PRÁCTICA 03

INTELIGENCIA ARTIFICIAL

DESCONECTA4 - BOOM

Jose Manuel Osuna Luque

Grupo 2ºA

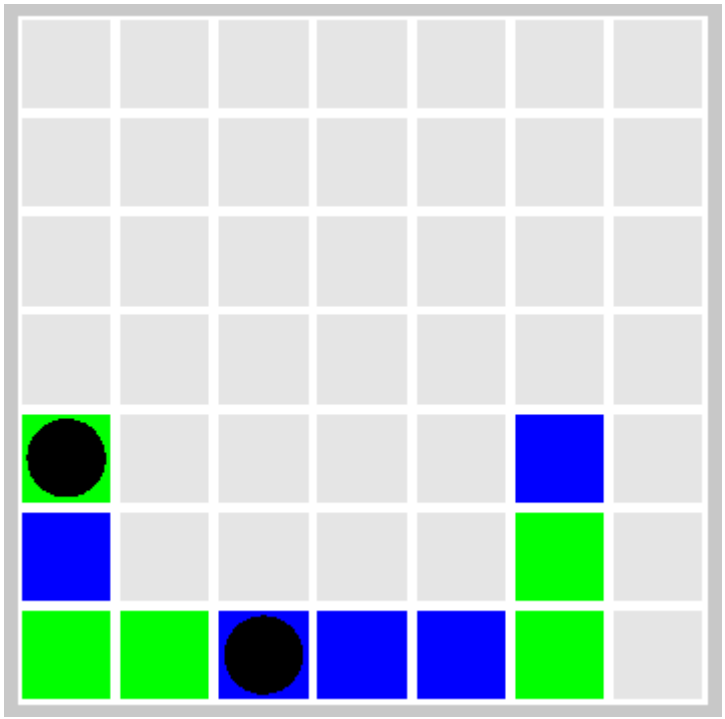
Grupo de Prácticas A3

Descripción del problema

El objetivo a conseguir en esta práctica es la configuración de un bot para ganar una partida (ya sea contra otro bot, o un humano) al juego conocido como Desconecta4-BOOM

¿Qué es el Desconecta4-BOOM?

Desconecta4-BOOM es una variante del conocido juego “Conecta4” o “4 En Raya”. El cometido del juego original es intentar crear una secuencia de cuatro fichas en horizontal, vertical o en diagonal. El tablero estará compuesto por SIETE filas y SIETE columnas. En esta variante, el objetivo será el contrario, pierde aquel que haga una conexión de cuatro fichas consecutivas. Además, se añade una ficha extra, denominada BOOM cada cinco turnos (y solo puede haber una de cada jugador), que al aplicarla elimina las fichas del jugador de esa fila en la que se encontraba. El tablero comienza vacío. El primer jugador siempre será el color VERDE y el segundo jugador el AZUL



Partida

Siempre comenzará la partida el jugador que sea las fichas VERDES, y se irá desarrollando a partir de ahí con turno por jugador.

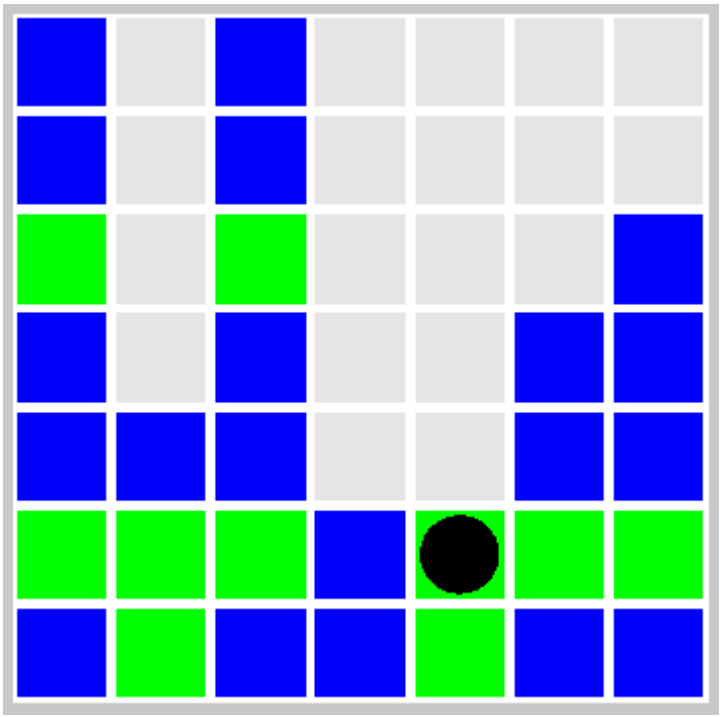
Casos posibles

Colocar una Ficha. El jugador decidirá en qué columna colocar la ficha de su color.

