



*ugr*

Universidad  
de Granada

## SEMINARIO 4

# Presentación Práctica 3

## Métodos de Búsqueda con Adversario (Juegos)

**Inteligencia Artificial**  
Dpto. Ciencias de la Computación e  
**Inteligencia Artificial**  
ETSI Informática y de Telecomunicación  
UNIVERSIDAD DE GRANADA  
Curso 2011/2012



**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. Método de evaluación de la práctica

# 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

# 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 profundidad limitada.*

# 1. Introducción

- Trabajaremos en un entorno de simulación para
  - poder representar dos agentes aspiradora que compiten entre sí con el objetivo de limpiar la mayor cantidad posible de basura
  - en un entorno representado por un mapa que hace las veces de un tablero de juego
  - modificado 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

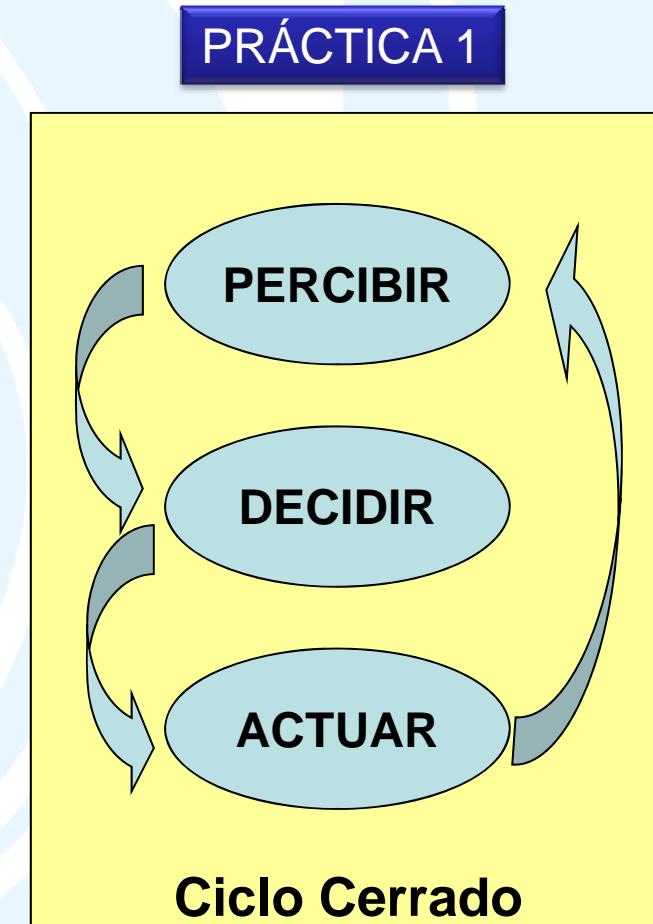
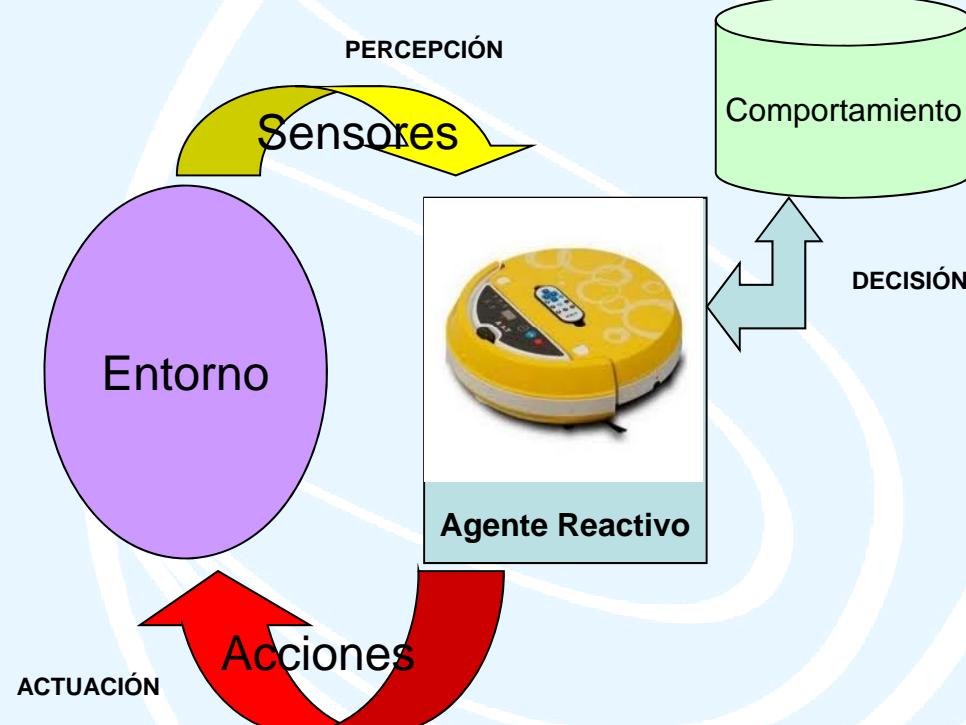
1. Introducción
2. Presentación del Problema
3. Presentación del Juego
4. Presentación del Simulador
5. Pasos del desarrollo de la práctica
6. Evaluación de la práctica

