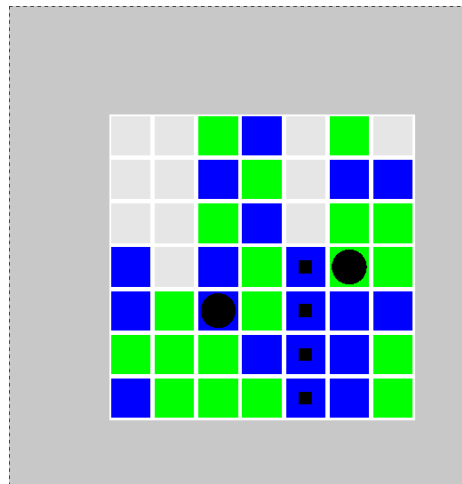


INTELIGENCIA ARTIFICIAL

E.T.S. de Ingenierías Informática y de
Telecomunicación

Práctica 3



Búsqueda con Adversario (Juegos)
CONECTA-4 BOOM

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN E INTELIGENCIA
ARTIFICIAL
UNIVERSIDAD DE GRANADA
Curso 2020-2021

1. Introducción

1.1. Motivación

La tercera práctica de la asignatura *Inteligencia Artificial* consiste en el diseño e implementación de alguna de las técnicas de búsqueda con adversario en un entorno de juegos. Al igual que en la práctica anterior, se trabajará con un simulador, pero en este caso adaptado para el juego **CONECTA-4** que pasamos a describir.

El juego **CONECTA-4 BOOM** se basa en el juego CONECTA-4 (también conocido como 4 en Raya). CONECTA-4 es un juego de mesa para dos jugadores distribuido por Hasbro, en el que se introducen fichas en un tablero vertical con el objetivo de alinear cuatro consecutivas de un mismo color. Fue creado en 1974 por Ned Strongin y Howard Wexler para Milton Bradley Company.



Para la realización de esta práctica, el alumno deberá conocer en primer lugar las técnicas de búsqueda con adversario explicadas en teoría. En concreto, **el objetivo de esta práctica es la implementación del algoritmo MINIMAX o del algoritmo de PODA ALFA-BETA**, para dotar de comportamiento inteligente deliberativo a un jugador artificial para este juego, de manera que esté en condiciones de competir y ganar a sus adversarios.

A continuación, explicamos cuáles son los requisitos de la práctica, los objetivos concretos que se persiguen, el software necesario junto con su instalación, y una guía para poder programar el simulador.

2. Requisitos

Para poder realizar la práctica, es necesario que el alumno disponga de:

1. Conocimientos básicos del lenguaje C/C++: tipos de datos, sentencias condicionales, sentencias repetitivas, funciones y procedimientos, clases, métodos de clases, constructores de clase.
2. El entorno de programación **CodeBlocks en el caso de trabajar bajo Sistema Operativo Windows**, que tendrá que estar instalado en el computador donde vaya a realizar la práctica. Este software se puede descargar desde la siguiente URL: <http://www.codeblocks.org/>. Es importante recordar que debe bajarse la **versión 17.12**. Para versiones más actuales, el software que se proporciona no es compatible.
3. **En el caso de trabajar bajo entorno Linux**, es necesario tener instalada la librería “GLUT3”, para los que tienen Ubuntu la instalación de esta biblioteca se hace poniendo la siguiente sentencia ‘**`sudo apt-get install freeglut3-dev`**’. Para otras versiones de Linux, buscar en internet como instalar esta librería¹.
4. El software para construir el simulador **Conecta4Boom** disponible en la plataforma docente de la asignatura.

3. Objetivo de la práctica

La práctica tiene como objetivo diseñar e implementar un agente deliberativo que pueda llevar a cabo un comportamiento inteligente dentro del juego **CONECTA-4 BOOM** que se explica a continuación.

El objetivo de **CONECTA-4 BOOM** es alinear cuatro fichas sobre un tablero formado por siete filas y siete columnas (en el juego original, el tablero es de seis filas). Cada jugador dispone de 25 fichas de un color (en nuestro caso, verdes y azules). Las jugadas entre los jugadores son alternas, empezando siempre el jugador 1, y en cada jugada el jugador hace dos movimientos, **excepto el jugador 1 en la primera jugada en el que sólo hace un**

¹ Para aquellos que hicieron la práctica anterior seguramente no sea necesaria esta instalación, ya que estás bibliotecas también eran requisito para ese software.

movimiento². Un movimiento consiste en introducir una ficha en la columna que prefiera (de la 1 a la 7, numeradas de izquierda a derecha, siempre que no esté completa) y ésta caerá a la posición más baja

Gana la partida **el primero que consiga alinear cuatro fichas consecutivas** de un mismo color en horizontal, vertical o diagonal. Si todas las columnas están ocupadas se produce un empate.

