ugr | Universidad de Granada

SEMINARIO 4

Presentación Práctica

Métodos de Búsqueda con Adversario (Juegos [conecta-4 CRUSH])

Inteligencia Artificial

Dpto. Ciencias de la Computación e Inteligencia Artificial

ETSI Informática y de Telecomunicación

UNIVERSIDAD DE GRANADA

Curso 2014/2015

DECSAI

Departamento de Ciencias de la Computación e I.A.

Universidad de Granada

Índice

1. Introducción
2. Presentación del Problema
3. Presentación del Juego
4. Presentación del Simulador
5. Pasos del desarrollo de la práctica
6. Evaluación de la práctica

Índice

1. Introducción

- El objetivo de esta práctica consiste en la implementación de:
  - *un agente deliberativo que pueda llevar a cabo un comportamiento inteligente en un entorno de juego*
  - *El proceso deliberativo está basado en el algoritmo de búsqueda MINIMAX con Poda ALFA-BETA y con profundidad limitada.*

1. Introducción

- Trabajaremos en un entorno de simulación para
  - poder representar dos agentes que compiten entre sí con el objetivo de ganar al juego CONECTA-4 CRUSH
  - el entorno representa un tablero de juego formado por 7 filas y 7 columnas
  - Este simulador es una versión modificada de la aspiradora inteligente basada en los ejemplos del libro *Stuart Russell, Peter Norvig, "Inteligencia Artificial: Un enfoque Moderno"*
- El simulador ha sido adaptado para la realización de esta práctica.

1. Introducción

- Esta práctica cubre los siguientes objetivos docentes:
  - Conocer la representación de problemas basados en estados (estado inicial, objetivo y espacio de búsqueda) para ser resueltos con técnicas computacionales.
  - Entender que la resolución de problemas en IA implica definir una representación del problema y un proceso de búsqueda de la solución.
  - Analizar las características de un problema dado y determinar si es susceptible de ser resuelto mediante técnicas de búsqueda. Decidir en base a criterios racionales la técnica más apropiada para resolverlo y saber aplicarla.
  - Entender el concepto de heurística y analizar las repercusiones en la eficiencia en tiempo y espacio de los algoritmos de búsqueda.
  - Conocer distintas aplicaciones reales de la IA. Explorar y analizar soluciones actuales basadas en técnicas de IA.
  - Conocer las técnicas básicas de búsqueda con adversario (minimax, poda alfa-beta) y su relación con los juegos.
  - Ser capaz de implementar cualquiera de estas técnicas en un lenguaje de programación de propósito general.

1. Introducción

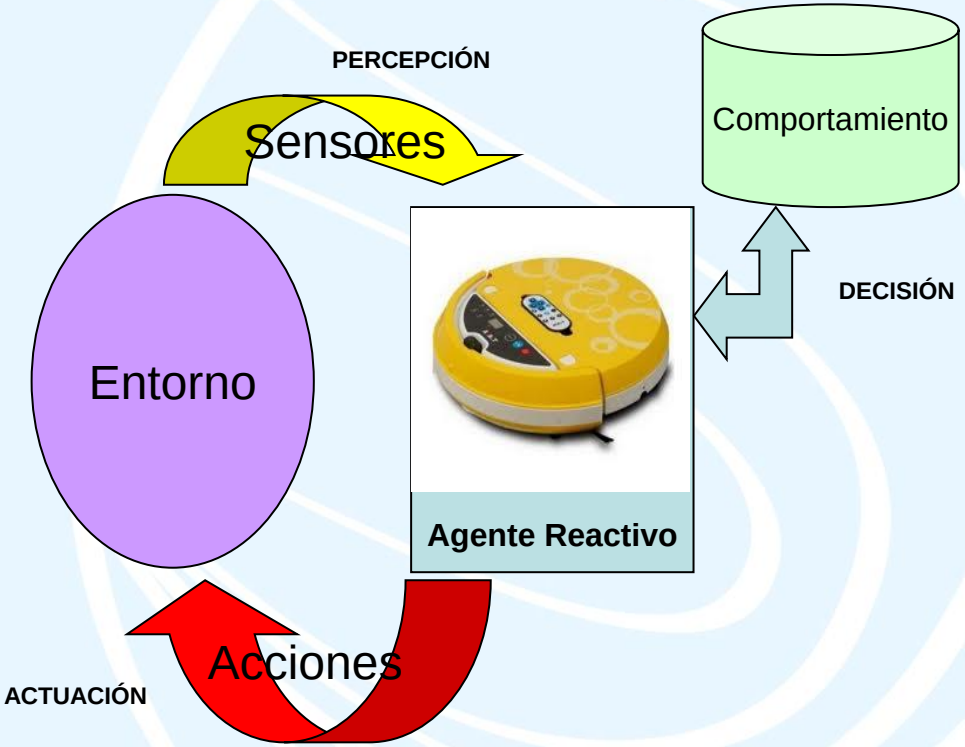
- Para seguir esta presentación:
  - Encender el ordenador
  - En la petición de identificación poned
    1. Vuestro identificador (Usuario)
    2. Vuestra contraseña (Password)
    3. Y en Código **codeblocks**
    4. Pulsar “Entrar”

Índice

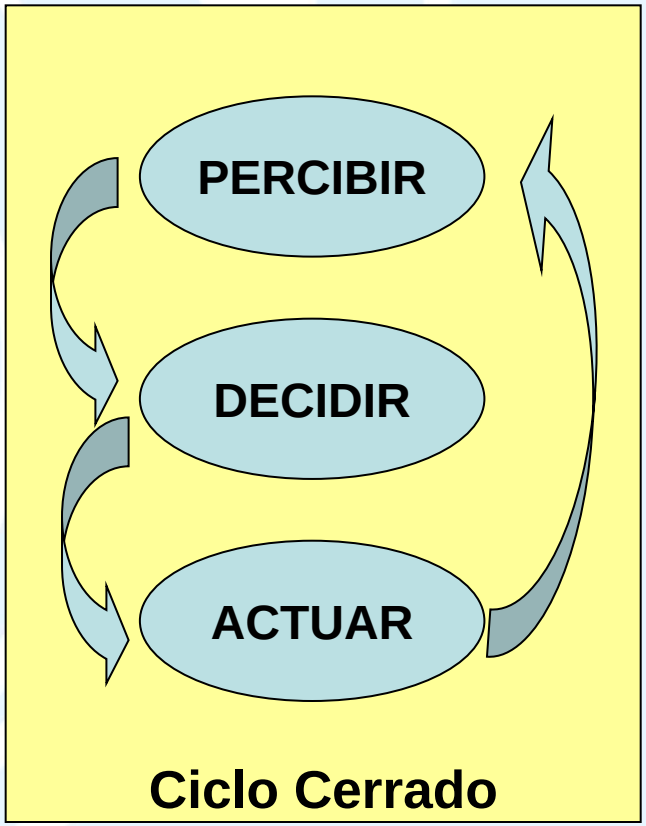
2. Presentación del Problema

- Conecta-4

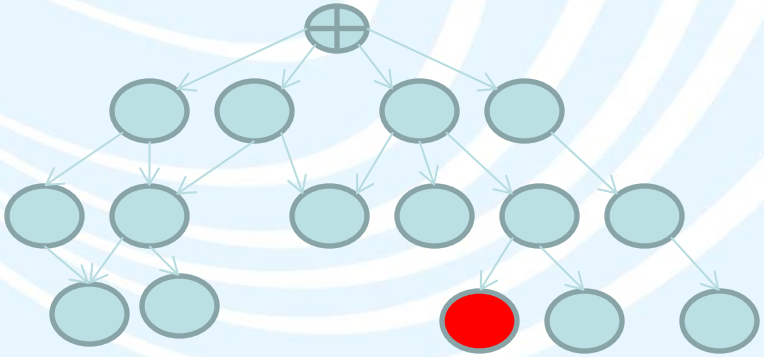
2. Presentación del Problema



PRÁCTICA 2



2. Presentación del Problema



2. Presentación del Problema

Problemas de Búsqueda

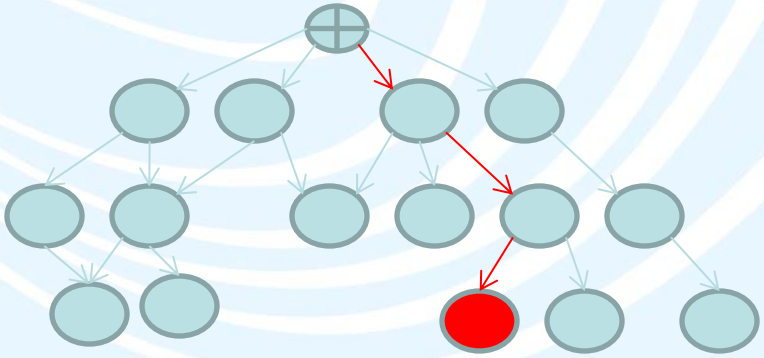
Estado inicial

Acciones

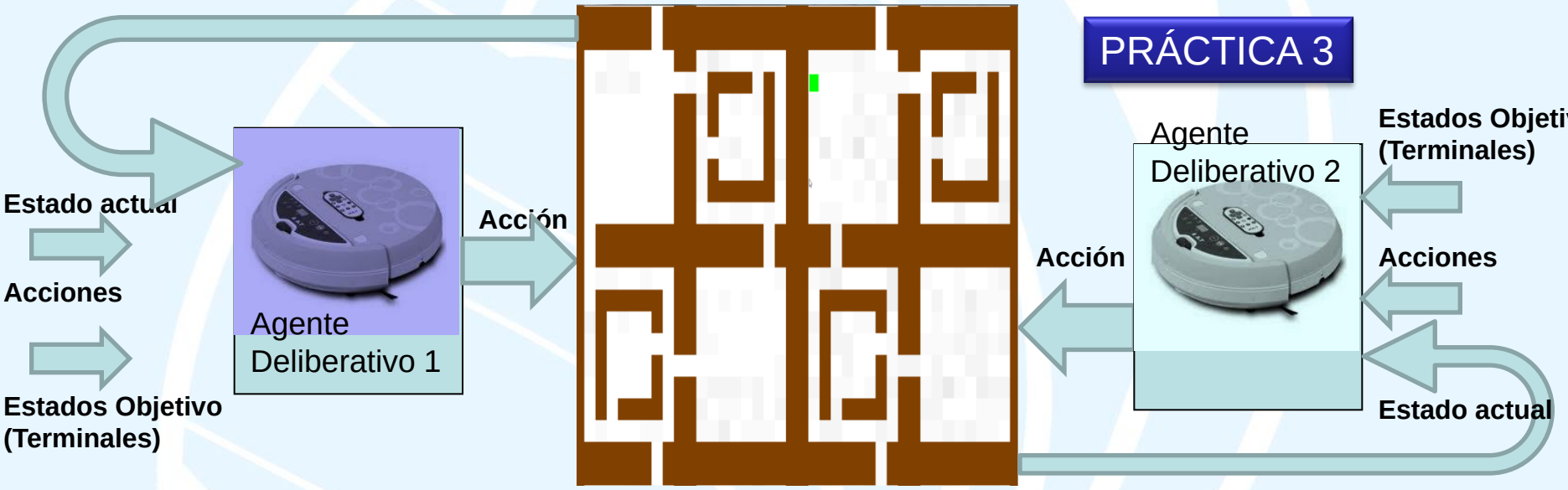
Estados Objetivo

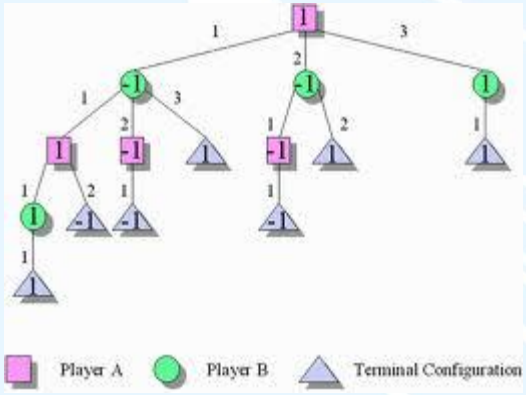
Agente Deliberativo

Secuencia de acciones

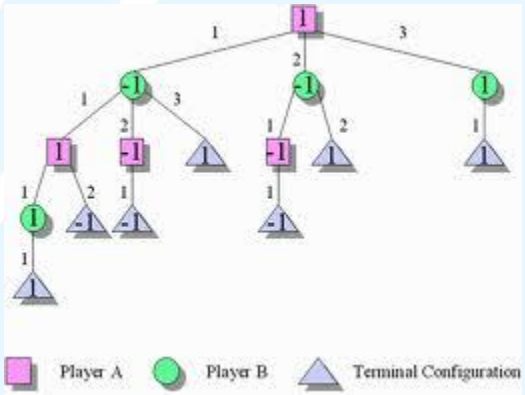


2. Presentación del Problema





- Entorno **multiagente**, dos agentes intervienen simultáneamente.
- Las acciones de un agente influyen en la percepción y toma de decisiones del otro.
- Entorno **competitivo**: los agentes tienen metas contrapuestas



