guardarPesosEnArchivo(mejoresPesos);
    }
    private static float[] generarNuevosPesos(float[] pesosActuales) {
        float[] nuevosPesos = new float[pesosActuales.length];
        for (int i = 0; i < pesosActuales.length; i++) {
            // Incrementar o decrementar el peso actual por un factor
            nuevosPesos[i] = (float) (pesosActuales[i] * (1 + (Math.random() * 2 - 1) * FACTOR_AJUSTE));
        }
        return nuevosPesos;
    }
    private static void setPonderaciones(float[] pesos) {
        PONDERACION_FICHAS_CONSECUTIVAS = pesos[0];
        PONDERACION_POSIBLES_ALINEACIONES = pesos[1];
        PONDERACION_FICHAS_CENTRALES = pesos[2];
        PONDERACION_FICHAS_DIAGONALES_ASCENDENTES = pesos[3];
    }
```

```java
private static int evaluarDesempeno() {
    int totalPartidas = 15; // Número total de partidas para evaluar el desempeño
    int victoriasActuales = 0;

    for (int i = 0; i < totalPartidas; i++) {
        // Crear una nueva instancia de Tablero para cada partida
        Tablero tablero = new Tablero();
        int resultado = simularPartida(tablero); // Simular la partida actual

        if (resultado == 1) {
            victoriasActuales++;
        }
    }

    // Calcular el porcentaje de victorias
    double porcentajeVictorias = (double) victoriasActuales / totalPartidas * 100;

    // Devolver un valor que represente el desempeño relativo de los pesos actuales
    return (int) porcentajeVictorias;
}

private static int simularPartida(Tablero tablero) {
    // Crear los jugadores con las heurísticas respectivas
    Jugador jugadorAleatorio = new Jugador(1);
    jugadorAleatorio.establecerEstrategia(new EstrategiaAlfaBeta(new EvaluadorPersonalizado(), 4));

    Jugador jugadorPersonalizado = new Jugador(2);
    jugadorPersonalizado.establecerEstrategia(new EstrategiaAlfaBeta(new EvaluadorPersonalizado(), 4));

    // Inicializar el turno aleatoriamente
    Random rand = new Random();
    int turno = rand.nextInt(2) + 1;

    while (!tablero.esFinal()) {
        int jugada;
        if (turno == 1) {
            jugada = jugadorAleatorio.obtenerJugada(tablero);
        } else {
            jugada = jugadorPersonalizado.obtenerJugada(tablero);
        }