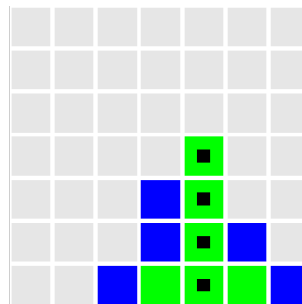


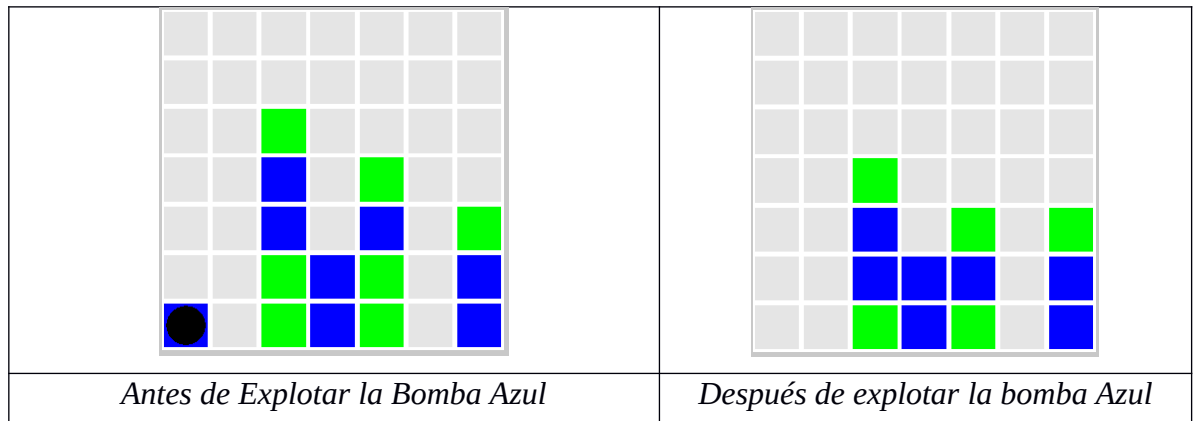
Figura 1. Imagen de una
partida de CONECTA-4
BOOM.

CONECTA-4 BOOM mantiene todas las normas del juego habitual del 4 en raya con una variante: *cada cuatro jugadas, es decir, en el cuarto turno de cada jugador*³, se coloca una ficha especial de su color que llamaremos “**ficha bomba**” (esto es en los turnos, 4, 8, 12, ...), pudiendo tener como máximo cada jugador una ficha de este tipo en el tablero (es decir, que si llega el turno 8 y el jugador al que le toca poner tiene ya una ficha bomba, la ficha que se pone en el tablero es una ficha normal). La ficha bomba, al igual que el resto de las fichas del jugador, sirven para confeccionar una posible alineación de 4 fichas para ganar el juego, pero tiene la peculiaridad de que el jugador la puede “**explotar**” en uno de los movimientos de su jugada. **Si ninguno de los dos jugadores consigue alinear las cuatro piezas de su color, perderá aquel que le toque hacer su jugada y no pueda realizar el movimiento.**

¿Cómo se explota una ficha? El jugador está en situación de jugar y tiene una ficha bomba colocada en el tablero. En este caso, tiene una acción adicional que consiste en explotarla. Esta acción consume uno de los movimientos del jugador en su jugada.

¿Qué efecto produce la explosión? La explosión elimina la propia ficha bomba y las fichas que están **en la misma fila que son del adversario**. Las fichas situadas encima de las casillas afectadas caerían por gravedad hasta situarse en sus posiciones estables.

- 2 La secuencia de juego será: empieza el jugador 1 con el movimiento 1, después el jugador 2 hará los movimientos 2 y 3, después el jugador 1 los movimientos 4 y 5, y así sucesivamente.
- 3 Para el primer jugador la cuarta jugada o turno corresponde con el movimiento 12. Para el segundo jugador es el movimiento 14.



OBJETIVO DE LA PRÁCTICA

A partir de estas consideraciones iniciales, el objetivo de la práctica es implementar **MINIMAX (con profundidad máxima de 6)** o **PODA ALFA-BETA (con profundidad máxima de 8)**, de manera que un jugador pueda determinar el movimiento más prometedor para ganar el juego, explorando el árbol de juego **desde** el estado actual **hasta** una profundidad máxima de 8 dada como entrada al algoritmo.

También forma parte del objetivo de esta práctica, la definición de una heurística apropiada, que asociada al algoritmo implementado proporcione un buen jugador artificial para el juego del **CONECTA-4 BOOM**.