Decisión

Decisión

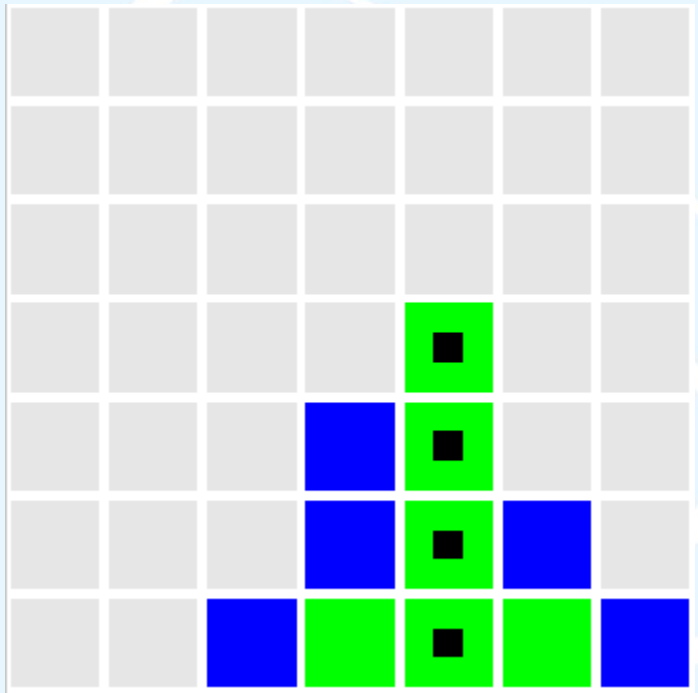
2. Presentación del Problema

Ahora el problema se plantea como:

- Dado un estado de una partida de CONECTA-4 CRUSH
- Determinar la acción a realizar que permita al agente conseguir la victoria en el juego
- Siguiendo un proceso deliberativo, basado en una búsqueda con adversario en un espacio de estados representado por un árbol de juego.

Índice

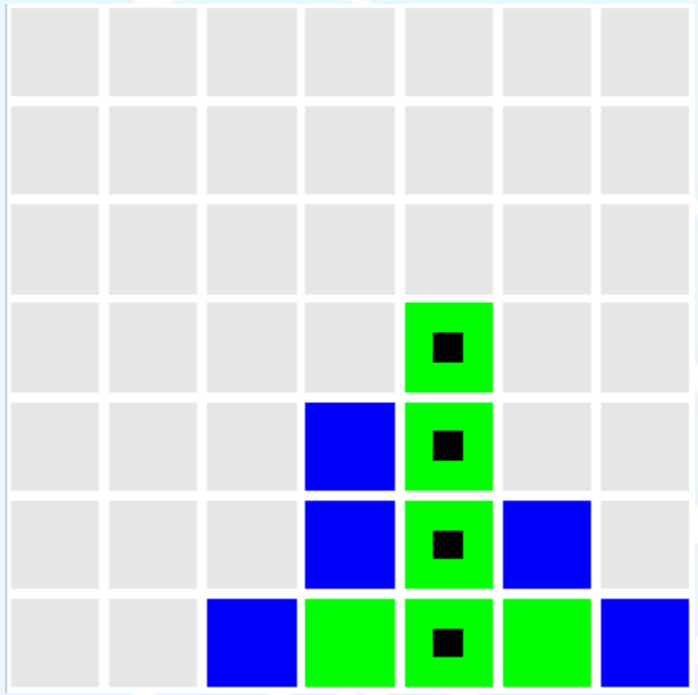
El juego: CONECTA-4 CRUSH



Una posible configuración del juego en el que gana el jugador con fichas verdes.

- Juego bipersonal con información completa
- Un tablero 7x7 casillas
- Los jugadores van poniendo fichas en el tablero de forma alternativa
- **Objetivo**: conseguir alinear 4 fichas del mismo color en horizontal, vertical o diagonal.

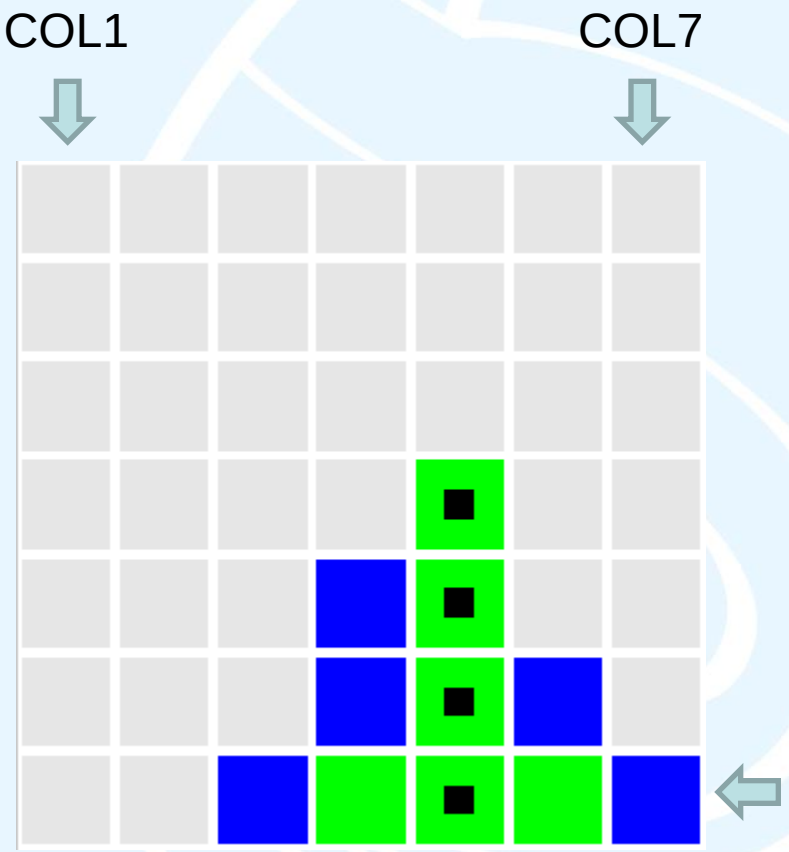
El juego: CONECTA-4 CRUSH



Una posible configuración del juego en el que gana el jugador con fichas verdes.

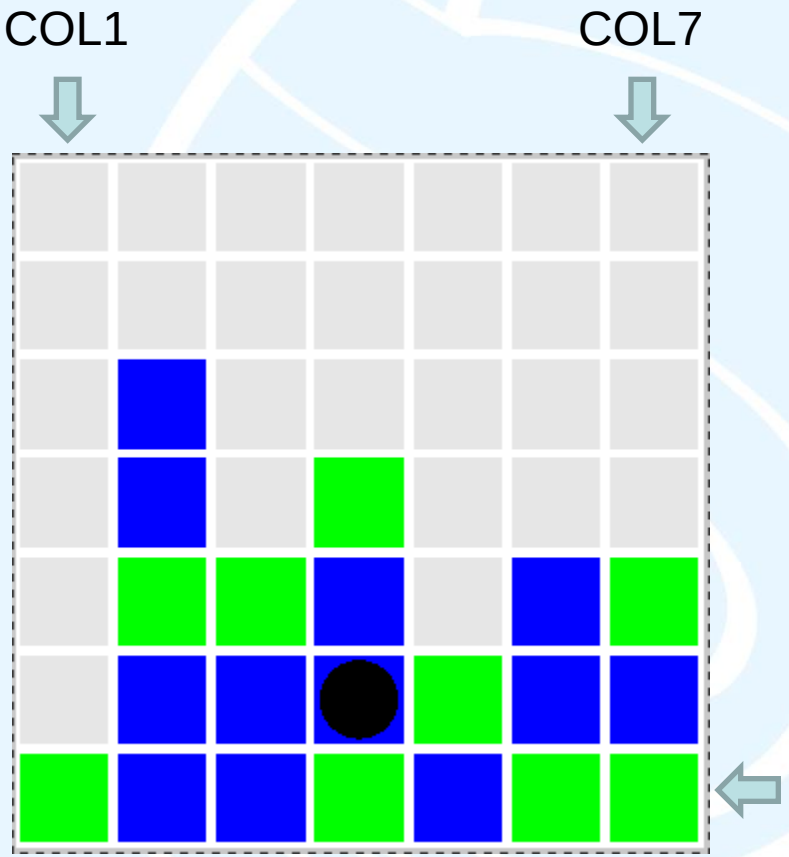
- Juego bipersonal con información completa
- Un tablero 7x7 casillas
- Los jugadores van poniendo fichas en el tablero de forma alternativa
- **Objetivo**: conseguir alinear 4 fichas del mismo color en horizontal, vertical o diagonal.

El juego: CONECTA-4 CRUSH



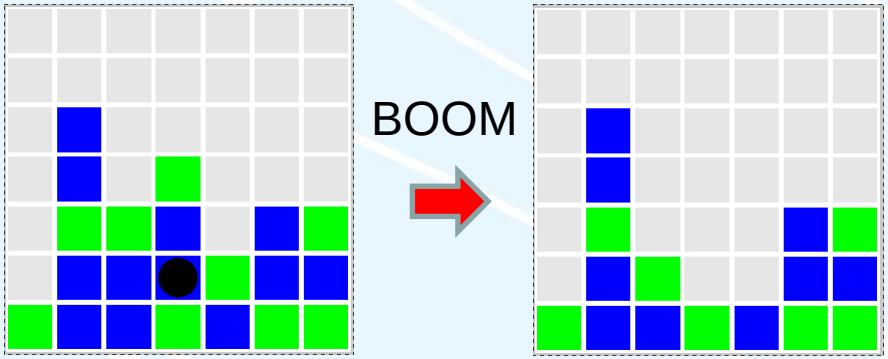
- Movimientos
  - Situar una ficha en alguna de las 7 columnas, siempre que la columna no esté completa.
  - Las fichas se sueltan por arriba y caen por gravedad hasta llegar al tope del tablero, o hasta que se sitúa encima de una ficha previamente colocada.
  - El número máximo de acciones de cada jugador por jugada es de 7.

