Universidad de Granada
Departamento de Ciencias de la
Computación e Inteligencia Artificial

INTELIGENCIA ARTIFICIAL

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

Práctica 3

Métodos de Búsqueda con Adversario (Juegos)

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL
UNIVERSIDAD DE GRANADA
Curso 2013-2014

Universidad de Granada

Departamento de Ciencias de la
Computación e Inteligencia Artificial

1. Introducción

1.1. Motivación

La tercera práctica de la asignatura ***Inteligencia Artificial*** consiste en el diseño e implementación de técnicas de búsqueda con adversario en un entorno de juegos. Al igual que en la práctica anterior, se trabajará con una versión modificada del simulador software que implementa una aspiradora, en ese caso adaptada para el juego CONECTA-4. Este simulador fue desarrollado inicialmente por el profesor Tsung-Che Chiang de la NTNU (Norwegian University of Science and Technology, Taiwán).

El entorno de simulación se ha adaptado para simular el juego CONECTA-4. CONECTA-4 (también conocido como 4 en Raya en algunas versiones) 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 (Tema 4). En concreto, el objetivo de esta práctica es la implementación de un algoritmo MINIMAX con PODA ALPHA-BETA, con profundidad limitada (con cota máxima 8), para dotar de comportamiento inteligente deliberativo a un jugador artificial para este juego, de manera que esté en condiciones de competir y ganar a su adversario.

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.

Universidad de Granada
Departamento de Ciencias de la
Computación e Inteligencia Artificial

2. Requisitos

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

- Conocimientos básicos del lenguaje C/C++: tipos de datos, sentencias condicionales, sentencias repetitivas, funciones y procedimientos, clases, métodos de clases, constructores de clase.
- El entorno de programación **CodeBlocks** (también es válida la alternativa del entorno **Dev-C++**), 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/** o desde la web de la asignatura facilitada por el profesor.
- El entorno de simulación **4raya**, disponible en la web de la asignatura.
- Las bibliotecas adicionales **libopengl32.a**, **libglu32.a**, **libglut32.a**, disponibles en la web de la asignatura.

La guía de instalación del software previamente mencionado puede consultarse en el guión de la práctica 2.

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 que se explica a continuación.

El objetivo de CONECTA-4 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). Por turnos, los jugadores deben introducir una ficha en la columna que prefieran (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 llenas pero nadie ha conseguido alinear 4 fichas de su color, entonces se produce empate.

Universidad de Granada

Departamento de Ciencias de la Computación e Inteligencia Artificial

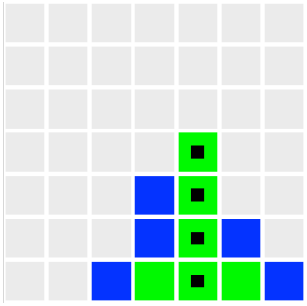


Figura 1. Imagen de una partida de CONECTA-4.

OBJETIVO DE LA PRÁCTICA:

A partir de estas consideraciones iniciales, el objetivo de la práctica es implementar un algoritmo MINIMAX con PODA ALPHA-BETA, **con profundidad limitada (con cota 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 CONECTA4.

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 **4Raya** 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 basada en ventanas.