## 2. Presentación del Problema

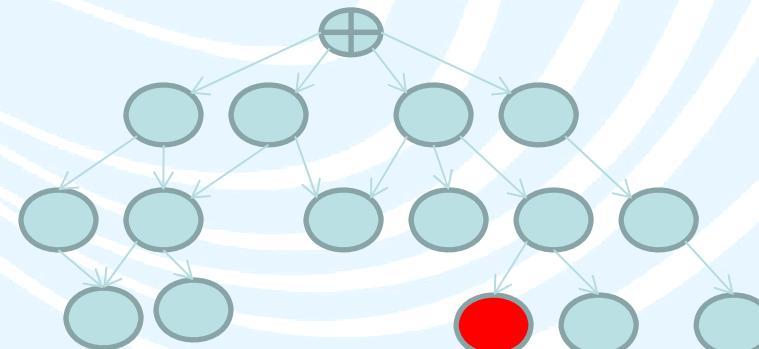
- Aspiradora Inteligente



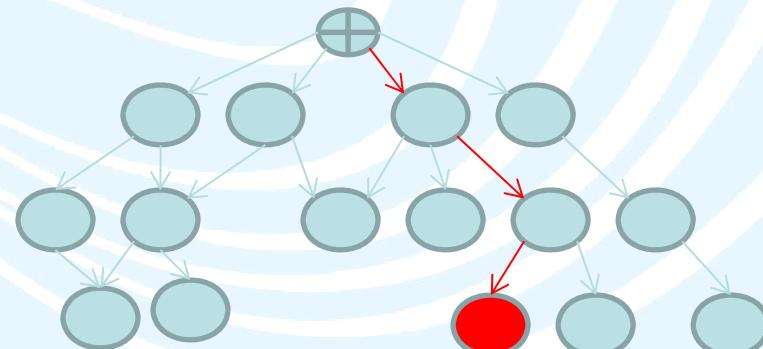
## 2. Presentación del Problema



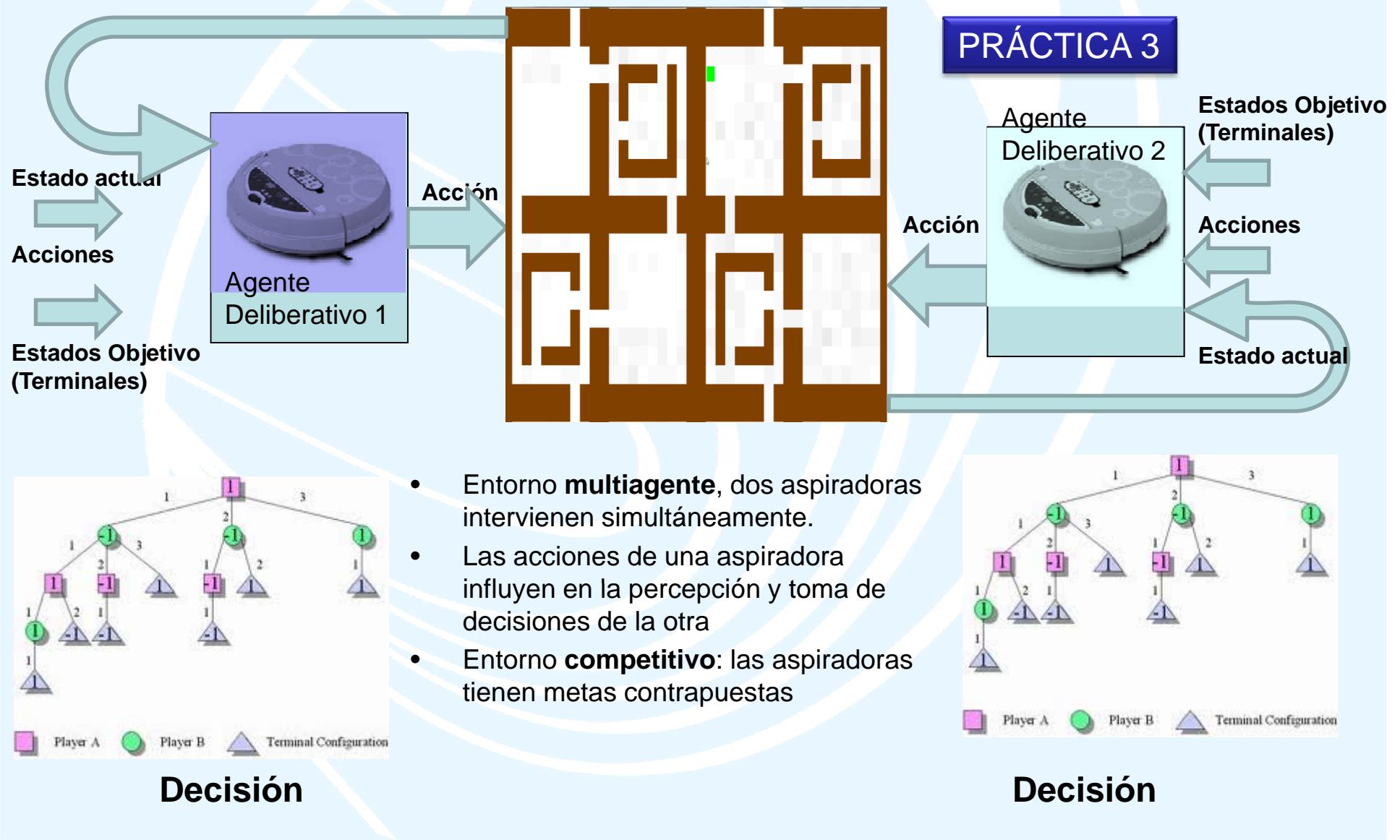
## 2. Presentación del Problema



## 2. Presentación del Problema



## 2. Presentación del Problema



## 2. Presentación del Problema

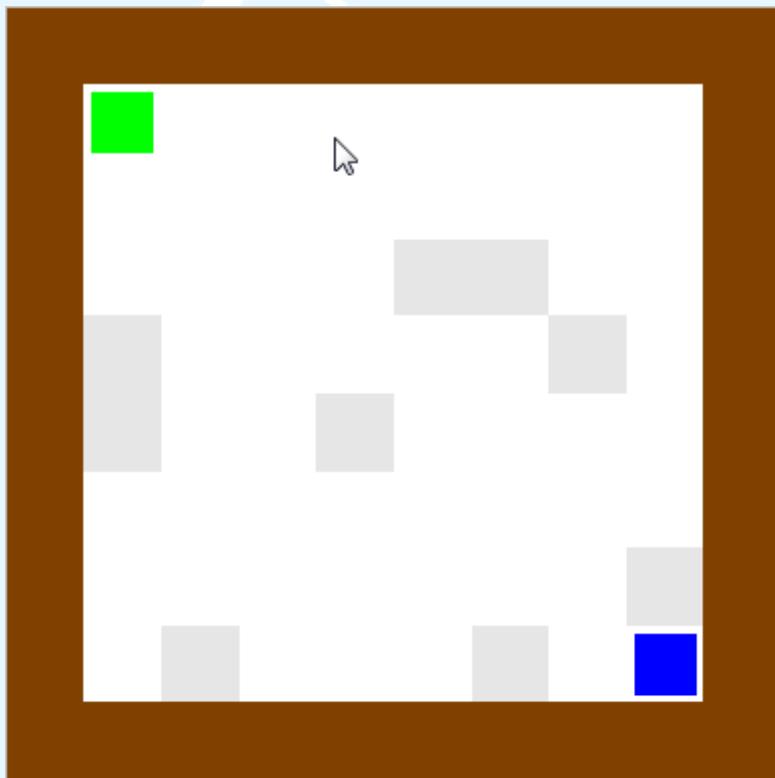
Ahora el problema se plantea como:

- Dado el estado actual percibido
- Determinar la acción a realizar que permita conseguir el mayor beneficio
  - Beneficio: número de unidades de basura conseguida
- Siguiendo un proceso deliberativo, basado en una búsqueda con adversario en un espacio de estados representado por un árbol de juego.

# 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

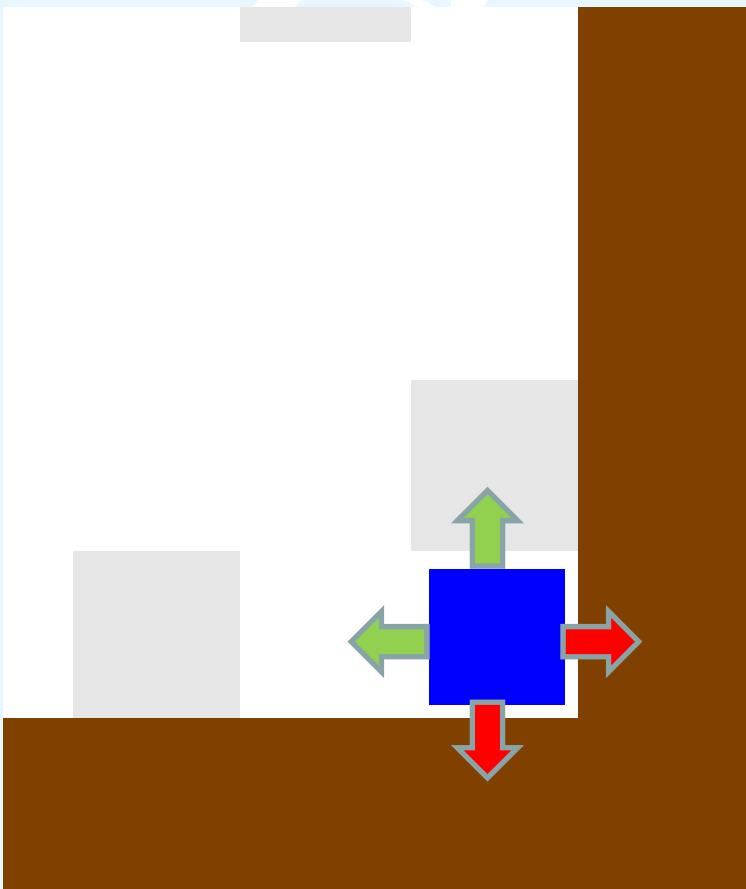
# El juego:ASPIRADORA TRON



Tablero inicial de 10x10 casillas en el que se muestran dos aspiradoras (jugadores) y 9 casillas con basura.

- Juego bipersonal con información completa
- Un tablero nxn casillas
  - Casillas con suciedad
  - Casillas con obstáculo
  - Casillas transitables
- Objetivo: conseguir la mayor cantidad posible de unidades de basura
  - Una casilla puede tener más de 1 unidad de basura

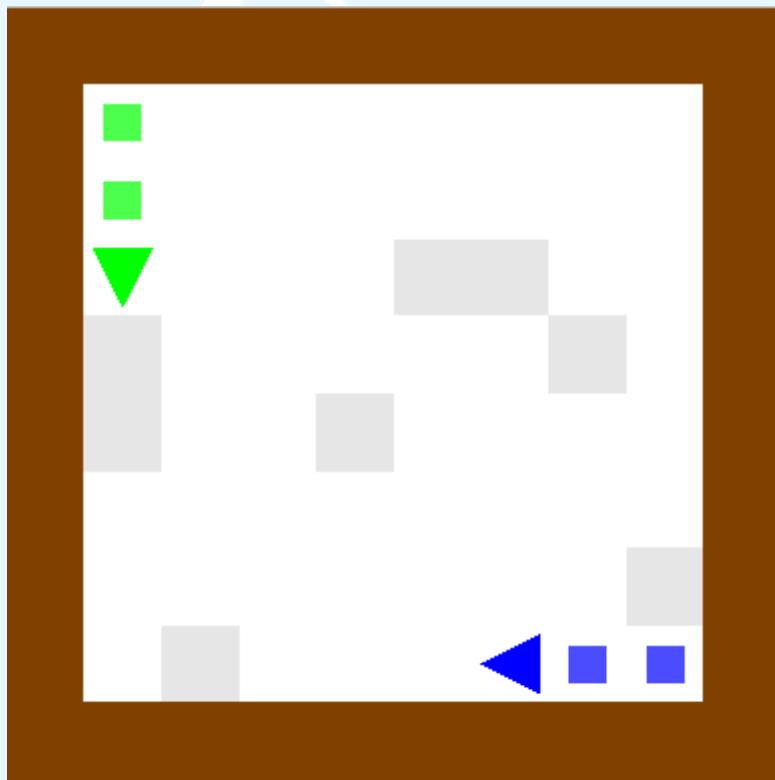
# El juego:ASPIRADORA TRON



Tablero inicial de 10x10 casillas en el que se muestran dos aspiradoras (jugadores) y 9 casillas con basura.

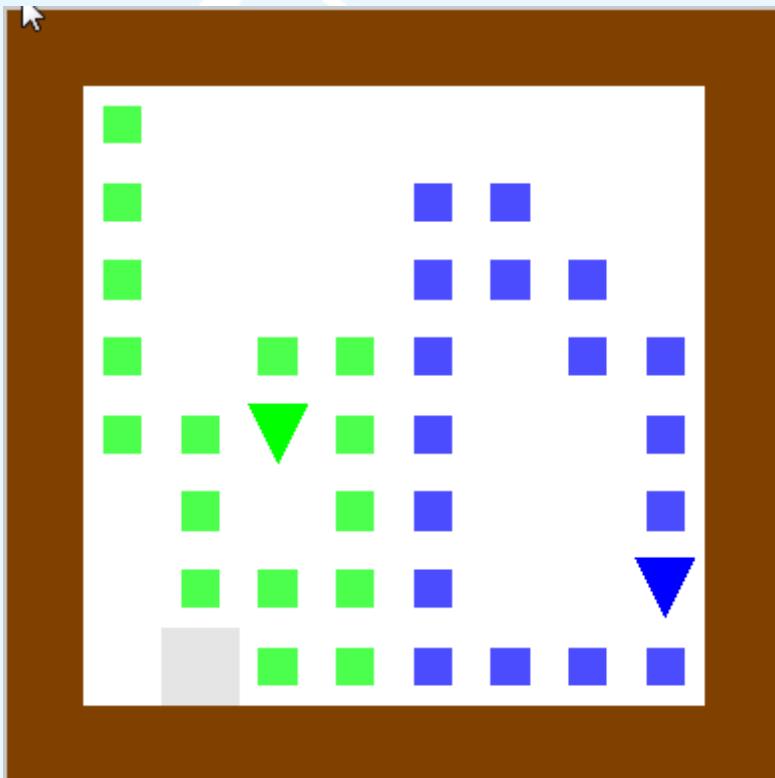
- **Movimientos**
  - Arriba, Abajo, Derecha, Izquierda
- **Válido**
  - Casilla destino no tiene obstáculo, no está ocupada por otro jugador y pertenece al tablero.
- **Aspiración automática:**
  - Si la casilla destino tiene  $k$  unidades de basura entonces  
 $\text{Marcador(jugador)} += k$
  - E.d: no se contempla la aspiración como movimiento posible.

# El juego:ASPIRADORA TRON



- Cada vez que un jugador mueve, la casilla origen es intransitable (aparece un obstáculo).

# El juego:ASPIRADORA TRON



## El juego termina cuando

1. No quedan unidades de suciedad en el mapa.
2. El número de unidades de basura de algún jugador es superior a la suma de unidades del otro jugador más las unidades restantes.
3. Los dos jugadores quedan bloqueados.
4. El jugador que queda bloqueado ha recogido menos unidades de suciedad que su adversario.