        if (tablero.esCasillaValida(jugada, 0)) {
            tablero.anadirFicha(jugada, turno);
            tablero.obtenerGanador();
            turno = turno == 1 ? 2 : 1; // Cambiar de turno
        }
    }
    // Devolver el resultado de la partida
    if (tablero.ganaJ1()) {
        return 1; // Jugador 1 gana
    } else if (tablero.ganaJ2()) {
        return -1; // Jugador 2 gana
    } else {
        return 0; // Empate
    }
}
```

Resultados Obtenidos

Resultados tras 1000 partidas con el algoritmo MiniMax con la poda AlfaBeta

Enfr	Prof.	V. E1	V. E2	E	T. E1	Nodos E1	Movs. E1	T. E2	Nodos E2	Movs. E2	Tiempo Total
1	1	632	368	0	0,038	19	4	0,036	15	4	
2	1	487	513	0	0,02	30	11	0,118	31	11	
3	1	595	405	0	0,008	18	4	0,006	16	4	36884
1	2	528	437	35	0,328	341	12	0,132	238	12	
2	2	462	538	0	0,128	232	16	0,165	270	16	
3	2	508	450	42	0,326	331	13	0,11	258	13	39291
1	3	405	575	20	0,911	980	11	0,914	1189	11	
2	3	486	514	0	0,611	937	11	0,928	990	11	
3	3	400	574	26	1,019	992	11	0,846	1138	11	46843
1	4	494	380	126	0,004	5297	16	0,005	7998	16	
2	4	519	481	0	0,002	3526	21	0,002	3267	21	
3	4	508	381	111	0,004	5292	16	0,006	7793	16	83381
1	5	395	551	54	12,475	13162	14	23,533	33241	15	
2	5	506	494	0	13,647	15972	17	13,93	15594	17	
3	5	407	542	51	11,92	13366	14	23,102	32941	14	221075
1	6	365	522	133	47,765	51290	16	124,884	17223	17	
2	6	474	525	0	38,781	44184	19	43,155	49032	19	
3	6	402	518	80	44,396	50230	16	117,05	162627	16	778334
1	7	210	747	43	70,52	78003	14	444,869	615757	15	
2	7	479	521	0	112,056	125871	19	123,96	136907	19	
3	7	209	753	38	68,659	75129	14	443,281	602289	14	2112834

Enfr → Tipo de enfrentamiento (especificado más abajo).
Prof → Profundidad.
V. E1 → Victorias Evaluador 1.
V. E2 → Victorias Evaluador 2.
E → Empates.
T. → Tiempo medio por evaluador en milisegundos (ms).
Nodos → Número medio de nodos expandidos evaluador.
Movs → Número de movimientos medio por evaluador.
Tiempo Total → Tiempo total por profundidad tras los 3 enfrentamientos en milisegundos (ms).

Tipo de enfrentamientos
Enfrentamiento 1: Evaluador personalizado **SIN** ajuste automático vs Evaluador aleatorio.

Enfrentamiento 2: Evaluador personalizado **SIN** ajuste automático vs Evaluador personalizado **CON** ajuste automático.

Enfrentamiento 3: Evaluador personalizado **CON** ajuste automático vs Evaluador aleatorio.

Todos los datos de esta tabla se encuentran en el archivo *"resultados_simulaciones.txt"*.

Resultados tras 1000 partidas con el algoritmo MiniMax sin la poda AlfaBeta

Enfr	Prof.	V. E1	V. E2	E	T. E1	Nodos E1	Movs. E1	T. E2	Nodos E2	Movs. E2	Tiempo Total
1	1	641	359	0	0,029	19	4	0,027	14	4	
2	1	498	502	0	0,019	30	11	0,029	31	11	
3	1	607	393	0	0,007	18	4	0,006	16	4	37974
1	2	500	469	31	0,356	321	12	0,185	254	12	
2	2	504	496	0	0,167	252	16	0,162	250	16	
3	2	515	459	26	0,304	346	12	0,24	243	12	39970
1	3	387	588	25	0,996	951	11	0,768	1185	11	
2	3	501	499	0	0,656	965	11	0,854	1400	11	
3	3	415	564	21	1,633	1809	11	0,596	1157	11	48360
1	4	483	389	128	4,434	5245	16	5,8	8128	16	
2	4	494	506	0	2,586	3356	21	6,663	8483	21	
3	4	485	407	108	13,766	16220	16	5,816	8166	16	108472
1	5	404	550	46	12,729	13120	14	25,981	33278	15	
2	5	521	479	0	15,329	16446	17	44,491	58563	17	
3	5	408	542	50	65,536	81938	14	23,935	32842	14	408778
1	6	378	535	87	45,372	48363	16	125,514	170746	16	
2	6	503	497	0	40,699	46860	18	299,271	399888	18	
3	6	387	513	100	412,125	551669	16	116,352	166919	16	2028296
1	7	212	749	39	72,696	79925	14	441,016	602542	14	
2	7	483	517	0	115,785	126911	19	1521,48	1977968	19	
3	7	185	770	45	1347,981	1761976	14	446,669	631575	14	10050813

Enfr → Tipo de enfrentamiento (especificado más abajo).
Prof → Profundidad.
V. E1 → Victorias Evaluador 1.
V. E2 → Victorias Evaluador 2.
E → Empates.
T. → Tiempo medio por evaluador en milisegundos (ms).
Nodos → Número medio de nodos expandidos evaluador.
Movs → Número de movimientos medio por evaluador.
Tiempo Total → Tiempo total por profundidad tras los 3 enfrentamientos en milisegundos (ms).

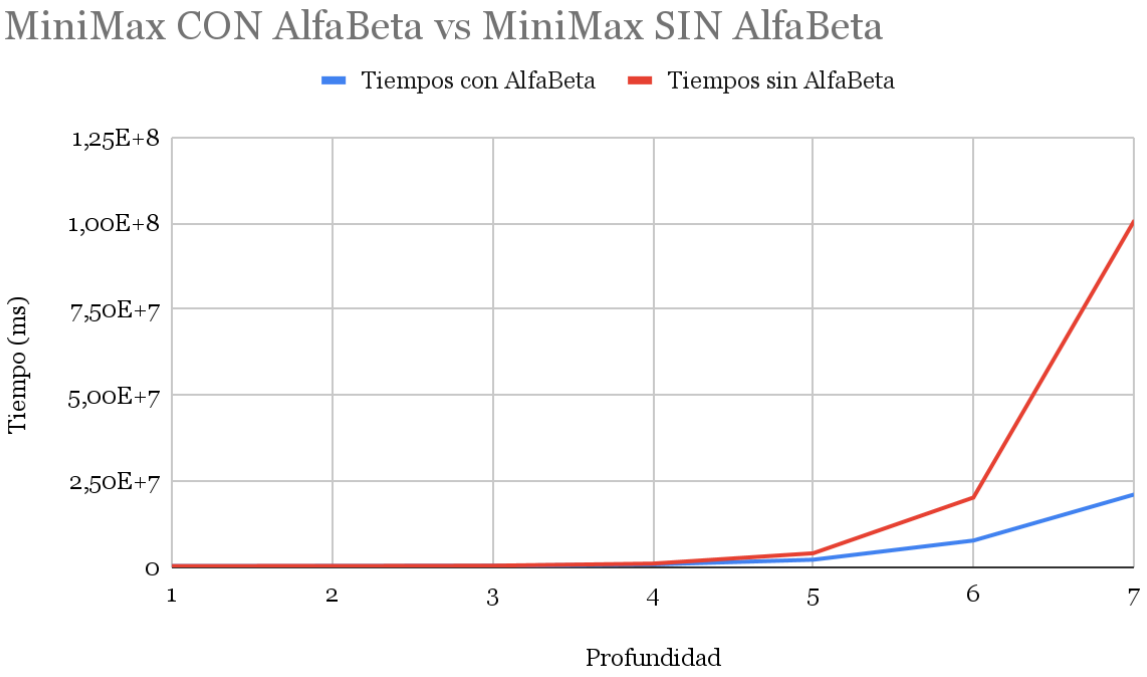
Tipo de enfrentamientos
Enfrentamiento 1: Evaluador personalizado **SIN** ajuste automático vs Evaluador aleatorio.