Colocar una Ficha Bomba. Cada turno múltiplo de 5 (5, 10, 15…) la siguiente ficha que el jugador colocará será BOMBA, salvo que ya tenga una en el tablero, y por tanto será normal.

Explotar una Ficha Bomba. Si el jugador tiene una ficha BOMBA en el tablero, puede tomar la decisión de detonarla. La consecuencia será que todas las fichas de SU COLOR de la FILA serán destruidas, y aquellas FICHAS encima de estas, caerán una a una FILA por debajo donde hayan dejado huecos las del jugador

Fin de Partida. El juego terminará cuando uno de los jugadores cree una conexión de cuatro fichas consecutivas. Aquel jugador que la cree, PERDERÁ. (En el siguiente caso ganan las fichas VERDES).



Bot Programado

Para la creación de una “Inteligencia Artificial” para que nuestro programa sea capaz de vencer, tanto a un humano como a los bots preparados por el profesor, se ha hecho uso del ***Algoritmo Poda Alfa-Beta***, junto a una función ***Heurística***.

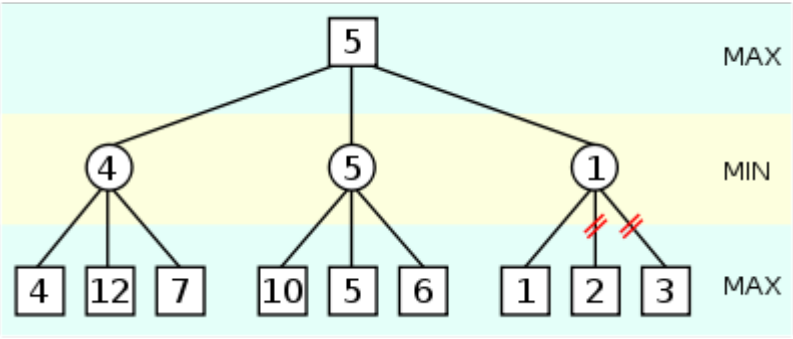
Alfa-Beta Poda

Consiste en analizar a través de un árbol binario analizando las distintas situaciones posibles. A diferencia del **Algoritmo Greedy**, este también tiene en cuenta el **turno del adversario**. Cada hijo generará a su vez sus posibles hijos, y estos generarán los suyos de cada, y así sucesivamente hasta que se llega a una profundidad máxima deseada. En ese momento se comenzará a analizar qué valoración tiene cada nodo.

Una característica de este algoritmo es que no comprueba todos los hijos de cada nodo. Y es que, el algoritmo guarda la ganancia máxima que ya ha recibido al analizar una parte del árbol, de forma que si al descender por alguno de sus hijos comprueba que ya va a ganar menos de lo almacenado, deja de valorar los demás hijos del nodo hijo.

Al analizar un nodo, pueden ocurrir dos situaciones posibles. Cuando es el *turno del jugador (del bot programado)* se intentará **maximizar** (y se considera nodo MAX) la ganancia obtenida en el movimiento mientras que, por la parte contraria, se buscará la situación menos perjudicial posible, es decir, **minimizar** (y se considera nodo MIN) la pérdida.

De esta forma, además de realizar un movimiento lo más beneficioso para el JUGADOR, habrá que tener en cuenta que no se desarrolla una situación demasiado favorable para el RIVAL.



Como se puede observar en este ejemplo, el tercer hijo del nodo raíz (que al ser MIN tomará el peor valor de sus hijos), analizando el primer hijo suyo, obtiene un valor de 1. No importa qué ocurra en los demás hijos, pues al ser un nodo MIN va a tomar el “peor resultado” que ya como máximo es 1, que es menor que 5. Por tanto, se cumple el criterio de PODA.

Además de dicho algoritmo, para establecer que casos serán beneficiosos para el bot y que movimientos resultarán desalentadores, habrá que establecer una Heurística.

Heurística

En computación, dos objetivos fundamentales son encontrar algoritmos con:

- buenos tiempos de ejecución
- buenas soluciones, usualmente las óptimas.