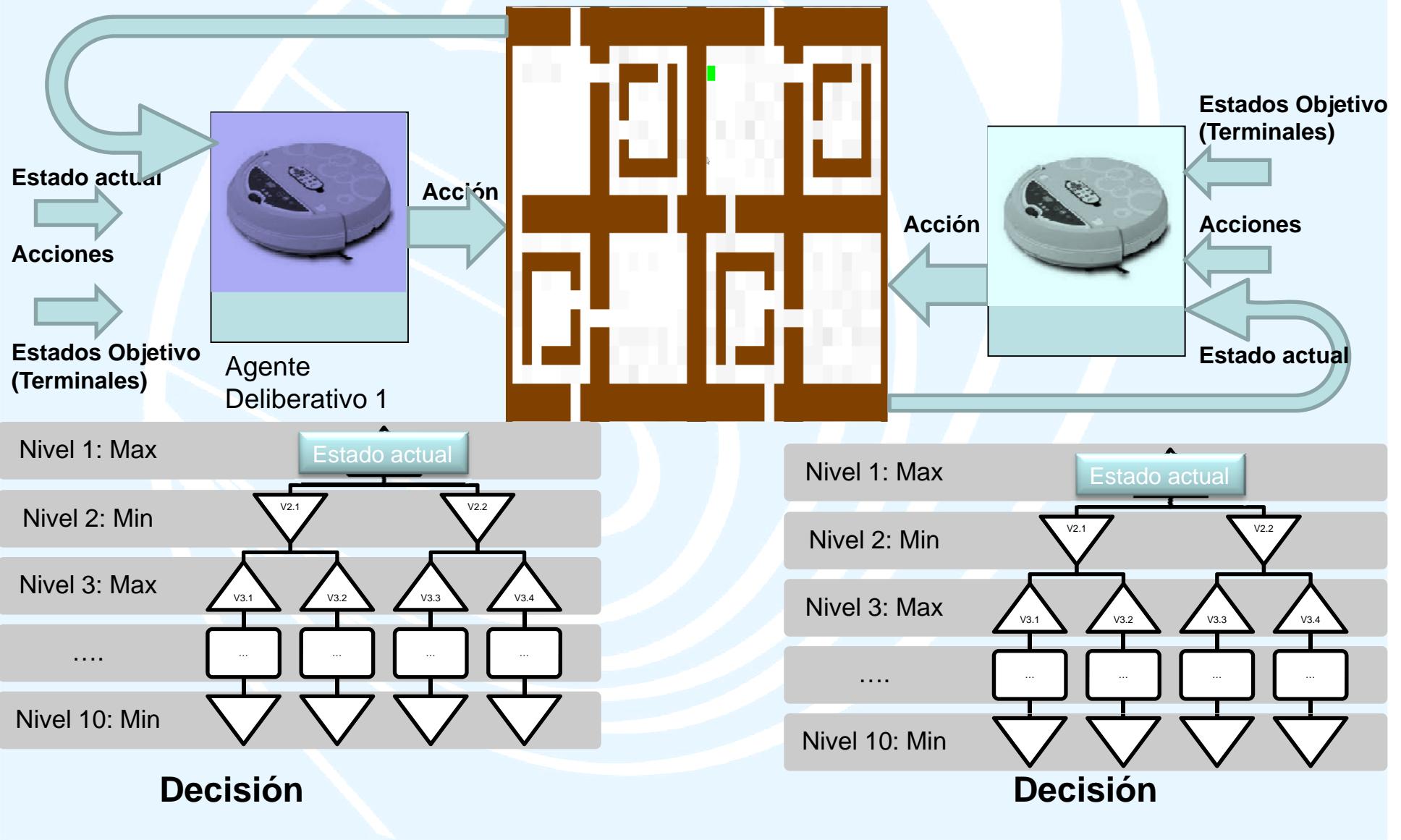
Ganador:

- Jugador con mayor número de unidades de basura aspirada

# El juego:ASPIRADORA TRON

- Objetivo
  - implementar un algoritmo MINIMAX, **con profundidad limitada (con cota máxima de 10),**
  - de manera que un jugador aspiradora pueda determinar el movimiento más prometedor para ganar el juego,
  - explorando el árbol de juego **desde** el estado actual **hasta** una profundidad máxima de 10 dada como entrada al algoritmo

# El juego:ASPIRADORA TRON



### 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 **Aspiradoras.zip** desde la **web** de la asignatura y cópielo en la carpeta anterior.

The screenshot shows the homepage of the Department of Computer Sciences and Artificial Intelligence (DECSAI) at the University of Granada. The header includes the university's seal and the text "Departamento de Ciencias de la Computación e I.A." and "Universidad de Granada". A sidebar on the left has links for "Inicio", "Docencia", "Miembros", "Investigación", "Información", and "Noticias". A login form under "Acceso Identificado" with fields for "Login" and "Contraseña" and a "acceder" button is also visible. The main content area features a welcome message "Bienvenidos a la página web del departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Granada" and the DECSAI logo, which is a stylized blue bird or dragon-like creature. The word "DECSAI" is written vertically below the logo, with "Universidad de Granada" underneath it.

- (a) <http://decsai.ugr.es>
- (b) Entrar en acceso identificado
- (c) Elegir la asignatura “Inteligencia Artificial”
- (d) Seleccionar “Material de la Asignatura”
- (e) Seleccionar “Práctica 3”
- (f) Seleccionar “Material para la Práctica 3”