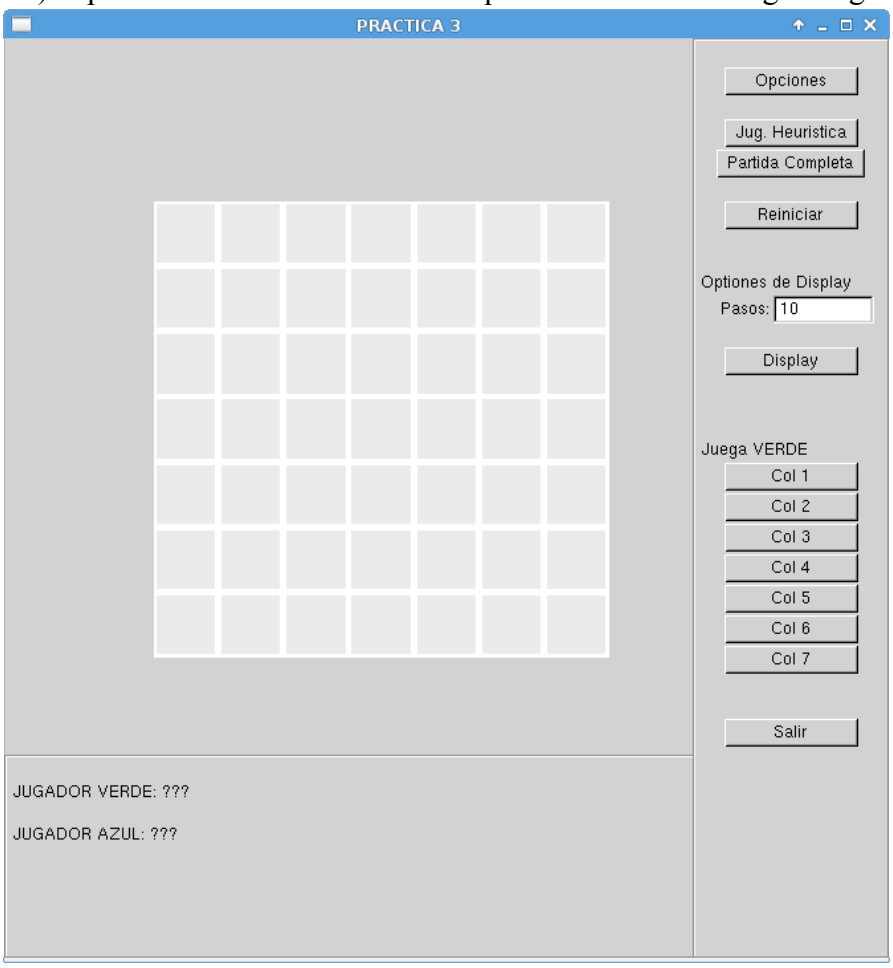
Para instalarlo, seguir estos pasos:

1. Descargue el fichero **4Raya.zip** desde la web de la asignatura, y cópielo en su carpeta personal dedicada a las prácticas de la asignatura de ***Inteligencia Artificial***. Supongamos, para los siguientes pasos, que esta carpeta se denomina "**U:\IA\practica3**".
2. Desempaquete el fichero en la raíz de esta carpeta.
3. Ya está instalado el simulador. A continuación, el siguiente paso es compilar el proyecto "**4Raya.cbp"** en el entorno **CodeBlocks**.

4.2. Ejecución del simulador

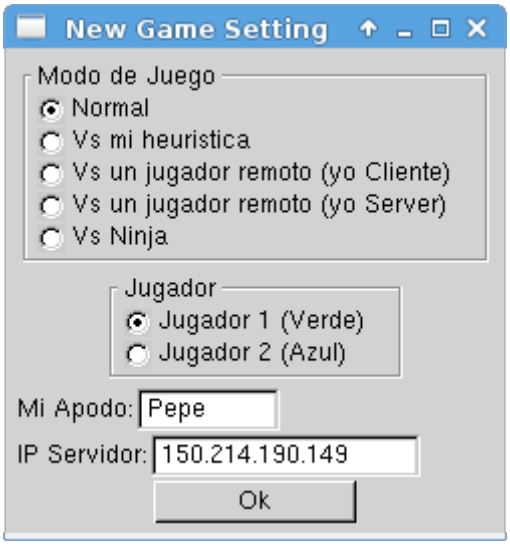
Tomamos la opción de abrir un proyecto existente y elegimos el proyecto ***"4Raya.cbp"*** y lo compilamos.