Los conceptos necesarios para poder llevar a cabo la implementación del algoritmo dentro del código fuente del simulador se explican en las siguientes secciones.

4. Instalación y descripción del simulador

4.1. Instalación del simulador

El simulador **CONECTA-4 BOOM** nos permitirá

- implementar el comportamiento de uno o dos jugadores en un entorno en el que el jugador (bien humano o bien máquina) podrá competir con otro jugador software o con otro humano.
- visualizar los movimientos decididos en una interfaz de usuario basado en ventanas.

Para instalarlo, seguir estos pasos:

- Descargue el software desde la plataforma docente PRADO, y cópielo en la carpeta de trabajo.
- Desempaquete el fichero en la raíz de esta carpeta.
- En Windows, el siguiente paso es compilar el proyecto “**Conecta4Boom.cbp**” en el entorno **CodeBlocks**, en el caso de Linux, acceder a la carpeta creada y ejecutar “**make**”.

4.2. Ejecución del simulador

Una vez compilado el simulador y tras su ejecución debe aparecernos la siguiente ventana:

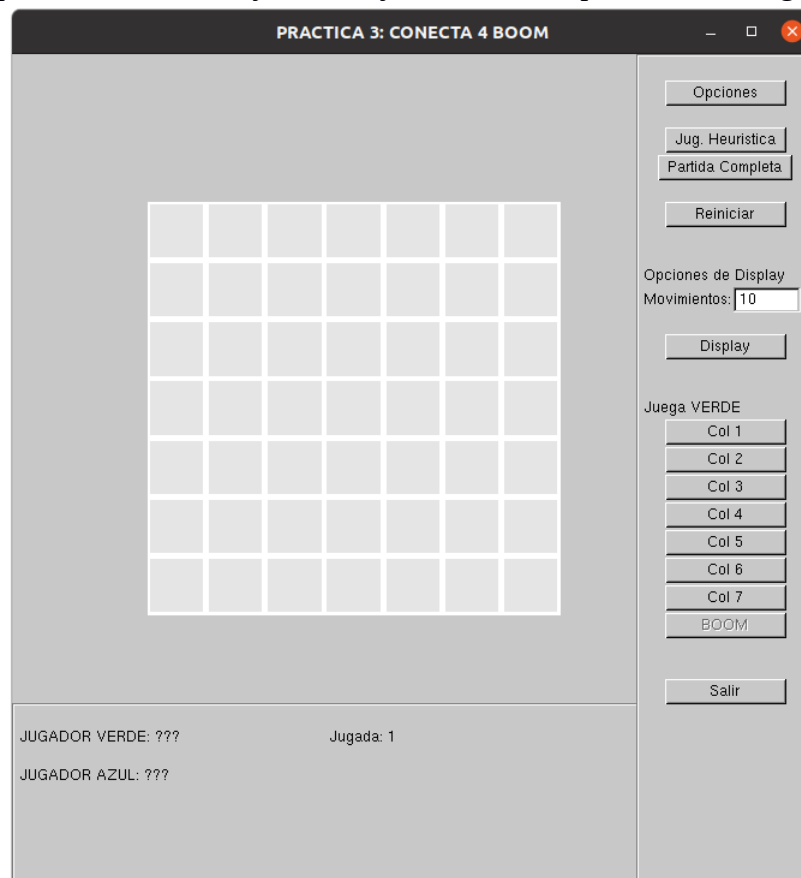


Figura 2. Ventana Principal del Simulador

La primera vez que se entra se pulsará el botón “**Opciones**”, que nos permite elegir el modo de juego. Una vez pulsado “**Opciones**” aparece una nueva ventana en la que podremos configurar la partida a jugar.

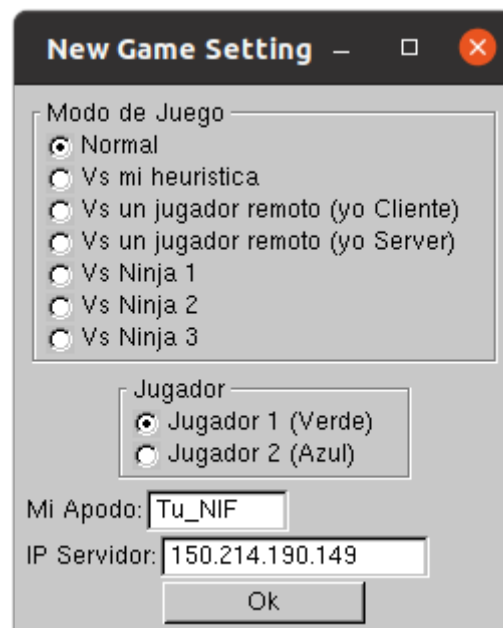


Figura 3. Ventana de Opciones.

