PRÁCTICA 2:
Búsqueda con adversario

Algoritmos Básicos de Inteligencia Artificial

2023/2024

Tomás Rial Costa - 35602105Z
tomas.rial@alumnos.uvigo.es

Carlos Sánchez Alcalá-Zamora - 47436230H
carlos.sanchez@alumnos.uvigo.es

ÍNDICE

1. Introducción

En esta práctica nos encargaremos de crear una partida de CONECTA-4 entre dos inteligencias artificiales, implementando y mejorando el algoritmo MINIMAX con poda ALFA-BETA. Comenzaremos generando el tablero, para a continuación, mediante una variante de MINIMAX, desarrollar la capacidad de seleccionar movimientos óptimos en cada turno, evaluando las posibles jugadas e intentando anticipar las decisiones del adversario.

2. Método ALFA-BETA

La poda ALFA-BETA es una técnica que mejora el rendimiento del algoritmo MINIMAX al reducir el número de nodos evaluados seleccionando la jugada que maximiza el valor de evaluación, consiguiendo una búsqueda más eficiente.

En su código comenzamos definiendo todos los parámetros (*tablero*, *jugador*, *capa*, *a* y *b*), a continuación indicamos las condiciones base del juego.

Damos lugar a una búsqueda recursiva, en la que el método explorará todas las posibles jugadas a partir del estado actual del tablero, evaluando recursivamente los nuevos tableros con el mismo método. Se alternarán entre capas de MAX (jugador actual) y MIN (jugador oponente), para así maximizar y minimizar los valores respectivamente.

La poda se irá realizando durante la búsqueda para evitar explorar ramas intrascendentes en la decisión final; en la capa MAX se actualizará el valor de ALFA (*a*) con el máximo entre el valor actual y el valor del nodo y lo mismo pasará en la capa MIN, actualizando el valor de BETA (*b*) con el mínimo entre el valor actual y el del nodo. La exploración de una rama se detendrá si *a* >= *b.*

3. Función de evaluación definida

La función de evaluación “*EvaluadorOptimo.java”* está definida de la siguiente forma, por una parte tenemos unas funciones que tras llamar a otras funciones definidas en “*Tablero.java”* devuelven cada una un valor heurístico y por otra parte tenemos la valoración().

a. Evaluador óptimo

El valorador se encarga de sumar los diferentes valores heurísticos que devuelve cada función de rasgo multiplicado por su peso asignado. Los pesos los asignamos leyendo un archivo “mejores_pesos.txt” que general el código de ajuste de pesos.

b. Funciones de la heurística

En "*Tablero.java*" vienen definidas todas las funciones que utilizaremos en la heurística explicada en el paso anterior, estas son:

i. ***contarFilasDeTres:*** Esta función utiliza otras tres funciones más para así buscar todas las posibles posiciones en las que podamos encontrar 3 fichas de nuestro color consecutivas, sumando 1 cada vez que lo hace.

1. ***contarFilasHorizontales:*** Itera sobre todas las filas y columnas del tablero, verificando si hay 3 fichas consecutivas del jugador en horizontal. Si encuentra la secuencia, incrementa el contador de filas, para luego devolver el total de filas encontradas.
2. ***contarFilasVerticales:*** Itera sobre todas las filas y columnas del tablero, verificando si hay 3 fichas consecutivas del jugador en vertical. Si encuentra la secuencia, incrementa el contador de filas, para luego devolver el total de filas encontradas.
3. ***contarFilasDiagonales:*** Itera sobre todas las filas y columnas del tablero, verificando si hay 3 fichas consecutivas del jugador en una diagonal ascendente o descendente. Si encuentra la secuencia, incrementa el contador de filas, para luego devolver el total de filas encontradas.

ii. ***contarFichasEnColumnas:*** Con esta función se cuentan la cantidad de fichas del jugador en columnas específicas del tablero. Itera sobre las filas de cada columna especificada, contando las filas del jugador encontradas, para luego devolver el total.

iii. ***contarFichasEnFilas:*** Con esta función se cuentan la cantidad de fichas del jugador en filas específicas del tablero. Itera sobre las columnas de cada fila especificada, contando las fichas del jugador encontradas, para luego devolver el total.

iv. ***contarBloqueos:*** Esta función cuenta la cantidad de movimientos del oponente que conducirían a una victoria, itera sobre las posibles columnas en las que el oponente podría colocar una ficha, para cada posibilidad clona el tablero y simula la jugada del oponente, si esta jugada provoca la victoria del rival, incrementa en 1 el contador de bloqueos. Al finalizar, devuelve el total de bloqueos encontrados.