Una vez compilado el proyecto del simulador, para ejecutarlo pulsaremos sobre la opción "**Run**" del menú **"Build**" (alternativamente, también podemos hacer doble clic sobre el programa **4enRaya.exe** generado en la carpeta del proyecto tras su compilación). Aparecerá una ventana como la que se muestra en la figura siguiente.



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.
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:



  - "***Normal***": Es el modo por defecto y dota al simulador de la máxima flexibilidad y todas las opciones están permitidas, lo que significa que en cada turno de juego el movimiento a realizar puede hacerse bien por la heurística definida o bien por un jugador humano. Si al entrar por primera vez al simulador no se pulsa "Opciones", el programa trabajará por defecto en este modo.
  - "***Vs mi heurística***": En este modo se van alternando turnos entre un jugador humano y la heurística programada. El orden en que juega el humano se define en el cuadro de opciones **Jugador.**
  - "***Vs un jugador remoto (yo Server)***": En este modo se permite jugar por red con otro adversario. 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.
  - "***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 IP corre un servidor que admite conexiones de clientes que quieren jugar partidas en red.
  - "**Vs Ninja**": Este es un modo especial de juego en red contra un jugador automático que tiene implementada una buena heurística. Se puede utilizar este modo para comparar como de buena es la heurística que uno 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).
- **Salir**: Abandona la aplicación

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 en detalle el contenido de los siguientes ficheros fuente, necesarios para poder implementar adecuadamente el algoritmo objeto de la 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:

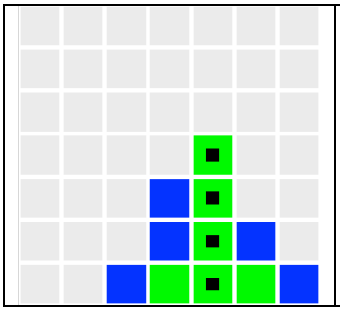
5.1. Representación de los estados del juego (Clase Environment)

Los estados del juego están representados con la clase Environment, definida en el fichero environment.h e implementada en el fichero environment.cpp. A continuación se describen las variables y métodos esenciales de esta clase, para la comprensión y la elaboración de la práctica:

Datos privados de la Clase Environment

- ***int MAZE_SIZE:*** Tamaño del mapa
- ***char **maze_:*** Matriz de 7x7 que representa el tablero de juego, dónde 0 representa que la casilla está vacía, 1 que hay una ficha del jugador 1 y 2 que hay una ficha del jugador 2. IMPORTANTE: son valores numéricos, es decir, no es el carácter '0','1' y '2', sino el valor entero 0, 1, 2.
- ***char *tope_:*** Vector de 7 componentes que indica el número de piezas ya puestas en cada columna del tablero.
- ***int last_action1_, last_action2_:*** Almacena la última acción realizada por cada jugador.
- ***int jugador_activo_:*** Indica que jugador está en posesión del turno para jugar.
- ***int casillas_libres_:*** Indica el número de casillas libres que aún quedan en el tablero.

NOTA: La matriz 7x7 que representa el tablero (maze_), codifica el tablero de forma que la fila cero está en la fila superior, es decir, en la forma normal en la que se expresan las matrices, aunque a la hora de ser pintada por el simulador, esa matriz se invierte SOLO para ser mostrada. Por ejemplo, la representación interna del tablero que se muestra a la izquierda, es la matriz del entorno que se muestra a la derecha en la siguiente imagen:



0	0	2	1	1	1	2
0	0	0	2	1	2	0
0	0	0	2	1	0	0
0	0	0	0	1	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

Universidad de Granada

Departamento de Ciencias de la
Computación e Inteligencia Artificial

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 esta misma clase. 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 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. Este método asume el siguiente orden en la aplicación de las acciones: 0 PUT 1, 1 PUT2, …, 6 PUT 7. Si no hay un siguiente movimiento, el método 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 número de acciones que puede realizar el jugador al que le toca jugar sobre el tablero. "VecAct" es un vector de datos lógicos de tamaño 7 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 PUT 2, y así sucesivamente.