# 3. Presentación del Simulador

## 3.1. Compilación del Simulador

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

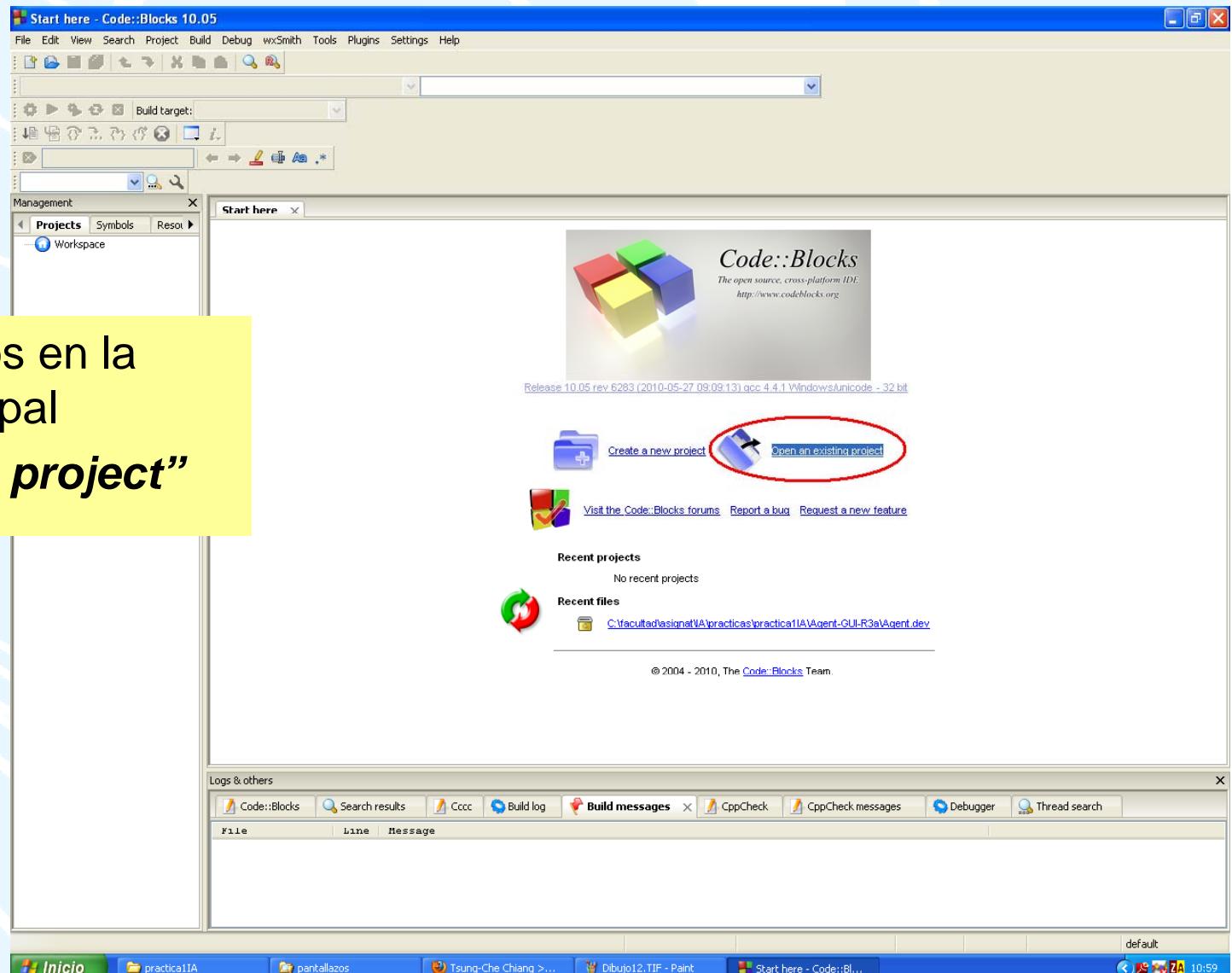
- Aspiradoras

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

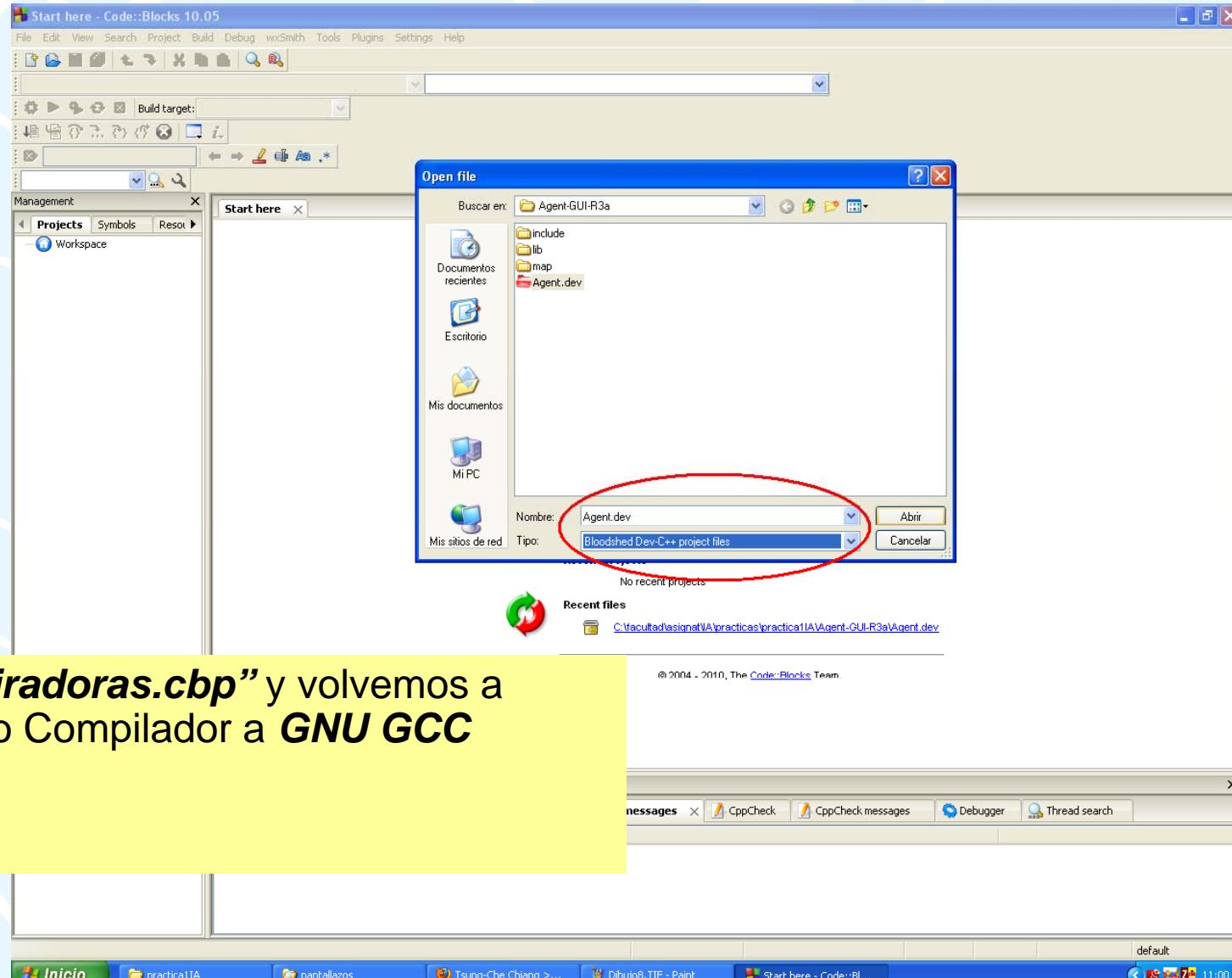
## 3.1. Compilación del Simulador



5. Seleccionamos en la pantalla principal  
**“Open an existing project”**

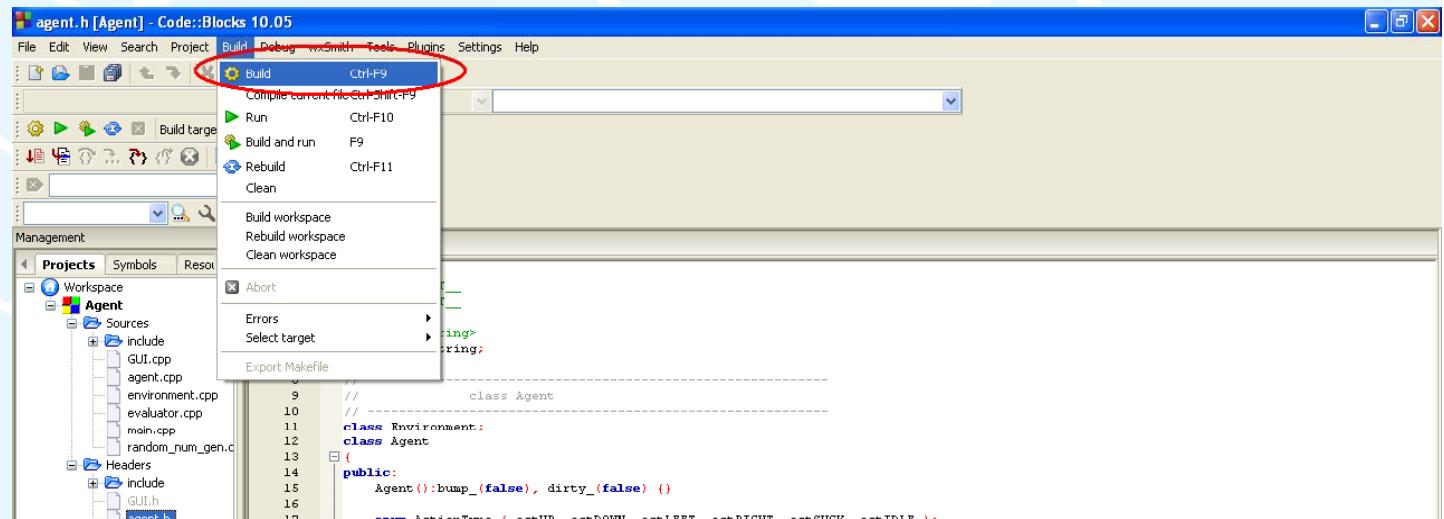
# 3. Presentación del Simulador

## 3.1. Compilación del Simulador

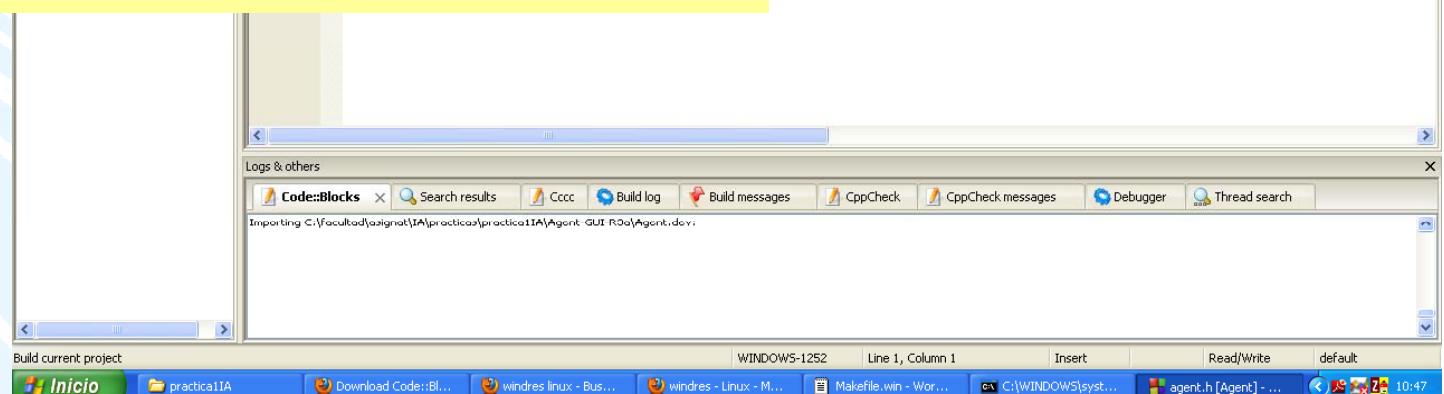


# 3. Presentación del Simulador

## 3.1. Compilación del Simulador

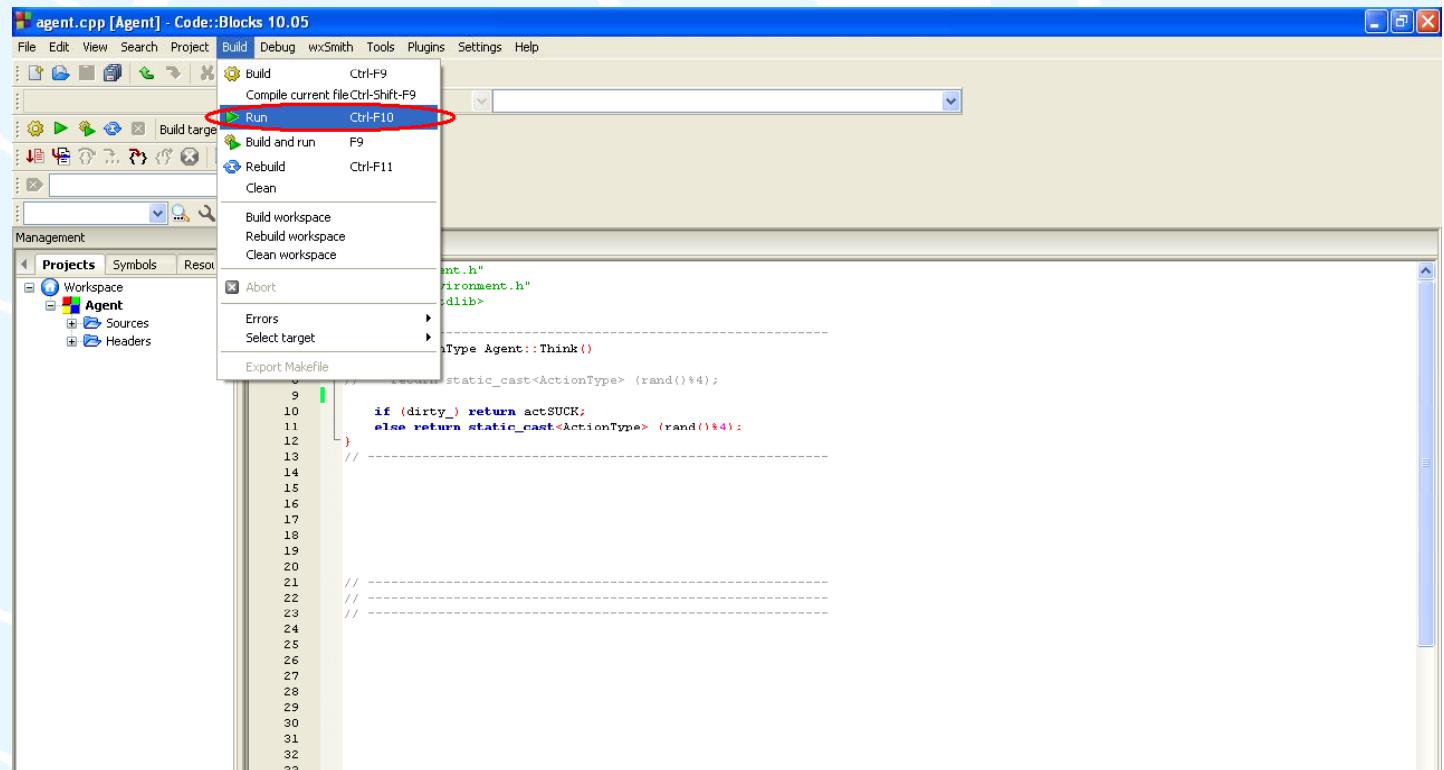


7. Compilamos el proyecto seleccionando  
**"Build → Build"**

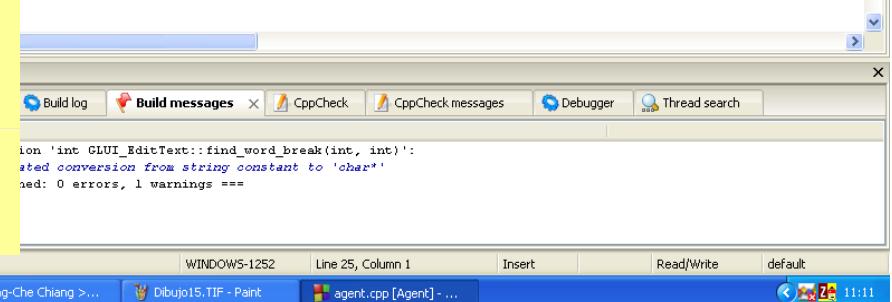


# 3. Presentación del Simulador

## 3.1. Compilación del Simulador

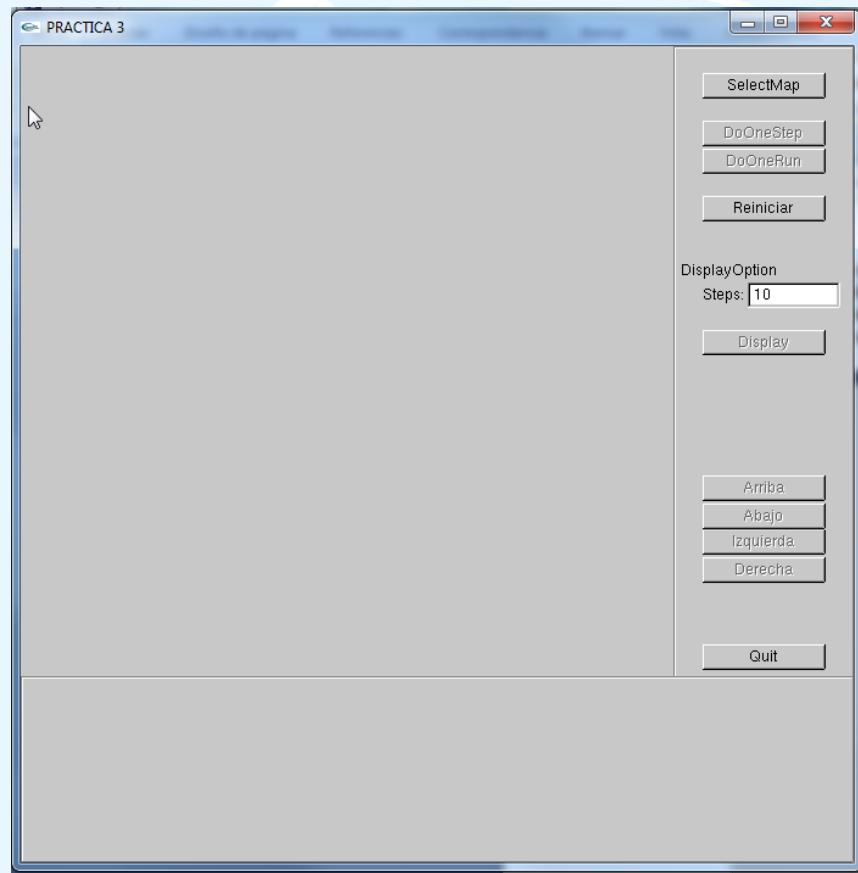


8. Ejecutamos el código, seleccionando  