Una función heurística es aquella que permite dar una valoración sobre lo tan buena o mal que es una decisión. Son reglas muy generales que permiten avanzar en el proceso de resolución de problemas. El uso de una regla heurística permite obtener soluciones aceptables, aunque no óptimas. Normalmente encuentran buenas soluciones, aunque no hay pruebas de que la solución no pueda ser arbitrariamente errónea en algunos casos. En otras ocasiones, se ejecuta razonablemente rápido, aunque no existe tampoco prueba de que siempre será así.

Cabeceras de algoritmo y heurística usados

```
/**
* Implementacion Algoritmo Poda AlfaBeta. Cuando BETA <= ALFA, se poda.
* @param mesa, estado actual en el desarrollo del juego
* @param jugador, entero que indica el jugador actual al que se asocia dicho estado
* @param miJugador, siempre almacena el valor que indica qué jugador es el propio
* @paran prof, entero que indica por qué profundidad va
* @param prof_maxima, entero que indica la máxima profundidad a la que se avanzará
* @param accion, es la acción que se ha tomado como la mejor opción del jugador
* @param alpha, double que indica el valor de alfa para la poda. Su primer valor es menosInfinito
* @param beta, double que indica el valor de beta para la poda. Su primer valor es másInfinito
* @return double, un valor de retorno con la valoracion para el Algoritmo
*/
double AlfaBeta(const Environment & mesa, int jugador, const int& miJugador, int prof, const int&
prof_alfabeta, Environment::ActionType & accion, double alfa, double beta)

/**
*Funcion Heurística para valorar un nodo al usar el algoritmo PodaAlfaBeta
* @param estado, es la situación actual del tablero de juego
* @param miJugador, es el entero que representa a mi jugador en la partida actual (1 VERDE, 2 AZUL)
* @return double, que es la valoracion que se la da a ese estado
*/
double Valoracion(const Environment &estado, const int& miJugador)
```

Heurística Bot

La Heurística usada para la programación del bot tiene en cuenta diversas situaciones en referencia al juego. Por cada razón **satisfecha** se sumarán puntos al valor de la jugada. A mayor puntuación, mejor será para el jugador la acción de dicho movimiento.

Se comienza estableciendo quién es el rival y quien es el jugador (si VERDE para el bot, o AZUL). Tras ello, la primera comprobación que se hace es si el juego se ha terminado. Si eso ocurre, entonces se analiza quién gana. Si lo hace el bot, entonces se le asigna al valor de ese movimiento un FACTOR_VICTORIA y no se realizan más comprobaciones. Si ocurre que ha ganado el rival se le asigna a la valoración de la heurística un FACTOR_VICTORIA pero negativo.

```
if(estado.JuegoTerminado()){

	if(estado.RevisarTablero2() == miJugador){
		return FACTOR_VICTORIA;
	}else{
		return FACTOR_VICTORIA*(-1);
	}
}
```

Si el juego no ha terminado. Se comprueban ciertas condiciones para dotar a dicho estado un valor heurístico positivo o negativo.

Si el jugador tiene una bomba, y al detonarla, obtiene un estado donde el bot gana, se le asigna un valor de FACTOR_VICTORIA. Si por el contrario al detonarla pierde el juego, se le asigna un FACTOR_VICTORIA negativo.

```
if(estado.Have_BOOM(miJugador)){

	//Si la tiene, se puede generar un nextMove(BOMBA);
	int accionBomba = 6; //En realidad es el PUT[7]
	Environment simulado = estado.GenerateNextMove(accionBomba);

	if(simulado.JuegoTerminado()){
		if(simulado.RevisarTablero2() == miJugador){
			return FACTOR_VICTORIA;
		}else{
			return FACTOR_VICTORIA*(-1);
		}
	}
}
```

Si esto tampoco sucede esto, se analiza la partida mediante una reglas y configuraciones básicas del juego. Se comprueba el número de fichas consecutivas tanto del rival como del jugador. Por cada consecutividad de 4 fichas del rival, se multiplicará por un valor muy positivo (ya que cuantas más consecutivas tenga el rival, es mejor para el jugador). Por cada secuencia de tres fichas, se multiplicará por un valor medio. Si solo tiene dos se multiplicará por un factor relativamente bajo, ya que no es nada

beneficioso que el jugador tenga filas de dos, querrá decir que sus fichas probablemente se encuentren dispersas por el tablero. Se realiza el mismo cálculo para el jugador, pero de forma inversa. A más fichas consecutivas, menor será el valor que recibirá dicho análisis.

```
//El valor indica el número de fichas consecutivas que se quiere buscar
double altura4 = ValorConsecutivas(estado, miJugador, 4); //4 consecutivas
double altura3 = ValorConsecutivas(estado, miJugador, 3); //3 consecutivas
double altura2 = ValorConsecutivas(estado, miJugador, 2); //2 consecutivas

double rival4 = ValorConsecutivas(estado, rival, 4);
double rival3 = ValorConsecutivas(estado, rival, 3);
double rival2 = ValorConsecutivas(estado, rival, 2);
```

En definitiva, para el jugador, conseguir una altura mayor de fichas consecutivas será negativo, mientras que cuantas más tenga el rival, más positiva será.