Las opciones configurables en el juego son las siguientes:

- “**Modo de Juego**”: Establece la forma en la que se va a comportar el simulador. Se puede elegir entre 5 modos diferentes:
 - o “*Normal*”: Es el modo por defecto y dota al simulador de la máxima flexibilidad. Si al entrar por primera vez al simulador no se pulsa “Opciones”, es en este modo en el que opera.
 - o “*Vs mi heurística*”: En este modo un jugador humano juega contra la heurística que el mismo ha programado en movimientos alternos.
 - o “*Vs un jugador remoto (yo Server)*”: En este modo se permite jugar por red con otro adversario. Obviamente, en este modo se requiere que un compañero tenga levantado su simulador, elija el modo *Vs un jugador remoto (yo Cliente)* y ponga en *IP Servidor* la dirección IP de la máquina donde se encuentra el compañero en el modo servidor.
 - o “*Vs un jugador remoto (yo Cliente)*”: El complementario de lo descrito justo anteriormente para jugar con un compañero en red. En este modo


existe una posibilidad más. Dos jugadores pueden jugar una partida estando en este modo, si ponen como *IP Servidor*, la dirección que aparece en la imagen. En esa máquina hay puesto un servidor que admite conexiones de clientes que quieren jugar partidas en red.

- “Vs Ninja 1”, “Vs Ninja 2”, “Vs Ninja 3” : Este es un modo especial de juego en red contra distintos jugadores automáticos . Se puede utilizar este modo para evaluar como de buena es la heurística que se ha implementado.
- “**Jugador**”: Decides que jugador quieres ser. El jugador 1 siempre juega primero.
- “**Mi Apodo**”: Un nombre que te identifique cuando juegas en red
- “**IP Servidor**”: La dirección IP del servidor contra el que quieres jugar en red.

Tras elegir el modo, se vuelve a la ventana principal del simulador (Figura 2). Dentro de esta ventana, tendremos opciones que se activarán o desactivarán en función del modo de juego elegido. En el caso del modo “Normal”, todas las opciones estarán disponibles.

Las opciones posibles son las siguientes:

- **Jug. Heurística**: Le pedimos al simulador que la siguiente jugada la calcule tomando la función heurística que nosotros hemos implementado.
- **Partida Completa**: La heurística que nosotros hemos implementado juega una partida completa contra ella misma.
- **Reiniciar**: Reinicia el juego.
- **Display**: Ejecuta el número de jugadas que se indica en el campo **Pasos**
- **Col 1 a Col 7**: De forma manual se selecciona la siguiente jugada. Col 1 indica que se quiere colocar una ficha en la primera columna (la columna más a la izquierda) y Col 7 en la última columna (la columna más a la derecha).
- **BOOM**: De forma manual se selecciona explotar la ficha bomba.
- **Salir**: Abandona el programa.



The image shows a vertical menu with the following elements from top to bottom:

- A button labeled "Opciones".
- Two buttons: "Jug. Heuristica" and "Partida Completa".
- A button labeled "Reiniciar".
- A section titled "Opciones de Display" containing a label "Movimientos:" followed by a text input field containing the number "10".
- A button labeled "Display".
- A section titled "Juega VERDE" containing a vertical stack of buttons labeled "Col 1", "Col 2", "Col 3", "Col 4", "Col 5", "Col 6", "Col 7", "BOOM", and "Salir".

En la parte inferior de la ventana de simulación aparece información relativa a la evolución del juego, en concreto, el último movimiento de cada uno de los jugadores, y en el caso de estar jugando en red, si tienes el turno o estas esperando a que juegue tu rival.

En la siguiente sección se explica el contenido de los ficheros fuente y los pasos a seguir para poder construir la práctica

5. Pasos para construir la práctica

En esta sección se explica de forma resumida el contenido de los siguientes ficheros fuente necesarios para poder implementar adecuadamente los métodos objeto de esta práctica:

- ***Environment.h y cpp***, donde se implementa la clase ***Environment*** usada para representar los diferentes estados del juego.
- ***Player.h y cpp***, donde se implementa la clase ***Player*** usada para representar a cada uno de los dos jugadores.
- ***GUI.h y cpp***, donde se implementan algunas utilidades necesarias para la ejecución correcta del juego y la visualización de los movimientos de los jugadores en el tablero de juego.
- ***Conexión.h y cpp***, donde se implementa la funcionalidad para jugar en red.