Enfrentamiento 2: Evaluador personalizado **SIN** ajuste automático vs Evaluador personalizado **CON** ajuste automático.

Enfrentamiento 3: Evaluador personalizado **CON** ajuste automático vs Evaluador aleatorio.

Todos los datos de esta tabla se encuentran en el archivo *"resultados_simulaciones.txt"*.

Gráfica comparativa



Conclusiones

Algoritmo MiniMax con poda AlfaBeta

Si nos fijamos en el tiempo de ejecución depende tanto de la profundidad como del evaluador empleado. A menor profundidad, el evaluador aleatorio presenta el mejor rendimiento en términos de tiempo, atribuible a su menor cantidad de operaciones. A partir de una profundidad de 4, se observa que el evaluador personalizado sin ajustes supera en velocidad al aleatorio, y en profundidades de 5 y más, el evaluador personalizado con ajuste también supera al aleatorio. Se han evaluado profundidades hasta 7, donde se confirma que el evaluador sin ajustes sigue siendo más rápido que el que sí utiliza ajuste. Sin embargo, se evidencia que el tiempo se reduce a medida que aumenta la profundidad. Por esta razón, creemos que, al aumentar considerablemente la profundidad, la mejor opción sería el evaluador personalizado con ajuste.

En cuanto a las victorias, se observa que el evaluador aleatorio pierde frente a los otros dos evaluadores cuando la profundidad es baja. Sin embargo, a medida que se incrementa la profundidad, el evaluador aleatorio comienza a obtener más victorias en ambos casos, superando significativamente al evaluador personalizado con ajuste en profundidades de 7, con una diferencia de más de 500 partidas.

Por otro lado, aunque el evaluador personalizado, ya sea con ajuste o sin él, pierde más partidas, la cantidad de nodos visitados por él es considerablemente menor que la cantidad de nodos explorados por el evaluador aleatorio. Este hecho puede ser un factor determinante a la hora de elegir el evaluador a utilizar.

Algoritmo MiniMax sin poda AlfaBeta

Al eliminar la Poda AlfaBeta podemos ver que nuestro evaluador pierde mucho, aumentando considerablemente la cantidad de tiempo necesario que utiliza y la cantidad de nodos explorados. Haciendo que a partir de profundidad 4 sea pŕacticamente inutil el evaluador personalizado.

En conclusión, la poda AlfaBeta permite reducir considerablemente la cantidad de nodos explorados y el tiempo empleado en nuestro evaluador personalizado. A pesar de esto, los resultados obtenidos a mayor profundidad no son excepcionales. Hasta una profundidad de 4, el evaluador personalizado supera en rendimiento al aleatorio, pero a profundidades mayores a 4, sería más conveniente utilizar el evaluador aleatorio si no se emplea la poda AlfaBeta y/o no hay limitaciones en la memoria disponible.

Problemas

Durante el desarrollo de este proyecto nos enfrentamos a varias dificultades respecto al código, siendo lo más difícil el ajuste de pesos automáticos en el que optamos por realizar un algoritmo que busca iterativamente los mejores pesos que maximizan el rendimiento del evaluador. Los pesos ajustados se almacenan en un archivo de texto para su posterior referencia y uso en evaluaciones futuras.

Otra de las dificultades pero que no nos llevó mucho tiempo en resolver, fue la creación de las 4 heurísticas para nuestro evaluador personalizado. En este caso optamos por realizar unas heurísticas simples pero que sirven perfectamente para encontrar las soluciones al problema planteado con distintas profundidades.

Manual de uso

Uso del programa

1. Para jugar contra la máquina:
    a. Ejecutar en consola `javac Conecta4.java`.
    b. Ejecutar en consola `java Conecta4`.
    c. Escribir por pantalla un número del 0 - 6 para colocar las fichas en el tablero.

2. Para hacer simulaciones (enfrentar los evaluadores):
    a. Ejecutar en consola `javac SimulacionPartidas.java`.
    b. Ejecutar en consola `java SimulacionPartidas`.

Aclaraciones:

- Podemos modificar qué evaluadores y qué profundidad va a utilizar la máquina modificando estos parámetros en el archivo “Conecta4.java”.

- Podemos modificar la profundidad con la que se van a enfrentar los evaluadores en el archivo “SimulacionPartidas.java” modificando la variable *profundidad*.