El juego: CONECTA-4 CRUSH



- Movimientos
  - Cada 4 movimientos aparece una “ficha bomba”.
  - Tener esta ficha en el tablero implica una acción más (BOOM)
  - Cada jugador sólo puede tener una ficha bomba como máximo en el tablero.
  - Esta acción de BOOM consume el turno de juego del jugador que la aplica.

El juego: CONECTA-4 CRUSH



- Movimientos
  - La explosión elimina la ficha bomba, junto con las fichas que se encuentren situadas encima, debajo, a derecha e izquierda de ella.
  - Si una explosión alcanza la ficha bomba del adversario, esta se elimina como si fuera una ficha más, sin encadenar las explosiones.

3. Presentación del Simulador

- Compilación del simulador
- Ejecución del simulador

3. Presentación del Simulador

3.1. Compilación del Simulador

Nota: En esta presentación, asumimos que el entorno de programación ***CodeBlocks*** está ya instalado. Si no es así, en el enunciado de la práctica se indica como proceder a su instalación.

1. Cread la carpeta "**U:\IA\practica3**"
2. Descargar **4Raya.zip** desde la web de la asignatura y cópielo en la carpeta anterior.

(a) http://decsai.ugr.es
(b) Entrar en acceso identificado
(c) Elegir la asignatura "Inteligencia Artificial"
(d) Seleccionar "Práctica 3"

3. Presentación del Simulador

3.1. Compilación del Simulador

3. Descomprimir en la raíz de esta carpeta y aparecerá la carpeta

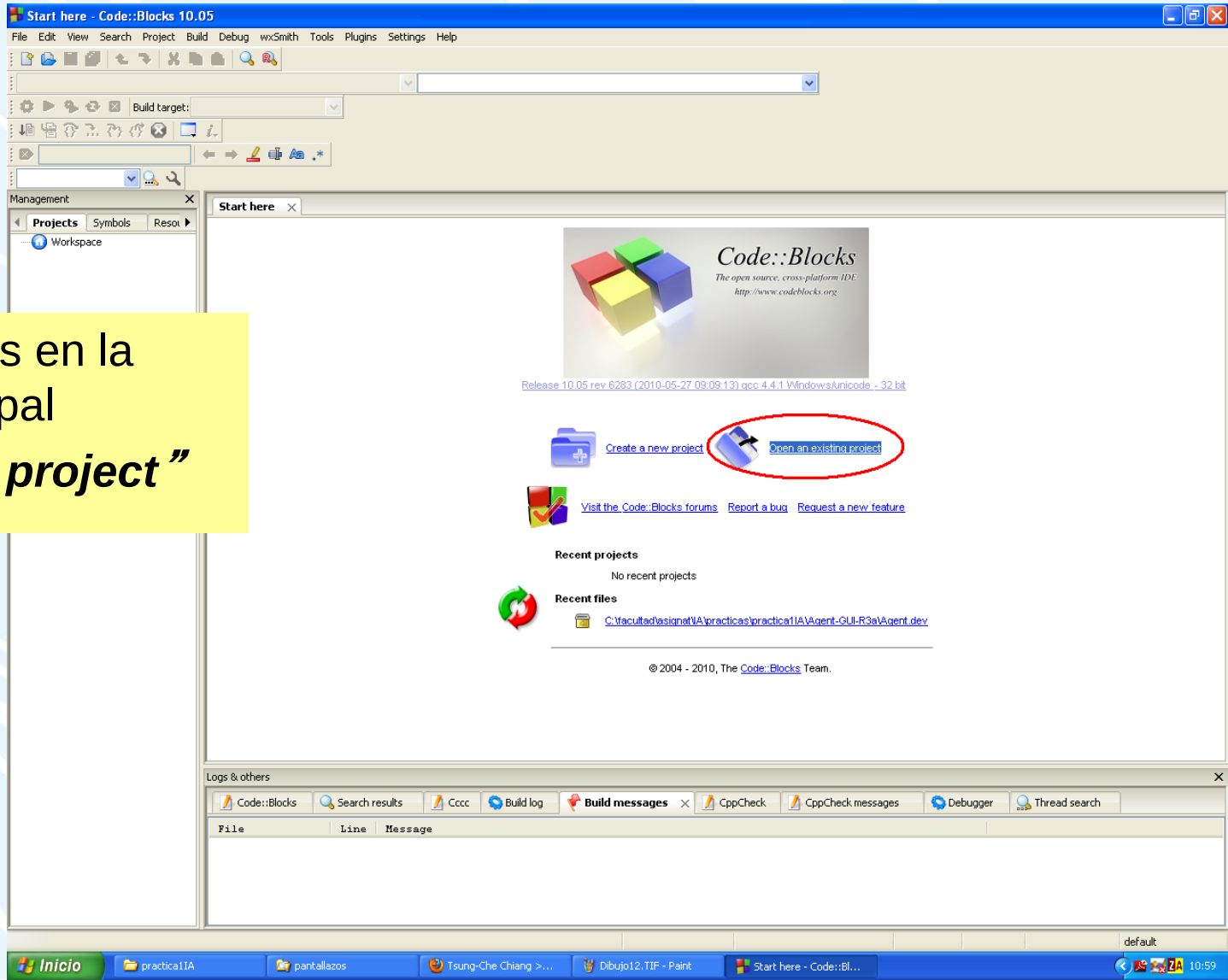
4. Abrimos “CodeBlocks”
   - Si es la primera vez que lo lanzamos nos preguntará el compilador de C/C++ a usar:
     - Seleccionaremos la primera opción, “GNU GCC Compilar”
   - Si es la primera vez, también nos preguntará si queremos asociar los ficheros C++ a este entorno de programación:
     - Seleccionaremos ***“Yes, associate Code::Blocks with every supported type (including project files from other IDEs)”***

3. Presentación del Simulador

3.1. Compilación del Simulador

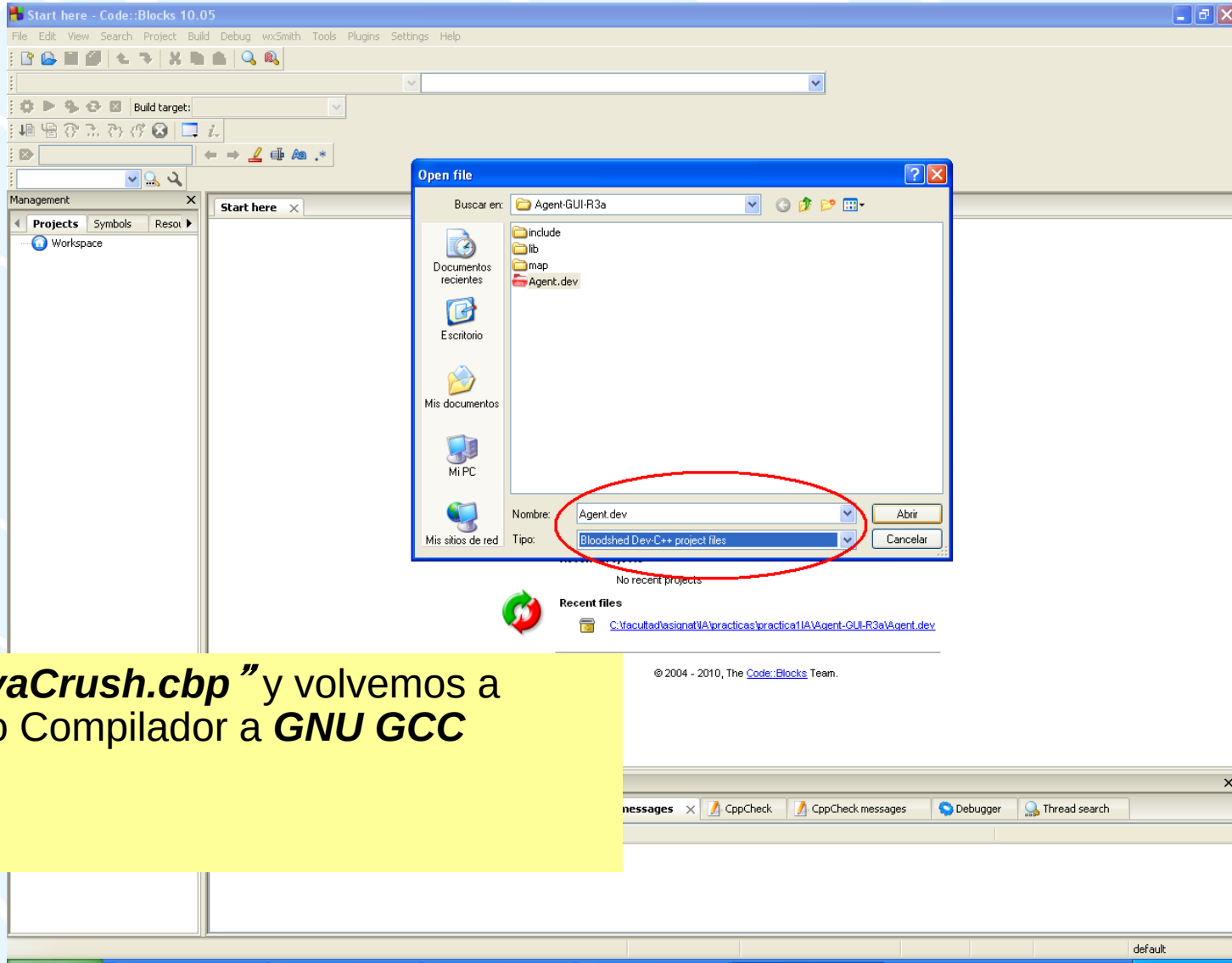
5. Seleccionamos en la pantalla principal

"Open an existing project"