De estos ficheros, los relevantes para hacer la práctica son “***Environment***” y “***Player***”, que pasamos a describir más detenidamente a continuación:

5.1. Representación de los estados del juego (Clase **Environment**)

Los estados del juego están representados con la clase **Environment**, definida en el fichero **environment.h** e implementada en el fichero **environment.cpp**.

El estado viene definido por los datos miembro que se muestran en la figura

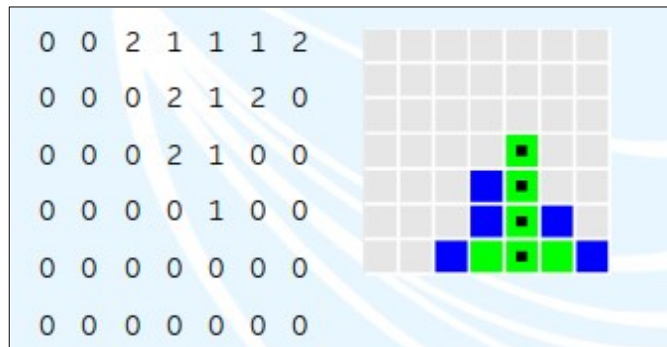
```
114 private:
115
116     // Tamano del mapa (Siempre el numero de filas (X) es igual al numero de columnas (Y)
117     int MAZE_SIZE;
118
119     // Matriz que codifica el tablero
120     char **maze_;
121
122     // Indica la ocupacion de cada columna
123     char *tope_;
124
125     // Ultimas acciones realizadas por los jugadores.
126     int last_action1_, last_action2_;
127
128     // Jugador al que le toca realizar el siguiente movimiento
129     int jugador_activo_;
130
131     // Cantidad de casillas que aun quedan libres.
132     int casillas_libres_;
133
134     // Posibilidad de explotar una ficha
135     bool explosion[3];
136
137     // Numero de jugada
138     int n_jugada;
139
140     // Numero de movimiento
141     int n_movimiento;
142
143     // Cada cuanto se puede poner una bomba
144     int n_veces;
```

donde

- **MAZE_SIZE** define el tamaño del tablero que es cuadrado.
- **maze** es la matriz que representa el tablero de juego. Las componentes de esa matriz pueden tomar los valores {0,1,2,4,5} donde
 - 0 representa que la casilla está vacía.
 - 1 y 2 que en esa casilla hay una pieza del jugador 1 o del jugador 2 respectivamente.
 - 4 y 5 que en la casilla hay una ficha bomba del jugador 1 y del jugador 2 respectivamente.

Una consideración importante sobre esta matriz es que cuando se pinta sobre el

simulador, las filas se invierten. Para ilustrarlo mirar en la siguiente imagen a que corresponde la codificación de la matriz.



- `tope_` son vectores que indican el nivel de ocupación de cada columna.
- `last_action1_` y `last_action2_` representan las dos últimas acciones realizadas por el jugador 1 y jugador 2 respectivamente.
- `jugador_activo_` toma el valor 1 o 2 e indica a que jugador le corresponde hacer la jugada sobre ese tablero.
- `casillas_libres_` indica el número de casillas en el tablero que aún están libres.
- `explosion`, es un vector de 3 componentes de booleanos e indica si el jugador 1 (`explosion[1]`) o el jugador 2 (`explosion[2]`) tienen alguna ficha bomba en el tablero.
- `n_jugada` indica el turno en el que se encuentra el jugador que juega.
- `n_movimiento` indica el número de movimientos al que corresponderá el que se realice sobre ese tablero.
- `n_veces` establece cada cuantas (`n_jugada`) se puede poner una nueva ficha bomba. En la práctica actual ese valor se ha fijado en 4.

A continuación, se describen los métodos esenciales de esta clase, para la comprensión y la elaboración de la práctica.

Métodos Destacables de la Clase `Environment`

`int GenerateAllMoves(Environment *V) const;`

Este método genera todas las situaciones resultantes de aplicar todas las acciones sobre el tablero actual para el jugador que le toca jugar. Cada nuevo tablero se

almacena en “V”, un vector de objetos de la clase Environment. El método devuelve el tamaño de ese vector, es decir, el número de movimientos posibles.

Environment GenerateNextMove(int &act) const;

Este método genera el siguiente movimiento que se puede realizar el jugador al que le toca jugar sobre el tablero actual devolviéndolo como un objeto de esta misma clase. El parámetro "act" indica cual fue el último movimiento que se realizó sobre el tablero. El método asume el siguiente orden en la aplicación de las acciones: 0 PUT_1, 1 PUT_2, ..., 6 PUT_7, 7 BOOM. Si no hay un siguiente movimiento, se devuelve como tablero el actual.