**“Build -> Run”**



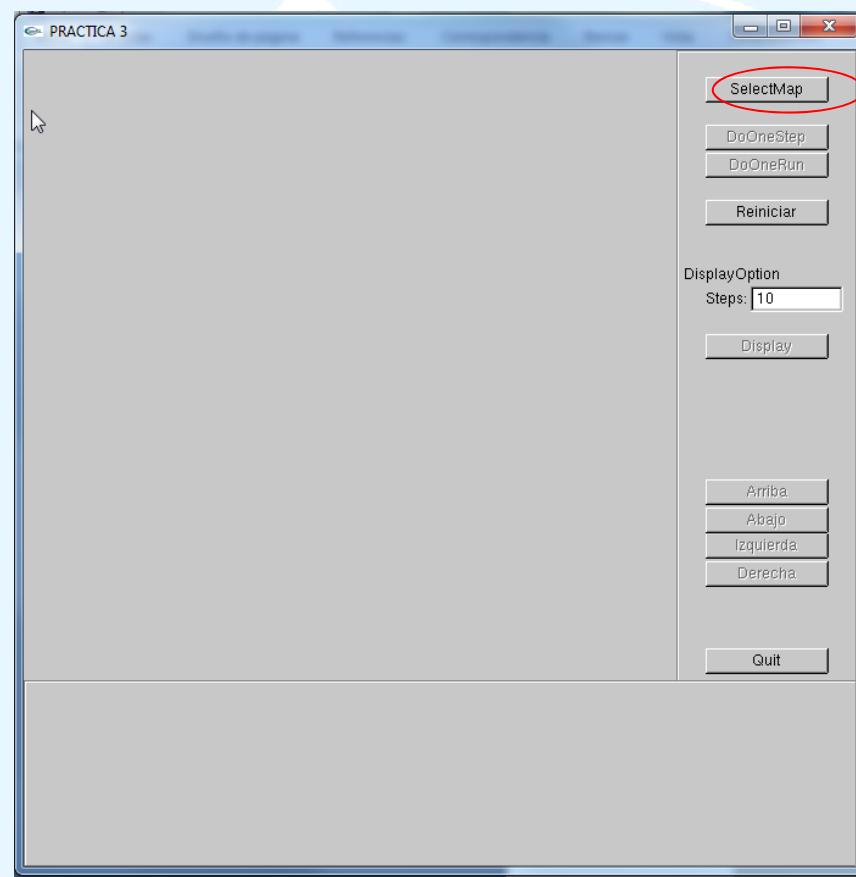
# 3. Presentación del Simulador

## 3.2. Ejecución del Simulador



# 3. Presentación del Simulador

## 3.2. Ejecución del Simulador



**"SelectMap"** Selecciona el fichero de problema sobre el que Realizar la simulación.

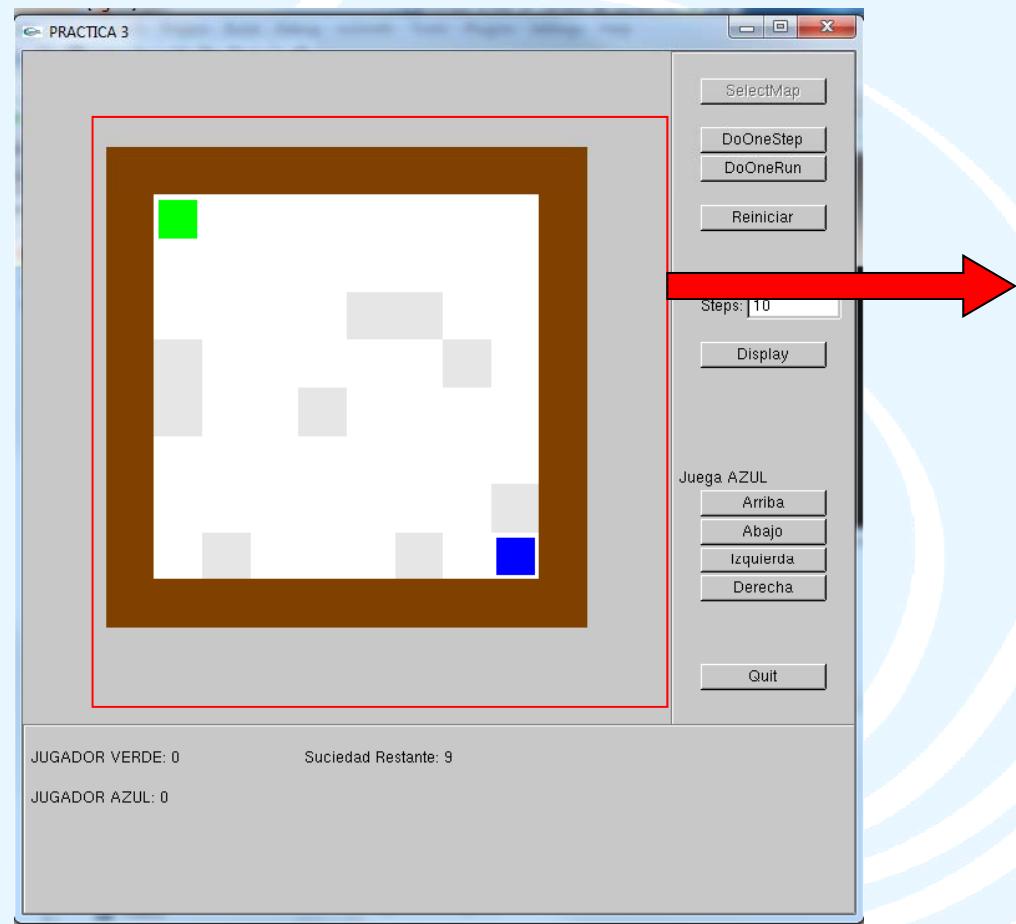
Seleccionamos el fichero  
***"mapita10a.map"***



- Seleccionamos el modo de actuación para cada jugador
  - Automático: sigue el proceso implementado
  - Humano: responde a las decisiones del humano

# 3. Presentación del Simulador

## 3.2. Ejecución del Simulador

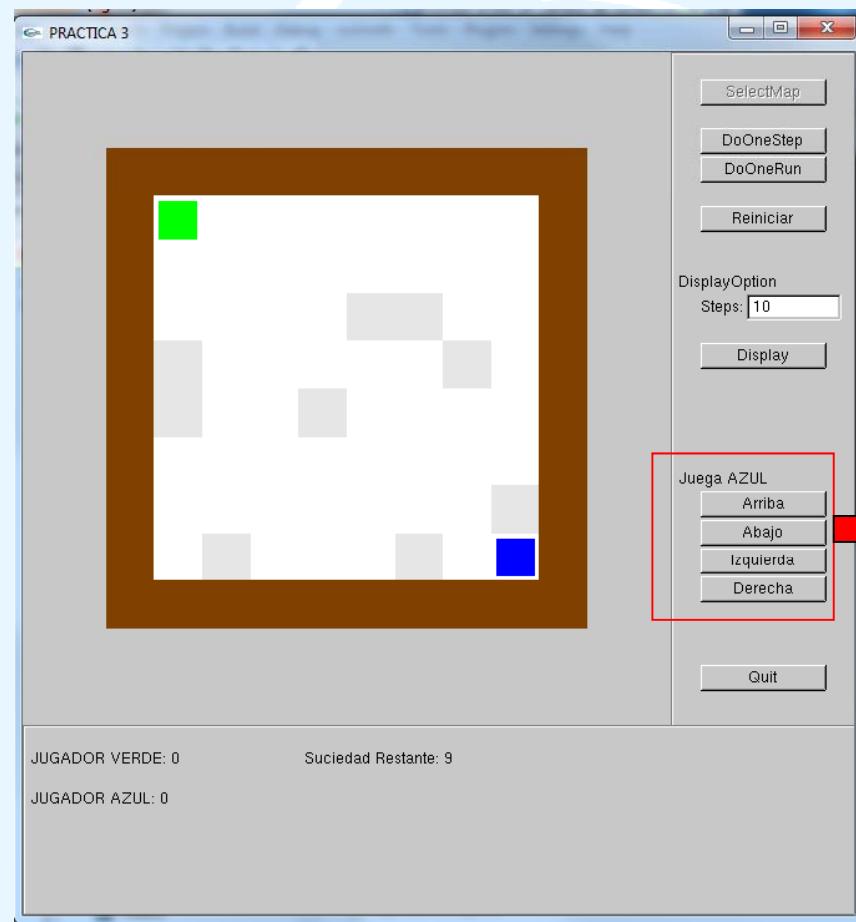


### **Mundo simulado:**

- Cuadrado verde: Jugador 1
- Cuadrado azul: Jugador 2
- Los cuadrados marrones representan las paredes de la habitación.
- Casillas grises con suciedad
- El resto de casillas representan la zona transitable.