3. Presentación del Simulador

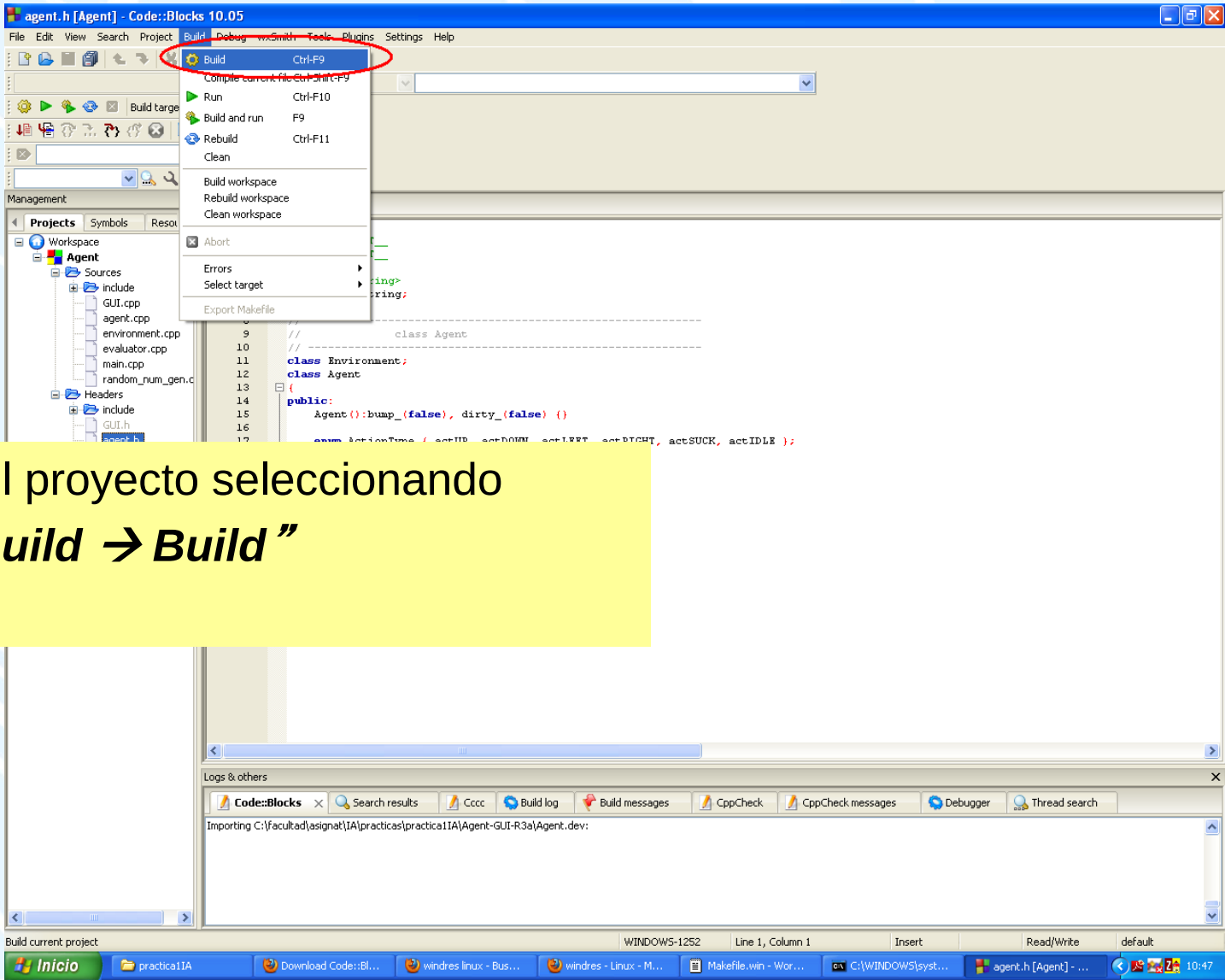
3.1. Compilación del Simulador



6. Abrimos ***"4RayaCrush.cbp"*** y volvemos a confirmar como Compilador a ***GNU GCC compiler***

3. Presentación del Simulador

3.1. Compilación del Simulador

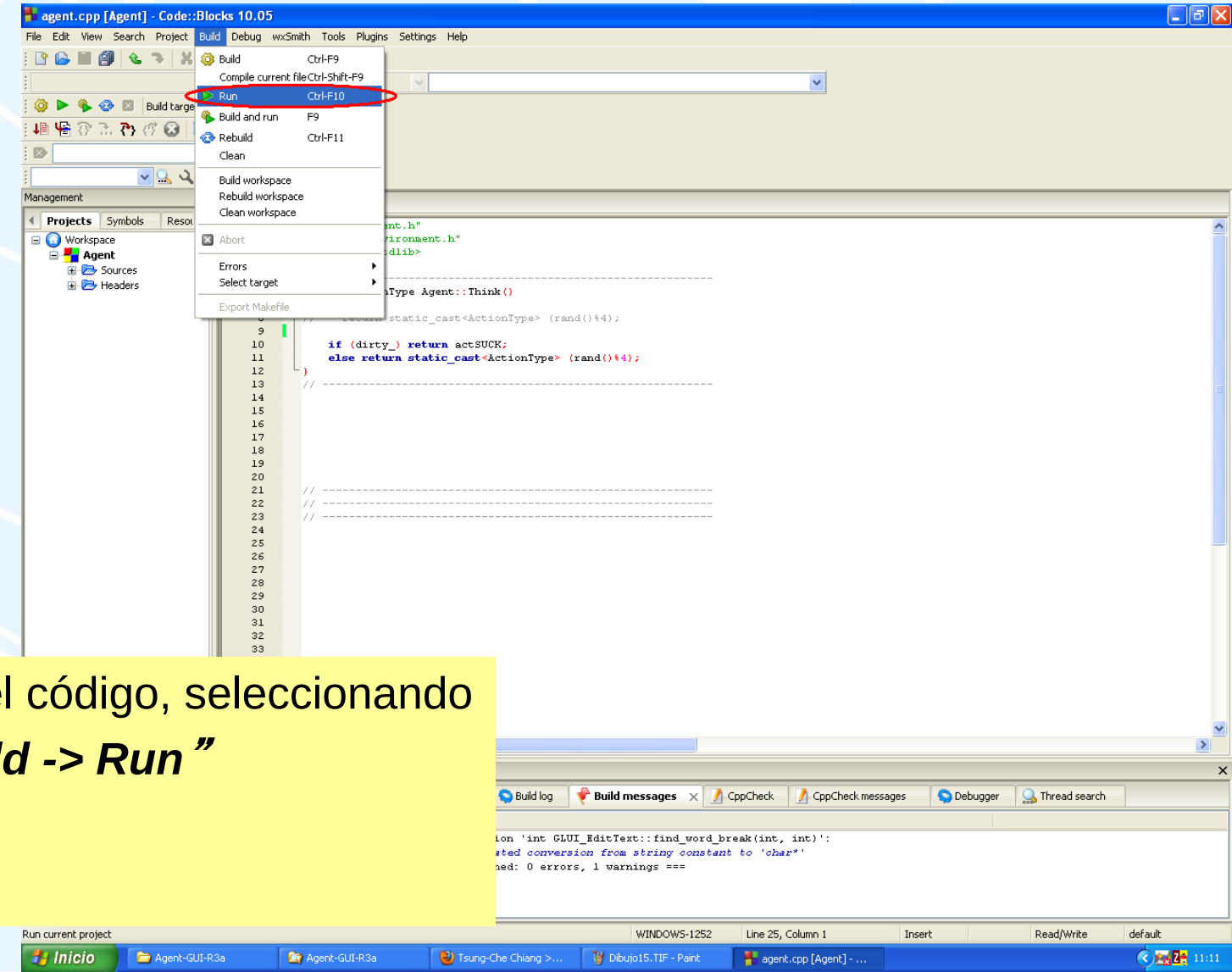


7. Compilamos el proyecto seleccionando

"Build → Build"