- ***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.

- ***string ActionStr(ActionType action)***: Expresa en una cadena de caracteres un dato del tipo enumerado *"ActionType*" que se pasa como argumento.

- ***int JugadorActivo()***: Devuelve el jugador al que le toca jugar, siendo 1 el jugador Verde y 2 el jugador Azul.

- ***int Get_Ocupacion_Columna(int columna) const***: Indica el nivel de ocupacion de una determinada columna.

- ***int Get_Casillas_Libres() const***: Devuelve el número de casillas libres que quedan en el tablero.

- ***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.

- ***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 Agent 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 representa el estado actual del entorno para un jugador dado).

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.

Universidad de Granada

Departamento de Ciencias de la Computación e Inteligencia Artificial

IMPORTANTE**:**
La implementación del algoritmo MINIMAX con poda ALPHA-BETA con profundidad limitada 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.

En el fichero "Player.cpp" hay dos funciones relevantes que no son miembros de la clase Player, que son ***double Valoracion (const Enviroment &estado, int jugador)*** que es dónde se debe implementar vuestra heurística (alterando la parametrización si lo consideráis oportuno). Otra ***Valoracion_Test (const Enviroment &estado, int jugador)*** que no debéis modificar. Esta última se utilizará para verificar si se ha implementado correctamente el algoritmo MINIMAX con Poda ALPHA-BETA.

La primera vez que os descargáis el software, la función ***Think()*** del fichero "*Player.cpp*", viene preparado para que cuando se pulse el botón "***Jug. Heuristica***", devuelva un movimiento aleatorio. Una vez que se haya implementado el algoritmo MINIMAX con Poda ALPHA-BETA, se marcará esa parte de la función ***Think()***, tal como se indica en los comentarios de la propia función, y se desmarcará la llamada a la función que implementa el algoritmo anterior.

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 guión) al finalizar las tareas a realizar.
- Se realizará una defensa de la práctica. El objetivo de esta defensa es verificar que la memoria entregada ha sido realizada por el alumno. Por tanto, esta defensa requerirá de la ejecución del simulador con los comportamientos realizados por los alumnos, en clase de prácticas, y de la respuesta a cuestiones del trabajo realizado. La calificación de la defensa será **APTO** o **NO APTO**. Una calificación **NO APTO** en la defensa implica el suspenso con calificación **0** en la práctica. Una calificación **APTO** permite al alumno obtener la calificación según los criterios explicados en el punto siguiente.
- 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 2.**
- Las cuestiones realizadas por el profesor durante la defensa de prácticas y correctamente respondidas por el alumno se evalúan de **0 a 2**.
- La eficacia del algoritmo se evaluará de **0 a 6** puntos y estará basado en un torneo de todos contra todos a dos partidas (en una, uno de ellos será el primer jugador y otro el segundo, y en la segunda se invertirán los papeles). En cada partida, la victoria da tres puntos al vencedor y cero al derrotado. En caso de empate, cada jugador se llevará un punto. Tras disputarse la competición completa, el jugador (o jugadores) con mayor puntuación obtendrá un 6, el jugador o jugadores con 0 puntos obtendrán un 0, el resto un valor proporcional a los puntos obtenidos.

- La **fecha límite de entrega de la práctica** será el día viernes 30 de Mayo a las 22:00 horas. No se admitirá ningún medio de entrega de la práctica que no sea el que se habilita a través de la página web dentro del plazo aquí establecido.
- **La defensa de la práctica se realizará en la semana del 2 al 6 de Junio.**

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 con poda ALPHA-BETA en los términos en que se ha explicado previamente. Estos ficheros deberán entregarse mediante la plataforma web de la asignatura, en un fichero ZIP, llamado practica3.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 exprese el esfuerzo realizado por el alumno en la realización del trabajo.