# 3. Presentación del Simulador

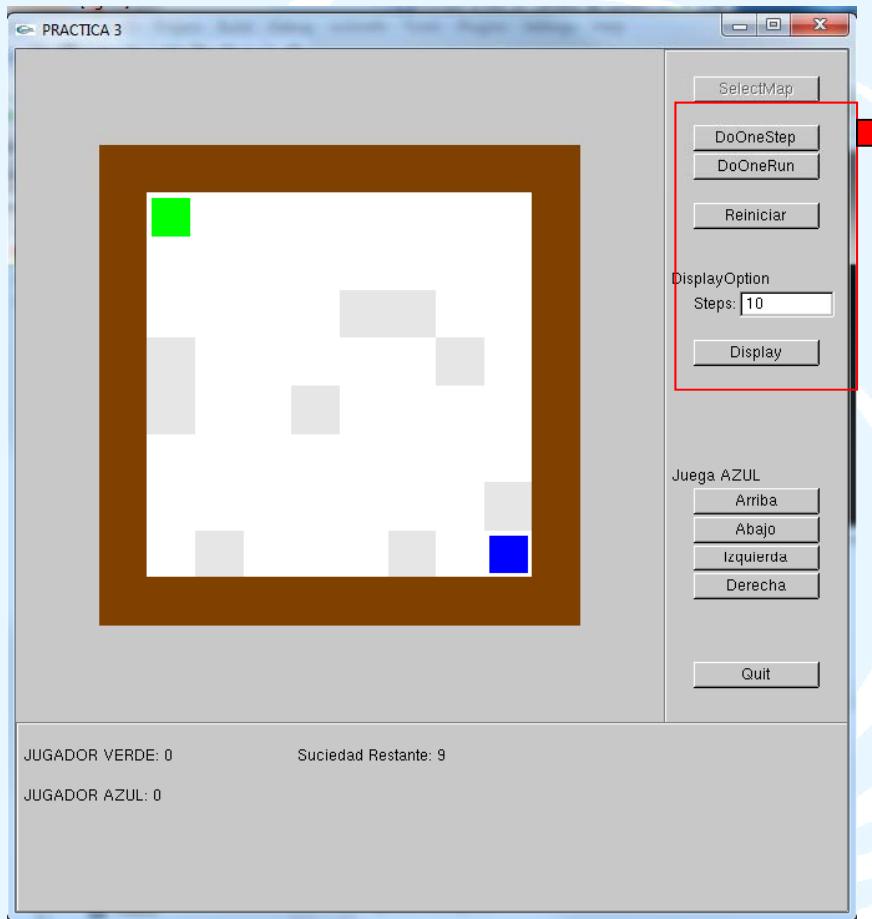
## 3.2. Ejecución del Simulador



- Indica el turno.
- Control de movimientos de jugador Humano.
- Si el jugador con turno es máquina, pulsar alguno de estos botones implica ejecutar la acción decidida internamente

# 3. Presentación del Simulador

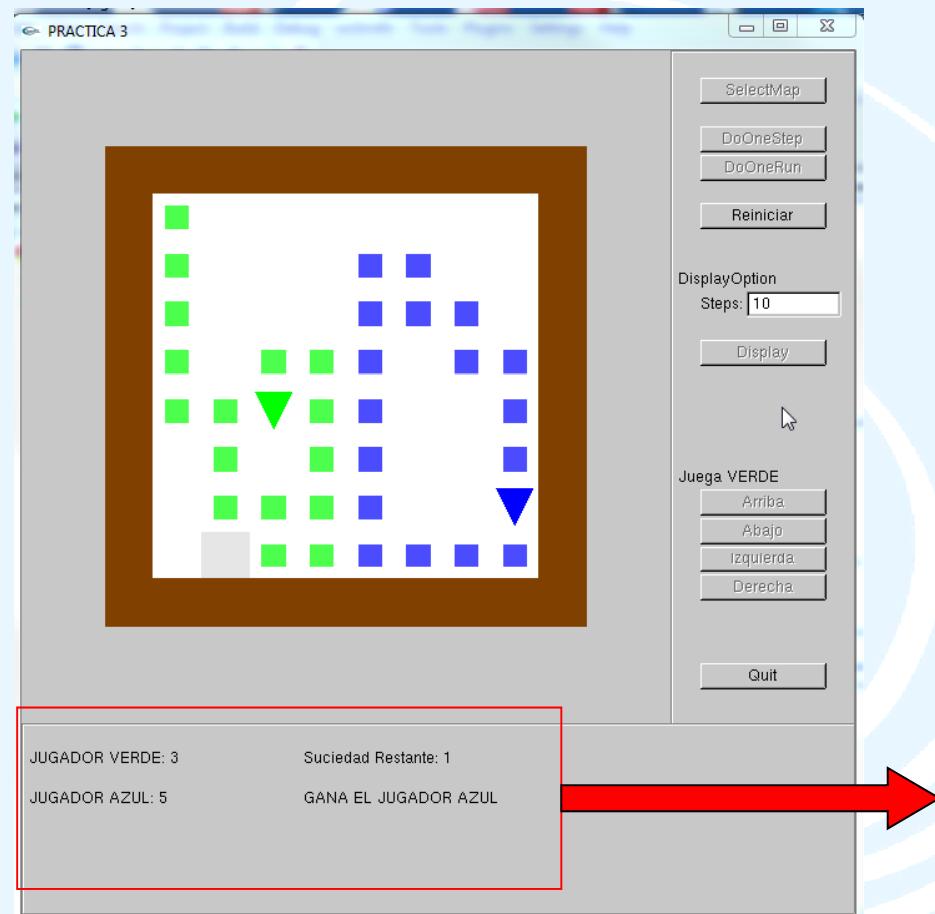
## 3.2. Ejecución del Simulador



- **DoOneStep:** Si el jugador con turno es Máquina, hace que ejecute la acción decidida internamente, en otro caso no hace nada.
- **DoOneRun:** Si ambos jugadores son automáticos, ejecuta una partida completa
- **Display:** Si ambos jugadores son automáticos, ejecuta el número de pasos indicado en el cuadro de texto “Steps”.
- **Reiniciar:** Reinicia el juego.
- **Quit:** Abandona la aplicación

# 3. Presentación del Simulador

## 3.2. Ejecución del Simulador



- Final del juego
- Información del número de unidades de suciedad para cada jugador y suciedad restante.
- Indica qué jugador ha ganado.

# Índice

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

## 4. Pasos del desarrollo de la práctica

1. Descripción de la clase ***Environment***
  - Extensión de clase Environment de otras prácticas
2. Descripción de la clase ***Player***.
  1. 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

```
14 class Environment
15 {
16     public:
17
18     enum ActionType { actUP, actDOWN, actLEFT, actRIGHT, actSUCK, actIDLE };
19
20     Environment();
21     Environment(ifstream &infile);
22     ~Environment();
23     Environment(const Environment &env);
24     Environment& operator=(const Environment& env);
25
26     void PintaAspiradora(int jugador, int w, int h, int x, int y) const;
27
28     void Show(int,int) const;
29     void Change();
30     void AcceptAction(int jugador, ActionType accion);
31
32     bool isCurrentPosDirty(int jugador) const { return (jugador==1?maze_[Position1_X
33 ()][Position1_Y()]:maze_[Position2_X()][Position2_Y()]) > 0; }
34     bool isJustBump(int jugador) const { return (jugador==1?bump1_:bump2_); }
35     int DirtAmount(int x, int y) const;
36     int DirtAmount() const;
37     //int RandomSeed() const { return randomSeed_; }
38     int SizeMaze() const {return MAZE_SIZE;}
39     int **SeeMap() const;
40     int Position1_X() const {return agent1PosX_;}
41     int Position1_Y() const {return agent1PosY_;}
42     int Position2_X() const {return agent2PosX_;}
43     int Position2_Y() const {return agent2PosY_;}
44     int Marcador(int jug) const;
45     int Total_Suciedad() const;
46     void SaveMap() const;
47     int GenerateNextMove(Environment *V, int jug) const;
48     void possible_actions(bool *act, int jug) const;
49     int Last_Action(int jug) const;
50     bool operator==(const Environment & env) const;
51     bool a_movement_is_possible(int jug) const;
52
53
54 private:
55     static const int OBSTACLE = -1, OBSTACLE_VERDE=-2, OBSTACLE_AZUL=-3;
56     static const char MAP_ROAD = '-', MAP_OBSTACLE = 'O';
57
58     int MAZE_SIZE;
59
60     bool bump1_, bump2_;
61     int agent1PosX_, agent1PosY_, agent2PosX_, agent2PosY_;
62     int marcador1_, marcador2_, unidades_suciedad_;
63     int **maze_; // -1: Obstacle, >=0: amount of dirt
64     //int randomSeed_;
65     //double dirtyProb_;
66
67     /**
68     ActionType preAction1_, preAction2_;
69 
```

- Ampliación de la misma clase en las prácticas anteriores.
- Información esencial para representar cada nodo del árbol de búsqueda (entorno del juego).
- **Los dos jugadores del entorno se declaran en el fichero GUI.cpp como dos variables globales player1 y player2, de la clase Player.**