La primera vez que se invoca en un nuevo estado se le pasa como argumento en act el valor -1.

int possible_actions(bool *VecAct) const;

Devuelve el número de acciones que puede realizar el jugador al que le toca jugar sobre el tablero. "VecAct" es un vector de datos lógicos que indican si una determinada acción es aplicable o no. Cada componente del vector está asociada con una acción. Así, la [0] indica si PUT 1 es aplicable, [1] si lo es PUT2, y así sucesivamente hasta [7] que indica si BOOM es aplicable.

int Last_Action(int jug) const;

Indica la última acción que se aplicó para llegar a la situación actual del tablero. El entero que se devuelve es el ordinal de la acción.

int JugadorActivo()const;

Devuelve el jugador al que le toca jugar, siendo 1 el jugador Verde y 2 el jugador Azul.

int Get_Ocupacion_Columna(int columna);

Indica el nivel de ocupación de una determinada columna del tablero, un valor entre 0 y 7 donde 0 indica que la columna está vacía, y 7 que la columna está llena.

bool Have_BOOM (int jugador) const;

Devuelve verdadero si “jugador” tiene una ficha bomba en el tablero.

void ChangePlayer();

Cambia el jugador activo.

int N_Jugada() const;

Devuelve el número de jugada del tablero actual. Recordar que cada jugada tiene dos movimientos excepto la primera para el jugador 1 que sólo tiene un movimiento.

bool Put_FichaBOOM_now() const;

Devuelve verdadero si le toca al jugador activo colocar una ficha bomba en esta jugada.

char See_Casilla(int row, int col) const;

Devuelve lo que hay en el tablero en la fila "row" columna "col":
0 vacía,
1 jugador1,
2 jugador2,
4 ficha bomba del jugador1,
5 ficha bomba del jugador 2.

bool JuegoTerminado()const;

Devuelve verdadero cuando el juego ha terminado.

int RevisarTablero() const;

Cuando el juego está terminado devuelve quien ha ganado:
0 Empate.
1 Gana Jugador 1.
2 Gana Jugador2.

5.2. Representación de los jugadores (Clase Player)

La clase Player se utiliza para representar un jugador (es la clase equivalente a ComportamientoJugador en la anterior práctica).

```
1  #ifndef PLAYER_H
2  #define PLAYER_H
3
4  #include "environment.h"
5
6  class Player{
7  public:
8      Player(int jug);
9      Environment::ActionType Think();
10     void Perceive(const Environment &env);
11 private:
12     int jugador_;
13     Environment actual_;
14 };
15 #endif
16
```

Contiene dos variables privadas

- **jugador_** (un entero representando el número de jugador, que puede ser 1 (el jugador verde) o 2 (el jugador azul) y
- **actual_** (variable tipo **Environment** que codifica el Tablero sobre el que tendrá que realizar su movimiento “jugador_”).

Contiene dos métodos:

Think(), que implementa el proceso de decisión del jugador para escoger la mejor jugada y devuelve una acción (clase **Environment::ActionType**) que representa el movimiento decidido por el jugador.

Perceive(const Environment &env), que implementa el proceso de percepción del jugador y que permite acceder al estado actual del juego que tiene el jugador. **ESTE MÉTODO NO PUEDE SER MODIFICADO!!!!**.

IMPORTANTE:

La invocación al MINIMAX o la PODA ALFA-BETA se debe hacer dentro del método Think()

Todos los recursos necesarios para poder implementar un proceso de búsqueda con adversario se suministran fundamentalmente en la clase **Environment**. Podrán definirse los métodos que el alumno estime oportunos, pero tendrán que estar implementados en el fichero **Player.cpp**.

5.3. Versión Inicial del método think()

El estudiante al compilar por primera vez el software de la práctica se encontrará con la implementación en el método think() de un jugador aleatorio. La idea es ilustrar de forma práctica el uso de algunas de las funciones descritas en apartados anteriores.

```
84 // Invoca el siguiente movimiento del jugador
85 Environment::ActionType Player::Think(){
86     const int PROFUNDIDAD_MINIMAX = 6; // Umbral maximo de profundidad para el metodo MiniMax
87     const int PROFUNDIDAD_ALFABETA = 8; // Umbral maximo de profundidad para la poda Alfa_Beta
88
89     Environment::ActionType accion; // acción que se va a devolver
90     bool aplicables[8]; // Vector bool usado para obtener las acciones que son aplicables en el estado actual. La interpretacion es
91         // aplicables[0]==true si PUT1 es aplicable
92         // aplicables[1]==true si PUT2 es aplicable
93         // aplicables[2]==true si PUT3 es aplicable
94         // aplicables[3]==true si PUT4 es aplicable
95         // aplicables[4]==true si PUT5 es aplicable
96         // aplicables[5]==true si PUT6 es aplicable
97         // aplicables[6]==true si PUT7 es aplicable
98         // aplicables[7]==true si BOOM es aplicable
99
100
101
102     double valor; // Almacena el valor con el que se etiqueta el estado tras el proceso de busqueda.
103     double alpha, beta; // Cotas de la poda AlfaBeta
104 }
```