Una vez se han obtenido dichos valores, se realizan las multiplicaciones pertinentes y se resta la heurística del rival con la del jugador, de forma que el valor positivo que obtegamos por parte del rival, se le restará a lo que penalice el estado del jugador. La heurística está planteada de forma que un buen movimiento es aquel que perjudica en gran medida al rival y apenas al jugador.

```
// Calculamos el valor heurístico de la mesa

resultadoHeuristico = (altura4*FACTOR4_JUGADOR + altura3*FACTOR3 + altura2*FACTOR2_JUGADOR);
resultadoRival = (rival4*FACTOR4_RIVAL + rival3*FACTOR3 + rival2*FACTOR2_RIVAL);
valorHeuristico = (resultadoHeuristico – resultadoRival);

return valorHeuristico;
```

Aunque se controla antes de realizar el cálculo heurístico si el juego se ha terminado, se podría pensar que no tiene sentido ver si algún jugador tiene 4 fichas consecutivas. Bien, tal y como está planteada la forma de calcular las fichas consecutivas tiene sentido.

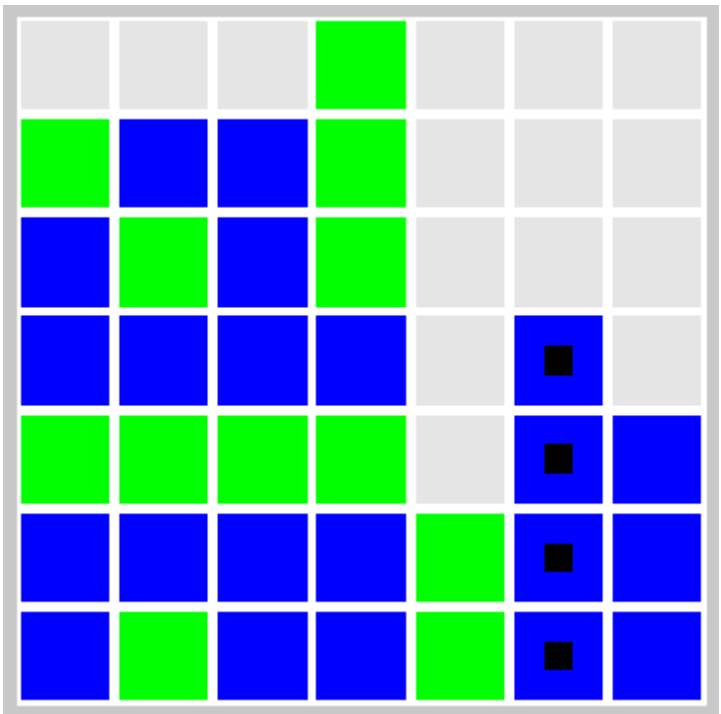
Además de hacer un cálculo normal para las fichas consecutivas (de la forma si la ficha(f,c) es igual a la ficha (f+1, c), por ejemplo), se ha controlado que si se rompe la consecutividad, antes de dejar de calcular si hay más fichas, se comprueba una serie de cosas.

En primer lugar, se comprueba si la secuencia de fichas se ha roto debido a que hay una ficha de otro color seguida. Si ocurre esto, se comprueba si encima de la ficha divisoria que hay entre estas existe una casilla del jugador. Si eso es así, el número de consecutivas no se rompe, sino que sigue sumando (es malo que haya fichas cerca del mismo color separadas por las del rival). Más negativo es aun si esto ocurre, y además el rival tiene una ficha bomba colocada en el tablero. Eso suma un plus de peligrosidad, ya que puede ocurrir que esa ficha bomba, al romperse haga que una ficha del jugador caiga, y formar así una linea de 4 fichas consecutivas y perder.