# 4. Pasos del desarrollo de la práctica

## 4.1. Descripción de la clase Environment

```
14 class Environment
15 {
16     public:
17
18     enum ActionType { actUP, actDOWN, actLEFT, actRIGHT, actSUCK, actIDLE };
19
20     Environment();
21     Environment(ifstream &infile);
22     ~Environment();
23     Environment(const Environment &env);
24     Environment& operator=(const Environment& env);
25
26     void PintaAspiradora(int jugador, int w, int h, int x, int y) const;
27
28     void Show(int,int) const;
29     void Change();
30     void AcceptAction(int jugador, ActionType accion);
31
32     bool isCurrentPosDirty(int jugador) const { return (jugador==1?maze_[Position1_X
33     ()][Position1_Y()]:maze_[Position2_X()][Position2_Y()]) > 0; }
34     bool isJustBump(int jugador) const { return (jugador==1?bump1_:bump2_); }
35     int DirtAmount(int x, int y) const;
36     int DirtAmount() const;
37     //int RandomSeed() const { return randomSeed_; }
38     int SizeMaze() const { return MAZE_SIZE; }
39
40     int **SeeMap() const;
41     int Position1_X() const { return agent1PosX_; }
42     int Position1_Y() const { return agent1PosY_; }
43     int Position2_X() const { return agent2PosX_; }
44     int Position2_Y() const { return agent2PosY_; }
45     int Marcador(int jug) const;
46     int Total_Suciedad() const;
47     void SaveMap() const;
48     int GenerateNextMove(Environment *V, int jug) const;
49     void possible_actions(bool *act, int jug) const;
50     int Last_Action(int jug) const;
51     bool operator==(const Environment & env) const;
52     bool a_movement_is_possible(int jug) const;
53
54 private:
55     static const int OBSTACLE = -1, OBSTACLE_VERDE=-2, OBSTACLE_AZUL=-3;
56     static const char MAP_ROAD = '-', MAP_OBSTACLE = 'O';
57
58     int MAZE_SIZE;
59
60     bool bump1_, bump2_;
61     int agent1PosX_, agent1PosY_, agent2PosX_, agent2PosY_;
62     int marcador1_, marcador2_, unidades_suciedad_;
63     int **maze_; // -1: Obstacle, >=0: amount of dirt
64     //int randomSeed_;
65     //double dirtyProb_;
66
67     /**
68     ActionType preAction1_, preAction2_;
69 }
```

### ***bool isCurrentPosDirty(int jugador)***

- Devuelve true si jugador se encuentra en una casilla con suciedad. False en otro caso
- Una casilla tiene suciedad si el valor que contiene la casilla es mayor que 0.

### ***bool isJustBump(int jugador)***

- Devuelve true si el jugador ha chocado con un obstáculo.

### ***int DirtAmount(int x, int y)***

- Devuelve la cantidad de suciedad que hay en la casilla (x,y). o 0 en otro caso.

### ***int SizeMaze()***

- Devuelve el tamaño del mapa o tablero.

# 4. Pasos del desarrollo de la práctica

## 4.1. Descripción de la clase Environment

```
14 class Environment
15 {
16     public:
17
18     enum ActionType { actUP, actDOWN, actLEFT, actRIGHT, actSUCK, actIDLE };
19
20     Environment();
21     Environment(ifstream &infile);
22     ~Environment();
23     Environment(const Environment &env);
24     Environment& operator=(const Environment& env);
25
26     void PintaAspiradora(int jugador, int w, int h, int x, int y) const;
27
28     void Show(int,int) const;
29     void Change();
30     void AcceptAction(int jugador, ActionType accion);
31
32     bool isCurrentPosDirty(int jugador) const { return (jugador==1?maze_[Position1_X
33 ()][Position1_Y()]:maze_[Position2_X()][Position2_Y()]) > 0; }
34     bool isJustBump(int jugador) const { return (jugador==1?bump1_:bump2_); }
35     int DirtAmount(int x, int y) const;
36     int DirtAmount() const;
37     //int RandomSeed() const { return randomSeed_; }
38     int SizeMap() const { return MAZE_SIZE; }
39     int **SeeMap() const;
40     int Position1_X() const {return agent1PosX_;}
41     int Position1_Y() const {return agent1PosY_;}
42     int Position2_X() const {return agent2PosX_;}
43     int Position2_Y() const {return agent2PosY_;}
44     int Marcador(int jug) const;
45     int Total_Suciedad() const;
46
47     void Savemap() const;
48     int GenerateNextMove(Environment *V, int jug) const;
49     void possible_actions(bool *act, int jug) const;
50     int Last_Action(int jug) const;
51     bool operator==(const Environment & env) const;
52     bool a_movement_is_possible(int jug) const;
53
54 private:
55     static const int OBSTACLE = -1, OBSTACLE_VERDE=-2, OBSTACLE_AZUL=-3;
56     static const char MAP_ROAD = '-', MAP_OBSTACLE = 'O';
57
58     int MAZE_SIZE;
59
60     bool bump1_, bump2_;
61     int agent1PosX_, agent1PosY_, agent2PosX_, agent2PosY_;
62     int marcador1_, marcador2_, unidades_suciedad_;
63     int **maze_; // -1: Obstacle, >=0: amount of dirt
64     //int randomSeed_;
65     //double dirtyProb_;
66
67     /**
68     ActionType preAction1_, preAction2_;
69 
```

### ***int \*\* SeeMap()***

- Devuelve el mapa del entorno

### ***int Position1\_X()***

- Devuelve la posición X (fila del tablero) del jugador 1.
- Análogo para las funciones:

- Position1\_Y(), Position2\_X(), Position2\_Y()

### ***int Marcador(int jug)***

- Devuelve la cantidad total actual de basura aspirada por un jugador.

### ***int Total\_Suciedad()***

- Devuelve la cantidad actual restante de suciedad en el tablero.

# 4. Pasos del desarrollo de la práctica

## 4.1. Descripción de la clase Environment

```
14 class Environment
15 {
16     public:
17
18     enum ActionType { actUP, actDOWN, actLEFT, actRIGHT, actSUCK, actIDLE };
19
20     Environment();
21     Environment(ifstream &infile);
22     ~Environment();
23     Environment(const Environment &env);
24     Environment& operator=(const Environment& env);
25
26     void PintaAspiradora(int jugador, int w, int h, int x, int y) const;
27
28     void Show(int,int) const;
29     void Change();
30     void AcceptAction(int jugador, ActionType accion);
31
32     bool isCurrentPosDirty(int jugador) const { return (jugador==1?maze_[Position1_X
33 ()][Position1_Y()]:maze_[Position2_X()][Position2_Y()]) > 0; }
34     bool isJustBump(int jugador) const { return (jugador==1?bump1_:bump2_); }
35     int DirtAmount(int x, int y) const;
36     int DirtAmount() const;
37     //int RandomSeed() const { return randomSeed_; }
38     int SizeMaze() const {return MAZE_SIZE;}
39     int **SeeMap() const;
40     int Position1_X() const {return agent1PosX_;}
41     int Position1_Y() const {return agent1PosY_;}
42     int Position2_X() const {return agent2PosX_;}
43     int Position2_Y() const {return agent2PosY_;}
44     int Marcador(int jug) const;
45     int Total_Suciedad() const;
46     void SaveMap() const;
47     int GenerateNextMove(Environment *V, int jug) const;
48     void possible_actions(bool *act, int jug) const;
49     int best_action(int jug) const;
50     bool operator==(const Environment & env) const;
51     bool a_movement_is_possible[int jug]] const;
52
53
54 private:
55     static const int OBSTACLE = -1, OBSTACLE_VERDE=-2, OBSTACLE_AZUL=-3;
56     static const char MAP_ROAD = '-', MAP_OBSTACLE = 'O';
57
58     int MAZE_SIZE;
59
60     bool bump1_, bump2_;
61     int agent1PosX_, agent1PosY_, agent2PosX_, agent2PosY_;
62     int marcador1_, marcador2_, unidades_suciedad_;
63     int **maze_; // -1: Obstacle, >=0: amount of dirt
64     //int randomSeed_;
65     //double dirtyProb_;
66
67     /**
68     ActionType preAction1_, preAction2_;
69 
```

### Void possible\_actions(bool \*act, int jug)

- Devuelve los posibles movimientos válidos (o acciones) que puede realizar el jugador *jug* en un estado concreto.
- *act* es un array de 4 valores booleanos. Cada *act[i]*,  $0 \leq i \leq 3$ , representa si *jug* puede llevar a cabo, respectivamente, la acción UP, DOWN, LEFT o RIGHT.

# 4. Pasos del desarrollo de la práctica

## 4.1. Descripción de la clase Environment

```
14 class Environment
15 {
16     public:
17
18     enum ActionType { actUP, actDOWN, actLEFT, actRIGHT, actSUCK, actIDLE };
19
20     Environment();
21     Environment(ifstream &infile);
22     ~Environment();
23     Environment(const Environment &env);
24     Environment& operator=(const Environment& env);
25
26     void PintaAspiradora(int jugador, int w, int h, int x, int y) const;
27
28     void Show(int,int) const;
29     void Change();
30     void AcceptAction(int jugador, ActionType accion);
31
32     bool isCurrentPosDirty(int jugador) const { return (jugador==1?maze_[Position1_X
33 ()][Position1_Y()]:maze_[Position2_X()][Position2_Y()]) > 0; }
34     bool isJustBump(int jugador) const { return (jugador==1?bump1_:bump2_); }
35     int DirtAmount(int x, int y) const;
36     int DirtAmount() const;
37     //int RandomSeed() const { return randomSeed_; }
38     int SizeMaze() const {return MAZE_SIZE;}
39     int **SeeMap() const;
40     int Position1_X() const {return agent1PosX_;}
41     int Position1_Y() const {return agent1PosY_;}
42     int Position2_X() const {return agent2PosX_;}
43     int Position2_Y() const {return agent2PosY_;}
44     int Marcador(int jug) const;
45     int Total_Suciedad() const;
46     void SaveMap() const;
47     int GenerateNextMove(Environment *V, int jug) const;
48     void possible_actions(bool *act, int jug) const;
49     int Last_Action(int jug) const;
50     bool operator==(const Environment & env) const;
51     bool a_movement_is_possible(int jug) const;
52
53
54 private:
55     static const int OBSTACLE = -1, OBSTACLE_VERDE=-2, OBSTACLE_AZUL=-3;
56     static const char MAP_ROAD = '-', MAP_OBSTACLE = 'O';
57
58     int MAZE_SIZE;
59
60     bool bump1_, bump2_;
61     int agent1PosX_, agent1PosY_, agent2PosX_, agent2PosY_;
62     int marcador1_, marcador2_, unidades_suciedad_;
63     int **maze_; // -1: Obstacle, >=0: amount of dirt
64     //int randomSeed_;
65     //double dirtyProb_;
66
67     /**
68     ActionType preAction1_, preAction2_;
69 
```