En una descripción breve de este método, podemos ver en la parte superior la declaración de los datos que se usarán en la implementación de la función. Entre ellos tenemos las constantes **PROFUNDIDAD_MINIMAX = 6** y **PROFUNDIDAD_ALFABETA = 8**. Estas serán las profundidades máximas que podrá elegir el estudiante en cada caso. Es decir, que si implementa Minimax la profundidad máxima que podrá poner será 6, siendo 8 la que se puede usar en el caso de tener implementada la poda AlfaBeta.

También podemos ver la variable **accion** que se usará para devolver la siguiente acción a realizar por el jugador. **aplicables** es un vector de booleanos que se usará para determinar que acciones son posibles realizar sobre el tablero actual. En los comentarios asociados se refleja la interpretación de cada valor.

Por último, **valor**, **alpha** y **beta** serán variables que se utilizarán en la invocación del algoritmo de búsqueda.

Departamento de Ciencias de la Computación e Inteligencia Artificial

```
105     int n_act; //Acciones posibles en el estado actual
106
107
108     n_act = actual_.possible_actions(aplicables); // Obtengo las acciones aplicables al estado actual en "aplicables"
109     int opciones[10];
110
111     // Muestra por la consola las acciones aplicable para el jugador activo
112     //actual_.PintaTablero();
113     cout << " Acciones aplicables ";
114     (jugador_==1) ? cout << "Verde: " : cout << "Azul: ";
115     for (int t=0; t<8; t++)
116         if (aplicables[t])
117             cout << " " << actual_.ActionStr( static_cast< Environment::ActionType > (t) );
118     cout << endl;
```

Después de la ejecución de la línea 108, en **n_act** tendremos el número de descendientes que se puede obtener del tablero **actual_**.

Las líneas de la 111 a la 118 sólo muestran por el terminal cuales son las acciones que realmente se pueden aplicar y a que jugador le tocaría hacer el movimiento.

```
121     //----- COMENTAR Desde aqui
122     cout << "\n\t";
123     int n_opciones=0;
124     JuegoAleatorio(aplicables, opciones, n_opciones);
125
126     if (n_act==0){
127         (jugador_==1) ? cout << "Verde: " : cout << "Azul: ";
128         cout << " No puede realizar ninguna accion!!!\n";
129         //accion = Environment::actIDLE;
130     }
131     else if (n_act==1){
132         (jugador_==1) ? cout << "Verde: " : cout << "Azul: ";
133         cout << " Solo se puede realizar la accion "
134             << actual_.ActionStr( static_cast< Environment::ActionType > (opciones[0]) ) << endl;
135         accion = static_cast< Environment::ActionType > (opciones[0]);
136     }
137     else { // Hay que elegir entre varias posibles acciones
138         int aleatorio = rand()%n_opciones;
139         cout << " -> " << actual_.ActionStr( static_cast< Environment::ActionType > (opciones[aleatorio]) ) << endl;
140         accion = static_cast< Environment::ActionType > (opciones[aleatorio]);
141     }
142
143
144     //----- COMENTAR Hasta aqui
```

El código que va desde la línea 121 a la 144 representa el comportamiento del jugador aleatorio. En base al valor de **n_act** que indica las acciones que se pueden realizar sobre este tablero se toman decisiones. Si **n_act == 0** entonces el jugador pierde ya que no puede realizar ninguna acción. Si **n_act == 1** entonces sólo una acción puede hacer el jugador y aplica directamente esa acción. En otro caso, hay varias acciones aplicables y en ese caso selecciona una de ellas aleatoriamente (línea 139).