3. Presentación del Simulador

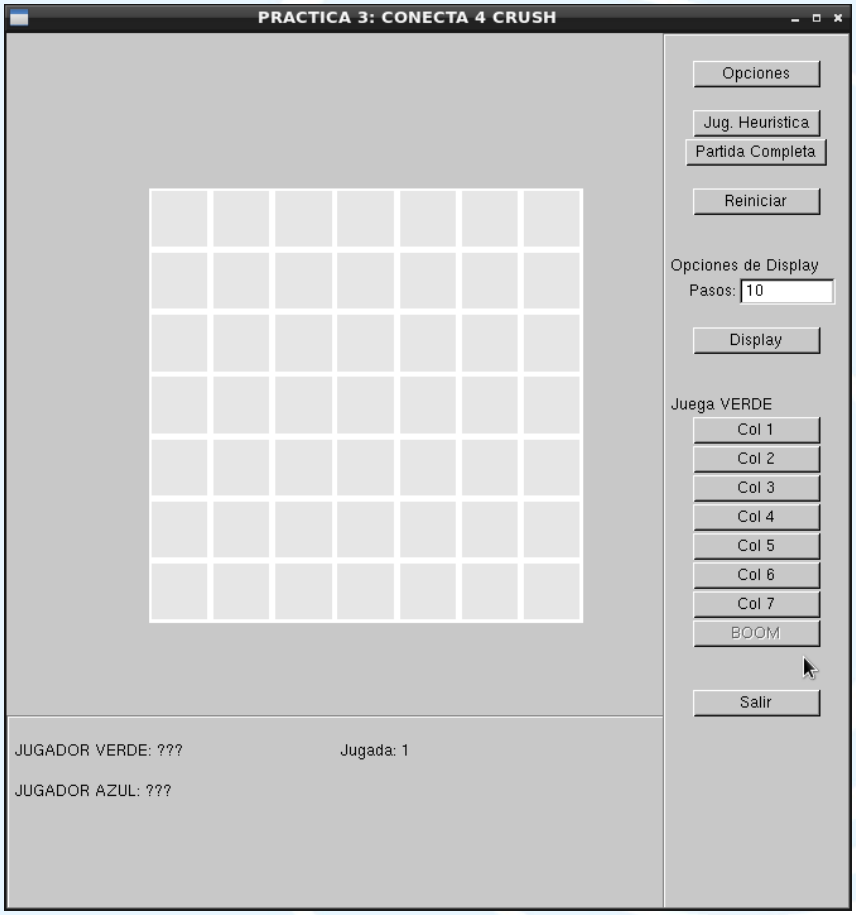
3.1. Compilación del Simulador



8. Ejecutamos el código, seleccionando ***"Build -> Run"***

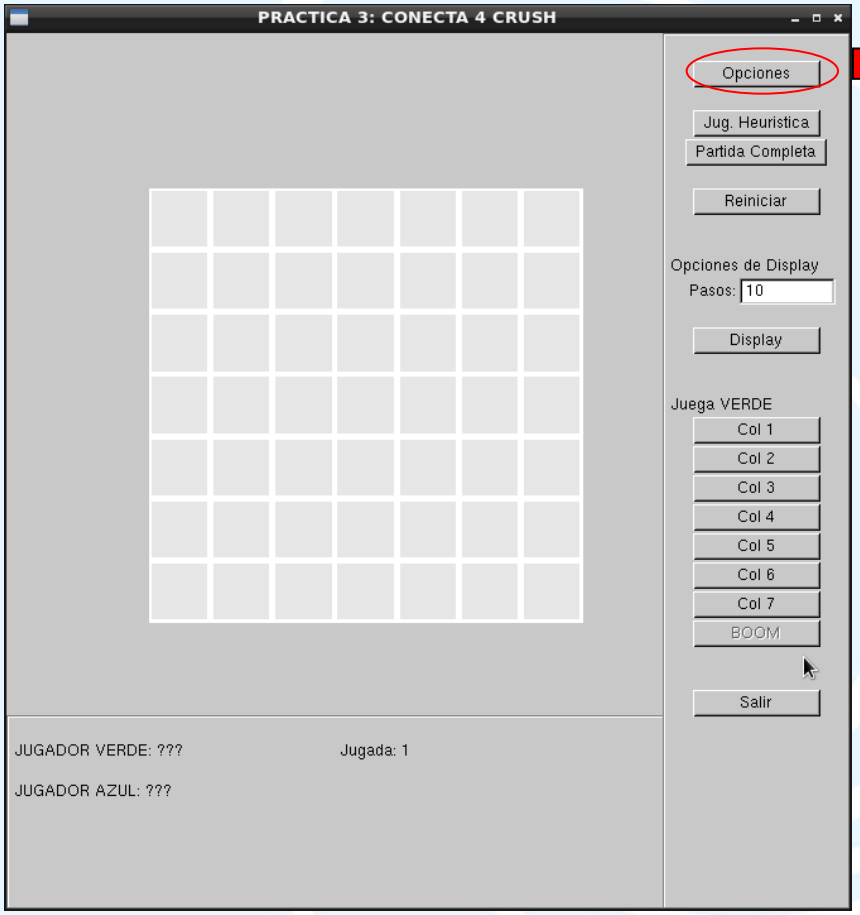
3. Presentación del Simulador

3.2. Ejecución del Simulador



3. Presentación del Simulador

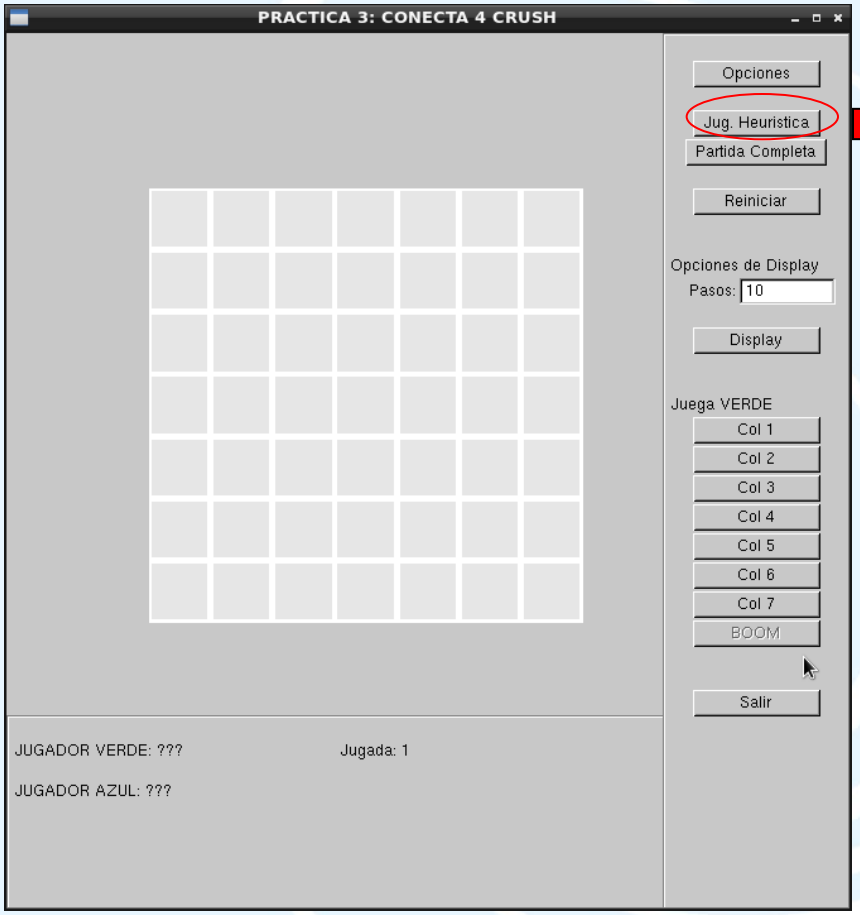
3.2. Ejecución del Simulador



"Opciones" Establece la configuración del juego.

3. Presentación del Simulador

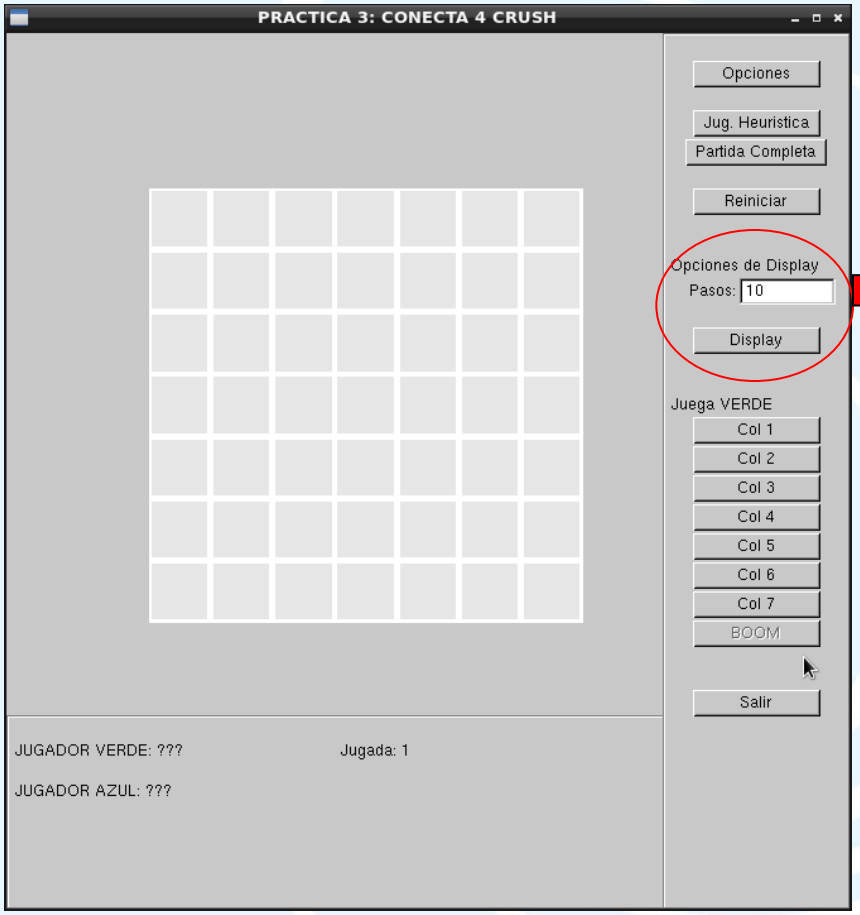
3.2. Ejecución del Simulador



"Jug. Heuristica" Pide que la siguiente jugada se obtenga a partir de la heurística definida por el alumno.

3. Presentación del Simulador

3.2. Ejecución del Simulador

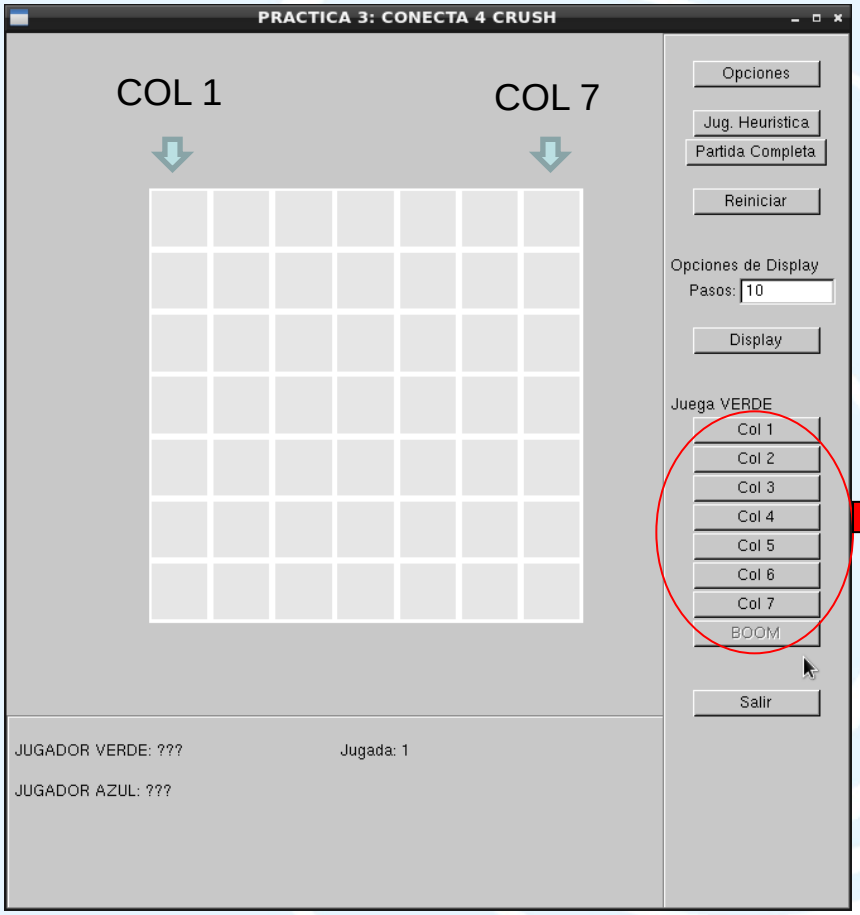


"Opciones de Display"

Muestra una secuencia de pasos del juego (número definido en Pasos), donde el control de los dos jugadores está guiado por la heurística que se haya implementado.

3. Presentación del Simulador

3.2. Ejecución del Simulador

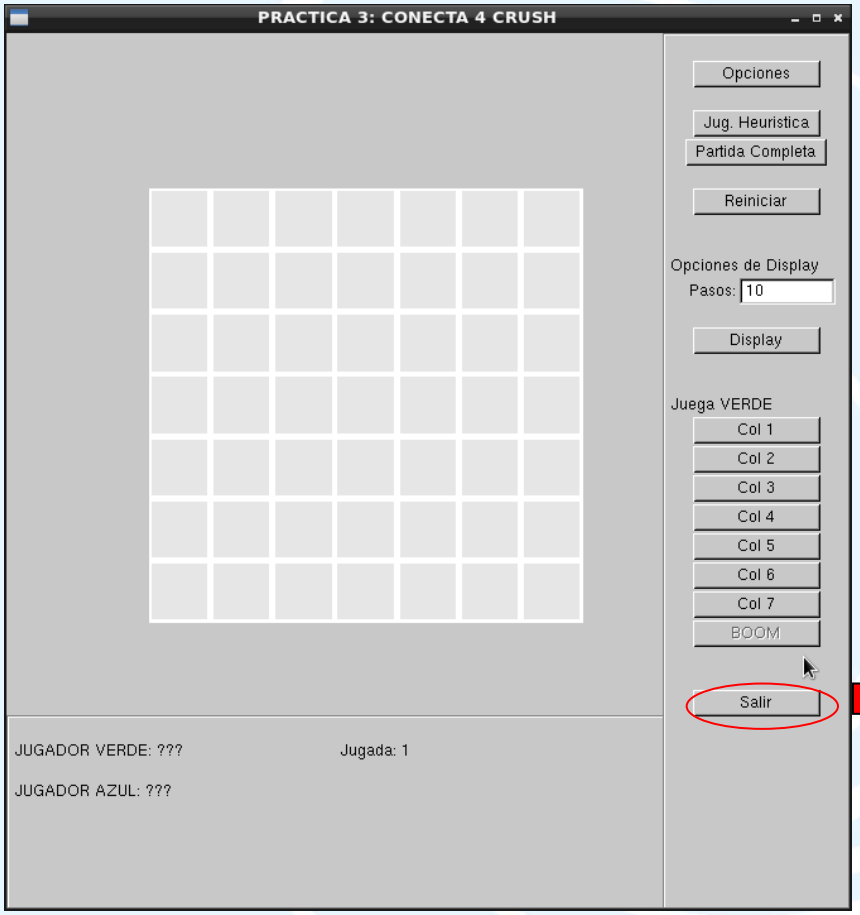


Controles del usuario humano: Cuando juega el jugador humano ha de pulsar uno de estos botones indicando en qué columna quiere realizar su movimiento.

La “COL 1” es la columna más a a la izquierda, “COL 7” la columna más a la derecha.