En el Evaluador Óptimo se ajusta como se devuelven estos valores:

- **Evaluar Fichas en Fila de 3**: Este rasgo que usa la función "*contarFilasDeTres*" devuelve un valor heurístico superior cuantas más filas de 3 tenga el jugador, ya que hacer filas de 3 implica que le queda solo una fichas más consecutiva para ganar y aumenta mucho las posibilidades.

- **Evaluar Bloqueos**: Este rasgo que usa la función *"contarBloqueos"* devuelve un valor heurístico menor cuantos más posibilidad de bloqueos haya en el tablero, aunque se espera que nunca haya más de 1 ya que entonces estaría

sentenciada la partida. Este rasgo es muy importante ya que ayuda al jugador a saber que es importante bloquear para que el contrincante no consiga conectar 4. Es importante no subir demasiado el peso de este rasgo ya que el jugador podría llegar a decidir bloquear una jugada del contrincante en vez de ganar conectando él cuatro.

- **Evaluar Ocupación Centro:** Este rasgo usa la función *"contarFichasEnColumnas"* pasando como columnas objetivo las céntricas (2, 3 y 4) ya que son las columnas que dan más amplitud y juego para poder conectar 4. El valor heurístico aumenta cuantas más fichas se hallen en esas columnas.

- **Evaluar Ocupación Altura:** Este rasgo usa la función *"contarFichasEnFilas"* pasando como filas las más inferiores (0,1, 2, 3) ya más arriba de esta fila se nos acaba el espacio y es más improbable y más complicado lograr conectar 4 fichas.

4. Método empleado para el ajuste de pesos

Para el ajuste de pesos utilizamos el código que se encuentra en *"PesosOptimos.java"*:

Comenzamos inicializando los 4 mejores pesos encontrados hasta el momento, establecidos ahora todos en 1, pasamos a 4 bucles "for" para así probar todas las posibles combinaciones en un rango de 1 a 4 (así lo decidimos porque hay 4 heurísticas y dicho número permitiría unos pesos escalonados por si así lo decidiera el algoritmo), asignando cada una a un evaluador.

En cada iteración del bucle for (para cada combinación de pesos) se llama a la función comparar evaluadores donde se enfrentan el evaluador al que se le asignó la combinación de pesos creada por el bucle contra la combinación guardada en la lista de mejores pesos.

El enfrentamiento consta de 2 partidos, en cada partido empieza un jugador diferente, si el jugador que usa la combinación generada supera en victorias al jugador de mejores pesos, pasará a sustituirle y entrará a combatir el siguiente peso generado, como si de un "rey de pista" se tratase, hasta que lleguemos a la última combinación.

Por último, los mejores pesos encontrados se guardarán en un archivo de texto, el cual incluirá las ponderaciones para las fichas en fila, los bloqueos, la ocupación del centro y la ocupación de la altura.

5. Tests

a. Pruebas ALFA-BETA

i. Tiempo medio de búsqueda de cada movimiento

	Inicial	Óptimo	Aleatorio
2	0.43ms	0.42ms	0.04ms
3	0.82ms	0.74ms	0.10ms
4	3.66ms	3.52ms	0.48ms
5	15.31ms	11.80ms	3.37

ii. Nº medio de nodos generados en cada búsqueda

	Inicial	Óptimo	Aleatorio
2	6	6	5
3	22	27	30
4	100	105	140
5	376	332	551

iii. Enfrentamientos

Dos partidas por cada enfrentamiento, turnando el jugador 1 para buscar igualdad.

Los resultados están escritos en el sistema Victoria/Empate/Derrota, visto desde el punto de vista de las heurísticas que se encuentran en la primera columna.