Por su peculiaridad, hay que indicar que en este código se juega con la doble posibilidad de trabajar con un dato enumerado o bien usando la definición propia o bien aprovechando

Departamento de Ciencias de la Computación e Inteligencia Artificial

que internamente cada elemento del enumerado está codificado con un número entero. El uso de `static_cast<>` es la forma de poder transformar estos tipos entre sí.

```
147 //----- AQUÍ EMPIEZA LA PARTE A REALIZAR POR EL ALUMNO -----
148
149
150 // Opcion: Poda AlfaBeta
151 // NOTA: La parametrizacion es solo orientativa
152 // valor = Poda AlfaBeta(actual_, jugador_, 0, PROFUNDIDAD_ALFABETA, accion, alpha, beta);
153 //cout << "Valor MiniMax: " << valor << " Accion: " << actual_.ActionStr(accion) << endl;
154
155 return accion;
156 }
```

La última parte del método `think()` está comentada, y tendrá que ser descomentada cuando el estudiante haga su implementación del Minimax o de la Poda Alfabeta. En ese momento, se comentará en su lugar la parte del código relativa al jugador aleatorio (marcada por las dos señales de -----COMENTAR desde aquí, y ---- COMENTAR hasta aquí).

La parametrización que se propone de la poda AlfaBeta es meramente indicativa. El estudiante es libre de decidir los parámetros que desea utilizar para realizar la búsqueda.

La primera tarea del estudiante en esta práctica es familiarizarse con el código y hacer una implementación de uno de los dos algoritmos de búsqueda. Para que resulte más fácil y el estudiante en este primer instante sólo piense en el algoritmo, se le ofrece una heurística para que la pueda utilizar, llamada **ValoracionTest**.

```
40 // Funcion de valoracion para testear Poda Alfabeta
41 double ValoracionTest(const Environment &estado, int jugador){
42     int ganador = estado.RevisarTablero();
43
44     if (ganador==jugador)
45         return 99999999.0; // Gana el jugador que pide la valoracion
46     else if (ganador!=0)
47         return -99999999.0; // Pierde el jugador que pide la valoracion
48     else if (estado.Get_Casillas_Libres()==0)
49         return 0; // Hay un empate global y se ha rellenado completamente el tablero
50     else
51         return Puntuacion(jugador,estado);
52 }
```

Una vez verificado que el algoritmo implementado funciona correctamente, en la segunda tarea el estudiante tiene que definir una heurística propia (distinta de `ValoracionTest`) que permita hacer un jugador automático que juegue lo mejor posible contra otros compañeros y en especial contra los jugadores automáticos que se proponen junto al software. Para la confección de dicha función heurística por parte del estudiante se ha dejado una función sólo con la cabecera llamada `Valoracion`. Se debe completar esta función para

establecer la heurística del estudiante. Como pasaba con el método de búsqueda, en este caso, la parametrización de la función es sólo orientativa y el estudiante es libre de hacer la parametrización que crea oportuna.

Importante: En ningún caso puede eliminarse ni alterarse la función **ValoracionTest**. Debe permanecer tal y como está cuando se produzca la entrega de la práctica. Esta función heurística será utilizada para validar el correcto funcionamiento de la implementación de la función de búsqueda realizada por el estudiante.

6. Evaluación y entrega de prácticas

La **calificación final** de la práctica se calculará de la siguiente forma:

- Se entregará una memoria de prácticas (ver apartado 6.1 de este guion) al finalizar las tareas a realizar. La fecha límite de la entrega de la memoria será comunicada con suficiente antelación por el profesor de prácticas en clase, y publicada en la página web de la asignatura.
- La práctica se califica numéricamente de **0 a 10**. Se evaluará como la suma de los siguientes criterios:
 - La memoria de prácticas se evalúa de **0 a 4**.
 - La eficacia del algoritmo se evaluará de **0 a 6** puntos y estará basado en competir frente a tres jugadores ninja tanto de jugador 1 como de jugador 2. Con estas 6 partidas (3 como primer jugador y 3 como segundo jugador) se definirá la capacidad de la heurística desarrollada por el estudiante. La calificación será de **un punto por cada victoria**.
 - La nota final será la suma de los dos apartados anteriores. Es requisito necesario entregar tanto el software como la memoria de prácticas. Si alguna de ellas falta en la entrega se entenderá por no entregada.
- La **fecha de entrega de la práctica**

Miércoles 9 de junio antes de las 23:00 horas.



6.1. Restricciones del software a entregar y representación.

Se pide desarrollar un programa (modificando el código de los ficheros del simulador **player.cpp**, **player.h**) que implemente el algoritmo MINIMAX o la PODA ALFA-BETA en los términos en que se ha explicado previamente. Estos ficheros deberán entregarse en la plataforma docente PRADO, en un fichero ZIP que NO contenga carpetas separadas, es decir, todos los ficheros aparecerán en la carpeta donde se descomprima el fichero ZIP. **No se evaluarán aquellas prácticas que contengan ficheros ejecutables o virus.**

El fichero ZIP debe contener una **memoria de prácticas** en formato PDF (no más de 5 páginas) que, como mínimo, contenga los siguientes apartados:

1. Análisis del problema
2. Descripción de la solución planteada