3. Presentación del Simulador

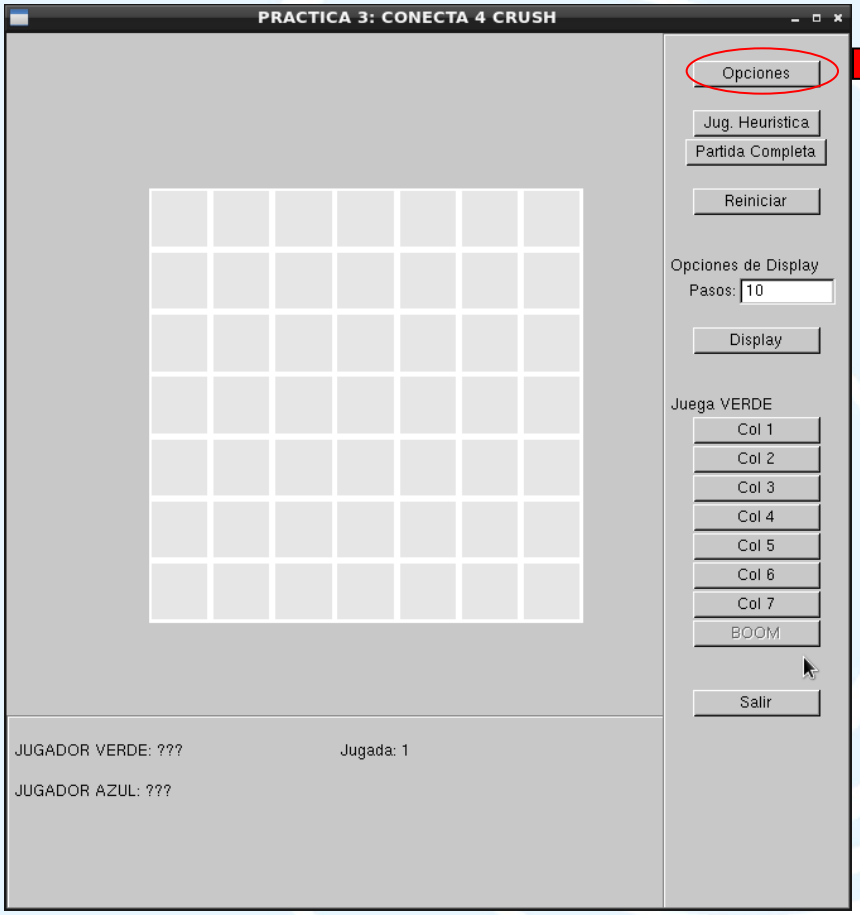
3.2. Ejecución del Simulador



"Salir" Termina la ejecución del simulador.

3. Presentación del Simulador

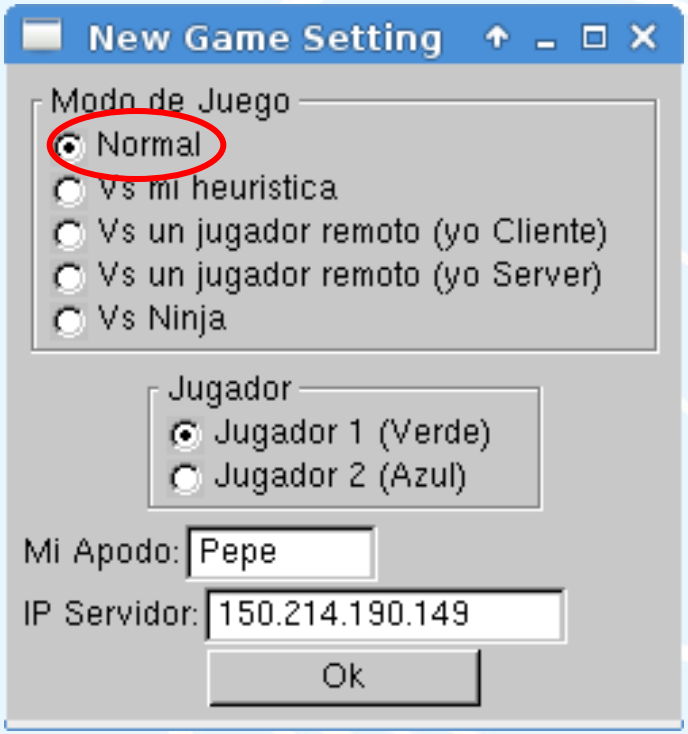
3.2. Ejecución del Simulador



Pulsando en Opciones, entramos en las distintas configuraciones del juego

3. Presentación del Simulador

3.2. Ejecución del Simulador



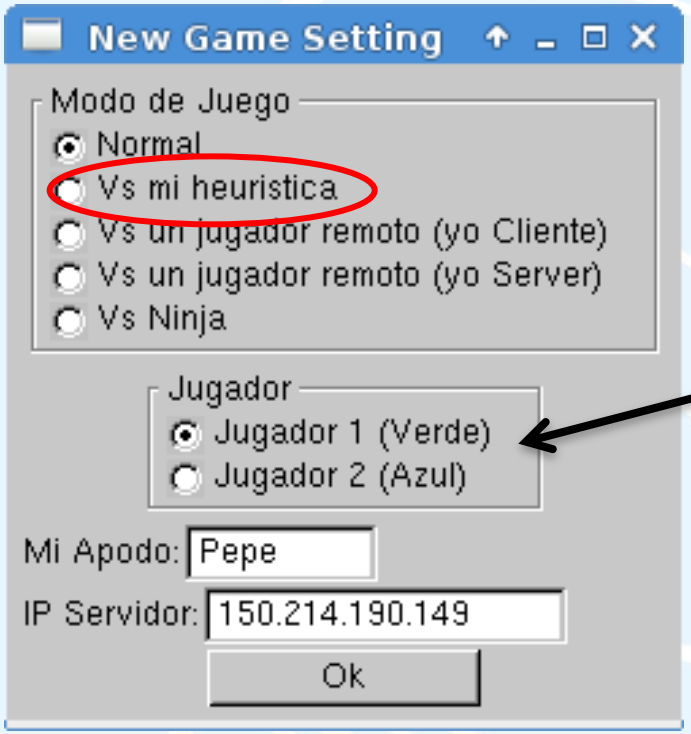
- Modos de Juego

En este modo, todos los botones de la ventana principal están activos, y es el modo de juego por defecto.

Permite la máxima flexibilidad, y se usa para testear que tanto la heurística como el algoritmo de juego están bien implementados.

3. Presentación del Simulador

3.2. Ejecución del Simulador



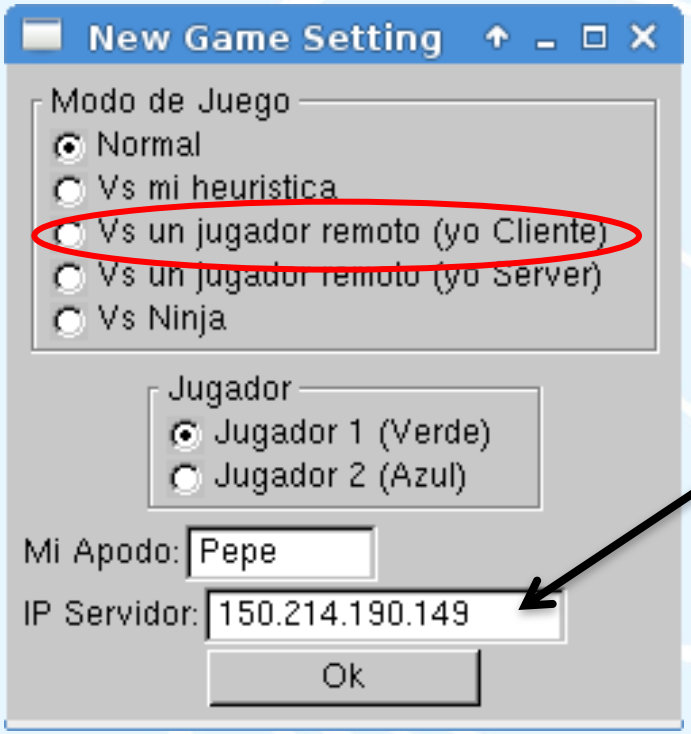
- Modos de Juego

En este modo, un jugador humano juega una partida contra una heurística implementada. El jugador humano juega el rol de jugador que se elija, la heurística el otro.

El turno cambia alternativamente después de cada jugada, y el jugador humano sólo puede usar los botones de selección de jugada.

3. Presentación del Simulador

3.2. Ejecución del Simulador



- Modos de Juego

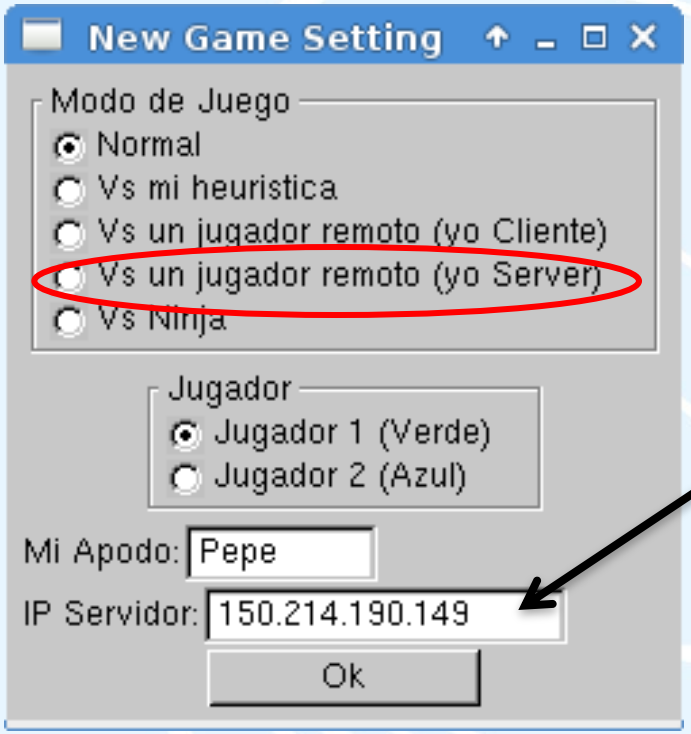
Este es un modo de juego en red entre dos compañeros.

En IP Servidor se debe colocar la dirección IP del simulador que hace el papel de servidor.

Dos jugadores en modo “Yo Cliente” pueden usar este modo para jugar entre ellos si colocan la dirección IP que aparece por defecto (150.214.190.149)

3. Presentación del Simulador

3.2. Ejecución del Simulador



- Modos de Juego

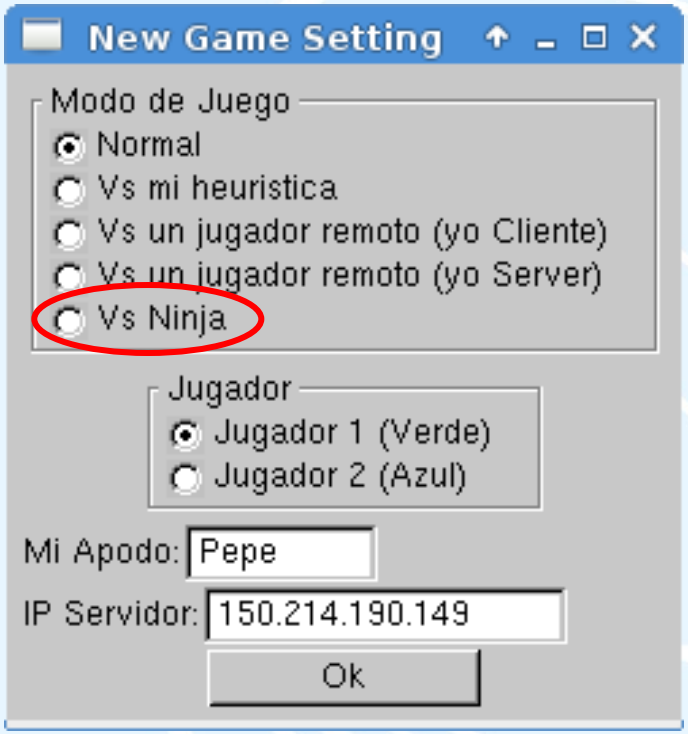
Este es un modo de juego en red entre dos compañeros.