Profundidad 2	Inicial	Óptimo	Aleatorio
Inicial	-	1/0/1	2/0/0
Óptimo	1/0/1	-	2/0/0
Aleatorio	0/0/2	0/0/2	-

Profundidad 3	Inicial	Óptimo	Aleatorio
Inicial	-	0/0/2	2/0/0
Óptimo	2/0/0	-	2/0/0
Aleatorio	0/0/2	0/0/2	-

Profundidad 4	Inicial	Óptimo	Aleatorio
Inicial	-	1/0/1	1/0/1
Óptimo	1/0/1	-	1/0/1
Aleatorio	1/0/1	1/0/1	-

Profundidad 5	Inicial	Óptimo	Aleatorio
Inicial	-	0/1/1	1/0/1
Óptimo	1/1/0	-	1/1/0
Aleatorio	1/0/1	0/1/1	-

b. Pruebas MINIMAX

i. Tiempo medio de búsqueda de cada movimiento

ii. Nº medio de nodos generados en cada búsqueda

Profundidad 4	Inicial	Óptimo	Aleatorio
Tiempo medio	11.26ms	10.53ms	1.18ms
Nº medio nodos	303	301	313

iii. Enfrentamientos

Profundidad 4	Inicial	Óptimo	Aleatorio
Inicial	-	1/0/1	2/0/0
Óptimo	1/0/1	-	1/1/0
Aleatorio	0/0/2	0/1/1	-

c. Códigos utilizados

i. Tiempo medio:

En la función jugar() hago un cálculo de tiempo para cada jugador:

```
            // obtener movimiento: llama al jugador que tenga el turno,
        // que, a su vez, llamará a la estrategia que se le asigno al crearlo
            long tiempoInicioBusqueda = System.currentTimeMillis();
            movimiento = jugadorActual.obtenerJugada(tablero);
            // comprobar si es correcto
            if ((movimiento>=0) && (movimiento<Tablero.NCOLUMNAS)) {
                posicionesPosibles = tablero.columnasLibres();
                if (posicionesPosibles[movimiento]) {
                    tablero.anadirFicha(movimiento, jugadorActual.getIdentificador());
                    // comprobar ganador
                    tablero.obtenerGanador();
                }
                else {
                    ERROR_FATAL(mensaje:"Columna completa. Juego Abortado.");
                }
            }
            else {
              ERROR_FATAL(mensaje:"Movimiento invalido. Juego Abortado.");
            }
            long tiempoFinBusqueda = System.currentTimeMillis();
            if(jugadorActual.getIdentificador() == 1) {
                tiempoTotalBusqueda1 += (tiempoFinBusqueda - tiempoInicioBusqueda);
                nummovimientos1++;
            } else {
                tiempoTotalBusqueda2 += (tiempoFinBusqueda - tiempoInicioBusqueda);
                nummovimientos2++;
            }
        }
        double tiempoMedioBusqueda1 = tiempoTotalBusqueda1 / nummovimientos1;
        double tiempoMedioBusqueda2 = tiempoTotalBusqueda2 / nummovimientos2;
        System.out.println("Tiempo medio de busqueda jugador 1: " + tiempoMedioBusqueda1);
        System.out.println("Tiempo medio de busqueda jugador 2: " + tiempoMedioBusqueda2);
    }
```

Al hacerlo de esta manera puedo hacer jugar dos evaluadores diferentes y sacar los datos individuales de cada uno, así aligero el proceso.

ii. Media de nodos

Modifiqué los códigos de MiniMax y AlphaBeta para que incremente nodos, en el caso de AlphaBeta en al ambos else ya que está estrategia contiene poda, y en el caso del MiniMax cada vez que se llama a su función. Se devuelve el valor dividido entre el número de búsquedas.

```
valorSucesor = AlphaBeta(nuevoTablero,Jugador.alternarJugador(jugador),capa:1,_evaluador.MINIMO, _evaluador.MAXIMO);
numbusquedas++;
```

```
if (a_actual >= b) {
    break;
}
else{
numNodos++;
```

```
if (b_actual <= a) {
    break;
}
else{
numNodos++;
```

```
public int MINIMAX(Tablero tablero, int jugador, int capa) {
    // Implementa la propagación de valores MINIMAX propiamente dicha
// a partir del segundo nivel (capa 1)
    numNodos++;
```

```
public int getNodos() {
    return numNodos/numbusquedas;
}
```

iii. Enfrentamientos:

Para sacar las victorias y derrotas uso el Comparador.java que usa la función ya creada en PesosOptimos.java para sacar los datos de victorias empates y derrotas.

```
public class Comparador {
    Run | Debug
    public static void main(String[] args) {
        EvaluadorOptimo evaluadorOptimo = new EvaluadorOptimo();
        EvaluadorOptimo evaluadorInicial = new EvaluadorOptimo();
        int[] pesos = {1, 1, 1, 1};
        PesosOptimos.cambiarPesos(evaluadorInicial, pesos);
        EvaluadorAleatorio evaluadorAleatorio = new EvaluadorAleatorio();

        int[] profundidades = {2, 3, 4, 5};
        for (int profundidad : profundidades) {
            System.out.println("Profundidad Máxima de Búsqueda: " + profundidad);

            // Comparación entre Heurística Óptima y Heurística Inicial
            System.out.println(x:"Optimo vs Inicial:");
            int[] resultadosOI = PesosOptimos.compararEvaluadores(evaluadorOptimo, evaluadorInicial, profundidad);
            System.out.println("Ganadas: " + resultadosOI[0] + ", Empates: " + resultadosOI[1] + ", Perdidos: " + resultadosOI[2]);

            // Comparación entre Heurística Óptima y Heurística Aleatoria
            System.out.println(x:"Optimo vs Aleatorio:");
            int[] resultadosOA = PesosOptimos.compararEvaluadores(evaluadorOptimo, evaluadorAleatorio, profundidad);
            System.out.println("Ganadas: " + resultadosOA[0] + ", Empates: " + resultadosOA[1] + ", Perdidos: " + resultadosOA[2]);

            // Comparación entre Heurística Inicial y Heurística Aleatoria
            System.out.println(x:"Inicial vs Aleatorio:");
            int[] resultadosIA = PesosOptimos.compararEvaluadores(evaluadorInicial, evaluadorAleatorio, profundidad);
            System.out.println("Ganadas: " + resultadosIA[0] + ", Empates: " + resultadosIA[1] + ", Perdidos: " + resultadosIA[2]);
        }
    }
}
```

6. Conclusiones

a. Resultados obtenidos en los tests

Sin duda se puede concluir que la mejor función es la óptima, esto es apreciable en prácticamente todas las tablas, gana la mayoría de enfrentamientos y además en el menor tiempo (no podemos tener en cuenta

la función aleatoria para este apartado ya que es prácticamente instantánea al no necesitar de heurísticas para calcular su siguiente movimiento). Por otro lado, el nº medio de nodos generado es bastante similar entre las distintas funciones, pero siendo siempre más elevado en el caso de la aleatoria.

Lo único apreciable con respecto al aumento de profundidad es un aumento del tiempo de ejecución y en el número de nodos (como es obvio), pero no se destacan otros datos claros en las tablas.

En cuanto a la comparación entre algoritmos, como era de esperar, la poda ALPHA-BETA consigue reducir mucho los tiempos y el nº de nodos generados.

b. Problemas

Uno de nuestros problemas es el uso de escribir y leer el archivo *"mejores_pesos.txt"*. Si el archivo no existe y intentamos ejecutar el código este no funcionará, ya que para usar el Evaluador Óptimo se necesitan los pesos.

Como la ubicación cambia dependiendo de si usamos el run del ejecutor (Visual Studio Code en nuestro caso) o el comando java, hay que estar pendiente de donde está el archivo para poder leerlo o sobreescribirlo.

Otros problemas que tuvimos fueron encontrar los rasgos para la heurística óptima, algunos de los que también pensamos fueron evaluar filas de 2 y ocupación de las esquinas, pero no daban buenos resultados.

Y por último, uno de los problemas más longevos que tuvimos fue en cómo hacer el evaluador de bloqueos, en algún momento incluso pensamos en dejarlo y poner otro, pero como no se nos ocurría nada mejor decidimos seguir intentándolo. Hasta que dimos con la idea de clonar el tablero y ver si el contrario ganaba en ese turno.