Se pondrá como ejemplo el caso de comprobar la continuidad de fichas en vertical

```
if(estado.See_Casilla(f, col) == rival){
        if(f+1 < FILAS_COLUMNAS){
        //Si la casilla donde se rompe la cadena es del jugador, pero el rival tiene una bomba, hay
        que tener cuidado
        if(estado.See_Casilla(f+1, col) == jugador){

                if(estado.Have_BOOM(rival)){
                        valoracion += FACTOR_BOMBAENEMIGACONSECUTIVAS;
                        consecutivas++; //Se contempla como que no se rompe la cadena
                }else{
                        valoracion += FACTOR_SUPERPUESTA;
                        break;
                }
        }
```



Implementación

Teniendo definido el algoritmo **Poda AlfaBeta**, y habiendo explicado la función que le da un valor a los nodos del algoritmo, denominada **Valoracion**, ya solo queda su implementación. Se hará de forma recursiva. Ella misma se irá llamando a sí misma hasta llegar a una límite establecido por el programador, una profundidad máxima, que, una vez haya llegado a ella, comenzará a realizar cálculos y compararlos con los ya obtenidos. De esta forma se asegura no solo tener en cuenta la mesa presente, con el desenlace de uno de los movimientos posibles en ese estado, si no que también se tienen en cuenta las posibles consecuencias durante un número de jugadas siguientes.

Lo ideal sería representar el árbol de la partida entero, desde el inicio hasta el fin, y elegir el mejor movimiento que asegurará la victoria. Pero como eso puede llegar a ser imposible, se genera un árbol de menor dimensión en cada estado actual del tablero, y a raíz de él, se analizan las posibles jugadas y sus respectivas consecuencias. No sólo atendiendo al jugador, si no también a los posibles movimientos del adversario.

Si es el turno del jugador, se intentará maximizar su ganancia.

Como se ha controlado el algoritmo iniciando en una profundidad 0 y avanzando hasta llegar a la profundidad máxima. Como en este caso el algoritmo recibe un jugador (que es el activo de la ronda) y el propio valor del jugador, cuando estas dos variables tengan el mismo valor, se considerará un nodo MAX, y se deberá MAXIMIZAR el resultado. En el momento en el que esa ganancia no la mejora el nodo que se evalua, esa rama del árbol se tomará como inaceptable y se “podará”. Para controlar esto se hará uso de las variables alfa y beta. Siendo alfa la mejor ganancia y beta la menor perdida.

```
if(jugador == miJugador){ //NODO MAX
	while(!(mesa==hijo)){
		if(maximizar(alfa, AlfaBeta(hijo, hijo.JugadorActivo(), miJugador, prof+1,
						prof_alfabeta, accion_anterior, alfa, beta))){
			//La mejor accion por ahora
			accion = static_cast <Environment::ActionType > (ult_act);
		}

		if(beta <= alfa){
			break;
		}

		hijo = mesa.GenerateNextMove(ult_act);

	}

	return alfa;
}
```

El método usado para crear un nuevo hijo del estado del juego se denomina *GenerateNextMove(int&ultimaAccion)*. Este método genera una situación en la que se ha aplicado la siguiente acción atendiendo a la que recibe. Es decir, si ha recibido la acción *PUT4*, entonces intentará hacer *PUT5*, y si no puede, *PUT6*, así hasta que pueda una. Si no puede ninguna acción, devolverá el mismo objeto que lo invoca. De esta forma, se puede aplicar en un bucle que mientras que esto no ocurra (que el hijo sea igual al padre) siga generando hijos.

Al llamar en la condición del *IF* de nuevo al método *AlfaBeta*, se genera la recursividad, aumentando la profundidad en uno.

En esa misma condición se hace uno de una función auxiliar “maximizar” que simplemente comprueba que si el valor que retorna el método *AlfaBeta* (su valor heurístico”) es mayor que **alfa**, entonces actualiza **alfa**.

Si en algún momento, antes de la generación de un nuevo hijo, ocurre que el valor de beta es menor o igual que alfa, entonces se cumple el criterio de poda y se termina de analizar esa rama del árbol (evitando de este modo analizar partes del árbol que no iban a interesar).

Si es el turno del adversario, se intentará buscar la pérdida mínima a causa de sus movimentos.

```
while(!(mesa==hijo)){
	if(minimizar(beta, AlfaBeta(hijo, hijo.JugadorActivo(), miJugador, prof+1,
					prof_alfabeta, accion_anterior, alfa, beta))){

		//La que menos perdidas genera
		accion = static_cast <Environment::ActionType > (ult_act);
	}

	if(beta <= alfa){
		break;
	}

	hijo = mesa.GenerateNextMove(ult_act);

}

return beta;
```

En el caso de ser un nodo MIN el proceso es inverso, y por ello se llama a la función auxiliar minimizar, que al contrario que la otra, esta actualiza el valor de beta.