***int GenerateNextMove(Environment \*V, int jug)***

- Es la función utilizada para generar los estados sucesores correspondientes a los movimientos que puede llevar a cabo el jugador *jug* en el estado actual.
- Devuelve un array *V* de *n* posiciones, donde *n* puede ser 1, 2, 3 o 4, 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

```
14 class Environment
15 {
16     public:
17
18     enum ActionType { actUP, actDOWN, actLEFT, actRIGHT, actSUCK, actIDLE };
19
20     Environment();
21     Environment(ifstream &infile);
22     ~Environment();
23     Environment(const Environment &env);
24     Environment& operator=(const Environment& env);
25
26     void PintaAspiradora(int jugador, int w, int h, int x, int y) const;
27
28     void Show(int,int) const;
29     void Change();
30     void AcceptAction(int jugador, ActionType accion);
31
32     bool isCurrentPosDirty(int jugador) const { return (jugador==1?maze_[Position1_X
33 ()][Position1_Y()]:maze_[Position2_X()][Position2_Y()]) > 0; }
34     bool isJustBump(int jugador) const { return (jugador==1?bump1_:bump2_); }
35     int DirtAmount(int x, int y) const;
36     int DirtAmount() const;
37     //int RandomSeed() const { return randomSeed_; }
38     int SizeMaze() const {return MAZE_SIZE;}
39     int **SeeMap() const;
40     int Position1_X() const {return agent1PosX_;}
41     int Position1_Y() const {return agent1PosY_;}
42     int Position2_X() const {return agent2PosX_;}
43     int Position2_Y() const {return agent2PosY_;}
44     int Marcador(int jug) const;
45     int Total_Suciedad() const;
46     void SaveMap() const;
47     int GenerateNextMove(Environment *V, int jug) const;
48     void possible_actions(bool *act, int jug) const;
49     int last_action(int jug) const;
50     bool operator==(const Environment & env) const;
51     bool a_movement_is_possible(int jug) const;
52
53
54 private:
55     static const int OBSTACLE = -1, OBSTACLE_VERDE=-2, OBSTACLE_AZUL=-3;
56     static const char MAP_ROAD = '-', MAP_OBSTACLE = 'O';
57
58     int MAZE_SIZE;
59
60     bool bump1_, bump2_;
61     int agent1PosX_, agent1PosY_, agent2PosX_, agent2PosY_;
62     int marcador1_, marcador2_, unidades_suciedad_;
63     int **maze_; // -1: Obstacle, >=0: amount of dirt
64     //int randomSeed_;
65     //double dirtyProb_;
66
67     /**
68     ActionType preAction1_, preAction2_;
69 
```

### *int Last\_Action(int jug)*

- Devuelve la última acción ejecutada por jug.

### *bool a\_movement\_is\_possible(int jug)*

- Devuelve true si el jugador jug puede llevar a cabo algún movimiento. False en otro caso.

# 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
```

- 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).

# 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
12     private:
13         int jugador_;
14         Environment actual_;
15     };
16 #endif
```

- 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 jugador y que permite acceder al estado actual del juego que tiene el jugador.

# 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
```

### IMPORTANTE:

- **La implementación del algoritmo MINIMAX 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.

# 4. Pasos del desarrollo de la práctica

## 4.2. Ejemplo Implementación *Player.cpp*

```
/**  
Esta funcion devuelve la siguiente mejor accion guiada por la heuristica usando el algoritmo minimax.  
Devuelve: La siguiente accion a realizar.  
Parametros:  
    "actual" contiene el estado del tablero.  
    "jug" indica que jugador esta pidiendo el valor heuristico.  
    "limite_profundidad" establece el limite de exploracion del grafo.  
OBSERVACION: esta parametrizacion es solo indicativa, y el alumno podra modificarla segun sus necesidades  
*/  
Environment::ActionType MiniMax(const Environment & actual, int jug, int limite_profundidad){  
    Environment::ActionType accion;  
  
    // Definicion del metodo minimax para obtener la siguiente accion a realizar  
  
    // OBSERVACION: la sentencia siguiente esta puesta para que haga algo. En la version final no debe aparecer.  
    accion = static_cast<Environment::ActionType>(rand()%4);  
  
    return accion;
```



**Possible indicación** de cabecera de función MiniMax y cómo llamarla en el Método *Think()*

- **Devuelve:** siguiente acción a realizar
- **Parámetros**
  - **actual:** estado del tablero
  - **jug:** jugador que está "pidiendo" el valor heurístico de *actual*
  - **limite\_profundidad:** establece el límite de exploración del grafo

- **Indicativo, puede modificarse según necesidades.**

# 4. Pasos del desarrollo de la práctica

## 4.2. Ejemplo Implementación *Player.cpp*

```
/**  
Esta funcion devuelve la siguiente mejor accion guiada por la heuristica usando el algoritmo minimax.  
Devuelve: La siguiente accion a realizar.  
Parametros:  
    "actual" contiene el estado del tablero.  
    "jug" indica que jugador esta pidiendo el valor heuristico.  
    "limite_profundidad" establece el limite de exploracion del grafo.  
OBSERVACION: esta parametrizacion es solo indicativa, y el alumno podra modificarla segun sus necesidades  
*/  
Environment::ActionType MiniMax(const Environment & actual, int jug, int limite_profundidad){  
    Environment::ActionTypeaccion;  
  
    // Definicion del metodo minimax para obtener la siguiente accion a realizar  
  
    // OBSERVACION: la sentencia siguiente esta puesta para que haga algo. En la version final no debe aparecer.  
    accion = static_cast<Environment::ActionType>(rand()%4);  
  
    return accion;
```

**Possible indicación** de cabecera de función MiniMax y cómo llamarla en el Método *Think()*

- **Devuelve:** siguiente acción a realizar
- **Parámetros**
  - *actual*: estado del tablero
  - *jug*: jugador que está "pidiendo" el valor heurístico de *actual*
  - *limite\_profundidad*: establece el límite de exploración del grafo

- **En la versión actual devuelve una acción aleatoria.**
- **En la versión a entregar esa sentencia no debe aparecer**

# 4. Pasos del desarrollo de la práctica

## 4.2. Ejemplo Implementación *Player.cpp*

```
/**  
Calcula el valor heuristico de un estado de la frontera de busqueda  
Devuelve: El valor heuristico asociado al estado "actual" desde el p  
Parametros:  
    "actual" estado que contiene el tablero a evaluar.  
    "jug" indica que jugador esta pidiendo el valor heuristico.  
OBSERVACION: esta parametrizacion es solo indicativa, y el alumno po  
*/  
double Heuristica (const Environment & actual, int jug){  
    int marcador_propio, marcador_rival;  
  
    if (jug==1){  
        marcador_propio=actual.Marcador(1);  
        marcador_rival=actual.Marcador(2);  
    }  
    else {  
        marcador_propio=actual.Marcador(2);  
        marcador_rival=actual.Marcador(1);  
    }  
  
    if (marcador_propio>marcador_rival+actual.Total_Suciedad()){  
        return 1000000;  
    }  
    else if (marcador_rival>marcador_propio+actual.Total_Sucied  
    else {  
        return 0;  
    }  
}
```

Ejemplo de implementación de evaluación heurística de un estado

- **Devuelve:** evaluación del estado *actual*, según *jug*
- **Parámetros:**
  - *actual*: estado que contiene el tablero a evaluar
  - *jug*: jugador que evalua el estado

Heurística muy simple, corresponde sólo a evaluación de nodo terminal del juego:

- ***Si estoy en un nodo terminal y gano***
  - *Entonces Máxima Evaluación*
  - *Si no Mínima Evaluación*
- ***Si no es nodo terminal Devolver 0***

# 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

## 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 o rar) que llamado “***practica3***” contenga dos carpetas:

- Una de las carpetas con la memoria de la práctica (en formato pdf)
- La otra carpeta con **6 archivos**
  - “*player.cpp*”, “*player.h*”,
  - Copias renombradas de ambos ficheros
  - Copia renombrada de “*GUI.cpp*”
  - Un fichero de proyecto codebloks “*aspiradoras2.cbproj*”
- siguiendo las siguientes instrucciones

# 5. Evaluación de la práctica

Sea <DNI> la cadena de texto del DNI del alumno (sin letra NIF)

1. Hacer copias de player.\* y renombrarlas a player<DNI>.\*
2. En player<DNI>.cpp hacer
  - Sustituir
    - `#include "player.h"` por `#include "player<DNI>.h"`
  - Reemplazar
    - “Player” por la palabra “Player<DNI>”
  - Guardar el fichero
3. En player<DNI>.h hacer
  - El mismo reemplazo que en player<DNI>.cpp
4. En GUI.cpp
  - Sustituir
    - `#include "player.h"` por `#include "player<DNI>.h"`
  - Buscar y sustituir
    - `Player *player1=NULL;` por `Player<DNI> *player1=NULL`
    - `Player *player2=NULL;` por `Player<DNI> *player2=NULL`
  - Guardar el fichero con el **nuevo nombre GUI<DNI>.cpp**

## 5. Evaluación de la práctica

Sea <DNI> la cadena de texto del DNI del alumno (sin letra NIF)

5. Crear un nuevo proyecto “aspiradoras2.cbz”
  1. Situarse sobre la ventana del proyecto (la que tiene una estructura arborescente), seleccionar los archivos “player.cpp”, “player.h”, “GUI.cpp”, cliquear con el botón derecho del ratón y seleccionar “Remove file from project”.
  2. Cliquear sobre el menú superior la opción “Project”, seleccionar “Add files...” y marcar los archivos “player<DNI>.cpp”, “player<DNI>.h” y “GUI<DNI>.cpp” y pulsar el botón “Abrir”.
  3. Por último, pulsar sobre el menú superior la opción “File”, seleccionar “Save Project as...” y nombrar al nuevo proyecto como “aspiradoras2.cbz”.

## 5. Evaluación de la práctica

- SI el DNI es **29874712-V** en el fichero comprimido que hay que entregar deben aparecer los siguientes ficheros fuente:
  - “**player.cpp**” y “**player.h**”
  - “**player29874712.cpp**”, “**player29874712.h**”
  - “**GUI29874712.cpp**”
  - El fichero de proyecto CodeBlocks **aspiradoras2.cbproj**

## 5. Evaluación de la práctica

¿Qué debe contener la memoria de la práctica?

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

***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 **APTO** o **NO APTO**. APTO equivale a 2 puntos, 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 estará basado **en un torneo eliminatorio**. El alumno vencedor del torneo obtendrá una calificación máxima en este apartado de **5** puntos. El resto de los alumnos obtendrá una calificación proporcional al número de rondas que consigan superar.

## 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](http://decsai.ugr.es) → Entrega de Prácticas
- La fecha de entrega será
  - Comunicada oportunamente.