En IP Servidor se debe colocar la dirección IP del simulador que hace el papel de servidor.

Dos jugadores pueden usar este modo para jugar entre ellos si uno de los jugadores está en modo “Yo Server”, el otro está en modo “Yo Cliente” y éste último coloca la IP del primero.

3. Presentación del Simulador

3.2. Ejecución del Simulador



- Modos de Juego

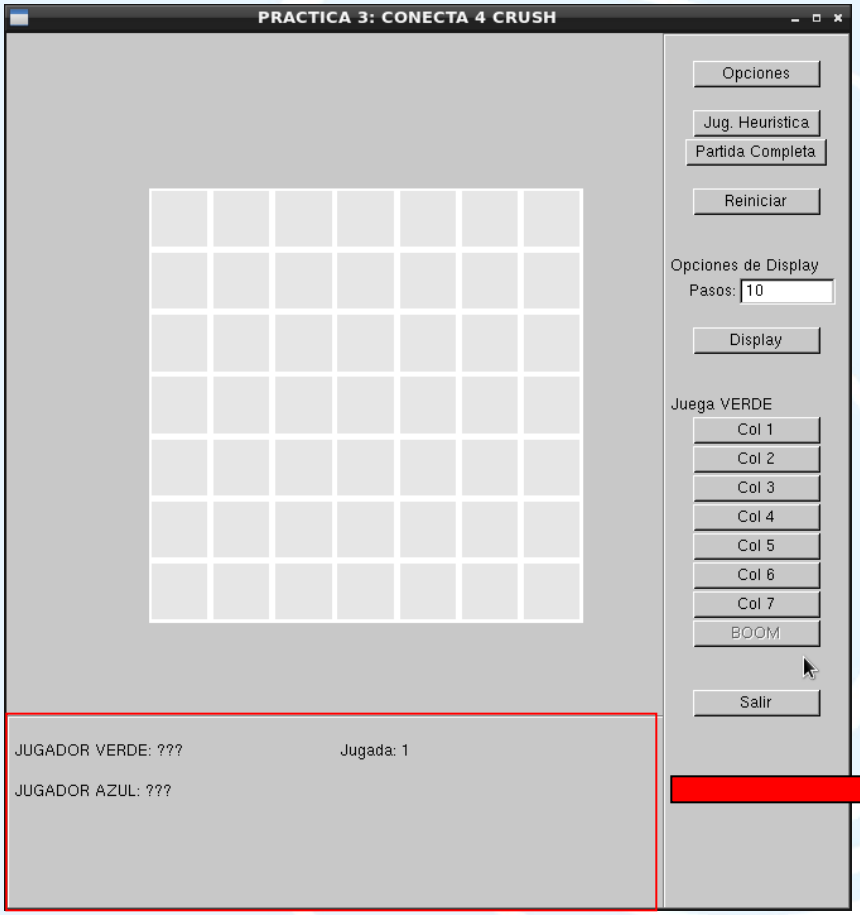
Este es el modo especial de juego en red.

Cuando se usa está opción, se juega contra un jugador automático externo que tiene una buena heurística para este juego.

Debe ser usado para comprobar como de buena es la heurística desarrollada.

3. Presentación del Simulador

3.2. Ejecución del Simulador



- Información sobre la

la evolución del juego, indicando los últimos movimientos de cada jugador, y en la fase final, quién termina ganando.

Índice

4. Pasos del desarrollo de la práctica

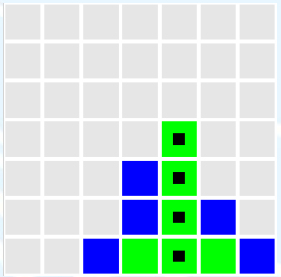
1. Descripción de la clase ***Environment***
   - Extensión de clase Environment
2. Descripción de la clase ***Player***.
   - Nueva clase que implementa la idea de la anterior clase Agent

4. Pasos del desarrollo de la práctica

4.1. Descripción de la clase **Environment**

```
96
97 private:
98
99    // Tamano del mapa (Siempre el numero de filas (X) es igual al numero de columnas (Y)
100      int MAZE_SIZE;
101
102
103      // Matriz que codifica el tablero
104      char **maze_;
105
106   // Indica la ocupación de cada columna
107      char *tope_;
108
109      // Ultimas acciones realizadas por las dos aspiradoras.
110      int last_action1_, last_action2_;
111
112      // Jugador al que le toca realizar el siguiente movimiento
113      int jugador_activo_;
114
115      // Cantidad de suciedad que debe contener el marcador de un jugador para ganar la partida.
116      int casillas_libres_;
117 };
118 // ----------------------------------------------------------------
```

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



Un estado queda definido por:

- Una matriz 7x7 (maze_)
- Un vector de tamaño 7 que indica el nivel de ocupación de cada columna (tope_)
- Las dos últimas acciones realizadas por cada jugador.
- El jugador que está en posesión del turno
- Número de casillas libres del tablero.

La matriz del tablero se representa en el sentido tradicional, pero se invierte cuando lo representa el simulador

4. Pasos del desarrollo de la práctica

4.1. Descripción de la clase **Environment**

```
51
52    // Este metodo genera todas las situaciones resultantes de aplicar todas las acciones sobre el tablero actual para el
53    // jugador que le toca jugar. Cada nuevo tablero se almacena en V, un vector de objetos de esta misma clase. El metodo
54    // devuelve el tamano de ese vector, es decir, el numero de movimientos posibles.
55    int GenerateAllMoves(Environment *V) const;
56
57    // Este metodo genera el siguiente movimiento que se puede realizar el jugador al que le toca jugar sobre el tablero actual
58    // devolviendolo como un objeto de esta misma clase. El parametro "act" indica cual fue el ultimo movimiento que se realizo
59    // sobre el tablero. Este metodo asume el siguiente orden en la aplicacion de las acciones: 0 PUT 1, 1 PUT 2, ...,
60    // 6 PUT7. Si no hay un siguiente movimiento, el metodo devuelve como tablero el actual.
61    // La primera vez que se invoca en un nuevo estado se le pasa como argumento en act el valor -1.
62    Environment GenerateNextMove(int &act) const;
63
64
65    // Devuelve numero de acciones que puede realizar el jugador al que le toca jugar sobre el tablero. "VecAct" es un vector de
66    // datos logicos que indican si una determinada accion es aplicable o no. Cada componente del vector esta asociada con una
67    // accion. Asi, la [0] indica si PUT 1 es aplicable, [1] si lo es PUT2,y asi sucesivamente.
68  int possible_actions(bool *VecAct) const;
69
70  // Indica la ultima accion que se aplico para llegar a la situacion actual del tablero. El entero que se devuelve es el
71  // ordinal de la acción.
72  int Last_Action(int jug) const;
73
74
75  // Expresa en una cadena de caracteres un dato del tipo enumerado "ActionType" que se pasa como argumento.
76  string ActionStr(ActionType action);
77
78  // Devuelve el jugador al que le toca jugar, siendo 1 el jugador Verde y 2 el jugador Azul.
79  int JugadorActivo(){return jugador_activo_;}
80
81  // Indica el nivel de ocupacion de una determinada columna
82  int Get_Ocupacion_Columna(int columna) const {return tope_[columna];}
83
84  // Devuelve el numero de casillas libres que quedan en el tablero
85  int Get_Casillas_Libres() const {return casillas_libres_;}
86
87  // Devuelve lo que hay en el tablero en la fila "row" columna "col": 0 vacia, 1 jugador1, 2 jugador2.
88  char See_Casilla(int row, int col) const {return maze_[row][col];}
89
90  // Devuelve verdadero cuando el juego ha terminado.
91    bool JuegoTerminado()const;
92
93    // Cuando el juego esta terminado devuelve quien ha ganado: 0 Empate, 1 Gana Jugador 1, 2 Gana Jugador2.
94    int RevisarTablero() const;
95
```

int GenerateAllMoves(Environment *V)

- Es la función utilizada para generar todos los estados sucesores correspondientes a los movimientos que puede llevar a cabo el jugador *activo* en el estado actual.
- Devuelve un array *V* de **n** posiciones, donde **n** puede ser 1, 2, …, 8 dependiendo del número de posibles acciones que puede realizar el jugador en ese momento.
- Cada posición de *V* es un nodo del espacio de búsqueda de tipo *Environment* que incluye la nueva configuración del tablero para cada movimiento válido dado.

4. Pasos del desarrollo de la práctica

4.1. Descripción de la clase **Environment**

```
51
52    // Este metodo genera todas las situaciones resultantes de aplicar todas las acciones sobre el tablero actual para el
53    // jugador que le toca jugar. Cada nuevo tablero se almacena en V, un vector de objetos de esta misma clase. El metodo
54    // devuelve el tamano de ese vector, es decir, el numero de movimientos posibles.
55    int GenerateAllMoves(Environment *V) const;
56
57    // Este metodo genera el siguiente movimiento que se puede realizar el jugador al que le toca jugar sobre el tablero actual
58    // devolviendolo como un objeto de esta misma clase. El parametro "act" indica cual fue el ultimo movimiento que se realizo
59    // sobre el tablero. Este metodo asume el siguiente orden en la aplicacion de las acciones: 0 PUT 1, 1 PUT 2, ...,
60    // 6 PUT7. Si no hay un siguiente movimiento, el metodo devuelve como tablero el actual.
61    // La primera vez que se invoca en un nuevo estado se le pasa como argumento en act el valor -1.
62    Environment GenerateNextMove(int &act) const;
63
64
65    // Devuelve numero de acciones que puede realizar el jugador al que le toca jugar sobre el tablero. "VecAct" es un vector de
66    // datos logicos que indican si una determinada accion es aplicable o no. Cada componente del vector esta asociada con una
67    // accion. Asi, la [0] indica si PUT 1 es aplicable, [1] si lo es PUT2,y asi sucesivamente.
68  int possible_actions(bool *VecAct) const;
69
70  // Indica la ultima accion que se aplico para llegar a la situacion actual del tablero. El entero que se devuelve es el
71  // ordinal de la acción.
72  int Last_Action(int jug) const;
73
74
75  // Expresa en una cadena de caracteres un dato del tipo enumerado "ActionType" que se pasa como argumento.
76  string ActionStr(ActionType action);
77
78  // Devuelve el jugador al que le toca jugar, siendo 1 el jugador Verde y 2 el jugador Azul.
79  int JugadorActivo(){return jugador_activo_;}
80
81  // Indica el nivel de ocupacion de una determinada columna
82  int Get_Ocupacion_Columna(int columna) const {return tope_[columna];}
83
84  // Devuelve el numero de casillas libres que quedan en el tablero
85  int Get_Casillas_Libres() const {return casillas_libres_;}
86
87  // Devuelve lo que hay en el tablero en la fila "row" columna "col": 0 vacia, 1 jugador1, 2 jugador2.
88  char See_Casilla(int row, int col) const {return maze_[row][col];}
89
90  // Devuelve verdadero cuando el juego ha terminado.
91    bool JuegoTerminado()const;
92
93    // Cuando el juego esta terminado devuelve quien ha ganado: 0 Empate, 1 Gana Jugador 1, 2 Gana Jugador2.
94    int RevisarTablero() const;
95
```

Environment GenerateNextMove(int &act)

- Esta función genera el siguiente sucesor del estado actual, donde ***act***, indica el último operador aplicado.
- El valor de act se actualiza dentro de la función indicando que operador se ha aplicado (0: PUT1, …, 6: PUT7, 7: BOOM)
- La primera vez que se invoca, act debe tomar el valor -1.
- Si no hay más descendientes, act toma un valor >=8 y la función devuelve el estado actual.

4. Pasos del desarrollo de la práctica

4.1. Descripción de la clase **Environment**

```
51
52    // Este metodo genera todas las situaciones resultantes de aplicar todas las acciones sobre el tablero actual para el
53    // jugador que le toca jugar. Cada nuevo tablero se almacena en V, un vector de objetos de esta misma clase. El metodo
54    // devuelve el tamano de ese vector, es decir, el numero de movimientos posibles.
55    int GenerateAllMoves(Environment *V) const;
56
57    // Este metodo genera el siguiente movimiento que se puede realizar el jugador al que le toca jugar sobre el tablero actual
58    // devolviendolo como un objeto de esta misma clase. El parametro "act" indica cual fue el ultimo movimiento que se realizo
59    // sobre el tablero. Este metodo asume el siguiente orden en la aplicacion de las acciones: 0 PUT 1, 1 PUT 2, ...,
60    // 6 PUT7. Si no hay un siguiente movimiento, el metodo devuelve como tablero el actual.
61    // La primera vez que se invoca en un nuevo estado se le pasa como argumento en act el valor -1.
62    Environment GenerateNextMove(int &act) const;
63
64
65    // Devuelve numero de acciones que puede realizar el jugador al que le toca jugar sobre el tablero. "VecAct" es un vector de
66    // datos logicos que indican si una determinada accion es aplicable o no. Cada componente del vector esta asociada con una
67    // accion. Asi, la [0] indica si PUT 1 es aplicable, [1] si lo es PUT2,y asi sucesivamente.
68  int possible_actions(bool *VecAct) const;
69
70    // Indica la ultima accion que se aplico para llegar a la situacion actual del tablero. El entero que se devuelve es el
71    // ordinal de la acción.
72    int Last_Action(int jug) const;
73
74
75    // Expresa en una cadena de caracteres un dato del tipo enumerado "ActionType" que se pasa como argumento.
76    string ActionStr(ActionType action);
77
78    // Devuelve el jugador al que le toca jugar, siendo 1 el jugador Verde y 2 el jugador Azul.
79    int JugadorActivo(){return jugador_activo_;}
80
81    // Indica el nivel de ocupacion de una determinada columna
82    int Get_Ocupacion_Columna(int columna) const {return tope_[columna];}
83
84    // Devuelve el numero de casillas libres que quedan en el tablero
85    int Get_Casillas_Libres() const {return casillas_libres_;}
86
87    // Devuelve lo que hay en el tablero en la fila "row" columna "col": 0 vacia, 1 jugador1, 2 jugador2.
88    char See_Casilla(int row, int col) const {return maze_[row][col];}
89
90    // Devuelve verdadero cuando el juego ha terminado.
91      bool JuegoTerminado()const;
92
93      // Cuando el juego esta terminado devuelve quien ha ganado: 0 Empate, 1 Gana Jugador 1, 2 Gana Jugador2.
94      int RevisarTablero() const;
95
```

void possible_actions(bool *VecAct)

- Devuelve los posibles movimientos válidos (o acciones) que puede realizar el jugador *activo* en un estado concreto.
- *VecAct* es un array de 8 valores booleanos. Cada act[i], 0<= i <= 7, representa si se puede llevar a cabo, respectivamente, la acción COL1, …, COL7, BOOM.

4. Pasos del desarrollo de la práctica

4.2. Descripción de la clase ***Player***

```
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 tablero para un jugador dado).

4. Pasos del desarrollo de la práctica

4.2. Descripción de la clase ***Player***

```
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 tres métodos:
- ***Player(int jug)***
  - Asigna a jugador_ el valor jug
- ***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 estado actual del tablero de juego.

4. Pasos del desarrollo de la práctica

4.2. Descripción de la clase ***Player***

```
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
```

IMPORTANTE:

- **La implementación del algoritmo MINIMAX con Poda ALFA-BETA y con profundidad limitada a 8 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***.

4. Pasos del desarrollo de la práctica

4.2. Descripción de la clase ***Player***

```
83 // Invoca el siguiente movimiento del jugador
84 Environment::ActionType Player::Think(){
85    const int PROFUNDIDAD_MINIMAX = 6;  // Umbral maximo de profundidad para el metodo MiniMax
86    const int PROFUNDIDAD_ALFABETA = 8; // Umbral maximo de profundidad para la poda Alfa_Beta
87
88    Environment::ActionType accion; // acción que se va a devolver
89    bool aplicables[7]; // Vector bool usado para obtener las acciones que son aplicables en el estado actual. La interpretacion es
90                  // aplicables[0]==true si PUT1 es aplicable
91                  // aplicables[1]==true si PUT2 es aplicable
92                  // aplicables[2]==true si PUT3 es aplicable
93                  // aplicables[3]==true si PUT4 es aplicable
94                  // aplicables[4]==true si PUT5 es aplicable
95                  // aplicables[5]==true si PUT6 es aplicable
96                  // aplicables[6]==true si PUT7 es aplicable
97
98
99
00    double valor; // Almacena el valor con el que se etiqueta el estado tras el proceso de busqueda.
01    double alpha, beta; // Cotas de la poda AlfaBeta
02
03    int n_act; //Acciones posibles en el estado actual
04
05
06    n_act = actual_.possible_actions(aplicables); // Obtengo las acciones aplicables al estado actual en "aplicables"
07    int opciones[10];
08
09    // Muestra por la consola las acciones aplicable para el jugador activo
10    cout << " Acciones aplicables ";
11    (jugador_==1) ? cout << "Verde: " : cout << "Azul: ";
12    for (int t=0; t<7; t++)
13      if (aplicables[t])
14        cout << " " << actual_.ActionStr( static_cast< Environment::ActionType > (t) );
15    cout << endl;
16
```

En la versión que se pasa:

- El método Think() implementa un jugador de conecta-4 aleatoria.

4. Pasos del desarrollo de la práctica

4.2. Descripción de la clase ***Player***

```
118  //------------------- COMENTAR Desde aqui
119  cout << "\n\t";
120  int n_opciones=0;
121  JuegoAleatorio(aplicables, opciones, n_opciones);
122
123  if (n_act==0){
124    (jugador_==1) ? cout << "Verde: " : cout << "Azul: ";
125    cout << " No puede realizar ninguna accion!!!\n";
126    //accion = Environment::actIDLE;
127  }
128  else if (n_act==1){
129       (jugador_==1) ? cout << "Verde: " : cout << "Azul: ";
130       cout << " Solo se puede realizar la accion "
131            << actual_.ActionStr( static_cast< Environment::ActionType > (opciones[0]) ) << endl;
132       accion = static_cast< Environment::ActionType > (opciones[0]);
133
134     }
135     else { // Hay que elegir entre varias posibles acciones
136       int aleatorio = rand()%n_opciones;
137       cout << " -> " << actual_.ActionStr( static_cast< Environment::ActionType > (opciones[aleatorio]) ) << endl;
138       accion = static_cast< Environment::ActionType > (opciones[aleatorio]);
139     }
140
141  //------------------- COMENTAR Hasta aqui
142
143
144  //------------------- AQUI EMPIEZA LA PARTE A REALIZAR POR EL ALUMNO ------------------------------------------
145
146
147  // Opcion: Poda AlfaBeta
148  // NOTA: La parametrizacion es solo orientativa
149  // valor = Poda_AlfaBeta(actual_, jugador_, 0, PROFUNDIDAD_ALFABETA, accion, alpha, beta);
150  //cout << "Valor MiniMax: " << valor << "  Accion: " << actual_.ActionStr(accion) << endl;
151
152  return accion;
153 }
```

En la versión que se pasa:

- El método Think() implementa un jugador de conecta-4 aleatoria.
- Esa parte se debe comentar tal como se indica en el código fuente, cuando se implemente la poda alpha-beta.
- Y se descomenta esta parte de aquí.
- La parametrización que se propone es sólo orientativa.

4. Pasos del desarrollo de la práctica

4.2. Descripción de la clase ***Player***

```
23 double Puntuacion(int jugador, const Environment &estado){
24   double suma=0;
25
26   for (int i=0; i<7; i++)
27     for (int j=0; j<7; j++){
28       if (estado.See_Casilla(i,j)==jugador){
29         if (j<3)
30           suma += j;
31         else
32           suma += (6-j);
33       }
34     }
35
36   return suma;
37 }
38
39
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 }
53
54
55
56 // Funcion heuristica (ESTA ES LA QUE TENEIS QUE MODIFICAR)
57 double Valoracion(const Environment &estado, int jugador){
58 }
```

En la versión que se pasa:

- Hay dos funciones definidas en player.cpp que **deben mantenerse en la versión que se entregue**.
- Estas funciones son:
  - ValoracionTest
  - Puntuacion

Estas funciones se definen para verificar si el alumno ha implementado correctamente la poda alpha-beta.

- Aparece un prototipo orientativo donde el alumno debe implementar su heurística llamada “Valoracion”.

Índice

5. Evaluación de la práctica

1. ¿Qué hay que entregar?
2. ¿Qué debe contener la memoria de la práctica?
3. ¿Cómo se evalúa la práctica?
4. ¿Dónde y cuándo se entrega?

5. Evaluación de la práctica

¿Qué hay que entregar?

No ficheros ejecutables

Un único archivo comprimido **zip** llamado "***practica3.zip***" que NO contenga carpetas en su interior, con 3 archivos:

- Una de las carpetas con la memoria de la práctica (en formato pdf)
- Los archivos "player.cpp" y "player.h" donde se encuentra implementado el método pedido y la heurística diseñada para el juego CONECTA-4 CRUSH.

5. Evaluación de la práctica

¿Qué debe contener la memoria de la práctica?

Debe expresar el esfuerzo realizado por el alumno para la realización del trabajo.

Como orientación, indicar heurísticas probadas previas a la versión definitiva, y el motivo de su no consideración.

Documento 5 páginas máximo

5. Evaluación de la práctica

¿Cómo se evalúa?

Se tendrán en cuenta tres aspectos:

1. El documento de la memoria de la práctica
   - se evalúa de 0 a 3 puntos.
2. La defensa de la práctica
   - se evalúa entre 0 y 2 o NO APTO. NO APTO implica tener un **0** en esta práctica.
3. Evaluación de la eficacia:
   - La eficacia del algoritmo se evaluará de **0 a 5** puntos y consistirá en una competición de todos contra todos a dos partidas.

5. Evaluación de la práctica
La competición

- Todos juegan contra todos en dos partidas (en una de primer jugador y en otra de segundo).
- Cada partida jugada da:
  - 3 puntos al ganador y 0 al perdedor
  - 1 punto a cada uno si hay empate

- Tras disputarse la competición, las notas se reparten de la siguiente manera:
  - El jugador con más puntos obtiene un 5
  - El jugador con 0 puntos obtiene un 0
  - El resto una linealmente proporcional a los puntos que ha obtenido

5. Evaluación de la práctica

¿Dónde y cuándo se entrega?

- Se entrega en la aplicación de gestión de prácticas de la asignatura
  decsai.ugr.es → Entrega de Prácticas
- La fecha límite de entrega será:

  Semana del 18 Mayo
- La fecha de la defensa será:

  Semana del 25 Mayo